SIGGRAPH 2002 COURSE 9

Simulating Nature: Realistic and Interactive Techniques

Course Speakers:

1. **Oliver Deusen**
   deussen@inf.tu-dresden.de
2. **David S. Ebert**
   ebertd@purdue.edu
3. **Ron Fedkiw**
   fedkiw@cs.stanford.edu
4. **F. Kenton Musgrave**
   musgrave@fractalworlds.com
5. **Przemyslaw Prusinkiewicz**
   pwp@cpsc.ucalgary.ca
6. **Jos Stam**
   jstam@aw.sgi.com
7. **Jerry Tessendorf**
   jerry@cinesite.com

Abstract

This course will impart a working knowledge of several techniques for simulating natural phenomena, including practical, as well as, research issues. The course presenters will provide both a research and production perspective to the difficult task of photo-realistic modeling, rendering, and animation of natural phenomena. Most of the techniques that will be presented can be adapted for interactive simulations and real-time applications. Physics-based approaches for modeling and animating water, waves, and oceanscapes, real-time, rapid, stable dynamics for water and gas animation, procedural and physics-based approaches for fast, realistic modeling of smoke, steam, and fire, procedural volumetric techniques and interactive approximation for modeling and animating clouds, interactive grammar-based techniques for modeling plants and plant ecosystems, and fractal techniques for simulating mountainous landscapes and interactive navigable worlds will be presented.

Speaker Contact Information:

1. **Oliver Deussen**
   Professor
   Dept. of Computer Science
   Dresden University of Technology
   Mommsenstr. 13
   01062 Dresden, Germany
   deussen@inf.tu-dresden.de

2. **David S. Ebert**
   Associate Professor
   School of Electrical and Computer Engineering
   1285 EE Building
   Purdue University
   West Lafayette, IN 47907
   ebertd@purdue.edu

3. **Ron Fedkiw**
   Assistant Professor
   Computer Science Department
   Stanford University
   Gates Computer Science Bldg., Room 207
   Stanford, CA 94305-9020
   fedkiw@cs.stanford.edu

4. **F. Kenton Musgrave**
   CEO
   Pandromeda, Inc.
   15724 Trapshire Ct.
   Waterford, VA 20197
   musgrave@fractalworlds.com

5. **Przemyslaw Prusinkiewicz**
   Professor
   Department of Computer Science
   University of Calgary
   2500 University Drive N.W.
   Calgary, Alberta T2N 1N4 Canada
   pwp@cpsc.ucalgary.ca

6. **Jos Stam**
   Alias | wavefront
   1218 Third Ave. , 8th floor
   Seattle, WA 98101
   jstam@aw.sgi.com

7. **Jerry Tessendorf**
   Research Scientist
   Cinesite Digital Studios
   1017 N. Las Palmas Ave, Suite 300
   Hollywood, CA 90038
   jerry@cinesite.com

Speaker Biographies:

Oliver Deussen graduated in computer science at the University of Karlsruhe in 1991 and finished his doctoral thesis in 1996 at the Institute for Operating and Dialog Systems of the University of Karlsruhe. From 1996 he was at the Department of Simulation and Graphics of the Otto-von-Guericke University of Magdeburg, now he is a professor for Media Design and Computer Graphics at the Technical University of Dresden. Oliver has published several papers at SIGGRAPH on the modeling, rendering, and non-photorealistic rendering of plants and plant communities at SIGGRAPH. He is also co-founder of the Greenworks company that produces the very widely used xfrog software for plant modeling.

David Ebert's research interests are procedural techniques for computer graphics, animation, scientific, medical, and information visualization. After receiving his Ph.D. from the Computer and Information Science Department at The Ohio State University in 1991, he was an Instructor in that department for two years. In 1993, Dr. Ebert joined the faculty of the Computer Science and Electrical Engineering Department at the University of Maryland, Baltimore County. Ebert is joining the School of Electrical and Computer Engineering at Purdue University on December 15, 2000. He has taught ten courses at SIGGRAPH since 1992 and has published numerous articles on procedural modeling, rendering, and animation of gaseous phenomena. David is also the co-author of the first and second editions of *Texturing and Modeling: A Procedural Approach*, published by AP Professional. His work in simulating gaseous phenomena has also been featured in the SIGGRAPH Electronic Theater and the Papers program. Most recently, Ebert has been working on realtime issues for procedural techniques and simulating natural phenomena, consulting with Electronic Arts, Inc., and collaborating on the Stanford Real-time Shading project.

Ron Fedkiw obtained his Ph.D. in Applied Mathematics at UCLA, then spent time as a member of both the UCLA Mathematics Department and the Caltech Aeronautics Department before joining the Stanford Computer Science Department. The work at UCLA and Caltech was focused on the design of new algorithms for a large variety of application areas mostly related to computational fluid dynamics. Sponsored by the Office of Naval Research (ONR) at UCLA and the Department of Energy (DOE) at Caltech, this work led to strong collaborative relationships with a number of scientists at the national laboratories including both Los Alamos National Laboratory (LANL) and Lawrence Livermore National Laboratory (LLNL). Besides the Department of Defense related work, he have been active in both computer graphics and image processing including 2 years of consulting with Arete Entertainment and 1 year of consulting with Centropolis FX.

Ken Musgrave is a world-renowned computer artist and computer graphics researcher. Musgrave lectures internationally on fractals, computer graphics and the visual arts, and his own research in computer graphics. He is an author of a standard textbook for the special effects industry, and developed digital effects for films such as Titanic and Apollo 13. His images have been widely published and exhibited internationally, including at the Lincoln Center and Guggenheim Museum in New York City. Musgrave spent six years in the Yale Department of Mathematics working with Benoit Mandelbrot, the inventor of fractal geometry, who credited him with being "the first true fractal-based artist." He is a founding member of the Algorist school of algorithmic artists.

Dr. Musgrave has served as Director of Advanced 3D Research at MetaCreations, Principal Software Engineer at Digital Domain, Senior Scientist at Bethesda Softworks, and Assistant Professor at The George Washington University. He is currently CEO and CTO of Pandromeda, Inc., which is making his groundbreaking artistic methods available in easy-to-use software products. Musgrave received his PhD in computer science from Yale University and his MS and BA in computer science from the University of California at Santa Cruz.

Przemyslaw Prusinkiewicz is a Professor of Computer Science at the University of Calgary. He has been conducting research in computer graphics since the late 1970s. In 1985, he originated a method for visualizing the structure and the development of plants based on L-systems, a mathematical model of development. He is a co-author of two books, "Lindenmayer Systems, Fractals and Plants" (Springer-Verlag 1990) and "The Algorithmic Beauty of Plants" (Springer-Verlag 1990), as well as numerous technical papers. In 1997, he received the ACM SIGGRAPH Computer Graphics Achievement Award for his work pertaining to the modeling and visualization of biological structures. Prusinkiewicz holds a M.S. and Ph.D., both in Computer Science, from the Technical University of Warsaw. He has lectured at SIGGRAPH courses on fractals, procedural modeling, modeling of natural phenomena, and artificial life.

Jos Stam is currently a Research Scientist at Alias|wavefront's Seattle office. He received BS degrees in computer science and mathematics from the University of Geneva, Switzerland in 1988 and 1989, and he received a MS and PhD in computer science both from the University of Toronto in 1991 and 1995, respectively. He also spent 18 months in Paris and Helsinki as an ERCIM fellow in national research labs. His research interests cover most areas of computer graphics such as natural phenomena modeling, physics-based animation, rendering and surface modeling. He has published papers at SIGGRAPH and elsewhere in all of these areas.

Recent results published at SIGGRAPH include a method to evaluate subdivision surfaces, a general class of reflection models based on wave physics and a method to animate fluids in real-time.

Jerry Tessendorf is a Research Scientist at Cinesite Digital Studios, where he creates new graphics technologies for Cinesite's visual effects business. Software based on his work has been used in a variety of feature films, including "Hard Rain", "Double Jeopardy", and "Red Planet". Recent research has focussed on the merging of particle and fluid dynamic simulation systems. Previously, Dr. Tessendorf was a Corporate Senior Scientist at Arete Associates, where he lead the development and implementation of ocean environment simulation and rendering for the products Renderworld and Digital Nature Tools. He supervised the implementation of this technology for the films "Waterworld" and "Titanic".

Course Syllabus:

1. **8:30-8:35 Course Introduction** - **Ebert** 1-1

2. **8:35-9:35 Fractal Models of Natural Phenomena** - **Musgrave** 2-1

A. What is a Fractal
   1. Definitions, self-similarity
   2. Dilation symmetry
   3. Examples in nature

B. Building Random Fractals
   1. Key variables
   2. Methods
   3. Visual artifacts

B. Fractal Terrain Models
   1. Monofractal
   2. Heterogeneous monofractal
   3. Multifractal

C. Level of Detail
   1. Nyquist Limit
   2. Clamping

D. Demonstration
   1. Building random fractals
   2. Real-time roaming on fractal terrains
   3. Imaging with level-of-detail

3. **9:35-10:15,**
   10:30-10:50 Water More Real Than Real- **Tessendorf** 3-1

A. Environment Components (waves, atmosphere, water volume)

B. Ocean Wave Phenomenology (deep water waves)
   1. Statistical wave models
   2. FFTs and random ocean realizations
   3. Wave animation
   4. Nonlinear wave action

C. Water Surface Optics
   1. Fresnel reflection and transmission of skylight
   2. Glitter modeling

D. Water Volume Reflection and Transmission
   1. Water volume reflection models
   2. Refraction focusing and defocusing
   3. Water attenuation models
   4. Underwater POV
      a. Refracted skylight
      b. Refracted sunlight - caustics
      c. Scattered sunlight - sunbeams
   5. Underwater imaging - scattering and blurring

4. **10:50-11:50 Stable Simulation of Fluids** - **Stam** 4-1
A. Introduction
1. The Equations of Fluid Dynamics
2. Traditional Approaches
3. Previous Work in computer graphics
B. A new Stable Solver
1. Stability (why is it important)
2. Stable solvers for the linear terms
3. Stable solver for the non-linear part
C. Applications
1. Simulation of Gaseous Phenomena
2. Simulation of Fluids (water, etc.)
D. Demos
(interactive demos of the simulations)
E. Future Work

5. **11:50-12:15**
1:30-2:05 Realistic Physics-based Smoke and Fire Simulation – Fedkiw 4-1
A. Numerical Simulations of CFD Equations
1. Introduction
2. Course grid difference equations vs. the continuous Navier-Stokes equations
B. Exploiting Smoke Physics For Rapid, Realistic Simulations
C. Simple, interactive FFT solvers
D. Fire and Water Simulations

6. **2:05-3:00 Procedural Volumetric Cloud Modeling, Animation, and Real-Time Techniques** - **Ebert** 5-1
A. Introduction to Volumetric Cloud Modeling
B. Volumetric Implicit Models
1. Basics of volumetric implicits
2. Modeling and rendering issues
C. Simulating Clouds
1. Different cloud types
2. Interesting Effects
D. Real-time Issues for Simulation of Clouds and other Natural Phenomena
1. Capabilities of the latest PC boards and game consoles
2. Flexibility, precision, and other issues
3. Vertex and fragment programming
4. Techniques for using flexible 2D and 3D texture mapping hardware

7. **3:00-3:15**
   3:30-4:15 The Science and Art of Plant Modeling - **Prusinkiewicz** 6-1
   A. Classification of Plant Modeling Techniques
   B. Simulation-based Modeling
      1. Abstractions and formalisms
         a. "Expanding canvas" and "dynamic platform" as metaphors for a growing plant
         b. Modularity of plant architecture
         c. L-systems: the concept, applications, and limitations
      2. Simulating physiological processes affecting plant development: lineage, signaling, and interaction with the environment
      3. Simulating mechanical and biomechanical factors (e.g., gravity, tropisms).
      4. Examples and applications of simulation-based plant models, from the level of genes to the level of plant ecosystems
      5. Interaction with plant models
   C. Inverse Modeling of Plants
      1. The essence of inverse modeling
      2. Abstractions and formalisms (positional information, organized and random variation, structural invariants)
      3. Techniques for model reconstruction
      4. Examples of applications of inverse modeling
      5. Modeling arrangements of plant organs (phyllotactic paterns)
      6. Modeling branching plant structures
      7. Modeling interactions between plants and their environment
   C. Modeling Plant Ecosystems
      1. Overview: The concept of multilevel modeling
      2. Modeling plant distribution
      3. Populating a scene with realistic plant models
      4. Harnessing scene complexity using approximate instancing
      5. Case stuides

8. **4:15-5:15 Fast Rendering and Modeling of Plants – Deussen** 7-1
   A. Level of detail issues with plant models
   B. Fast Rendering of plants
   C. Point-based rendering
   D. Non-realistic rendering of plants

Table of Contents

1. Introduction 1-1

2. Fractal Models of Natural Phenomena 2-1

3. Simulating Ocean Water 3-1

4. 3D Fluid Simulation
   - Stable Fluids 4-1
   - A Simple Fluid Solver Based on the FFT 4-9
   - Visual Simulation of Smoke 4-13
   - Physically-Based Modeling and Animation of Fire 4-21
   - Practical Animation of Liquids 4-30
   - Animation and Rendering of Complex Water Surfaces 4-38
   - Stam Talk Slides 4-47
   - Stam Talk Notes on CD-Rom

5. Procedural Volumetric Modeling and Animation of Clouds and Other Gaseous Phenomena 5-1
   - Talk Slides 5-27

6. The Science and Art of Plant Modeling 6-1

7. Fast Rendering and Modeling of Plants 7-1

Course Introduction

The subject of this course is one of the challenging open problems in computer graphics: the photorealistic simulation of nature. In 1994, there was a SIGGRAPH course on natural phenomena, discussing this problem purely from a research perspective. There have been tremendous advances in modeling nature since 1994. This course will highlight the state-of-the-art in several areas of natural phenomena: gases/clouds, water, organic environments, plant ecosystems, fractal landscapes and planet models.

This course also offers a new perspective on the problem of simulating natural phenomena. The course speakers come from both the academic research community and from the commercial production industry, providing opportunities for contrasting and comparing techniques used by both groups. Research issues as well as practical considerations will be presented to show how to tractably simulate complex natural environments.

We have now entered a new era in computer graphics: hardware accelerated programmable rendering and shading at interactive rates on desktop PCs and game consoles. With the programmability of the graphics processing unit (GPU) that has recently become available, combined with the increased performance of PC CPUs, we can now start to simulate natural phenomena and other complex effects at interactive or near interactive rates. PC board such as the Nvidia GeForce3 and the ATI Radeon are beginning to enable complex, programmable graphics at interactive rates. Throughout this course and these notes, you notice that all of the speakers / authors present techniques for simulating aspects of nature at interactive or near interactive rates.

Ken Musgrave will describe techniques for simulating fractal landscape and entire planets both from a fractal research perspective and from a commercial software perspective. Two speakers will discuss different aspects of modeling and animation water. Jerry Tessendorf will describe ocean wave models, water wave optics, and volume optics aspects of water simulation. Jos Stam will discuss his work for rapid stable fluid solvers for both water and gases. Ron Fedkiw will then talk about 3D Navier Stokes solutions for simulating gases and fluids. David Ebert will describe volumetric techniques for simulating clouds and other types of gaseous phenomena from a procedural perspective, as well as issues related to adapting these techniques to interactive modeling.

The final course speakers, Przemek Prusinkiewicz and Oliver Deussen, will describe plants, ecosystems, and organic environments. Przemek will discuss techniques from modeling individual plants to entire ecosystems. Olver will concentrate on fast rendering and modeling of plants

From attending this course or reading these notes, you should get a fundamental understanding of the important research approaches, practical considerations, and design strategies for simulating nature.

Fractal Models of Natural Phenomena

SIGGRAPH 01 "Simulating Nature" Course

F. Kenton "Doc Mojo" Musgrave

Introduction

Nature is visually complex. Capturing and reproducing that complexity in synthetic imagery is one of the principal research problems in computer graphics. In recent years we have made impressive progress, but nevertheless, most computer graphics are still considerably less complex and varied than the average scene you see in Nature. Personally, I don't expect computer graphics to be able match the visual richness of Nature in my own lifetime—there's just too much complexity and variety to be seen in our universe. But that certainly doesn't mean we shouldn't try—only that we can keep at it for a long time to come.

So how do we make a first stab at creating visual complexity in synthetic imagery? Fractals, in a word. Fractal geometry is a potent language of complex visual form. It is wonderful in that it reduces much of the staggering complexity we see in Nature to some very simple mathematics. I'm going to try to convey, as simply as I can, the intuition behind fractals in this chapter. I know it's a little confusing the first time around. It took me several re-readings of the standard texts when I was a graduate student, to get it straight in my head. But after I "got it," it was clear that the important parts are very simple. I'm going to try to convey that simple view of fractals in this chapter. First a little motivation, though.

Building Mountains

One of the most common fractals we see in Nature is the Earth we walk on. Mountains are fractal, and we can make very convincing synthetic mountains with some pretty simple fractal computer programs. Benoit Mandelbrot, the inventor/discoverer of fractals and fractal geometry calls such things "fractal forgeries" of Nature. My own career in fractals began with making some new kinds of fractal terrains through novel computer programs, to create fractal forgeries the likes of which had not been seen before.

When I began to crank out some very realistic images, people immediately started asking me "why don't you animate them?" I thought, well, there aren't many moving parts in a landscape. Landscapes tend to just sit there, rather peacefully. What *is* free to roam about is the camera, or your point of view. This in turn begs the questions: Where do you come from? Where do you go? If the point of view is free to roam, the landscapes need to be in a proper global context. What's the global context for a landscape? A globe, of course!

Building Planets

Naturally, the next thing to do was to build a planet, so that these realistic landscapes would have a geometrically correct context in which to reside. Fortunately planets, being

covered with terrains that are fractal, are themselves fractal. So we just need to build bigger fractal terrains, to cover a planet. There are some other interesting points about the planet model, too.

Figure 1. A preliminary Earth-like planet model.

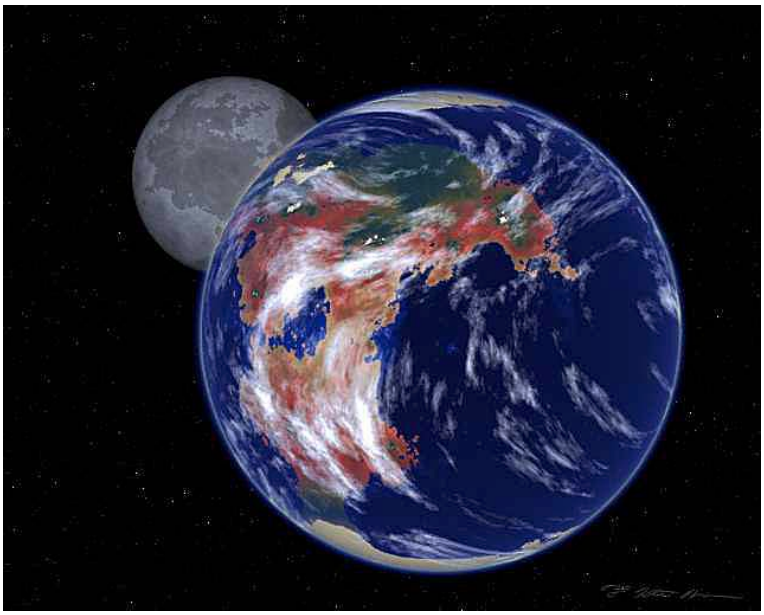


Figure 2. Gaea: a multifractal model of an Earth-like planet.

When I set out to build my first planet, seen here in Figure 1, it was apparent that we needed better fractal terrains. The kind of fractals we had all been so pleased with up to that time weren't really up to the job of modeling a whole planet—they were just too monotonous. So I created some new fractals—*multifractals*—that had a little more variety, as seen in Figure 2. Here's the difference: See how the coastline in Figure 1 is has pretty much the same "wiggliness" everywhere on the planet? You can see that it's pretty smooth and uncomplicated in some places on the planet in Figure 2, and pretty "rough" or "wiggly" in other places. That's the pretty much all you need to know about multifractals—no kidding! They're another step towards the true richness and

complexity we see in Nature. The cool thing is that they don't complicate the math much at all, which is a Very Good Thing, if you ask me.

Ah, but I said there are other interesting *points*, plural, about a planet model. One that is not so obvious is the atmosphere. Landscape painters have known for hundreds of years that the atmosphere gives the only visual indication of truly large scale in a rendering. Leonardo wrote about it in his journal. Computer graphics have used atmospheric effects for as long as we've been making realistic renderings of fractal mountains. But it turns out that, in order to get the atmospherics to work really well, even on a local scale, you simply can't use the simplest and easiest atmospheric model—a flatland model. You have to use an atmosphere that curves around a planet. You've seen the sun lighting clouds from underneath well after it's set—you can't get that with a flatland model! So for practical reasons of getting things just so, you end up having to model the Earth's atmosphere quite accurately, just to get sunsets to look right. Heck, go look up the word "atmosphere"—it literally means, "sphere of vapor." Another connection like "global context."

Is the atmosphere fractal? Not in any obvious way, even though clouds certainly are. When I was a graduate student working under Mandelbrot, who was basically paying me to invent new fractal terrain models and make beautiful images of them, I worried that I was spending too much time on my atmosphere models. When I asked him about it he, in true form, quipped mysteriously, "Well, many things that do not appear to be fractal are, in fact, fractal." It turns out that global circulation and the distributions of pollutants and density layers are fractal. Someday we'll have the power to model these things in MojoWorld, but not yet in this, the first year of the third millennium AD. Also, the path of photons as they are scattered by the atmosphere is fractal (not that it's of any consequence to us in making our pictures). Fractals are, indeed, everywhere.

Building a Virtual Universe

If landscapes need a global context, well then, so do planets. Planets orbit suns in solar systems, stars tend to live in clusters which in turn live in and around galaxies, which live in clusters and superclusters, right up the largest-scale features of our universe, which are in turn attributable to quantum fluctuations in the early universe, according to current cosmological theory. (If you want an explanation of *that*, you'd better ask Jim Bardeen, who wrote the part of MojoWorld that gives us continents with rivers and lakes. Jim is an astrophysicist who helped Stephen Hawking work out the original theory of black holes, and now works on exactly that aspect of cosmology. He does rivers for us on the side, in his spare time.) Fortunately, the distribution of stars and galaxies, and the beautiful shapes of the stellar and interstellar nebulae that we're constantly getting ever-better pictures of, are all quite fractal. In coming years, we here at Pandromeda have every intention of generating an entire synthetic universe that lives inside your computer. The path is clear, as for how to do it. It will just take time to develop it and a *lot* of computer power to make it so. I, for one, am anxious for the future to arrive already!

Okay, so there's the Big Picture. Now how are we going to build this universe? What does it take? Fractals. Lots of fractals.

What is a Fractal?

Let's get to first things first: what exactly *is* a fractal? Let me offer this definition:

> fractal: a complex object, the complexity of which arises from the repetition of a given shape at a variety of scales

Here's another definition, from www.dictionary.com:

> fractal (noun): A geometric pattern that is repeated at ever smaller scales to produce irregular shapes and surfaces that cannot be represented by classical geometry. Fractals are used especially in computer modeling of irregular patterns and structures in nature.
> [French from Latin fractus, past participle of frangere, to break; see fraction.]

It's really that simple. One of the easiest examples of a fractal is the Koch snowflake.

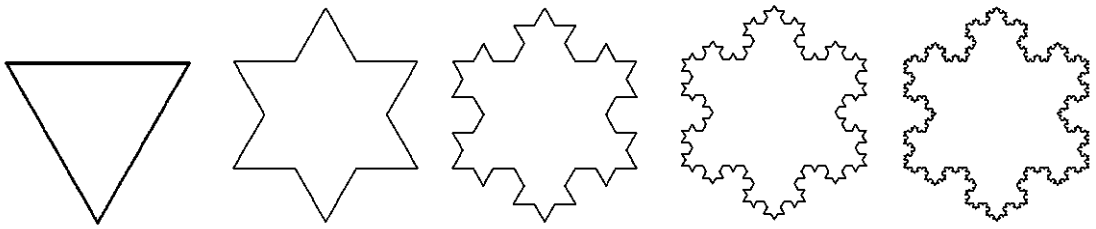


Figure 3. The Koch snowflake: a canonical fractal.

In the Koch snowflake the repeated shape is an equilateral triangle. Each time it is repeated on a smaller scale, it is reduced in size by exactly 1/3. This repetition can be repeated at smaller scales ad infinitum, leading to a curve—the edge of the snowflake—that is wonderfully complex (and also exhibits some bizarre mathematical properties, but we won't go into that here).

There's lots of math we could go into about fractals, but perhaps the neatest thing about fractal geometry is that you don't need to learn any math at all to understand and use it. You can think of it as an artist, entirely in terms of visual form. Let me describe a few easy ways to think of fractals.

Self-Similarity

The repetition of form over a variety of scales is called *self-similarity*: A fractal looks similar to itself on a variety of scales. A little piece of a mountain looks a lot like a bigger piece of a mountain, and vice-versa. The bigger eddies in a turbulent flow look much the same as the smaller ones, and vice-versa (see Figure 4). Small rocks look the same as big rocks. In fact, in geology textbooks, you'll always see a rock hammer or a ruler in photographs of rock formations, something to give you sense of scale in the picture. Why? Because rock formations are fractal: They have no inherent scale; you simply cannot tell how big a rock formation is unless you're told. Hence another synonym for the adjective "fractal" is "scaling:" a fractal is an object that is invariant under change of scale.

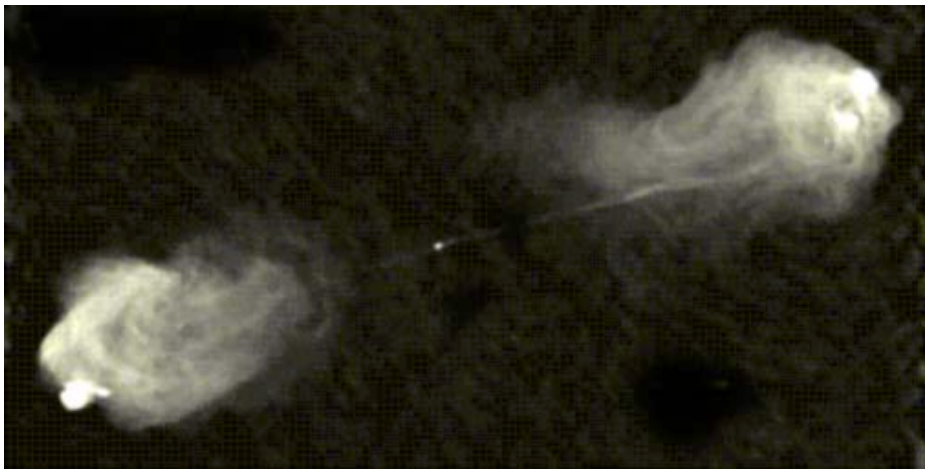


Figure 4. A fractal on a truly grand scale: jets of gas from an active galactic nucleus. VLA radio image of Cygnus A at 6 cm courtesy NRAO.

Figure 4 shows my favorite example of a fractal in Nature. Looks kind of like a puff of smoke from a smoker's mouth at arm's length, or a drop of milk in water, doesn't it? Guess how big it is. It's about 400,000 *light years* wide—that's roughly four thousand trillion kilometers or twenty four hundred trillion miles from one end to the other. Too big for *me* to imagine! In the 1970's when I first saw this picture, taken at radio wavelengths by the Very Large Array radio telescope, I considered it the most mind-blowing image I'd ever seen. I don't think Benoit had even coined the term "fractal" yet; if he had, I certainly hadn't heard it yet. But I found it stunning that this incomprehensibly large object looked so *ordinary*, even *small*. That's a fractal for you. You can recognize one immediately, even if you don't know it's called a "fractal."

Dilation Symmetry

My favorite, easy way to grasp the idea of "fractal" is as a new form of symmetry: *dilation symmetry*. You're probably already familiar with symmetries such as the mirror symmetry by which the human body is pretty much the same on both sides when mirrored across a line down the middle, and perhaps the rotational symmetry whereby a square remains unchanged by a rotation of 90°. Dilation symmetry is when an object is unchanged by zooming in and out. Turbulence is like that; hence we can't tell how big that turbulent puff of gas in Figure 4 is until we're told.

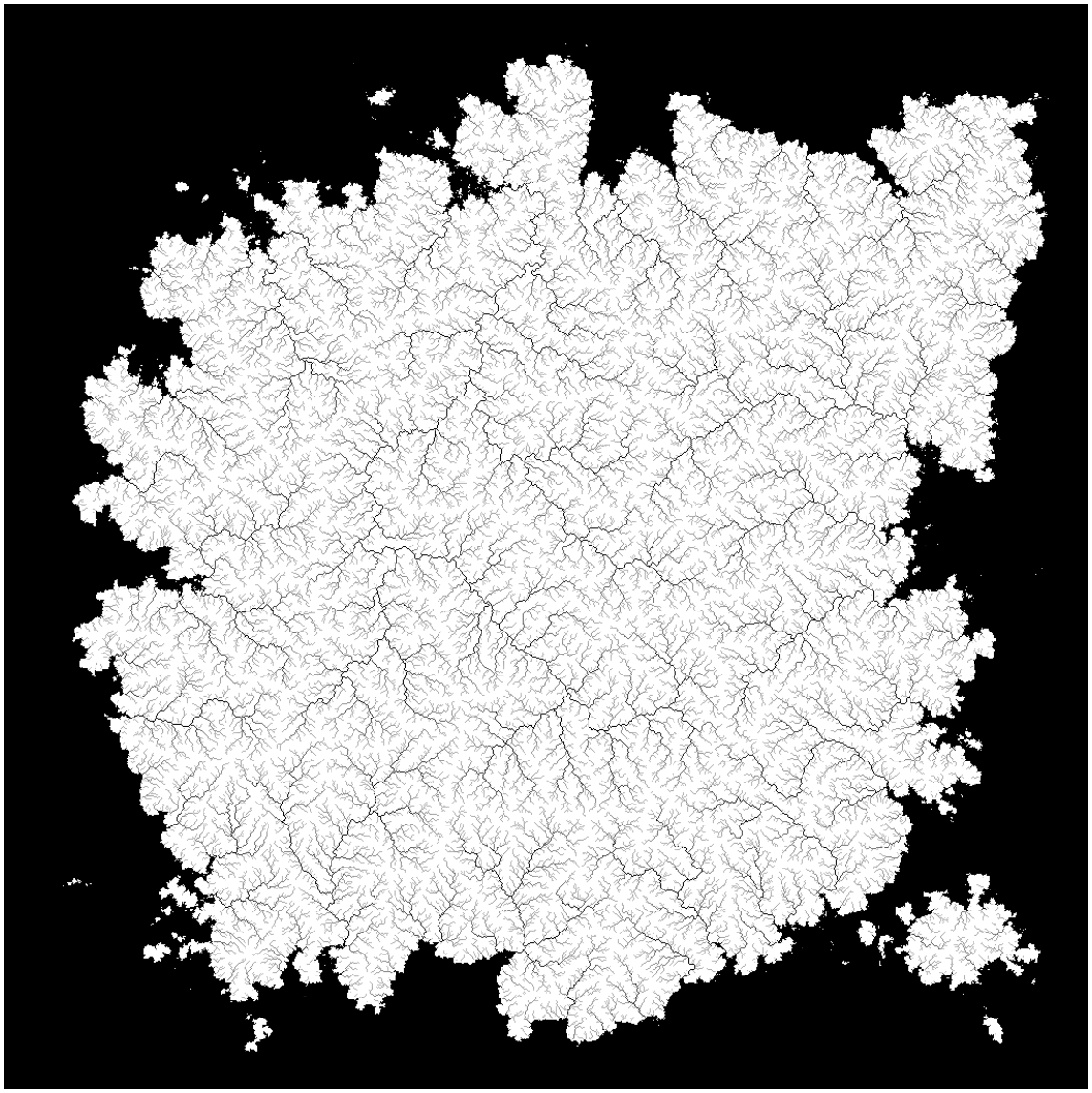


Figure 5. A fractal river drainage network for a MojoWorld continent.

Imagine, if you will for a moment, a tree branch. A smaller branch looks pretty much like a larger branch, right down to the level of the twigs. This is dilation symmetry. The same goes for river networks: smaller tributaries and their networks look much like the larger river networks. Figure 5 shows this in some of Jim Bardeen's river networks on a MojoWorld continent. Clouds, mountains, coastlines and lightning are like that, too: smaller parts look just like larger parts. There is a catch: unlike the Koch snowflake, they aren't *exactly* the same at different scales, only qualitatively so. This leads to our next distinction in fractals: *random fractals.*

Random Fractals

Random fractals may be loosely defined as *fractals that incorporate random variables in their construction.* The random variable may be some quantum process, like the probability of a given air molecule scattering a passing photon, or a pseudo-random

variable in a computer program, as we might use to determine the altitude of a point on a fractal terrain. Computers are always deterministic, so we don't have truly random variables in computer programs, only ones that are designed to look random while being in fact deterministic. "Deterministic" means that a given input always generates the same output. This determinism is a good thing: It is why we always get the same MojoWorld from a given scene file, even though what is found there is unpredictable. If the computer were producing truly random variables, we might get slightly better MojoWorlds (for very obscure mathematical reasons), but we wouldn't be able to roam around and come back to the same place again.

The point is that self-similarity comes in at least two flavors: *exact* self-similarity, as in the Koch snowflake where every part is exactly the same as every other, if you rotate it properly, and *statistical* self-similarity, as in all the natural phenomena I've mentioned. In Nature, self-similarity is usually of the statistical sort, where the statistics of random behaviors don't change with scale. But you needn't worry any about statistics—to the human eye these fractals look similar at different scales, no doubt about it, without any reference to numbers, statistics or any other fancy mathematics.

A Bit of History of Fractal Terrains

Like all intellectual revolutions fractal geometry did not happen overnight; rather, it was an extension of the work and observations of many people working in many different fields. But let me perform the standard practice of historians and make a complex story simple.

The Mathematics

Fractals were noticed by mathematicians around the turn of the twentieth century. They noted their mathematically bizarre behavior, labeled them "monsters," and left them for posterity.

Benoit Mandelbrot had an uncle who was a famous mathematician. He assured the young Benoit that the person cracked this mathematical case could make a name for himself. Benoit was not immediately interested, but the ideas festered (my word, not his!) in his mind, and he eventually came to work on such things as a researcher at IBM. In 1982 he published his classic book "The Fractal Geometry of Nature" which introduced the world to fractals. In 1987, I was fortunate to be hired to be Benoit's programmer in the Yale math department. I was to work on fractal terrain models that included river networks, a research project that ultimately failed. In 1988 "The Science of Fractal Images," edited by Heinz-Otto Peitgen and Dietmar Saupe—whom I had gotten to know at UC Santa Cruz in 1986—was published. In it Mandelbrot issued a challenge to the world to succeed at the difficult task of creating fractal terrains with rivers. In 1989 I published a paper, with Craig Kolb and Rob Mace, titled "The Synthesis and Rendering of Eroded Fractal Terrains" the described one way to create rivers in fractal terrains, though that method remains mathematically and computationally intractable to this day. In 1993 I completed my doctoral dissertation "Methods for Realistic Landscape Imaging" which was pretty much a compilation of the various papers

and course notes that I had published over the last six years on various aspects of modeling and rendering realistic forgeries of Nature. In it I described my multifractal terrain models. That's a very brief sketch of the mathematical academic history of fractal terrains. Now for the mathematical imaging track.

The Mathematical Imaging

Mandelbrot divides the history of computer images of fractal terrains into three eras: the Heroic, the Classical and the Romantic. The Heroic era is characterized by the work of Sig Handelman who made the first wireframe renderings of Benoit's terrain models. According to Benoit, it was a heroic effort in the 1970's to get even a wireframe image out of the computer, hence the name of that era. Handelman's images are mostly lost in the mists of time, alas, though one or two appear in "The Fractal Geometry of Nature." Next came the work of Richard Voss who made the realistic (at least, for that time) images such as the classic "Fractal Planetrise" that graces the back cover of "The Fractal Geometry of Nature." Voss' work comprises the Classical Era. Richard fleshed out the mathematics and rendering algorithms required to make beautiful and convincing forgeries of Nature. Next came my work, in which I brought various artistic subtleties to the forefront. As I went on at length about artistic self-expression, Benoit calls my work the Romantic Era. Benoit has generously credited me with being "the first true fractal-based artist." Figures 6, 7, and 8 are a few of my personal favorites from my own body of work of that era.

Figure 6. "Blessed State," 1988.

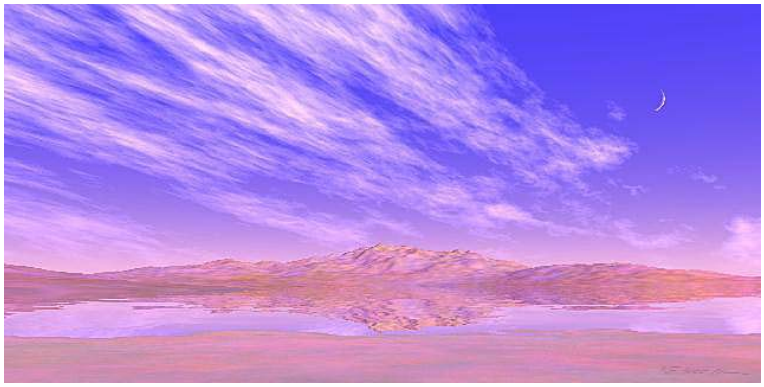


Figure 7. "Zabriskie Point," 1990.

Figure 8. "Pleiades," 1996.

You may notice my fixation on planets right from the start. ;-)

The Computer Graphics Community

Then there's the computer graphics track. In 1979 Loren Carpenter, then at Boeing, made the groundbreaking computer animation "Vol Libre," the first animated flyby of a synthetic fractal terrain. In 1982 Loren, now Senior Scientist at Pixar and a member of Pandromeda's distinguished Board of Advisors, published with Alain Fournier and Donald Fussell the paper "Computer Rendering of Stochastic Models" which introduced the commonly used polygon subdivision method for generating fractal terrains. That precipitated a bit of a feud with Mandelbrot, but it was before my time in the field, so I'll say no more about *that*. Also in 1982, Loren and the rest of the distinguished crew at Pixar showed us the first MojoWorld (if I may be so presumptuous) on the big screen in the Genesis Sequence in "Star Trek II: The Wrath of Kahn." That blew our minds. In 1985 Alain Fournier came to UC Santa Cruz to teach a course "Modeling Natural Phenomena" that changed my life, and set the course of my career. Alain mentored me in

the area and advised me in my Masters research. Later in 1985 Ken Perlin and Darwyn Peachey published twin papers that introduced the procedural methods that have pretty much driven my entire career. Thanks, guys! They had some really cool pictures in those papers; I saw a world of possibility (so to speak) in their methods and the rest is, well, "history." In 1986 Dietmar Saupe and Heinz-Otto Peitgen came to the UC Santa Cruz math department, and I took Dietmar's course "Fractals in Computer Graphics." In 1987 Dietmar recommended me to Mandelbrot for the job at Yale. (I always tell my students: "there's no substitute for dumb luck.") In 1993 I finally graduate from Yale with a terminal degree—a PhD, but I prefer that other term for it!—at the ripe old age of 33. Can you say "professional student?" In 1994 Matt Pharr, Rob Cook and I created the "Gaea Zoom" computer animation that was, I think, the first MojoWorld with adaptive level of detail; that is, a synthetic fractal planet that you could zoom in and out from, without nasty artifacts described by obscure mathematics that I won't go into here. Suffice it to say, it's not so easy to make a planet that looks good from both near and far, doesn't overload your computer's memory, and renders in a reasonable amount of time. The Gaea Zoom took two weeks to render on four supercomputers, so we weren't quite there yet in 1994. Finally, in 2001, we are. Hence MojoWorld, which really couldn't have been launched even a year earlier.

The Literature

And then there's the track of explanations of fractal terrains. First came the technical papers that even *I* never could really understand. Then came "The Science of Fractal Images" in 1988 (which I even wrote a tiny part of), in which Richard Voss and Dietmar Saupe cover everything you'd ever want to know about the mathematics of these and other kinds of fractals. Next came our book: David Ebert, Darwyn Peachey, Ken Perlin, Steve Worley and me, "Texturing and Modeling: A Procedural Approach," first edition in 1994, second in 1998. In it I explain how to build fractal terrains from a programming perspective. Still pretty technical, that book, but the standard reference on how to program up fractal terrains and even entire MojoWorlds. And now there's this little exposition, in which I'm trying to explain it all to the non-technical reader

Software

Last but not least, there's the software track of the history of fractal terrains. For some time there existed only the various experimental programs created by we academics. They couldn't be used by the average person or on the average computer. The first commercial software that addressed fractal terrains was high-end stuff like Alias. I'm afraid I must plead ignorance of that high-end stuff, because I was just a lowly graduate student at the time and, while I had access to some mighty fancy computers to run my own programs on, we certainly couldn't afford such top-of-the-line commercial software, and they weren't giving it away to university types like us despite our constant and pathetic pleas of poverty and need. The first affordable commercial fractal terrain program that came to my attention was Vistapro, around 1992. With its flight-simulator interface for making animations, it was really cool. You can still buy Vistapro, though it's a bit quaint by today's technological standards. Next, I believe, came Bryce 1.0, in 1994, written primarily by Eric Wenger and Kai Krause for what was then HSC

Software, then MetaTools, then MetaCreations, now defunct. (Corel now owns the Bryce name and is carrying the product forward.) Bryce 1.0 was Macintosh-only software, and mighty cool. It put a user-friendly interface on all the procedural methods that my colleagues and I had been going on about in the academic literature for years, and made it all accessible to Average Joe with a home computer. Since then, Animatek's World Builder and 3D Nature's World Construction Set have released powerful, semi-high end products priced around $1,000 US. Natural Graphics' Natural Scene Designer, E-on Software's Vue d'Esprit and Mathew Fairclough's Terragen shareware program have filled out the low end at $200 US and less. Each product has its specialty; each can create and render fractal terrains. Meanwhile, Bryce is up to version 5.0. I worked on Bryce 4.0 for MetaCreations until the greedy management decided to change coats to do another dot-com IPO, and sacked the lot of us. The next day FractalWorlds, now Pandromeda, was launched to make MojoWorld a reality. We're the first to come to market with entire planets with level of detail, thus opening the door to cyberspace.

Disclaimers and Apologies

So there's Doc Mojo's Close Cover Before Striking History of Fractal Terrains. Sure, it's biased. I worked for Mandelbrot. I come from the academic side, and was camped more with the mathematicians than with my real colleagues, the computer science/computer graphics people. I've left out a lot of important contributions by friends and colleagues like Gavin Miller, Jim Kajiya and many, many more. I'm keeping it brief here; if you want the exhaustive listing of who's done what in the field, see the bibliography of my dissertation. And I must state that, though many people think so, I am *not* a mathematician. Believe me, having a faculty office in the Yale math department for six years drove *that* point home. My degrees are in computer science. But I'm really a computer artist, more than a computer scientist. My contribution has been mostly to the artistic methods, ways to make our images of fractal terrains more beautiful and realistic. I'm mathematical lightweight; I just do what I have to do to get what I want. And all my reasoning is visual: I think terms of shape and proportion, even if I do translate it into math in order to make my pictures. To me all the equations just provide shapes and way to combine them.

The Present and Future

The abstract of my 1993 doctoral dissertation ends with this sentence:

> Procedural textures are developed as models of mountains and clouds, culminating in a procedural model of an Earth-like planet that in the future may be explored interactively in a virtual reality setting.

In MojoWorld we finally have that planet, and a whole lot of others as well. It's not quite yet what I'd call "virtual reality"—the real-time part is just not that realistic. Yet. Getting there is just a matter of a lot of hardware and software engineering.

Gertrude Stein once said of Oakland, California, "there's no 'there' there." If that's true for Oakland, I say it's way more true (to put it in the California

vernacular) of all implementations of synthetic environments up to now.

MojoWorld puts the "there" there. In MojoWorld, as Buckaroo Bonzai said, "everywhere you go, there you are." You never run out of detail, and there's always someplace new and interesting to visit. I think of MojoWorld as a window on a parallel universe, a universe that already exists, always has and always will exist, in the timeless truth of mathematical logic. All we've done is create the machinery that reveals it, and the beauty to be found there. It's been my great privilege to play a part in discovery/creation of this possibility. It's been my vision for years now to make this experience accessible to everyone. With MojoWorld, we're on our way, and I, for one, am very excited about that. I hope you enjoy it half as much as I will!

Now let's get on to explaining how we build a MojoWorld, so that you understand the controls that you'll be using.

Building Random Fractals

The construction of fractal terrains is remarkably simple: It is an iterative loop involving only four important factors, and one of those generally a non-issue. First we have the *basis function*, or the shape that we build the fractal out of. Next there's the *fractal dimension*, which controls the roughness of the fractal by simply modulating the vertical size of the basis function in each iteration (i.e., each time you go through the loop). Then there's the *octaves*, or the number of times that we iterate in building the fractal. Finally, we have the *lacunarity*, or the factor by which we change the horizontal size of the basis function in each iteration. Usually we leave the lacunarity at 2.0 and never think about it.

Let's see what the effect of each of these four factors is, and how all they fit together.

The Basis Function

The basis function is perhaps the most interesting choice you get to make when building a random fractal, whether for creating terrain, clouds, water, nebulae or surface textures. The shape of the basis function largely determines the visual qualities of the resulting fractal, so "choose wisely." It's fun to experiment and see the subtle and not-so-subtle effects the choice of basis function has, visually, in the result. I have certainly gotten a lot of artistic mileage over the years through careful choice and modulation of basis functions. And MojoWorld has plenty of basis functions, that's for sure.

For obscure but important mathematical reasons, basis functions should (A) have shapes that are not too complicated, (B) never return values smaller than –1.0 or larger than +1.0, and (C) have an average value of 0.0. As Mick once said, "you can't always get what you want," but the basis functions in MojoWorld are designed to obey these constraints in most cases.

The thing to keep in mind about the basis function is that its shape will show through clearly in the fractal you're building. So your choice of basis function is the most significant decision you make when building a fractal.

Fractal Dimension: "Roughness"

Fractal dimension is a cool, if slippery, beast. I'll leave mathematical explanations to the other texts. In MojoWorld we call it "roughness." For our purposes, just think of the Roughness control as a slider that controls visual complexity. It does this by varying the roughness of terrain, the wiggliness of coastlines, the raggedness of clouds, and the busy-ness of textures. The Roughness control in MojoWorld is an extremely potent control.* Use it a lot! You'll find it a powerful and subtle way to affect the aesthetics of your fractals. And don't be surprised if you find yourself setting its value via text entry, down the third digit after the decimal point. It's that sensitive and powerful. Play with it and see.

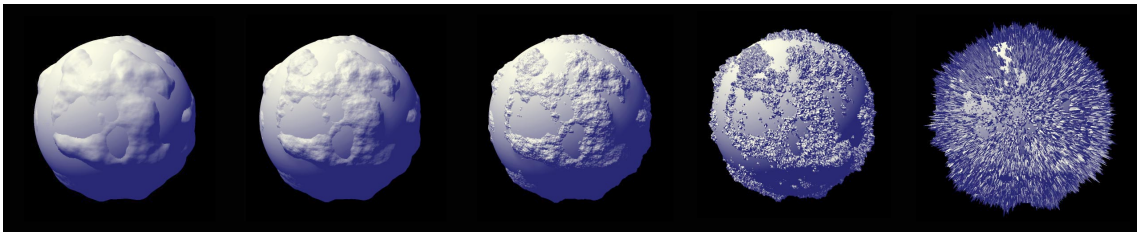


Figure 9. Planets with fractal roughness of –0.5, 0.0, 0.5, 1.0 and 1.5.

Larger values make for rougher, busier, more detailed fractals. They tend to get visually "noisy" at values over about 0.5. I generally prefer to use smaller values than most people, but hey, it's strictly a matter of taste.

Octaves: Limits to Detail

We call the number of times we iterate, adding in more detail, the *octaves*. Fractals can have potentially unlimited detail. But that detail has to be built by the computer, so it has to have limits if you want your computation to finish. In Nature, fractals are always *band limited*: There is a scale above which the fractal behavior vanishes (this is even true of the largest structures in the universe) and a scale below which it also goes away (as when we get to the scale of quantum physics). For imaging purposes, MojoWorld has to be in control of the number of times each sample of a fractal goes through the construction loop. The explanation for why this is so is *way* beyond the scope of this presentation. Suffice it to say that controlling the number of times we go through the loop controls the amount if detail, and the amount of detail required at a given point in the image depends on its distance from the camera, the screen resolution, field of view and more subtle factors as well. Furthermore, too much detail not only wastes computation time, but also causes *aliasing*—nasty visual artifacts that we go to great lengths to eliminate in MojoWorld. So, bottom line, MojoWorld has to control the number of octaves in the fractals. That's just the way it is.

---

* The way we've implemented it in MojoWorld, the Roughness control doesn't necessarily have an accurate relationship to the numerical value of the fractal dimension in the fractal you're building, but who cares? That number isn't important to us, only the qualitative effect we're getting.

We can, however, play games with the octaves. The Detail control can reduce or increase the number of octaves, and hence the fine detail, in fractals. Its effects can look pretty strange in animations and when you change the rendering resolution, but it can keep your fractals from being annoyingly "busy," visually, all the time. The Largest Feature Size and Smallest Feature Size controls set the band limits to the fractal. You'll get no more fractal details above and below these scales. You have to set the Largest Feature Size to something reasonable. Keep in mind that it shouldn't be any larger than the planet, or you're just wasting computation time. The Smallest Feature Size can be left at zero. MojoWorld will eventually decide that "enough is enough," but you'll probably get tired of zooming in long before that.

Lacunarity: The Gap Between Successive Frequencies

This one is usually a non-issue, but we've made it an input parameter in MojoWorld "just to be thorough." When you're going through the iterative loop that builds the fractal, you have to change the scale of features at each iteration, because that's how we get features at a variety of scales. The *lacunarity* determines how much the scale is changed at each iteration. Since "scale" is in this case synonymous with "spatial frequency" (of the features in the basis function), it's easiest to think of the lacunarity as the gap between successive spatial frequencies in the construction of the fractal. Indeed, "lacuna" is Latin for "gap."

Usually we double the frequency at each iteration, corresponding to a lacunarity of 2.0. Musically, this corresponds to raising the frequency by one octave, hence the term "octaves" for the number of scales at which we create detail. Why the value of 2.0, and not something else? Well, it has to be bigger than 1.0, or you go nowhere or even backward. On the other hand, the bigger you make it, the faster you can cover a given range of scales, because you're taking a bigger step at each iteration. Each iteration takes time, and when you're building a planet, you have big range of scales to cover. So a clever person might think "well, then, just crank up the lacunarity!" Not so fast, Bucko. It turns out that for lacunarity values much over 2.0, you start to see the individual frequencies of the basis function. It just looks bad—try, say, 5.0 and see. We've gone with a default lacunarity just over 2.2, to eek out a little more speed. If you want images that are as good as they can be, I'd recommend a value more like 1.9.[*] My best advice: Don't mess with lacunarity until you really know what you're doing.

---

[*] For pointy-headed technical reasons, it's best not to use a lacunarity of exactly 2.0, but something close to it, like 1.9 or 2.1. Transcendental numbers are best. MojoWorld's default is the natural logarithm *e* minus one half, or 2.718… - 0.5 = 2.218… You might try changing the 2 after the decimal point to 1. Keep in mind you should set your lacunarity permanently before you use your fractal, because changing it will change all the features except those on the largest scale, and this could completely disrupt some specific feature in your MojoWorld that you've become interested in.

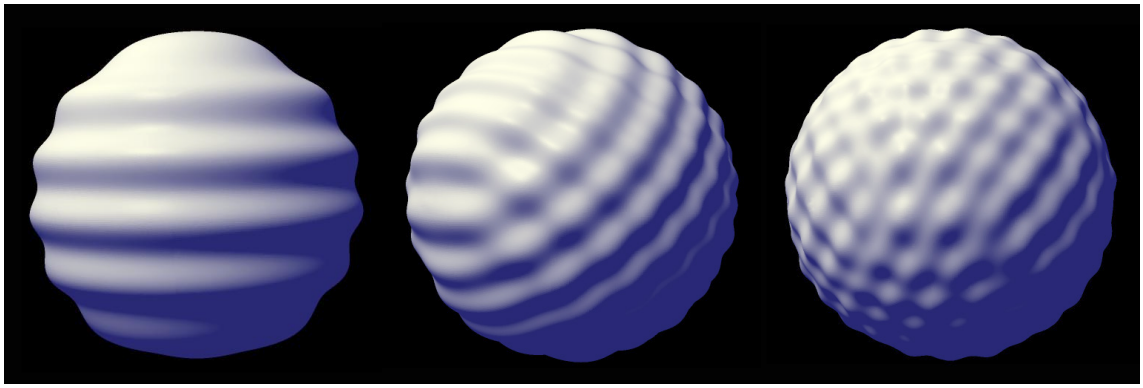
Advanced Topics

Dimensions: Domain and Range

The various functions used to create textures and geometry (for example, mountains) in MojoWorld are implemented in several dimensions. What does this mean? Pay close attention, as this can be a little confusing and counterintuitive. A *function* is an entity that maps some input, called the *domain* of the function, to some arbitrary output, called the *range*. Inside the MojoWorld program, the domain and range are all a bunch of numbers. As users, we usually think of the output as color, the height of mountains, the density of the atmosphere, and other such things. We also think of the input as things like position in space, altitude and color, too, or as numbers like the ones we can type in in the UI.

Both the domain and range of functions have *dimensions*. One dimension corresponds to a line, two to a plane, three to space, and four to space plus time. When you're creating a new function in MojoWorld, you'll sometimes have to choose the dimensionality of the domain of the function. This seems a little backward, as what you're really interested in is the dimensionality of the output, or the range of the function. Here's the catch: You can't have meaningful output of dimensionality greater than that of the input. There's just no way to make up the difference in information content.

Usually we're working in three dimensions in MojoWorld, so that's the correct default choice to make when you're confronted with this decision. But every added dimension doubles the time required to compute the function, in general. So you want to use as few dimensions as you can get away with. You might also want to do some special effects using lower dimensions, like determining the climate zones of your planet (implemented as a three-dimensional texture) by latitude (a one-dimensional variable). Figures 10, 11 and 12 illustrate MojoWorlds made from the same function, with domains of one, two and three dimensions, respectively.



Figures 10, 11 & 12. Planet made from a sine function with a one-, two- and three-dimensional domains.

The fact is MojoWorld also has a full compliment of functions with four-dimensional domains "under the hood." These will be useful for animating three-dimensional models over time to simulate things like continental drift and billowing volumetric clouds. The user interface for animation is a complicated thing, though, so we decided to leave it for a future version of MojoWorld, when we've had time to do it right.

Hyperspace

You may have noticed that we go on about *hyperspace* a lot in our MojoWorld propaganda. Hyperspace is whenever you go up one dimension: a plane is a hyperspace to a line, and three dimensions is a hyperspace to a plane. Add time to space, and you have a hyperspace to the three dimensions we're most familiar with. Well, it's really easy to keep adding more dimensions. Take the three dimensions of space and add a color, for example. Because the human eye has three different color receptors in the retina, one each for red, green and blue light, human vision has three color dimensions—hence the rgb color space used in computer graphics. (Some birds have six; they live is a world with of far richer color than we monkeys.) It takes three values to specify a color for the human eye: one each for red, green and blue. Each is independent—part of the definition of a *dimension*. From www.dictionary.com:

> **dimension:** … 4. *Mathematics.* a. One of the least number of independent coordinates required to specify a point in space or in space and time.

In our example we have three dimensions for space and three for color, for a total of six dimensions. Presto—a hyperspace! Not hard to do at all, eh?

Now think for a minute of all the values you may assign to make a MojoWorld—things like planet radius, sea level, color, atmospheric density, fractal dimension, etc., etc., etc. For an interesting planet, there'll be hundreds, even thousands of variables involved. From a mathematical standpoint, each independent variable adds another dimension to the space that the planet resides in. The more complicated the MojoWorld, the higher the dimensionality of the space it resides in. Each lower dimensional space, as for the same MojoWorld with one less color specified, is a called a *subspace* in mathematics. And, of course, each higher dimensional space is a *hyperspace*. There's no limit to the amount of complication you can add to a MojoWorld, and so, there's *no limit* to the dimensionality of the master MojoWorld hyperspace. Pretty mind-bending, ain't it? *I* certainly think so!

The various variables used to specify a MojoWorld are called *parameters*. So the master hyperspace spanned by the possible parameters is rightly called *Parametric Hyperspace*™. We took out a trademark on the name, because that's what pink-boy corporations do. "Stay off our turf." >;^) Or, to paraphrase Mick: "Hey! You! Get off of my hypercloud!" Anyway, it sounds like more hyper-hype from the dweebs in the Marketing Department, but it's not. It's simply the most succinct and accurate way to think and talk about how MojoWorld works. A pure MojoWorld scene file, uncomplicated by content such as plants, cities, monkeys and the like, needs only encode the numbers that specify the parameter settings. Everything else is generated at run time from these values. The set of parameter values specifies the point in Parametric Hyperspace™ where the MojoWorld resides. Load them into MojoWorld, and MojoWorld "beams" you there—hence we call them *transporter coordinates*. Very sci-fi, but very scientifically and mathematically accurate. MojoWorld Transporter™, our free software, let's you beam yourself to these places and explore them in three dimensions. If you want access to all of Parametric Hyperspace™—have the ability to mess with all of the parameters—you have to pony up for MojoWorld Generator™. I'm an old hippie at heart, so we're giving the Transporter away for free. But we have to

support our habit, so I got in touch with my inner Pink Boy and we're charging $249 for the Generator. But hey, where else can you get so many dimensions for so few dollars?

The Various Basis Functions

MojoWorld has by far the richest set of basis functions ever seen in a random fractal engine, so don't be surprised to find the choices a bit bewildering for a while. They're all based on methods from the academic literature in computer graphics. If you're an advanced graduate student in the field, they should all be familiar. If you're not, don't worry—not everyone needs a PhD in computer graphics! (Very few do, indeed.) You can familiarize yourself with the choices by least two routes: one, you can plunge into the lists of basis functions and their controlling parameters in the MojoWorld Texture Editor, or, if you're less patient and intrepid (like myself), you can keep examining the way various MojoWorlds that you like have been built and note the basis functions used in fractals that you particularly like. The variety of basis functions available in MojoWorld far surpasses anything I, personally, have ever had before, and it will be a long time—probably *never*—before I get around to trying every basis function that's possible in there. (In fact, once you get to filtering the basis functions, the possibilities are basically infinite, so no one will ever try them all.)

Because there are so many, I'm not going to try to describe all of MojoWorld's basis functions here. Rather, I'll describe the basic classes into which they fall, and the fundamental visual qualities of each.

Perlin

The best and fastest basis function, in general, is a Perlin "noise." Ken Perlin introduced this famous basis function in his classic 1985 SIGGRAPH paper "An Image Synthesizer," and it's the basis function that launched a thousand pictures (and my own career). Ken—and everybody else—calls it a "noise function." In the context of fractal mathematics this term is quite misleading, so in MojoWorld we call "Perlin noise" a "Perlin basis." At any rate, there are several flavors of the Perlin basis, and we have them all in MojoWorld, plus a new one.

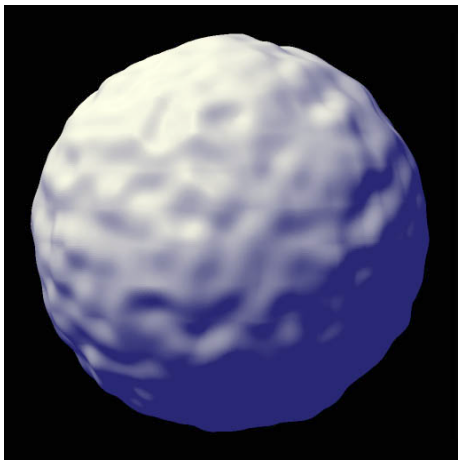


Figure 13. A planet with relief from a Perlin basis.

The Perlin basis consists of nice, smooth, random lumps of a very limited range of sizes. Its output values range between –1.0 and 1.0. It's ideal for building smooth, rounded fractals, such as ancient, heavily eroded terrains as seen in Figure 14. One of everyone's favorite terrain models is the so-called "ridged multifractal" seen in Figure 16. It looks completely unlike an ordinary Perlin fractal, but is closely related: It's made by taking the absolute value of the Perlin basis—that is, by changing the sign of all negative values to positive—and turning that upside down. Figure 17 shows visually how this process works. The result is a basis that has sharp ridges. You can use it in a monofractal function to get terrain as seen in Figure 15, or a multifractal to get terrain as in Figure 16.

Figure 14. "Carolina" is a rendition of the ancient Blue Ridge Mountains, using a subtle multifractal made from a Perlin basis function.

Figures 15 & 16. "Slickrock" is a monofractal built from a ridged Perlin basis. "Emil" is a terrain made from a ridged multifractal, a variety of Perlin fractal.

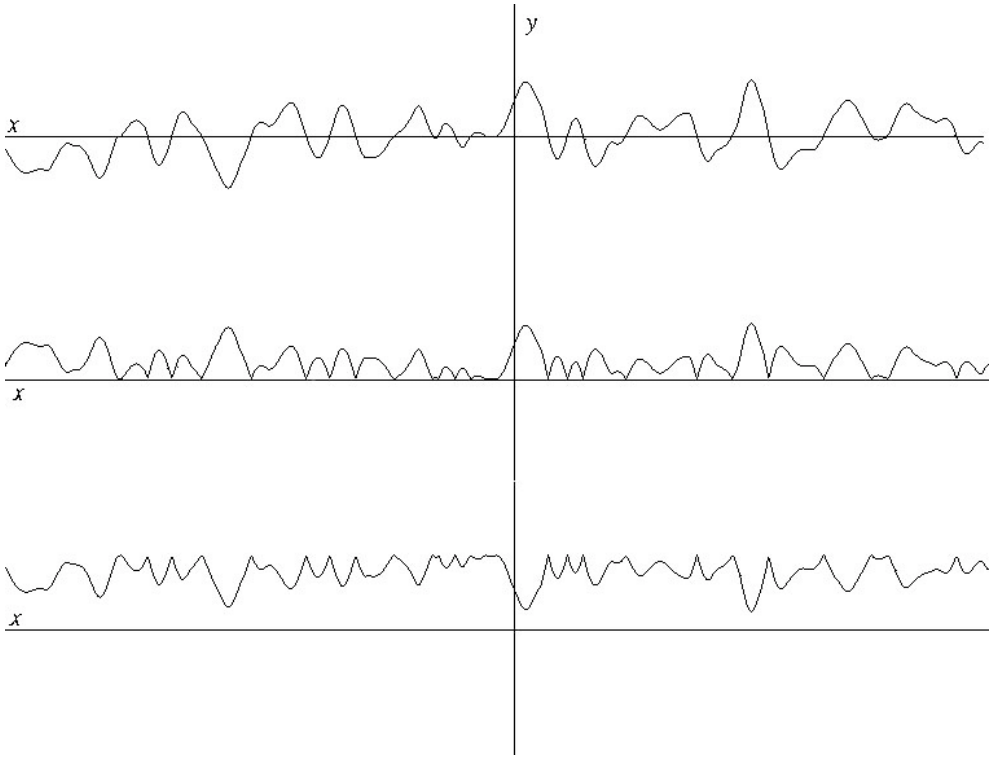


Figure 17. Building a ridged basis function from a Perlin basis: the Perlin basis, its absolute value, and one minus the absolute value.

Voronoi

The Voronoi basis functions are cool and useful, but slow. Steve Worley introduced the Voronoi basis in his 1996 SIGGRAPH paper. It has a cell-like character kind of like mud

cracks, with pits drilled into the middle of each tile of the mud (See Figure 18.) The pits are conical when you use the "distance" Voronoi and rounded (actually, parabolic) when you use the "distance squared" Voronoi. The value of the Voronoi basis is based on the distance from a random point in space. It has a ridge at the perpendicular bisector of the line between the random point and one of its neighbors. You can choose that neighbor to be the first, second, third or fourth closest neighbor to the point. You don't have to worry about which number you choose; rather, just look at the quality of the resulting texture and choose one you like. You can also choose the differences between the first and second, second and third, or third and fourth neighbors. Again, figure out what that *means* only if you want to; otherwise, just examine the quality of the texture you get and choose one you like for aesthetic reasons.

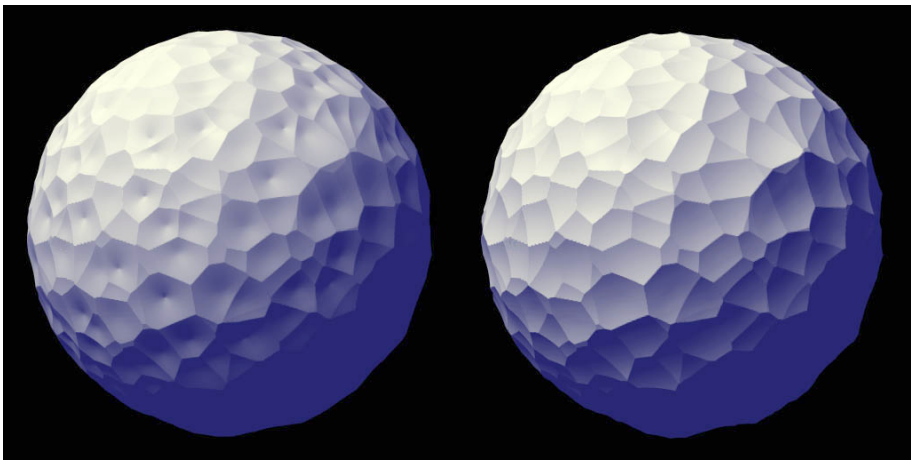


Figure 18. Planets made from the Voronoi basis, 1st neighbor. "Distance" Voronoi on the left, "distance squared" on the right. "Distance" has conical pits; "distance squared" has smooth pits.

Voronoi basis functions have ridges like the ridged Perlin basis, only they're all straight lines. Usually, you'll want to apply a fractal domain distortion to Voronoi fractals, to make those straight lines more natural—read: wiggly.

Sparse Convolution

In 1989 John Lewis introduced the sparse convolution basis: the slowest, technically "best" and most flexible of all. I'd say its time has not yet come—we simply need faster computers before this basis is going to be practical. But I'm personally obsessed with basis functions and I wanted it in there, so I put it in there. "So sue me." ;-)

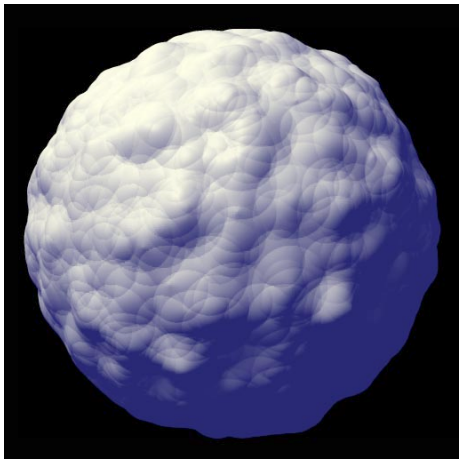


Figure 19. A planet made from the sparse convolution basis, using a cone for the kernel.

The sparse convolution basis generates random points in space and splats down a *convolution kernel* around those points. (That funny name comes from some more pointy-headed math terminology.) The kernel can be literally anything. In MojoWorld version 1.0 it can be a simple, radially symmetric shape that you choose from a list of options or building the curve editor. In later versions, we'll get into some bizarre and powerful kernels like bitmaps. Personally, I look forward to building planets out of Dobbsheads.

Various

Then there are the various other basis functions that I've thrown in for fun. See if you can find a creative use for them! Some of you may wonder why I haven't included some of the ones found in other texture engines such as that in Corel's Bryce 5.0 (which was written by Eric Wenger and myself). Well, some of those "noises" are really textures, not basis functions. You can obtain similar results, with far more flexibility, using MojoWorld's *function fractals*, which will build a fractal from whatever function you pass to them—and probably alias like a mother while they're at it. Such aliasing is why many things that are used as basis functions in Bryce, shouldn't be used for that. Others, like Eric's "techno noise," don't lend themselves to the level of detail schemes used in MojoWorld. As a rule, for deep mathematical reasons that I won't go into here, basis functions should be visually simple. So, with the exception of sparse convolution when using a complex kernel, all of MojoWorld's basis functions are. Keeping to this constraint of simplicity, here are a few more basis functions.

Sine

First there's the venerable sine wave. It's a little ambiguous what a sine wave should be, as a function of more than one variable. In MojoWorld we multiply sine waves along the various dimensions. As the sine function is periodic, anything built using the sine basis will be periodic. Periodic phenomena are quite common in Nature, but they tend to look unnatural in synthetic images. Nature can get away with things that we can't. >:-|

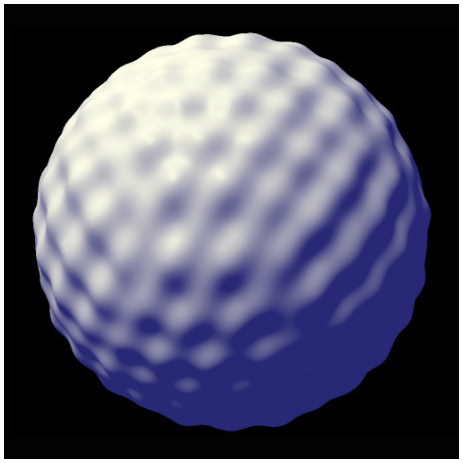


Figure 20. A planet made from a 3D sine basis.

Linear

The "linear" basis function is a simpler version of the simplest Perlin basis. It uses linear interpolation of random offsets, rather than the smooth cubic spline used in the Perlin bases. It will give you straight, sharp creases. Not very natural looking, but I'm sure someone will find a use for it.

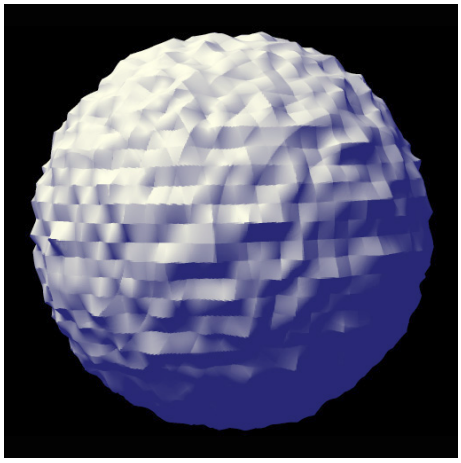


Figure 21. A planet made from the linear basis function.

Steps

The "steps" basis is simpler still: it's just a bunch of random levels in a cubic lattice. Turned to the correct angle and evaluated on a two-dimensional plane, it can yield hexagonal tiles. "We leave that as an exercise for the reader." (Don't you just hate that? I do!)

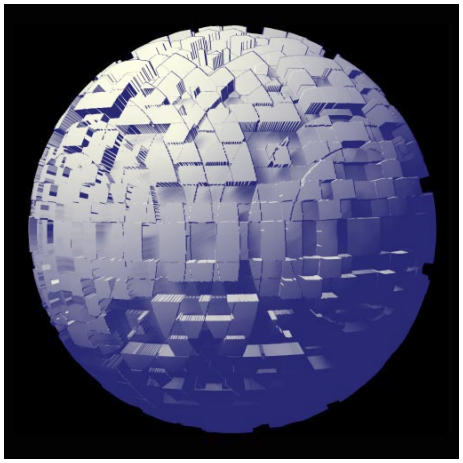


Figure 22. A planet made from the steps basis function.

Checkerboard

And then there's the procedural texture that any student of computer graphics programs first: the common checkerboard. In MojoWorld, the checkerboard basis alternates between 1.0 and –1.0. The little saw tooth artifacts you see on the cliff faces in the steps and checkerboard planets are the micropolygons that compose them. They can be made smaller, at the expense of longer render times, but they'll never go away completely. Discontinuous basis functions like these are Mathematically Evil. And so they really shouldn't be used, but what the heck, MojoWorld is for play as well as serious work, so not everything has to be *perfect*.

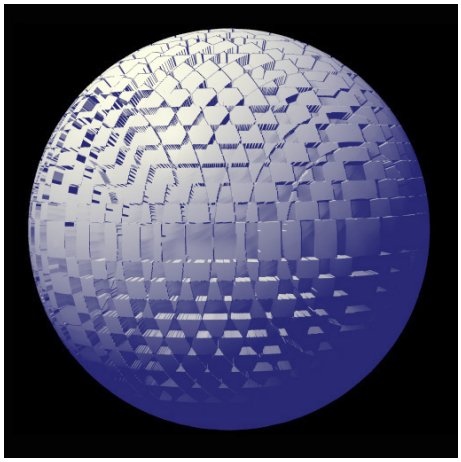


Figure 23. A planet made from the checkerboard basis function.

The Seed Tables

Here's another MojoWorld first. Call me a Noise Dweeb, a Basis Function Junkie, or just plain ill advised, but…"I just had to." <8^) The Voronoi and sparse convolution basis functions are built from tables of random points in space. Well, there are many forms "random" can take. (Take a class in Probability or Statistics and learn to hate them.) I won't try to describe the ramifications of various random distributions, but suffice it to say, they don't all *look* the same. So I implemented a mess of different ones

and whacked ‘em into MojoWorld. Play with them and try to suss out the subtle visual differences between them. In general, the ones with larger numbers have a denser spatial distribution. The ones with a ‘v’ (for “variable”) are more dense some areas and less dense in others—more heterogeneous, in a word. Ones with a “g” (for “Gaussian”) have heterogeneous heterogeneity (see why I don’t want to explain it here?) Again, just evaluate them visually and use ‘em if you like, or just ignore them—they are a *very* advanced feature for very subtle effects. Figure 24 illustrates some of the extremes in the visual consequences available with different random seed tables. Note that even these “extremes” are only subtly different; there are other tables with intermediate values to make the possible transitions all but imperceptible. A few Perfect Masters of MojoWorld will someday find these subtle differences useful, I predict.

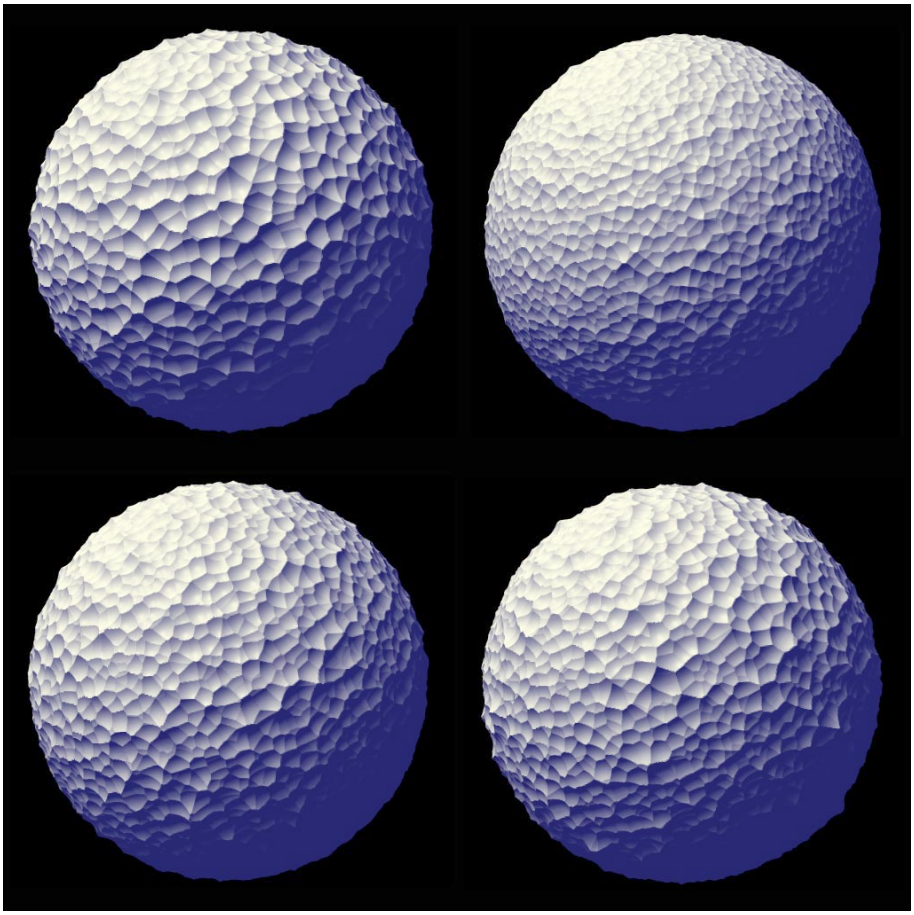


Figure 24. Planets made from Voronoi 1st neighbor with, going clockwise from upper left, seed table 1, 4, 4v and 4g.

Monofractals

Much like the various possible distributions of random numbers I just glossed over so quickly, there are even more complicated mathematical measures that characterize the randomness in random fractals. I’ll do my best to simplify and clarify the complicated and obscure here, so please bear with me.

Early fractal terrains were derived from a mathematical function called *fractional Brownian motion*, or *fBm* for short. FBm is a generalization of Brownian motion, which you may remember from your high school science classes: Brownian motion is the random walk of a very small particle being jostled about by the thermal motions of the much smaller particles comprising the fluid in which it is suspended. It's a lot like the random walk of an aimless drunk staggering about on a desert plain. (Don't you hate it when that happens to you at Burning Man?) FBm has a bunch of specific properties, and foremost among them are its uniformity: It's designed to look the same at all places and in all directions.* You can think of it as "statistically monotonous;" hence the name *mono*fractal.

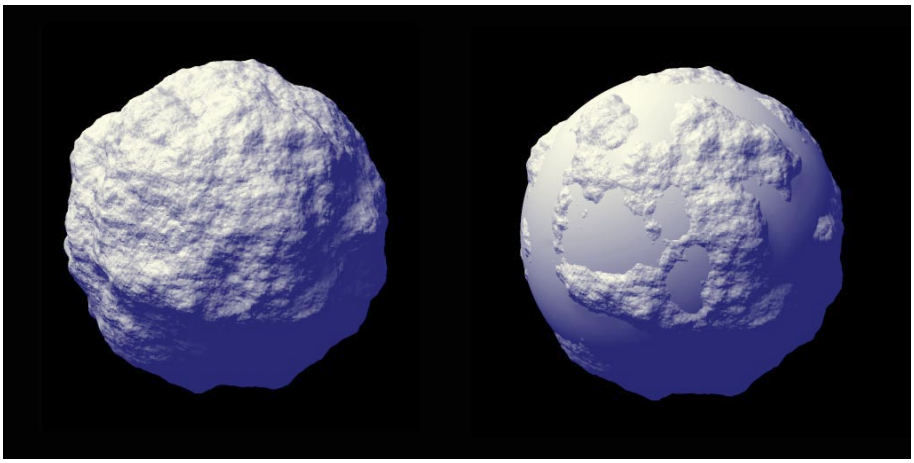


Figure 25. A planet made from monofractal fBm, a Perlin basis and roughness of 0.3, with and without "oceans."

This was a good start, and variations on fBm, generated from a variety of different basis functions, remains the standard random fractal used in MojoWorld and other fractal programs that create models of natural phenomena like mountains, clouds, fire and water.

Multifractals

Real terrains are not the same everywhere. Alpine mountains can rise out of flat plains—the eastern margin of America's Rocky Mountains being a conspicuous example. Early in my work with Mandelbrot, I wanted to capture some more of that kind of variety in fractal terrains, without complicating the very simple mathematical model that gives us fBm. As usual, I was reasoning as an artist, not a mathematician. I had some ideas about how the fractal dimension, or roughness, of terrain should vary with altitude and slope. For example, I knew that lakes fill in with sediment and become meadows as geologic time passes, so I thought "low areas should remain smooth, while peaks should be rough." Interestingly, the opposite appears to be more common in Nature, but what the heck, I was working in a dark closet (no kidding) on a landing on a staircase in the Yale math department at the time, so I was just working from memory, not active field work. I putzed about with my math—more at, my programs that implemented the math—and one

* If you care, this property is called *statistical stationarity*, in math-lingo.

day Benoit came in, saw a picture of what I had wrought and exclaimed "Oh! A multifractal!" I astutely replied, "What's a multifractal?" and proceeded on my merry way.

In order to escape Yale with my PhD and not get roasted like a pig on spit at my thesis defense, I tightened up my descriptions of these multifractal functions and did some interesting experiments that lead to dead ends, from the standpoint of making MojoWorlds (my ultimate goal). I did get a little attention from some physicists interested in multifractals, which I thought was cool, as an artist. At any rate, I've packaged up the two best-behaved multifractals I've devised and stuck 'em in MojoWorld.

"What do I care?" you ask. Well, monofractals get pretty boring pretty fast, because they're the same everywhere, all the time. Multifractals are a little more interesting, visually: they're *not* the same everywhere. They're smoother in some places and rougher in others. Nature, of course, is far more complex than this, but hey, it's a second step in the right direction.

In general, I always use multifractals for my terrains and usually use monofractals for clouds and textures.

The Heterofractal Function

The first of the two multifractal functions in MojoWorld I call *heterofractal*. Its roughness is a function of how far it is above or below "sea level" (where it tends to be quite smooth and boring). You can add in another function to a heterofractal terrain to move the terrain around vertically, so that the smooth areas don't always occur at the same altitude.

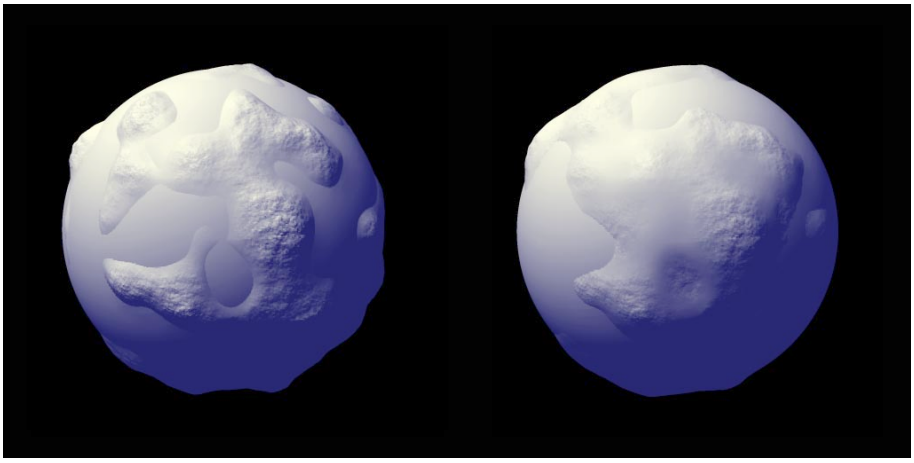


Figure 26. Planets made from heterofractal 3 with a Perlin basis and a smooth sphere at sea level. Straight heterofractal on the left, heterofractal plus a random vertical displacement on the right.

The Multifractal Function

The second of the two multifractal functions in MojoWorld is simply named *multifractal*. Back when I was still trying to figure out the Byzantine mathematics of multifractals, as in my dissertation and our book "Texturing and Modeling: A Procedural Approach," I called this function a "hybrid multifractal," for technical reasons. What I was then

calling "multifractal" turned out to be a dead end, for practical purposes, so I've since promoted "hybrid multifractal" to "multifractal" in my personal lexicon, and in MojoWorld's, by transitivity. ;-) It's a good workhorse, and my first choice for terrains on a planetary scale, most every time.

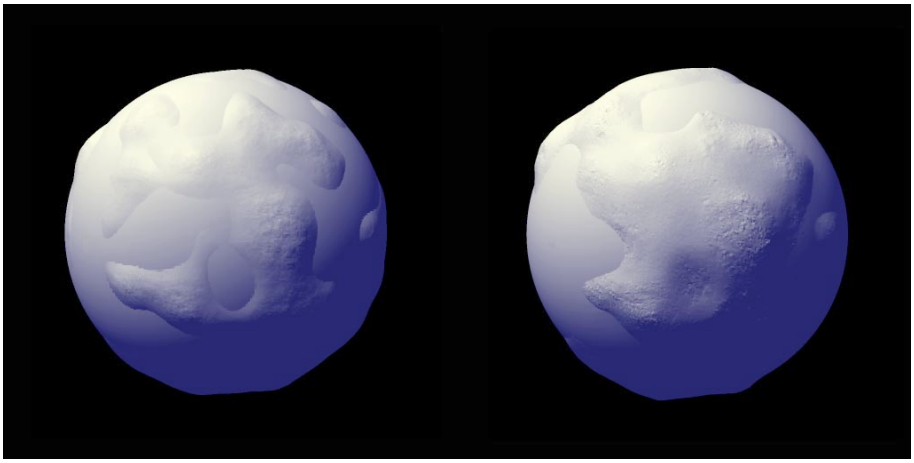


Figure 27. Planets made from multifractal 3 with a Perlin basis and a smooth sphere at sea level. Straight multifractal on the left, multifractal plus a random vertical displacement on the right.

Function Fractals

A fractal consists of some form repeated of a range of scales. There is no inherent restriction on what that form might be. In good, safe practice of image synthesis, though, there are. More math being glossed right over: There is a highest spatial frequency that can be used when synthesizing images on the computer. There are mathematics that tell us what that frequency is—the *Nyquist sampling theorem*. Frequencies higher than that limit, the *Nyquist limit*, will *alias* (read: turn to crap) in our images. The upshot: You don't want to go building fractals out of just anything; you really want to know what the highest spatial frequency is in your basis function. But alas, we all have a little Curious George in us, and we don't really give a damn about all that stupid math stuff when we're just making pictures, so GIVE US WHAT WE WANT AND PUT A CORK IN IT. "We hear you." Hence we have *function fractals* in MojoWorld. Rather than limiting you to using the carefully crafted basis functions built by well-informed professional engineers, with function fractals you can build 'em out of whatever ill-informed, ill-advised basis function you can come up with. >8-)

But be advised: There are two predictable consequences of using function fractals. First, they may be slow. The arbitrary basis function you put in may already be a fractal that requires thousands of CPU cycles to evaluate. Repeat that at a hundred different scales, and watch your fingernails grow as your image renders. Second, such fractals will almost certainly alias badly. This looks kind of like sandpaper in a still image; not too evil. But when you animate it, it can sizzle in a most annoying way.

Don't say we didn't warn you. But then, we have a motto here at Pandromeda: "Give 'em enough rope." >;^)

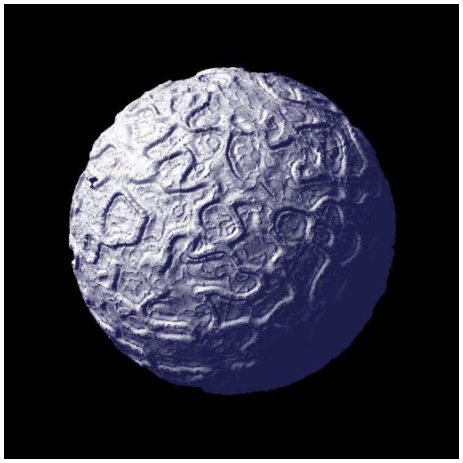


Figure 28. A planet made from a 3D function fractal.

Domain Distortion

One of the first things I tried when I started playing with procedural textures like we're using in MojoWorld is distorting one texture with another. Because we accomplish this by adding the output of one fractal function to the input of another, we call it *domain distortion*. Imagine a function with a one-dimensional domain, say the sine wave. Undistorted, it looks like this:

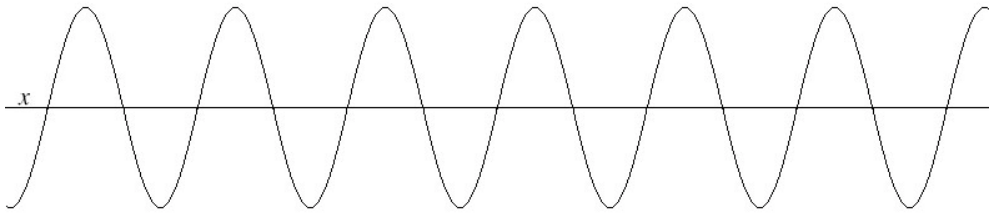


Figure 29. An undistorted sine wave: y = sine(x).

Imagine adding the cosine to x before we pass x to the sine function. What we're computing is then not the sine of x, but the sine of x plus the cosine of x. As the value of the cosine function varies from –1.0 to 1.0 and is added to x, it has the effect of displacing the x value that gets passed to the sine function, moving it back and forth from its undistorted value.

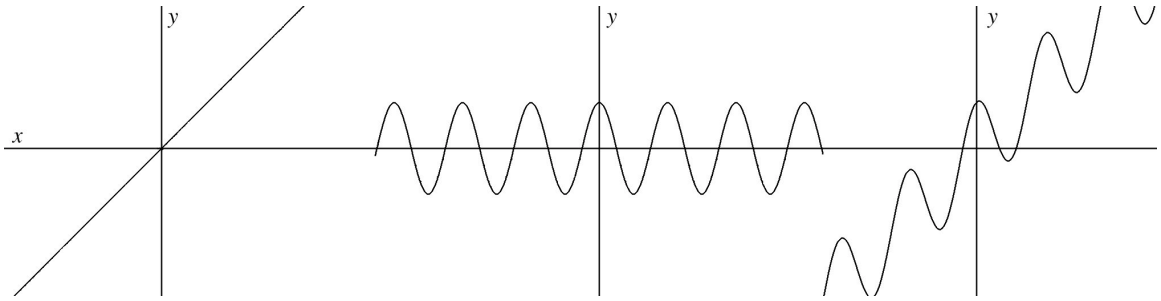


Figure 30. The undistorted domain y = x; a distortion function: y = cosine(x), and the distorted domain y = x + cosine(x).

Distorting the input of a function has the effect of distorting the output. We see such a result in Figure 30.

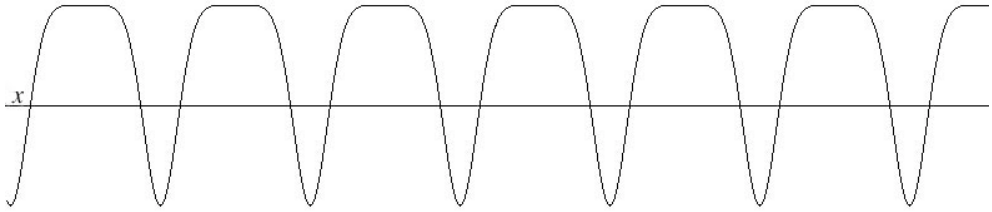


Figure 30. A distorted sine wave y = sine(x + cosine(x)).

Of course, that example is what mathematicians call "trivial." It's just to illustrate the process simply and clearly. Domain distortion in MojoWorld will generally involve more complex functions and take place in higher dimensions, but the way it's done is exactly the same. Figure 31 shows planets made from an undistorted fractal, and the same fractal with domain distortion. The domain distortion has the effect of stretching the distorted texture out in some places, and pinching it together in others.

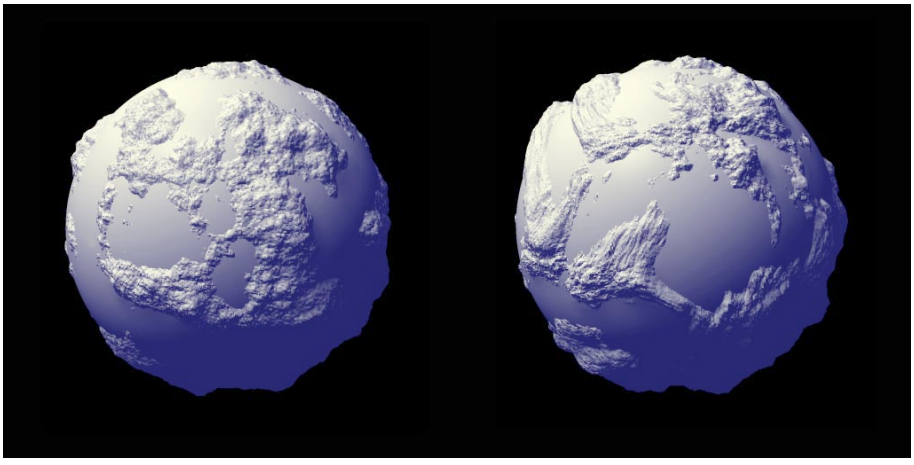


Figure 31. A planet with undistorted and distorted fractal relief.

MojoWorld's Texture Editor allows domain distortion both to the basis functions and to the aggregate fractal. For efficiency and to avoid severe aliasing, basis functions can only be distorted with other basis functions. On the other hand, fractal functions can be

distorted with other fractals; indeed, with any function. There are at least two reasons for them being treated differently. The first is that, for three-dimensional functions, the distortion function is three times as expensive to evaluate as the distorted function, as it has to be evaluated once for each of the three dimensions. The second is that the distortion function has to be evaluated once per octave of the distorted function, in the case of distortion applied to the basis function, but only once for distorting the aggregate fractal. As MojoWorld is churning through around 25 to 30 octaves—and potentially many more—when you're down close to the surface of a planet, performing anything other than simple distortion on a per-octave basis is simply not computationally practical, yet. No doubt it will become so in years to come, with faster computers, but it will still present aliasing problems.

Distorted Fractal Functions

You can use domain distortion to make long, linear mountain ranges in the areas where the distorted texture stretched in one direction and pinched in the other. But when you zoom in to those mountains, it's not very realistic to have all the little details all stretched and pinched in the same way—real mountains just don't look like that. So I borrowed an idea from turbulent flow, *viscous damping*, to get around that. Turbulent flow is damped, or slowed, by viscosity in the fluid at small scales.[*] Since domain distortion is kind of like turbulence, I thought "why not do the same thing with the domain distortion—taper it off at smaller scales." So MojoWorld has what I call *distorted fractal* functions available in the Graph Editor, or Pro UI. These are complicated little beasts; definitely a very advanced feature. Beginners will find them hopelessly confusing, I fear. But they have two fields, *onset* and *viscosity*, where you can specify where the viscous damping begins and is total (no distortion), respectively. The scales are specified in meters, the default unit of scale in MojoWorld.

Crossover Scales

This idea of scales for the onset of viscosity and where viscous damping is complete, leads to our next and last advanced feature in MojoWorld fractals: *crossover scales*. A crossover scale is simply a scale where the behavior of a fractal changes. The simplest examples are the *upper crossover scale* and *lower crossover scale*, above and below which fractal behavior vanishes. Mandelbrot makes the striking observation that the fractal dimension—the roughness—of the Himalayas and the runway at JFK airport are about the same; it's in their crossover scales that they differ.

MojoWorld has such crossover scales. There's always a largest feature size for any MojoWorld fractal, and if you don't explicitly set a lower crossover scale, MojoWorld will eventually say "enough" and quit adding detail. (That limit is up to the MojoWorld programmers.) In the *distorted fractal* functions we employ another kind of crossover scale, in another very advanced MojoWorld feature. I figured that zooming in and in on a

---

[*] "Small" is a relative term here; the scales at which viscosity damps turbulence depends on the viscosity of the turbulent fluid, which can be anything from molten rock, as in plumes in the Earth's mantle, to the tenuous gas in a near-perfect vacuum that we see in Figure 4.

single fractal, from the scale of continents to that of dirt, is neither very interesting nor realistic. In Nature, the character of terrain is different at different scales. So, in MojoWorld's distorted fractals, I included the ability to use different basis functions at different scales. You can use up to three different basis function in these fractals. When you use more than one, you have to specify the scale, in meters, where the crossover between basis functions begins and ends. It's not easy to show how this works, other than in an animated zoom. So no illustration here. :-)

Driving Function Parameters with Functions

One of the most powerful features of the MojoWorld Graph Editor, or Pro UI, is the ability to drive the value of almost any parameter of any function with the output of any other function. This can give some really wild and complicated results! Once you've become and advanced MojoWorld user, I recommend going into the Pro UI and playing with this. (It will probably be hopelessly confusing until you learn to think in the new, purely procedural MojoWorld paradigm.)

For example, using a "blend" node you can easily make a texture whose color is white above a certain altitude—the snow line. But a straight, horizontal snow line is not very natural looking. So you might create an "add" node, with "altitude" as one input and a fractal as the other. The add node is then a function with inputs and an output. You can hook the output of the add node to the parameter that controls where the blend node makes the transition to white. Now the snow line will be fractal and quite natural looking! See the tutorials in the MojoWorld manual for explicit examples of how to put together such function graphs.

A potent hidden aspect of the MojoWorld Graph Editor is that each node knows the dimensionality of the input it needs. All nodes will automatically provide output of the dimensionality requested by the parameter they are hooked into. Note that this doesn't free the user from having to make the right choice of dimensionality for certain function modes, as MojoWorld can't know what kind of effect you're out to create, and so it can't always make the choice for you.

Using Fractals

I've talked a lot about fractals and all their wonderful complexities in MojoWorld. Now let me talk a little about the specific uses for fractals.

Textures

Perhaps the main use for fractals in MojoWorld is in surface textures. Sure, MojoWorld can create some very complex geometry, but you can still get most of your interesting visual information from textures applied to surfaces. The procedural methods used in MojoWorld were originally used by Peachey and Perlin for creating such surface textures. You can create some really beautiful effects in color alone, using fractal procedural textures.

The development and artful use of fractal textures has pretty much made my career.

Such texture functions can be used to control surface color, shininess, displacement, transparency, you name it. Pretty much everything that makes a MojoWorld interesting and beautiful is some form of a fractal procedure, or *procedural texture*. That's why the Texture Editor is the very heart of MojoWorld. Doc Mojo's advice: Spend time mastering the Texture Editor. It is by far the most powerful tool in MojoWorld.

Terrains

Even the terrains that comprise a MojoWorld's planetary landscape are just procedural textures, used in this case to determine elevation. For consistency, that texture is evaluated on the surface of a sphere of constant radius, and the result is used to raise or lower the planet's terrain surface. The multifractal functions in MojoWorld were originally designed for modeling terrain, so I recommend using them when you're creating the terrain for a MojoWorld.

Displacement Maps

MojoWorld also features *displacement maps*: textures that can actually displace the surfaces they're applied to. Yes, a MojoWorld is just a displacement-mapped sphere. But the algorithm used to displace the planet's surface is a special one, designed for speed (at the expense of memory). There are two consequences to this: First, you can make displacement-mapped spheres for moons, but you can't zoom into them like you can a MojoWorld. (Well, you *could*, but the renderer would crawl to a halt, as you got close.) Second, you can't do lateral displacements on the MojoWorld terrain to get overhangs. "We'll fix both those things in a future version."

Clouds

MojoWorld 1.0 only has two-dimensional clouds, mapped onto spheres concentric with the planet. You can put any texture you like on those spheres, to represent your clouds. (You can do some really wild clouds—go for it!) Future versions will have full volumetric three-dimensional clouds. "We have the technology." See the "Great Balls of Fire" section of my web site—www.pandromeda.com/musgrave—for some animated examples of what I've done in the past.

More exciting still, in a future version we'll put in four-dimensional clouds. This will allow you to animate volumetric clouds over time. Woo-hoo!

Planets

Obviously, MojoWorld is an exercise in modeling entire planets with fractals. The possibilities are endless. I'm looking forward with great anticipation to seeing what people create and find in Parametric Hyperspace™. They're all out there, virtually, just waiting to be discovered.

Nebulae

Figure 4 illustrates rather convincingly that astrophysical nebulae are fractal in nature. Figure 32 illustrates an early experiment of mine in modeling with a multifractal texture, interstellar dust clouds like we see in the dark lanes in the Milky Way.

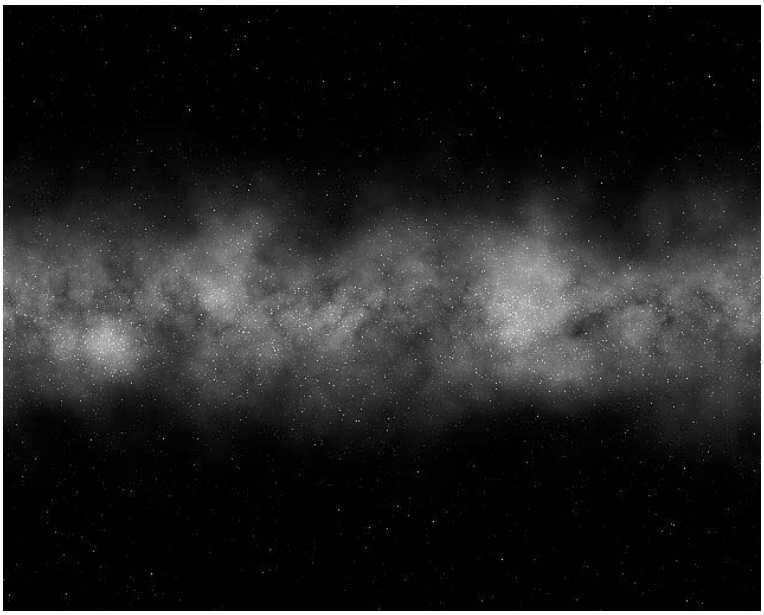


Figure 32. A multifractal texture as a model of the Milky Way.

Planets reside in solar systems, solar systems in galaxies, and galaxies in clusters. In future versions of MojoWorld we plan to model all these things. Volumetric nebulae are something we're all looking forward to seeing, playing with, and zipping through!

The Expressive Vocabulary of Random Fractals

People often ask me, "can you do people with fractals?" The answer is "no." Not everything is fractal. People don't look the same on a variety of scales (though parts of us, like our lungs and vascular systems, do). There is a limit to what can be done with fractals. We can do a lot, but certainly far from *everything*. MojoWorld is designed to do most of what can be done with random fractals today.

Perhaps the most fascinating aspect of MojoWorld is that all pure MojoWorlds—those without added content such as cities, monkeys, etc.—*already exist* in Parametric Hyperspace™. They are inherent in the timeless truth of mathematical logic. This is mind-boggling. So, do you *create* MojoWorlds, or do you merely *find* them? An interesting question indeed. I've had this conversation with Mandelbrot: did he invent fractal geometry, or did he discover it? He's not comfortable with either title, the "inventor of" or the "creator of" fractal geometry. How can you invent what's true? If it's true now, it's always been true and always will be. You can't "invent" what already exists!

But you don't just stumble across fractal geometry like a dollar bill on the sidewalk. Nor will you just stumble across any really great MojoWorlds in Parametric Hyperspace™—

it's way too big. It takes a very intelligent search to find great MojoWorlds in the vastness of Parametric Hyperspace™. So once you've isolated a set of transporter coordinates that really rock, did you *find* them or did you *create* them? It seems to be one of those conundrums in life: both are kind of true, neither is really true.

I've made a career of finding beauty in this hyperspace. I call myself an artist. I know it took creativity to…bring them home. The question of discovery versus creation is an interesting academic exercise, but on this I am clear: We've created MojoWorld to empower your creativity and to pique your imagination, your sense of adventure, and your sense of wonder. If you make great images/places/animations in MojoWorld, *I* will call you an "artist," without hesitation.

Experiment!

Get in there and experiment. Although Parametric Hyperspace™ as spanned by MojoWorld version 1.0 certainly doesn't include every world we'd ever like to see and explore, it is an infinitely vast virtual universe, much larger in fact than the one we inhabit, simply because it has so many dimensions. Get in there and find/create cool images/places and *share them* with the rest of us! That's part of why we're giving the Transporter away for free—so that everyone (with a fast enough computer) can go there and check it out. And you never know: Planets are big places, someone might find a more beautiful view on a planet you've created, than any you've yet found. That's one great thing about MojoWorld—it comes with a built-in audience, all the people who've downloaded the Transporter.

The Future

I've put into MojoWorld everything I can think of that's practical, and even a few things—like the sparse convolution basis—that probably aren't. Yet. What's practically doable on your home computer is a fast-moving target; they get faster at an amazing rate. We've designed MojoWorld to push the state of the art. And we can bring any processor in the world to its knees quite easily. Of course, we have other algorithms like radiosity (super-accurate illumination), physically-based erosion and fluid dynamics that we could stick in there, just in case we need to slow things down some more. >;^)

The Holy Grail we seek is virtual reality. Not the lame anything-but-real stuff we've seen called "VR" to date, but *believable* virtual reality—interactive MojoWorlds as beautiful and realistic as those we can render currently in non-real time. It will happen. And not too long from now. It's only a matter of engineering. So buy MojoWorld Generator™ and help power the cause! ;^)

Simulating Ocean Water

Jerry Tessendorf

Copyright ©1999 – 2001, Jerry Tessendorf

1 Introduction and Goals

These notes are intended to give computer graphics programmers and artists an introduction to methods of simulating, animating, and rendering ocean water environments. CG water has become a common tool in visual effects work at all levels of computer graphics, from print media to feature films. Several commercial products are now available for nearly any computer platform and work environment, in addition to proprietary tools held by a few companies, which generate high quality geometry and images. In order for an artist to exploit the tools to maximum benefit, it is important that he or she become familiar with concepts, terminology, a little oceanography, and the present state of the art.

As demonstrated by the pioneering efforts in the films *Waterworld* and *Titanic*, as well as several other films made since about 1995, images of cg water can be generated with a high degree of realism. However, this level of realism is limited to relatively calm, nice ocean conditions. Conditions with large amounts of spray, breaking waves, foam, splashing, and wakes are approaching the same realistic look.

Broadly, the reader should come away from this material with (1) an understanding of some algorithms that generate/animate water surface height fields suitable for modeling waves as big as storm surges and as small as tiny capillaries; (2) an understanding of the basic optical processes of reflection and refraction from a water surface; (3) an introduction to the color filtering behavior of ocean water; (4) an introduction to complex lighting effects known as caustics and godrays, produced when sunlight passes through the rough surface into the water volume underneath; and (5) some rules of thumb for which choices make nice looking images and what are the tradeoffs of quality versus computational resources. Some example shaders are provided, and example renderings demonstrate the content of the discussion.

Before diving into it, I first want to be more concrete about what aspect of the ocean environment we cover (or not cover) in these notes. Figure 1 is a rendering of an oceanscape produced from models of water, air, and clouds. Light from the clouds is reflected from the surface. On the extreme left, sun glitter is also present. The generally bluish color of the water is due to the reflection of blue skylight, and to light coming out of the water after scattering from the volume. Although these notes do not tackle the modeling and rendering of clouds and air, there is a discussion of how skylight from the clouds and air is reflected from, or refracted through, the water surface. These notes will tell you how to make a height-field displacement-mapped surface for the ocean waves with the detail and quality shown in the figure. The notes also discuss several effects of the underwater environment and how to model/render them. The primary four effects are sunbeams (also called godrays), caustics on underwater surfaces, blurring by the scattering of light, and color filtering.

There are also many other complex and interesting aspects of the ocean environment that will not be covered. These include breaking waves, spray, foam, wakes around objects in the water, splashes from bodies that impact the surface, and global illumination of the entire ocean-atmosphere environment. There is substantial research underway on these topics, and so it is possible that future versions of this or other lecture notes will include them. I have included a brief section on advanced modifications to the basic wave height algorithm that produce choppy waves. The modification could feasibly lead to a complete description of the surface portion of breaking waves, and possibly serve to drive the spray and foam dynamics as well.

There is, of course, a substantial body of literature on ocean surface simulation and animation, both in computer graphics circles and in oceanography. One of the first descriptions of water waves in computer graphics was by Fournier and Reeves[8] , who modeled

Figure 1: Rendered image of an oceanscape.

a shoreline with waves coming up on it using a water surface model called *Gerstner waves*. In that same issue, Darwin Peachey[9] presented a variation on this approach using basis shapes other than sinusoids.

In the oceanographic literature, ocean optics became an intensive topic of research in the 1940s. S.Q. Duntley published[13] in 1963 papers containing optical data of relevance to computer graphics. Work continues today. The field of optical oceanography has grown into a mature quantitative science with subdisciplines and many different applications. One excellent review of the state of the science was written by Curtis Mobley[14].

In these lectures the approach we take to creating surface waves is close to the one outlined by Masten, Watterberg, and Mareda[7], although the technique had been in use for many years prior to their paper in the optical oceanography community. This approach synthesizes a patch of ocean waves from a Fast Fourier Transform (FFT) prescription, with user-controllable size and resolution, and which can be tiled seamlessly over a larger domain. The patch contains many octaves of sinusoidal waves that all add up at each point to produce the synthesized height. The mixture of sinusoidal amplitudes and phases however, comes from statistical, emperically-based models of the ocean. What makes these sinusoids look like waves and not just a bunch of sine waves is the large collection of sinusoids that are used, the relative amplitudes of the sinusoids, and their animation using the dispersion relation. We examine the impact of the number of sinusoids and resolution on the quality of the rendered image.

In the next section we begin the discussion of the ocean environment with a broad introduction to the global illumination problem. The radiosity equations for this environment look much like those of any other radiosity problem, although the volumetric character of some of the environmental components complicate a general implementation considerably. However, we simplify the issues by ignoring some interactions and replacing others with models generated by remote sensing data.

Practical methods are presented in section 3 for creating realizations of ocean surfaces. We present two methods, one based on a simple model of water structure and movement, and one based on summing up large numbers of sine waves with amplitudes that are related to each other based on experimental evidence. This second method carries out the sum using the technique of Fast Fourier Transformation (fft), and has been used effectively in projects for commercials, television, and motion pictures.

After the discussion of the structure and animation of the water surface, we focus on the optical properties of water relevant to the

graphics problem. First, we discuss the interaction at the air-water interface: reflection and refraction. This leaves us with a simple but effective Renderman-style shader suitable for rendering water surfaces in BMRT, for example. Next, the optical characteristics of the underwater environment are explored.

2 Radiosity of the Ocean Environment

The ocean environment, for our purposes, consists of only four components: The water surface, the air, the sun, and the water below the surface. In this section we trace the flow of light through the environment, both mathematically and schematically, from the light source to the camera. In general, the radiosity equations here are as coupled as any other radiosity problem. To a reasonable degree, however, the coupling can be truncated and the simplified radiosity problem has a relatively fast solution .

The light seen by a camera is dependent on the flow of light energy from the source(s) (i.e. the sun and sky) to the surface and into the camera. In addition to specular reflection of direct sunlight and skylight from the surface, some fraction of the incident light is transmitted through the surface. Ultimately, some fraction of the transmitted light is scattered by the water volume back up into the air. Some of the light that is reflected or refracted at the surface may strike the surface a second time, producing more reflection and refraction events. Under some viewing conditions, multiple reflections and refractions can have a noticeable impact on images. For our part however, we will ignore more than one reflection or refraction from the surface at a time. This not only makes the algorithms and computation easier and faster, but also is reasonably accurate most of the time.

At any point in the environment above the surface, including at the camera, the total light intensity (radiance) coming from any direction has three contributions:

$$L_{ABOVE} = rL_S + rL_A + t_U L_U \ , \tag{1}$$

with the following definitions of the terms:

- r is the Fresnel reflectivity for reflection from a spot on the surface of the ocean to the camera.
- t_U is the transmission coefficient for the light L_U coming up from the ocean volume, refracted at the surface into the camera.
- L_S is the amount of light coming directly from the sun, through the atmosphere, to the spot on the ocean surface where it is reflected by the surface to the camera.
- L_A is the (diffuse) atmospheric skylight
- L_U is the light just below the surface that is transmitted through the surface into the air.

Equation 1 has intentionally been written in a shorthand way that hides the dependences on position in space and the direction the light is traveling.

While equation 1 appears to have a relatively simple structure, the terms L_S, L_A, and L_U can in principle have complex dependencies on each other, as well on the reflectivity and transmissivity. There is a large body of research literature investigating these dependencies in detail [15], but we will not at this point pursue these quantitative methods. But we can elaborate further on the coupling while continuing with the same shorthand notation. The direct light from the sun L_S is

$$L_S = L_{TOA} \exp\{-\tau\} \ , \tag{2}$$

where L_{TOA} is the intensity of the direct sunlight at the top of the atmosphere, and τ is the "optical thickness" of the atmosphere for the direction of the sunlight and the point on the earth. Both the diffuse atmospheric skylight L_A and the upwelling light L_U can be written as the sum of two terms:

$$L_A = L_A^0(L_S) + L_A^1(L_U) \tag{3}$$

$$L_U = L_U^0(L_S) + L_U^1(L_A) \tag{4}$$

These equations reveal the potential complexity of the problem. While both L_A and L_U depend on the direct sunlight, they also depend on each other. For example, the total amount of light penetrating into the ocean comes from the direct sunlight and from the atmospheric sunlight. Some of the light coming into the ocean is scattered by particulates and molecules in the ocean, back up into the atmosphere. Some of that upwelling light in turn is scattered in the atmosphere and becomes a part of the skylight shining on the surface, and on and on. This is a classic problem in radiosity. It is not particularly special for this case, as opposed to other radiosity problems, except perhaps for the fact that the upwelling light is difficult to compute because it comes from volumetric multiple scattering.

Our approach, for the purposes of these notes, to solving this radiosity problem is straightforward: take the skylight to depend only on the light from the sun, since the upwelling contribution represents a "tertiary" dependence on the sunlight; and completely replace the equation for L_U with an empirical formula, based on scientific observations of the oceans, that depends only on the direct sunlight and a few other parameters that dictate water type and clarity.

Under the water surface, the radiosity equation has the schematic form

$$L_{BELOW} = tL_D + tL_I + L_{SS} + L_M \ , \tag{5}$$

with the meaning

- t is the Fresnel transmissivity for transmission through the water surface at each point and angle on the surface.
- L_D The "direct" light from the sun that penetrates into the water.
- L_I The "indirect" light from the atmosphere that penetrates into the water.
- L_{SS} The single-scattered light, from both the sun and the atmosphere, that is scattered once in the water volume before arriving at any point.
- L_M The multiply-scattered light. This is the single-scattered light that undergoes more scattering events in the volume.

Just as for the above water case, these terms are all related to each other is relative complex ways. For example, the single scattered light depends on the direct and indirect light:

$$L_{SS} = P(tL_I) + P(tL_D) \tag{6}$$

with the quantity P being a linear functional operator of its argument, containing information about the single scattering event and the attenuation of the scattered light as it passes from the scatter point to the camera. Similarly, the multiply-scattered light is dependent on the single scattered:

$$L_M = G(tL_I) + G(tL_D) \ . \tag{7}$$

The functional schematic quantities P and G are related, since multiple scattering is just a series of single scatters. Formally, the two have an operator dependence that has the form

$$\begin{aligned} G &\sim P \otimes P \otimes \left\{1 + P + \frac{1}{2!}P \otimes P + \frac{1}{3!}P \otimes P \otimes P + \ldots\right\} \\ &\sim P \otimes P \otimes \exp(P) \ . \end{aligned} \tag{8}$$

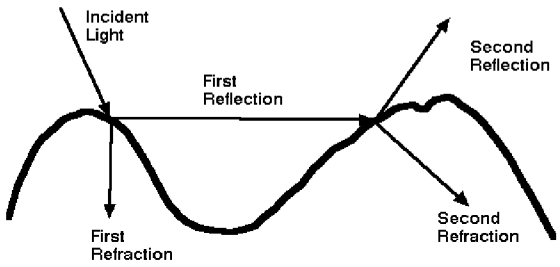


Figure 2: Illustration of multiple reflections and transmission through the air-water interface.

At this point, the schematic representation may have outlived its usefullness because of the complex (and not here defined) meaning of the convolution-like operator $\otimes$, and because the expression for G in terms of P has created an even more schematic view in terms of an exponentiated P. So for now we will leave the schematic representation, and journey on with more concrete quantities the rest of the way through.

The formal schematic discussion put forward here does have a mathematically and physically precise counterpart. The field of study in Radiative Transfer has been applied for some time to water optics, by a large number researchers. The references cited are excellent reading for further information.

As mentioned, there is one additional radiosity scenario that can be important to ocean rendering under certain circumstances, but which we will not consider. The situation is illustrated in figure 2. Following the trail of the arrows, which track the direction light is travelling, we see that sometimes light coming to the surface (from above or below), can reflect and/or transmit through the surface more than once. The conditions which produce this behavior in significant amounts are: the wave heights must be fairly high, and the direction of viewing the waves, or the direction of the light source must be nearly grazing the surface. The higher the waves are, the less grazing the light source or camera need to be. This phenonmenon has been examined experimentally and in computer simulations. It is reasonably well understood, and we will ignore it from this point on.

3 Practical Ocean Wave Algorithms

In this section we focus on algorithms and practical steps to building height fields for ocean waves. Although we will be occupied mostly by a method based on Fast Fourier Transforms (FFTs), we begin by introducing a simpler description called Gerstner Waves. This is a good starting point for several reasons: the mathematics is relatively light compared to FFTs, several important oceanographic concepts can be introduced, and they give us a chance to discuss wave animation. After this discussion of Gerstner waves, we go after the more complex FFT method, which produces wave height fields that are more realistic. These waves, called "linear waves" or "gravity waves" are a fairly realistic representation of typical waves on the ocean when the weather is not too stormy. Linear waves are certainly not the whole story, and so we discuss also some methods by which oceanographers expand the description to "nonlinear waves", waves passing over a shallow bottom, and very tiny waves about one millimeter across called capillary waves.

In the course of this discussion, we will see how quantities like windspeed, surface tension, and gravitational acceleration come into the practical implementation of the algorithms.

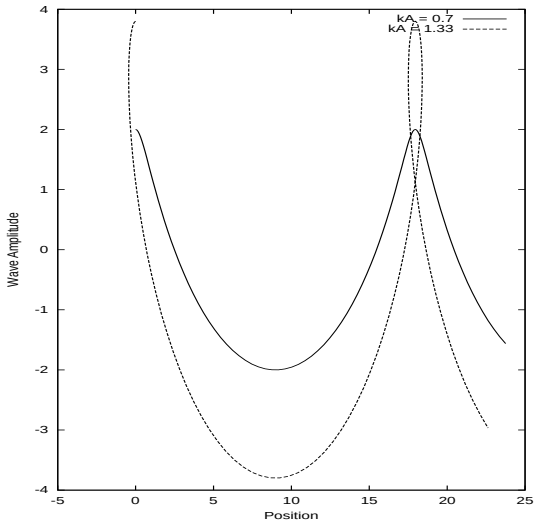


Figure 3: Profiles of two single-mode Gerstner waves, with different relative amplitudes and wavelengths.

3.1 Gerstner Waves

Gerstner waves were first found as an approximate solution to the fluid dynamic equations almost 200 years ago. There first application in computer graphics seems to be the work by Fournier and Reeves in 1986 (cited previously). The physical model is to describe the surface in terms of the motion of individual points on the surface. To a good approximation, points on the surface of the water go through a circular motion as a wave passes by. If a point on the undisturbed surface is labelled $\mathbf{x}_0 = (x_0, z_0)$ and the undisturbed height is $y_0 = 0$, then as a single wave with amplitude A passes by, the point on the surface is displaced at time t to

$$\mathbf{x} = \mathbf{x}_0 - (\mathbf{k}/k)A\sin(\mathbf{k}\cdot\mathbf{x}_0 - \omega t) \quad (9)$$

$$y = A\cos(\mathbf{k}\cdot\mathbf{x}_0 - \omega t)\ . \quad (10)$$

In these expressions, the vector $\mathbf{k}$, called the wavevector, is a horizontal vector that points in the direction of travel of the wave, and has magnitude k related to the length of the wave (λ) by

$$k = 2\pi/\lambda \quad (11)$$

The frequency w is related to the wavevector, as discussed later.

Figure 3 shows two example wave profiles, each with a different value of the dimensionless amplitude kA. For values $kA < 1$, the wave is periodic and shows a steepening at the tops of the waves as kA approaches 1. For $kA > 1$, a loop forms at the tops of the wave, and the "insides of the wave surface are outside", not a particularly desirable or realistic effect.

As presented so far, Gerstner waves are rather limited because they are a single sine wave horizontally and vertically. However, this can be generalized to a more complex profile by summing a set of sine waves. One picks a set of wavevectors $\mathbf{k}_i$, amplitudes A_i, frequencies ω_i, and phases ϕ_i, for $i = 1, \ldots, N$, to get the expressions

$$\mathbf{x} = \mathbf{x}_0 - \sum_{i=1}^{N}(\mathbf{k}_i/k_i)A_i\sin(\mathbf{k}_i\cdot\mathbf{x}_0 - \omega_i t + \phi_i) \quad (12)$$

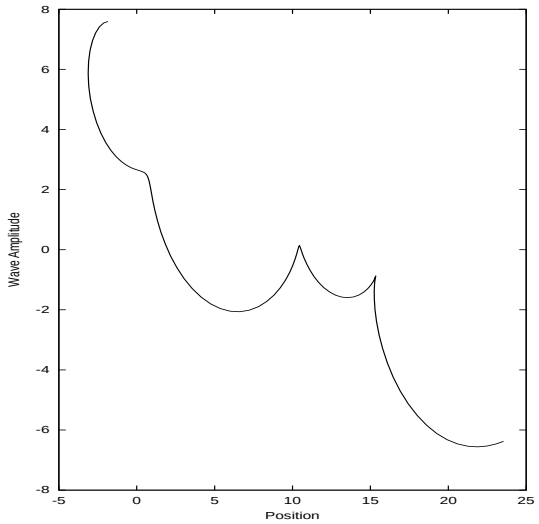


Figure 4: Profile of a 3-mode Gerstner wave.

$$y = \sum_{i=1}^{N} A_i \cos(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i) \; . \tag{13}$$

Figure 4 shows an example with three waves in the set. Interesting and complex shapes can be obtained in this way.

3.2 Animating Waves: The Dispersion Relation

The animated behavior of Gerstner waves is determined by the set of frequencies ω_i chosen for each component. For water waves, there is a well-known relationship between these frequencies and the magnitude of their corresponding wavevectors, k_i. In deep water, where the bottom may be ignored, that relationship is

$$\omega^2(k) = gk \; . \tag{14}$$

The parameter g is the gravitational constant, nominally $9.8m/sec^2$. This dispersion relationship holds for Gerstner waves, and also for the FFT-based waves introduced next.

There are several conditions in which the dispersion relationship is modified. When the bottom is relatively shallow compared to the length of the waves, the bottom has a retarding affect on the waves. For a bottom at a depth D below the mean water level, the dispersion relation is

$$\omega^2(k) = gk \tanh(kD) \tag{15}$$

Notice that if the bottom is very deep, the behavior of the $\tanh$ function reduces this dispersion relation to the previous one.

A second situation which modifies the dispersion relation is surface tension. Very small waves, with a wavelength of about 1 cm or less, have an additional term:

$$\omega^2(k) = gk(1 + k^2 L^2) \; , \tag{16}$$

and the parameter L has units of length. Its magnitude is the scale for the surface tension to have effect.

Using these dispersion relationships, it is very difficult to create a sequence of frames of water surface which for a continuous loop. In order to have the sequence repeat after a certain amount of time

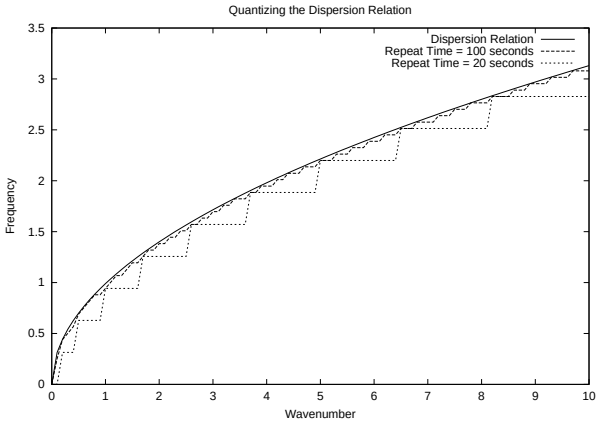


Figure 5: A comparison of the continuous dispersion curve $\omega = \sqrt{gk}$ and quantized dispersion curves, for repeat times of 20 seconds and 100 seconds. Note that for a longer repeat time, the quantized is a closer approximation to the original curve.

T for example, it is necessary that all frequencies be multiples of the basic frequence

$$\omega_0 \equiv \frac{2\pi}{T} \; . \tag{17}$$

However, when the wavevectors $\mathbf{k}$ are distributed on a regular lattice, itis impossible to arrange the dispersion-generated frequencies to also be on a uniform lattce with spacing ω_0.

The solution to that is to not use the dispersion frequences, but instead a set that is close to them. For a given wavenumber k, we use the frequency

$$\bar{\omega}(k) = \left[\left[\frac{\omega(k)}{\omega_0}\right]\right] \omega_0 \; , \tag{18}$$

where $[[a]]$ means take the integer part of the value of a, and $\omega(k)$ is any dispersion relationship of interest. The frequencies $\bar{\omega}(k)$ are a *quantization* of the dispersion surface, and the animation of the water surface loops after a time T because the quantized frequencies are all integer multiples of ω_0. Figure 5 plots the original dispersion curve, along with quantized dispersion curves for two choices of the repeat time T.

3.3 Statistical Wave Models and the Fourier Transform

Oceanographic literature tends to downplay Gerstner waves as a realistic model of the ocean. Instead, statistical models are used, in combination with experimental observations. In the statistical models, the wave height is considered a random variable of horizontal position and time, $h(\mathbf{x}, t)$.

Statistical models are also based on the ability to decompose the wave height field as a sum of sine and cosine waves. The value of this decomposition is that the amplitudes of the waves have nice mathematical and statistical properties, making it simpler to build models. Computationally, the decomposition uses Fast Fourier Transforms (ffts), which are a rapid method of evaluating the sums.

The fft-based representation of a wave height field expresses the wave height $h(\mathbf{x}, t)$ at the horizontal position $\mathbf{x} = (x, z)$ as the sum of sinusoids with complex, time-dependent amplitudes:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t)\ \exp(i\mathbf{k} \cdot \mathbf{x}) \tag{19}$$

where t is the time and $\mathbf{k}$ is a two-dimensional vector with components $\mathbf{k} = (k_x, k_z)$, $k_x = 2\pi n/L_x$, $k_z = 2\pi m/L_z$, and n and m are integers with bounds $-N/2 \le n < N/2$ and $-M/2 \le m < M/2$. The fft process generates the height field at discrete points $\mathbf{x} = (nL_x/N, mL_z/M)$. The value at other points can also be obtained by switching to a *discrete* fourier transform, but under many circumstances this is unnecessary and is not applied here. The height amplitude Fourier components, $\tilde{h}(\mathbf{k}, t)$, determine the structure of the surface. The remainder of this subsection is concerned with generating random sets of amplitudes in a way that is consistent with oceanographic phenomenology.

For computer graphics purposes, the slope vector of the wave-height field is also needed in order to find the surface normal, angles of incidence, and other aspects of optical modeling as well. One way to compute the slope is though a finite difference between fft grid points, separated horizontally by some 2D vector $\Delta\mathbf{x}$. While a finite difference is efficient in terms of memory requirements, it can be a poor approximation to the slope of waves with small wavelength. An exact computation of the slope vector can be obtained by using more ffts:

$$\epsilon(\mathbf{x}, t) = \nabla h(\mathbf{x}, t) = \sum_{\mathbf{k}} i\mathbf{k}\ \tilde{h}(\mathbf{k}, t)\ \exp(i\mathbf{k} \cdot \mathbf{x})\ . \tag{20}$$

In terms of this fft representation, the finite difference approach would replace the term $i\mathbf{k}$ with terms proportional to

$$\exp(i\mathbf{k} \cdot \Delta\mathbf{x}) - 1 \tag{21}$$

which, for small wavelength waves, does not well approximate the gradient of the wave height. Whenever possible, slope computation via the fft in equation 20 is the prefered method.

The fft representation produces waves on a patch with horizontal dimensions $L_x \times L_z$, outside of which the surface is perfectly periodic. In practical applications, patch sizes vary from 10 meters to 2 kilometers on a side, with the number of discrete sample points as high as 2048 in each direction (i.e. grids that are 2048×2048, or over 4 million waves). The patch can be tiled seamlessly as desired over an area. The consequence of such a tiled extension, however, is that an artificial periodicity in the wave field is present. As long as the patch size is large compared to the field of view, this periodicity is unnoticeable. Also, if the camera is near the surface so that the effective horizon is one or two patch lengths away, the periodicity will not be noticeable in the look-direction, but it may be apparent as repeated structures across the field of view.

Oceanographic research has demonstrated that equation 19 is a reasonable representation of naturally occurring wind-waves in the open ocean. Statistical analysis of a number of wave-buoy, photographic, and radar measurements of the ocean surface demonstrates that the wave height amplitudes $\tilde{h}(\mathbf{k}, t)$ are nearly statistically stationary, independent, gaussian fluctuations with a spatial spectrum denoted by

$$P_h(\mathbf{k}) = \left\langle \left| \tilde{h}^*(\mathbf{k}, t) \right|^2 \right\rangle \tag{22}$$

for data-estimated ensemble averages denoted by the brackets $\langle\ \rangle$.

There are several analytical semi-empirical models for the wave spectrum $P_h(\mathbf{k})$. A useful model for wind-driven waves larger than capillary waves in a fully developed sea is the *Phillips* spectrum

$$P_h(\mathbf{k}) = A \frac{\exp\left(-1/(kL)^2\right)}{k^4} |\hat{\mathbf{k}} \cdot \hat{w}|^2\ , \tag{23}$$

where $L = V^2/g$ is the largest possible waves arising from a continuous wind of speed V, g is the gravitational constant, and $\hat{w}$ is the direction of the wind. A is a numeric constant. The cosine factor $|\hat{\mathbf{k}} \cdot \hat{w}|^2$ in the Phillips spectrum eliminates waves that move perpendicular to the wind direction. This model, while relatively simple, has poor convergence properties at high values of the wavenumber $|\mathbf{k}|$. A simple fix is to suppress waves smaller that a small length $\ell \ll L$, and modify the Phillips spectrum by the multiplicative factor

$$\exp\left(-k^2\ell^2\right)\ . \tag{24}$$

Of course, you are free to "roll your own" spectrum to try out various effects.

3.4 Building a Random Ocean Wave Height Field

Realizations of water wave height fields are created from the principles elaborated up to this point: gaussian random numbers with spatial spectra of a prescribed form. This is most efficiently accomplished directly in the fourier domain. The fourier amplitudes of a wave height field can be produced as

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}} (\xi_r + i\xi_i) \sqrt{P_h(\mathbf{k})}\ , \tag{25}$$

where ξ_r and ξ_i are ordinary independent draws from a gaussian random number generator, with mean 0 and standard deviation 1. Gaussian distributed random numbers tend to follow the experimental data on ocean waves, but of course other random number distributions could be used. For example, log-normal distributions could be used to produce height fields that are vary "intermittent", i.e. the waves are very high or nearly flat, with relatively little in between.

Given a dispersion relation $\omega(k)$, the Fourier amplitudes of the wave field realization at time t are

$$\begin{aligned} \tilde{h}(\mathbf{k}, t) &= \tilde{h}_0(\mathbf{k}) \exp\{i\omega(k)t\} \\ &+ \tilde{h}_0^*(-\mathbf{k}) \exp\{-i\omega(k)t\} \end{aligned} \tag{26}$$

This form preserves the complex conjugation property $\tilde{h}^*(\mathbf{k}, t) = \tilde{h}(-\mathbf{k}, t)$ by propagating waves "to the left" and "to the right". In addition to being simple to implement, this expression is also efficient for computing $h(\mathbf{x}, t)$, since it relies on ffts, and because the wave field at any chosen time can be computed without computing the field at any other time.

In practice, how big does the Fourier grid need to be? What range of scales is reasonable to choose? If you want to generate wave heights faster, what do you do? Lets take a look at these questions.

How big should the Fourier grid be? The values of N and M can be between 16 and 2048, in powers of two. For many situations, values in the range 128 to 512 are sufficient. For extremely detailed surfaces, 1024 and 2048 can be used. For example, the wave fields used in the motion pictures *Waterworld* and *Titanic* were 2048×2048 in size, with the spacing between grid points at about 3 cm. Above a value of 2048, one should be careful because the limits of numerical accuracy for floating point calculations can become noticeable.

What range of scales is reasonable to choose? The answer to this question comes down to choosing values for L_x, L_z, M, and N. The smallest facet in either direction is $dx \equiv L_x/M$ or $dz \equiv L_z/N$. Generally, dx and dz need never go below 2 cm or so. Below this scale, the amount of wave action is small compared to the rest of the waves. Also, the physics

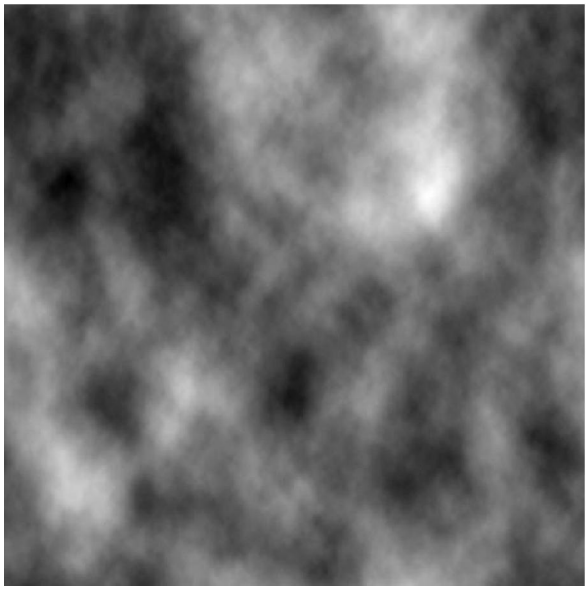


Figure 6: A surface wave height realization, displayed in greyscale.

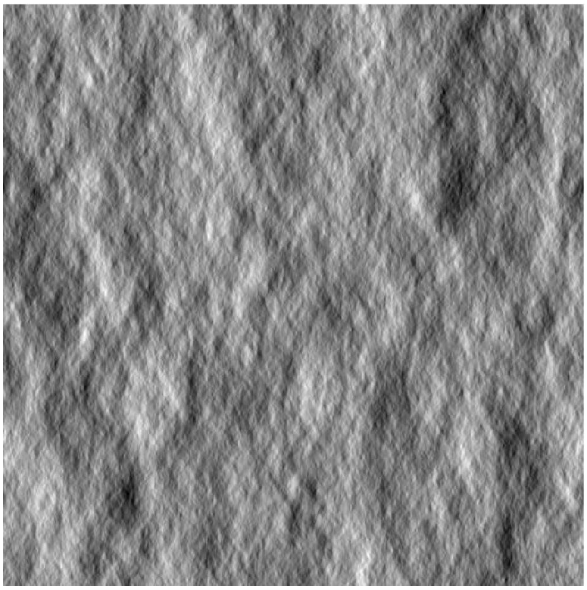


Figure 7: The x-component of the slope for the wave height realization in figure 6.

of wave behavior below 2 cm begins to take on a very different character, involving surface tension and "nonlinear" processes. From the form of the spectrum, waves with a wavelength larger than V^2/g are suppressed. So make sure that dx and dz are smaller than V^2/g by a substantial amount (10 - 1000) or most of the interesting waves will be lost. The secret to realistic looking waves (e.g. figure 9 (a) compared to figure 9 (c)) is to have M and N as large as reasonable.

How do you generate wave height fields in the fastest time? The time consuming part of the computation is the fast fourier transform. Running on a 180 MHz PowerPC 603e processor (under LinuxPPC r4), a 1024x1024 fft takes less than a minute. However, faster times are achieved by setting M and N to smaller powers of 2.

3.5 Examples: Height Fields and Renderings

We now turn to some examples of waves created using the fft approach discussed above. We will show waves in two formats: as greyscale images in which the grey level is proportional to wave height; and renderings of oceanscapes using several different rendering packages to illustrate what is possible.

In the first set of examples, the grid size is set to $M = N = 512$, with $L_x = L_z = 1000$ meters. The wind speed is a gale force at $V = 31$ meters/second, moving in the x-direction. The small-wave cutoff of $\ell = 1$ meter was also used. Figure 6 is a greyscale representation of the wave height: brighter means higher and darker means lower height. Although produced by the fft algorithms described here, figure 6 is not obviously a water height field. It may help to examine figure 7, which is a greyscale depiction of the x-component of the slope. This looks more like water waves that figure 6. What is going on?

Figures 6 and 7 demonstrate a consequence of water surface optics, discussed in the next section: the visible qualities of the surface structure tend to be strongly influenced by the slope of the waves. We will discuss this in quantitative detail, but for now we willl summarize it by saying that the reflectivity of the water is a strong function of the slope of the waves, as well as the directions of the light(s) and camera.

To illustrate a simple effect of customizing the spectrum model, figure 8 is the greyscale display of a height field identical to figure 6, with the exception that the directional factor $|\hat{\mathbf{k}} \cdot \hat{w}|^2$ in equation 23 has been changed to $|\hat{\mathbf{k}} \cdot \hat{w}|^6$. The surface is clearly more aligned with the direction of the wind.

The next example of a height field uses a relatively simple shader in BMRT, the Renderman-compliant raytracer. The shader is shown in the next section. Figure 9 shows three renderings of water surfaces, varying the size of the grid numbers M and N and making the facet sizes dx and dz proportional to $1/M$ and $1/N$. So as we go from the top image to the bottom, the facet sizes become smaller, and we see the effect of increasing amount of detail in the renderings. Clearly, more wave detail helps to build a realistic-looking surface.

As a final example, figure 10 is an image rendered in the commercial package RenderWorld by Arete Entertainment. This rendering includes the effect of an atmosphere, and water volume scattered light. These are discussed in the next section. But clearly, wave height fields generated from random numbers using an fft prescription can produce some nice images.

3.6 Experimental Evidence

This subsection is a diversion away from the main graphics thrust of these notes. It is an account of a relatively simple remote sensing experiment that demonstrates that the algorithms discussed in this section are grounded (wetted ?) in reality. Figure 11 is a frame from a video segment showing water coming into the beach near Zuma Beach, California. The video camera was located on hill overlooking the beach, in 1986. In 1993, the region of video frames indicated in the figure was digitized, to produce a time series of frames containing just water surface.

The analysis of the image data has a statistical character. Recall that the Fourier Transform method of creating surface simulations arises from statistical arguments about the averaged structure of the surface. Recall also that, to some degree, the grayscale display of

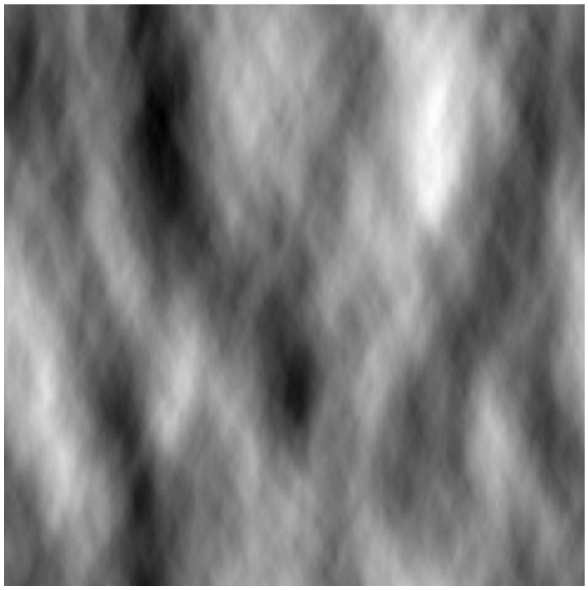


Figure 8: Wave height realization with increased directional dependence.

the slope field is reminiscent of an overhead view of a water surface. Our data analysis here attempts to use these assumptions to compute statistical quantities of the images, that should be related to the wave statistical properties.

From the multiple frames, a three dimensional Power Spectral Density (PSD) was created. The PSD is computed from the images by a two step processes (1) Fourier Transform the images in space and time to create the quantity

$$\tilde{h}(\mathbf{k},\omega) = \int d^2x\, dt\, h(\mathbf{x},t) \exp\left\{-i\mathbf{k}\cdot\mathbf{x} + i\omega t\right\} \ , \qquad (27)$$

and (2) form the estimated PSD by smoothing the absolute square of $\tilde{h}$. In mathematical notation, this is

$$PSD(\mathbf{k},\omega) = \overline{\left|\tilde{h}(\mathbf{k},\omega)\right|^2} \qquad (28)$$

There is a good reason for creating the 3D PSD as defined here. If the waves animate in time as prescribed in equation 26, that is, with the dispersion relationship $\omega = \omega(k)$, then the PSD will have a large value in the regions where $\mathbf{k}$ and ω satisfy that dispersion relationship, and a smaller value anywhere else.

Figure 12 shows a plot of the dispersion relationship in equation 14. There are two branches, for $+\sqrt{gk}$ and $-\sqrt{gk}$. If the waves in the image data are animating with that dispersion relation, then the PSD will show most of its strength along these two branches, although the actual magnitude of the PSD should vary from point to point. Figure 13 shows the actual 3D PSD from the image data. There are two clear branches along the dispersion relationship we have discussed, with no apparent modification by shallow water affects. There is also a third branch that is approximately a straight line lying between the first two. Examination of the video shows that this branch comes from a surfactant layer floating on the water in part of the video frame, and moving with a constant speed. Excluding the surface layer, this data clearly demonstrates the validity of the dispersion relationship, and demonstrates the usefulness of the statistical, Fourier Transform oriented model of ocean waves.

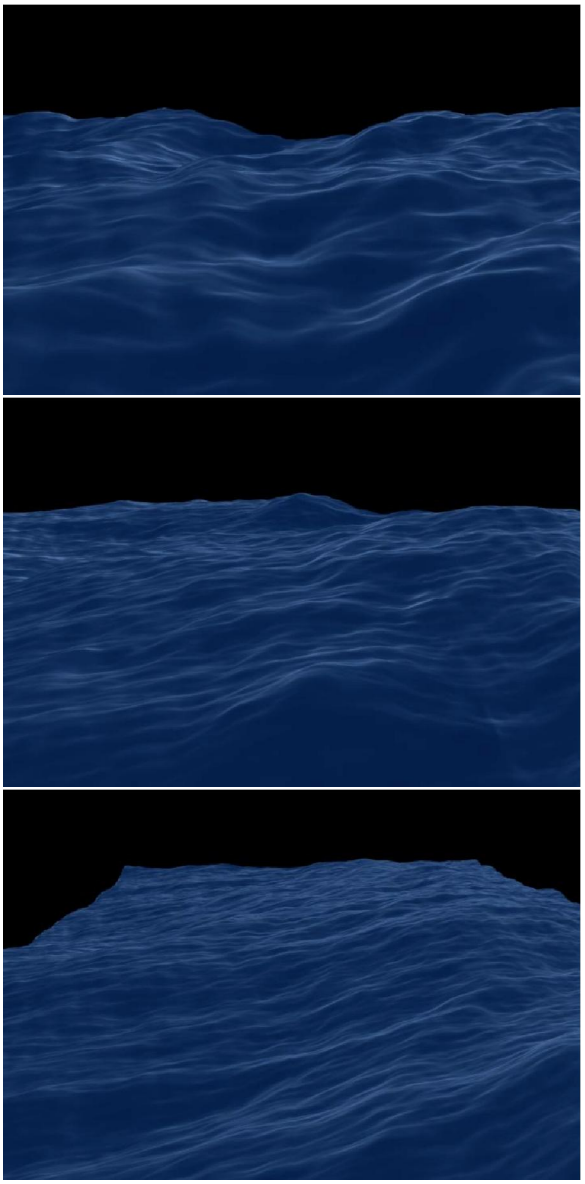


Figure 9: Rendering of waves with (top) a fairly low number of waves (facet size = 10 cm), with little detail; (middle) a reasonably good number of waves (facet size = 5 cm); (bottom) a high number of waves with the most detail (facet size = 2.5 cm).

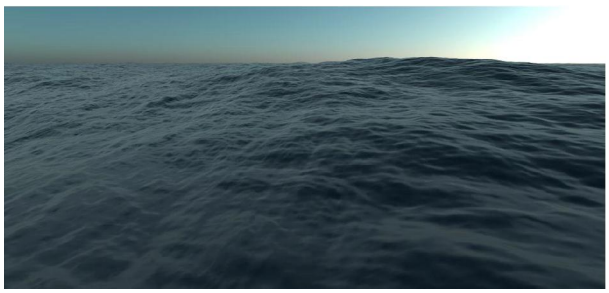


Figure 10: An image of a wave height field rendered in a commercial package with a model atmosphere and sophisticated shading.

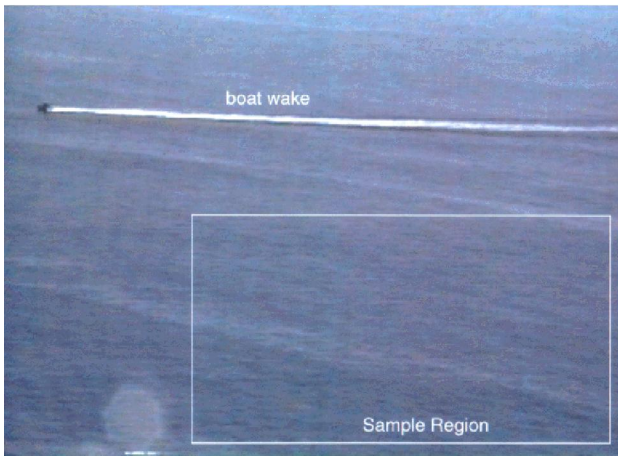


Figure 11: Site at which video data was collected in 1986, near Zuma Beach, California.

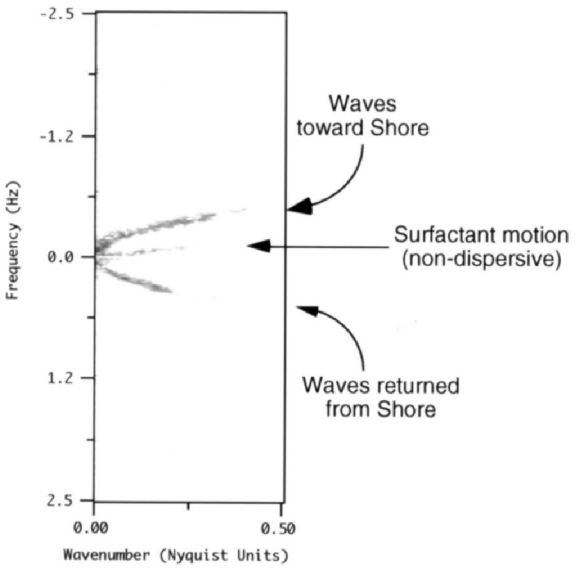


Figure 13: Slice from a 3D Power Spectral Density grayscale plot, from processed video data.

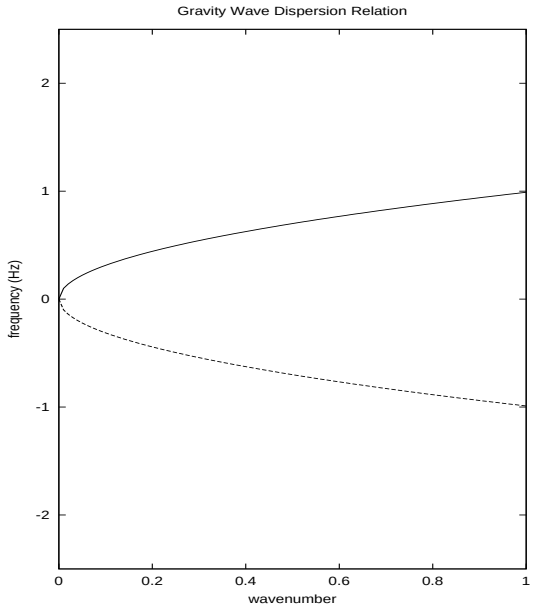


Figure 12: The theoretical dispersion relationship for deep water gravity waves.

3.7 Choppy Waves

We turn briefly in this section to the subject of creating choppy looking waves. The waves produced by the fft methods presented up to this point have rounded peaks and troughs that give them the appearance of fair-weather conditions. Even in fairly good weather, and particularly in a good wind or storm, the waves are sharply peaked at their tops, and flattened at the bottoms. The extent of this chopping of the wave profile depends on the environmental conditions, the wavelengths and heights of the waves. Waves that are sufficiently high (e.g. with a slope greater than about 1/6) eventually break at the top, generating a new set of physical phenonema in foam, splash, bubbles, and spray.

The starting point for this method is the fundamental fluid dynamic equations of motion for the surface. These equations are expressed in terms of two dynamical fields: the surface elevation and the velocity potential on the surface, and derive from the Navier-Stokes description of the fluid throughout the volume of the water and air, including both above and below the interface. Creamer et al[12] set out to apply a mathematical approach called the ”Lie Transform technique” to generate a sequence of ”canonical transformations” of the elevation and velocity potential. The benefit of this complex mathematical procedure is to convert the elevation and velocity potential into new dynamical fields that have a simpler dynamics. The transformed case is in fact just the simple ocean height field we have been discussing, including evolution with the same dispersion relation we have been using in this paper. Starting from there, the inverse Lie Transform in principle converts our phenomenological solution into a dynamically more accurate one. However, the Lie Transform is difficult to manipulate in 3 dimensions, while in two dimensions exact results have been obtained. Based on those exact results in two dimensions, an extrapolation for the form of the 3D solution has been proposed: a horizontal displacement of the waves, with the displacement locally varying with the waves.

In the fft representation, the 2D displacement vector field is com-

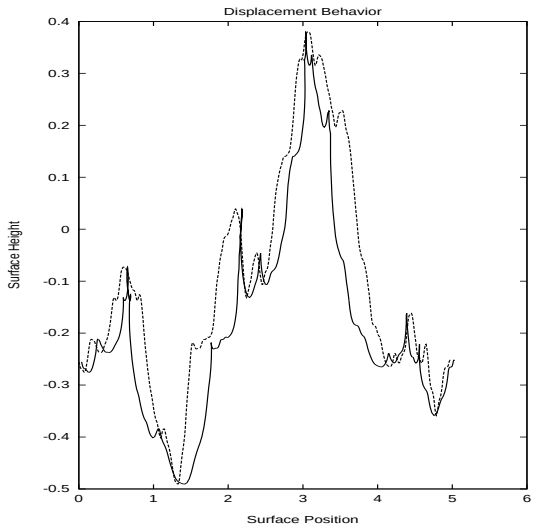


Figure 14: A comparison of a wave height profile with and without the displacement. The dashed curve is the wave height produced by the fft representation. The solid curve is the height field displaced using equation 29.

puted using the Fourier amplitudes of the height field, as

$$\mathbf{D}(\mathbf{x},t) = \sum_{\mathbf{k}} -i\frac{\mathbf{k}}{k}\,\tilde{h}(\mathbf{k},t)\,\exp\left(i\mathbf{k}\cdot\mathbf{x}\right) \tag{29}$$

Using this vector field, the horizontal position of a grid point of the surface is now $\mathbf{x} + \lambda\mathbf{D}(\mathbf{x},t)$, with height $h(\mathbf{x})$ as before. The parameter λ is not part of the original conjecture, but is a convenient method of scaling the importance of the displacement vector. This conjectured solution does not alter the wave heights directly, but instead warps the horizontal positions of the surface points in a way that depends on the spatial structure of the height field. The particular form of this warping however, actually sharpens peaks in the height field and broadens valleys, which is the kind of nonlinear behavior that should make the fft representation more realistic. Figure 14 shows a profile of the wave height along one direction in a simulated surface. This clearly shows that the "displacement conjecture" can dramatically alter the surface.

The displacment form of the this solution is similar to the algorithm for building Gerstner waves [8] discussed in section 3. In that case however, the displacement behavior, applied to sinusoid shapes, was the principle method of characterizing the water surface structure, and here it is a modifier to an already useful wave height representation.

Figure 15 illustrates how these choppy waves behave as they evolve. The tops of waves form a sharp cusp, which rounds out and disappears shortly afterward.

One "problem" with this method of generating choppy waves can be seen in figure 14. Near the tops of some of the waves, the surface actally passes through itself and inverts, so that the outward normal to the surface points inward. This is because the amplitudes of the wave components can be large enough to create large displacements that overlap. This is easily defeated simply by reducing the magnitude of the scaling factor λ. For the purposes of computer graphics, this might actually be a useful effect to signal the production of spray, foam and/or breaking waves. We will not discuss here how to carry out such an extension, except to note that in order

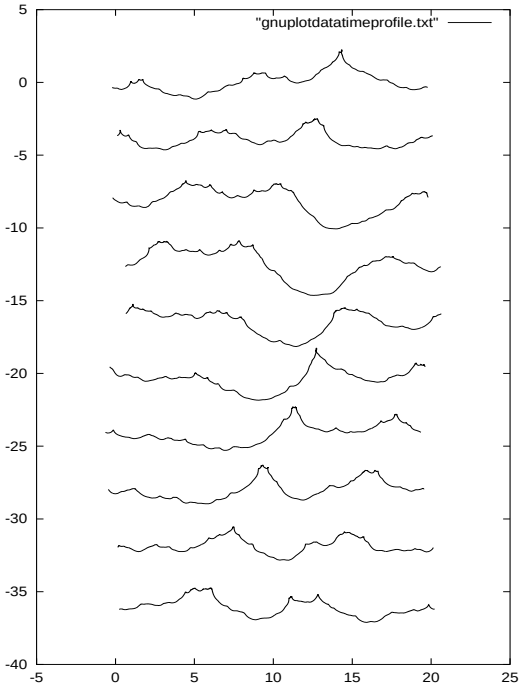


Figure 15: A time sequence of profiles of a wave surface. From top to bottom, the time between profiles is 0.5 seconds.

to use this region of overlap, a simple and quick test is needed for deciding that the effect is taking place. Fortunately, there is such a simple test in the form of the Jacobian of the transformation from $\mathbf{x}$ to $\mathbf{x} + \lambda\mathbf{D}(\mathbf{x},t)$. The Jacobian is a measure of the uniqueness of the transformation. When the displacement is zero, the Jacobian is 1. When there is displacement, the Jacobian has the form

$$J(\mathbf{x}) = J_{xx}J_{yy} - J_{xy}J_{yx}\ , \tag{30}$$

with individual terms

$$\begin{aligned} J_{xx}(\mathbf{x}) &= 1 + \lambda\frac{\partial D_x(\mathbf{x})}{\partial x} \\ J_{yy}(\mathbf{x}) &= 1 + \lambda\frac{\partial D_y(\mathbf{x})}{\partial y} \\ J_{yx}(\mathbf{x}) &= \lambda\frac{\partial D_y(\mathbf{x})}{\partial x} \\ J_{xy}(\mathbf{x}) &= \lambda\frac{\partial D_x(\mathbf{x})}{\partial y} = J_{yx} \end{aligned}$$

and $\mathbf{D} = (D_x, D_y)$. The Jacobian signals the presence of the overlapping wave bacause its value is less than zero in the overlap region. For example, figure 16 plots a profile of a basic wave without displacement, the wave with displacement, and the value of J for the choppy wave (labeled "Folding Map"). The "folds" or overlaps in the choppy surface are clearly visible, and align with the regions in which $J < 0$. With this information, it should be relatively easy to extract the overlapping region and use it for other purposes if desired.

But there is more that can be learned from these folded waves from a closer examination of this folding criterion. The Jacobian

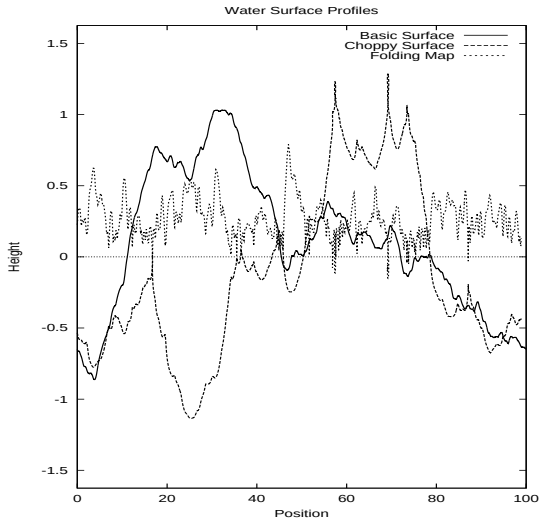


Figure 16: Wave height profile with and without the displacement. Also plotted is the Jacobian map for choppy wave profile.

derives from a 2×2 matrix which measures the local uniqueness of the choppy wave map $\mathbf{x} \rightarrow \mathbf{x} + \lambda \mathbf{D}$. This matrix can in general be written in terms of eigenvectors and eigenvalues as:

$$J_{ab} = J_{-}\hat{e}_a^{-}\hat{e}_b^{-} + J_{+}\hat{e}_a^{+}\hat{e}_b^{+}\ , \quad (a, b = x, y) \tag{31}$$

where J_{-} and J_{+} are the two eigenvalues of the matrix, ordered so that $J_{-} \leq J_{+}$. The corresponding orthonormal eigenvectors are $\hat{e}^{-}$ and $\hat{e}^{+}$ respectively. From this expression, the Jacobian is just $J = J_{-}J_{+}$.

The criterion for folding that $J < 0$ means that $J_{-} < 0$ and $J_{+} > 0$. So the minimum eigenvalue is the actual signal of the onset of folding. Further, the eigenvector $\hat{e}^{-}$ points in the horizontal direction in which the folding is taking place. So, the prescription now is to watch the minimum eigenvalue for when it becomes negative, and the alignment of the folded wave is parallel to the minimum eigenvector.

We can illustrate this phenomenon with an example. Figures 17 and 18 show two images of an ocean surface, one without choppy waves, and the other with the choppy waves strongly applied. These two surfaces are identical except for the choppy wave algorithm. Figure 19 shows the wave profiles of both surfaces along a slice through the surfaces. Finally, the profile of the choppy wave is plotted together with the value of the minimum eigenvalue in figure 20, showing the clear connection between folding and the negative value of J_{-}.

Incidentally, computing the eigenvalues and eigenvectors of this matrix is fast because they have analytic expressions as

$$J_{\pm} = \frac{1}{2}(J_{xx} + J_{yy}) \pm \frac{1}{2}\left\{(J_{xx} - J_{yy})^2 + 4J_{xy}^2\right\}^{1/2} \tag{32}$$

for the eigenvalues and

$$\hat{e}^{\pm} = \frac{(1, q_{\pm})}{\sqrt{1 + q_{\pm}^2}} \tag{33}$$

with

$$q_{\pm} = \frac{J_{\pm} - J_{xx}}{J_{xy}} \tag{34}$$

for the eigenvectors.

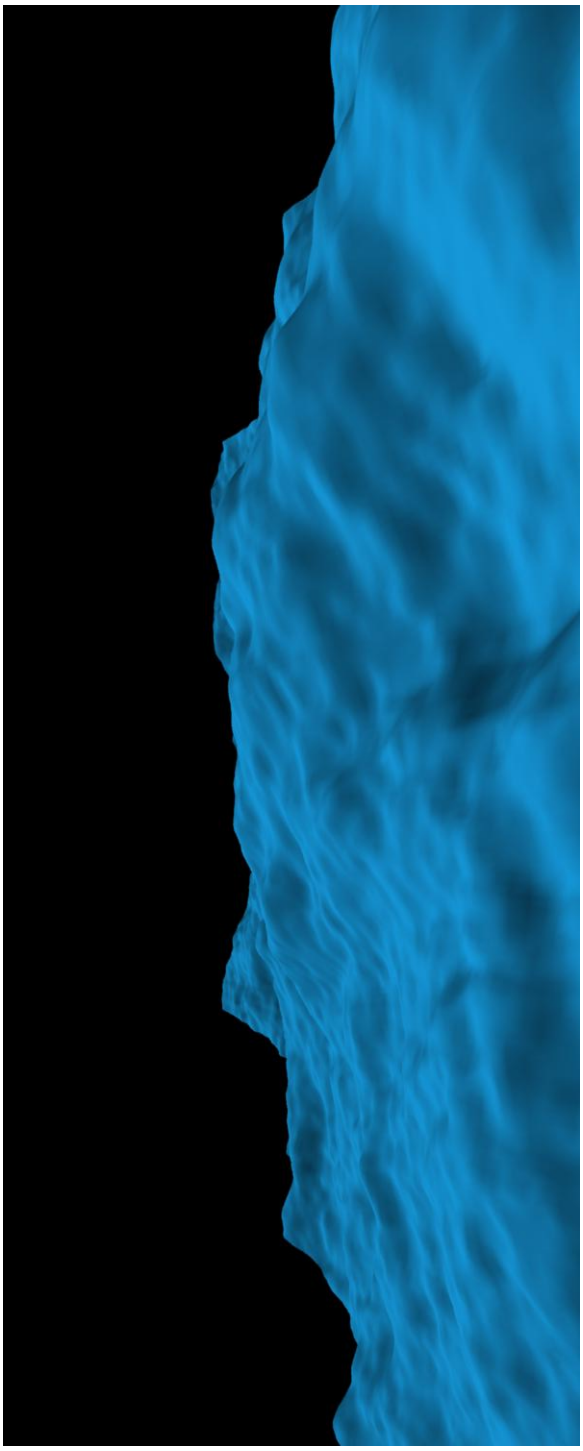


Figure 17: Simulated wave surface without the choppy algorithm applied. Rendered in BMRT with a generic plastic shader.

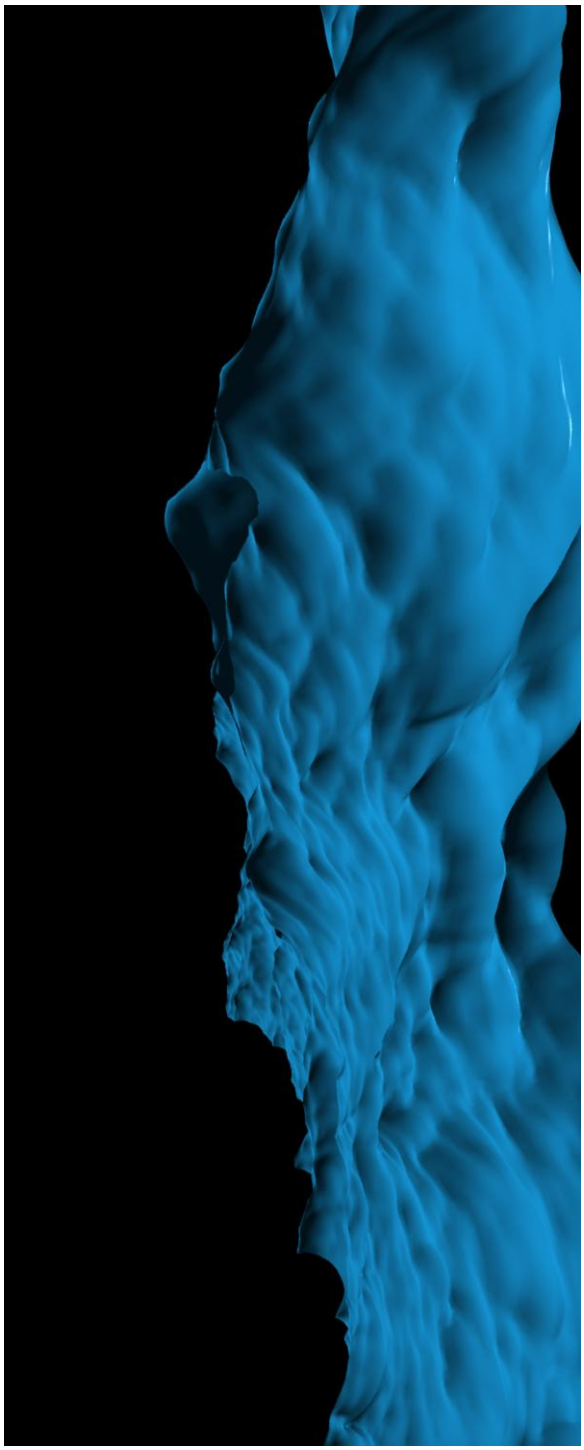


Figure 18: Same wave surface with strong chop applied. Rendered in BMRT with a generic plastic shader.

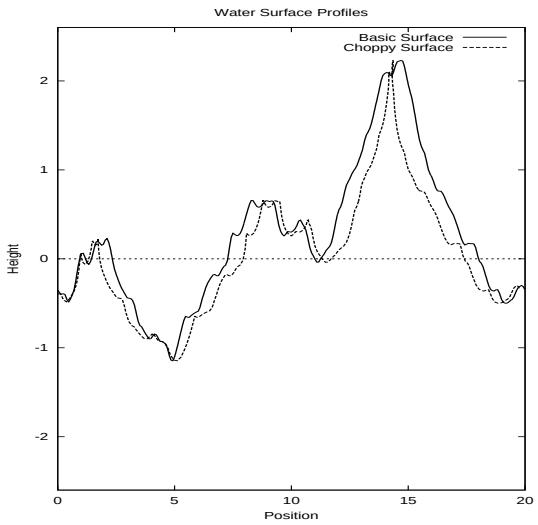


Figure 19: Profiles of the two surfaces, showing the effect of the choppy mapping.

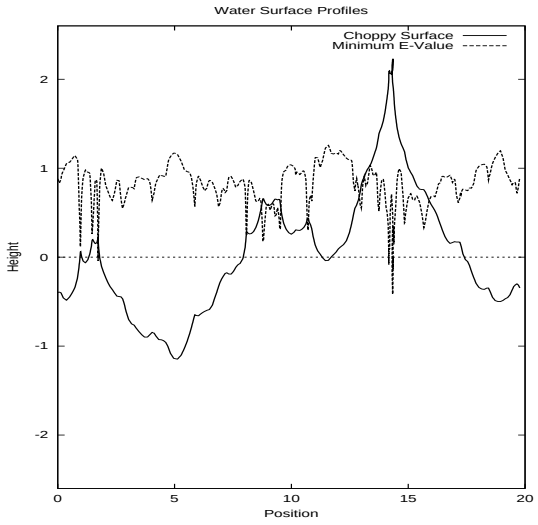


Figure 20: Plot of the choppy surface profile and the minimum eigenvalue. The locations of folds of the surface are clearly the same as where the eigenvalue is negative.

4 Surface Wave Optics

The optical behavior of the ocean surface is fairly well understood, at least for the kinds of quiescent wave structure that we consider in these notes. Fundamentally, the ocean surface is a near perfect specular reflector, with well-understand relectivity and transmissity functions. In this section these properties are summarized, and combined into a simple shader for Renderman. There are circumstances when the surface does not appear to be a specular reflector. In particular, direct sunlight reflected from waves at a large distance from the camera appear to be spread out and made diffuse. This is due to the collection of waves that are smaller than the camera can resolve at large distances. The mechanism is somewhat similar to the underlying microscopic reflection mechnanisms in solid surfaces that lead to the Torrance-Sparrow model of BRDFs. Although the study of glitter patterns in the ocean was pioneered by Cox and Munk many years ago, the first models of this BRDF behavior that I am aware of were developed in the early 1980's. At the end of this section, we introduce the concepts and conditions, state the results, and ignore the in-between analysis and derivation.

Throughout these notes, and particularly in this section, we ignore one optical phenomenon completely: polarization. Polarization effects can be strong at a boundary interface like a water surface. However, since most of computer graphics under consideration ignores polarization, we will continue in that long tradition. Of course, interested readers can find literature on polarization effects at the air-water interface.

4.1 Specular Reflection and Transmission

Rays of light incident from above or below at the air-water interface are split into two components: a transmitted ray continuing through the interface at a refracted angle, and a reflected ray. The intensity of each of these two rays is diminished by reflectivity and transmissivity coefficients. Here we discussed the directions of the two outgoing rays. In the next subsection the coefficients are discussed.

4.1.1 Reflection

As is well known, in a perfect specular reflection the reflected ray and the incident ray have the same angle with respect to the surface normal. This is true for all specular reflections (ignoring roughening effects), regardless of the material. We build here a compact expression for the outgoing reflected ray. First, we need to build up some notation and geometric quantities.

The three-dimensional points on the ocean surface can be labelled by the horizontal position $\mathbf{x}$ and the waveheight $h(\mathbf{x}, t)$ as

$$\mathbf{r}(\mathbf{x}, t) = \mathbf{x} + \hat{\mathbf{y}} h(\mathbf{x}, t) \, , \tag{35}$$

where $\hat{\mathbf{y}}$ is the unit vector pointing straight up. At the point $\mathbf{r}$, the normal to the surface is computed directly from the surface slope $\epsilon(\mathbf{x}, t) \equiv \nabla h(\mathbf{x}, t)$ as

$$\hat{\mathbf{n}}_S(\mathbf{x}, t) = \frac{\hat{\mathbf{y}} - \epsilon(\mathbf{x}, t)}{\sqrt{1 + \epsilon^2(\mathbf{x}, t)}} \tag{36}$$

For a ray intersecting the surface at $\mathbf{r}$ from direction $\hat{\mathbf{n}}_i$, the direction of the reflected ray can depend only on the incident direction and the surface normal. Also, as mentioned before, the angle between the surface normal and the reflected ray must be the same as the angle between incident ray and the surface normal. You can verify for yourself that the reflected direction $\hat{\mathbf{n}}_r$ is

$$\hat{\mathbf{n}}_r(\mathbf{x}, t) = \hat{\mathbf{n}}_i - 2\hat{\mathbf{n}}_S(\mathbf{x}, t) \, (\hat{\mathbf{n}}_S(\mathbf{x}, t) \cdot \hat{\mathbf{n}}_i) \ . \tag{37}$$

Note that this expression is valid for incident ray directions on either side of the surface.

4.1.2 Transmission

Unfortunately, the direction of the transmitted ray is not expressed as simply as for the reflected ray. In this case we have two guiding principles: the transmitted direction is dependent only on the surface normal and incident directions, and Snell's Law relating the sines of the angles of the incident and transmitted angles to the indices of refraction of the two materials.

Suppose the incident ray is coming from one of the two media with index of refraction n_i (for air, $n = 1$, for water, $n = 4/3$ approximately), and the transmitted ray is in the medium with index of refraction n_r. For the incident ray at angle θ_i to the normal,

$$\sin\theta_i = \sqrt{1 - (\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S)^2} = |\hat{\mathbf{n}}_i \times \hat{\mathbf{n}}_S| \tag{38}$$

the transmitted ray will be at an angle θ_t with

$$\sin\theta_t = |\hat{\mathbf{n}}_t \times \hat{\mathbf{n}}_S| \ . \tag{39}$$

Snell's Law states that these two angles are related by

$$n_i \sin\theta_i = n_t \sin\theta_t \ . \tag{40}$$

We now have all the pieces needed to derive the direction of transmission. The direction vector can only be a linear combination of $\hat{\mathbf{n}}_i$ and $\hat{\mathbf{n}}_S$. It must satisfy Snell's Law, and it must be a unit vector (by definition). This is adequate to obtain the expression

$$\hat{\mathbf{n}}_t(\mathbf{x}, t) = \frac{n_i}{n_t}\hat{\mathbf{n}}_i + \Gamma(\mathbf{x}, t) \, \hat{\mathbf{n}}_S(\mathbf{x}, t) \tag{41}$$

with the function Γ defined as

$$\begin{aligned} \Gamma(\mathbf{x}, t) \quad &\equiv \quad \frac{n_i}{n_t}\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S(\mathbf{x}, t) \\ &\pm \quad \left\{ 1 - \left(\frac{n_i}{n_t}\right)^2 |\hat{\mathbf{n}}_i \times \hat{\mathbf{n}}_S(\mathbf{x}, t)|^2 \right\}^{1/2} \ . \end{aligned} \tag{42}$$

The plus sign is used in Γ when $\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S < 0$, and the minus sign is used when $\hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_S > 0$.

4.2 Fresnel Reflectivity and Transmissivity

Accompanying the process of reflection and transmission through the interface is a pair of coefficients that describe their efficiency. The reflectivity R and transmissivity T are related by the constraint that no light is lost at the interface. This leads to the relationship

$$R + T = 1 \, . \tag{43}$$

The derivation of the expressions for R and T is based on the electromagnetic theory of dielectrics. We will not carry out the derivations, but merely write down the solution

$$R(\hat{\mathbf{n}}_i, \hat{\mathbf{n}}_r) = \frac{1}{2}\left\{ \frac{\sin^2(\theta_t - \theta_i)}{\sin^2(\theta_t + \theta_i)} + \frac{\tan^2(\theta_t - \theta_i)}{\tan^2(\theta_t + \theta_i)} \right\} \tag{44}$$

Figure 21 is a plot of the reflectivity for rays of light traveling down onto a water surface as a function of the angle of incidence to the surface. The plot extends from a grazing angle of 0 degrees to perpendicular incidence at 90 degrees. As should be clear, variation of the reflectivity across an image is an important source of the "texture" or feel of water. Notice that reflectivity is a function of the angle of incidence relative to the wave normal, which in turn is directly related to the slope of the surface. So we can expect that a strong contributor to the texture of water surface is the pattern of slope, while variation of the wave height serves primarily as a wave hiding mechanism. This is the quantitative explanation of why the

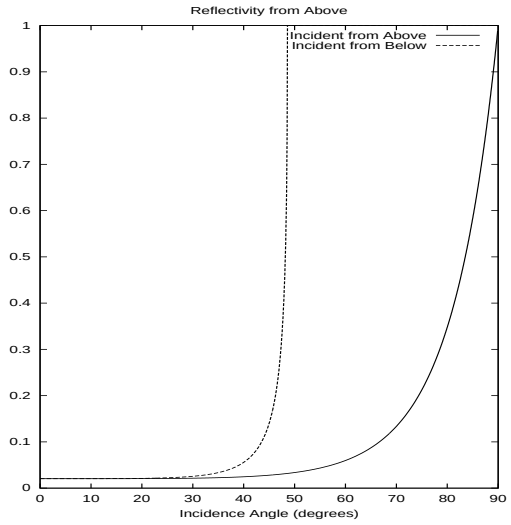


Figure 21: Reflectivity for light coming from the air down to the water surface, as a function of the angle of incidence of the light.

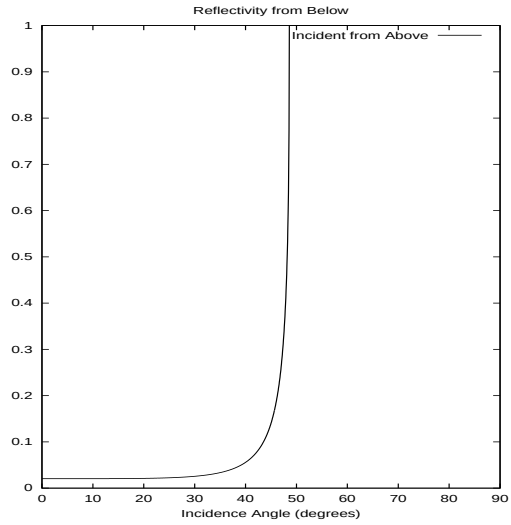


Figure 22: Reflectivity for light coming from below the water surface, as a function of the angle of incidence of the light.

surface slope more closely resembles rendered water than the wave height does, as we saw in the previous section when discussing figure 7.

When the incident ray comes from below the water surface, there are important differences in the reflectivity and transmissivity. Figure 22 shows the reflectivity as a function of incidence angle again, but this time for incident light from below. In this case, the reflectivity reaches unity at a fairly large angle, near 41 degrees. At incidence angles below that, the reflectivity is one and so there is no transmission of light through the interface. This phenomenon is *total internal reflection*, and can be seen just by swimming around in a pool. The angle at which total internal reflection begins is called Brewster's angle, and is given by, from Snell's Law,

$$\sin\theta_i^B = \frac{n_t}{n_i} = 0.75 \tag{45}$$

or $\theta_i^B = 48.6\,\mathrm{deg}$. In our plots, this angle is $90 - \theta_i^B = 41.1\,\mathrm{deg}$.

4.3 Building a Shader for Renderman

From the discussion so far, one of the most important features a rendering must emulate is the textures of the surface due to the strong slope-dependence of reflectivity and transmissivity. In this section we construct a simple Renderman-compliant shader using just these features. Readers who have experience with shaders will know how to extend this one immediately.

The shader exploits that fact that the Renderman interface already provides a built-in Fresnel quantity calculator, which provides R, T, $\hat{\mathbf{n}}_r$, and $\hat{\mathbf{n}}_t$ using the surface normal, incident direction vector, and index of refraction. The shader for the air-to-water case is as follows:

```
surface watercolorshader(
    color upwelling = color(0, 0.2, 0.3);
    color sky       = color(0.69,0.84,1);
    color air       = color(0.1,0.1,0.1);
    float nSnell    = 1.34;
    float Kdiffuse  = 0.91;
    string envmap   = "";
                        )
{
  float reflectivity;
  vector nI = normalize(I);
  vector nN = normalize(Ng);
  float costhetai = abs(nI . nN);
  float thetai = acos(costhetai);
  float sinthetat = sin(thetai)/nSnell;
  float thetat = asin(sinthetat);
  if(thetai == 0.0)
  {
    reflectivity = (nSnell - 1)/(nSnell + 1);
    reflectivity = reflectivity * reflectivity;
  }
  else
  {
    float fs = sin(thetat - thetai)
             / sin(thetat + thetai);
    float ts = tan(thetat - thetai)
             / tan(thetat + thetai);
    reflectivity = 0.5 * ( fs*fs + ts*ts );
  }
  vector dPE = P-E;
  float dist = length(dPE) * Kdiffuse;
  dist = exp(-dist);

  if(envmap != "")
  {
    sky = color environment(envmap, nN);
  }
  Ci =  dist * ( reflectivity * sky
                + (1-reflectivity) * upwelling )
     +  (1-dist)* air;
}
```

There are two contributions to the color: light coming downward onto the surface with the default color of the sky, and light coming upward from the depths with a default color. This second term will be discussed in the next section. It is important for incidence angles that are high in the sky, because the reflectivity is low and transmissivity is high.

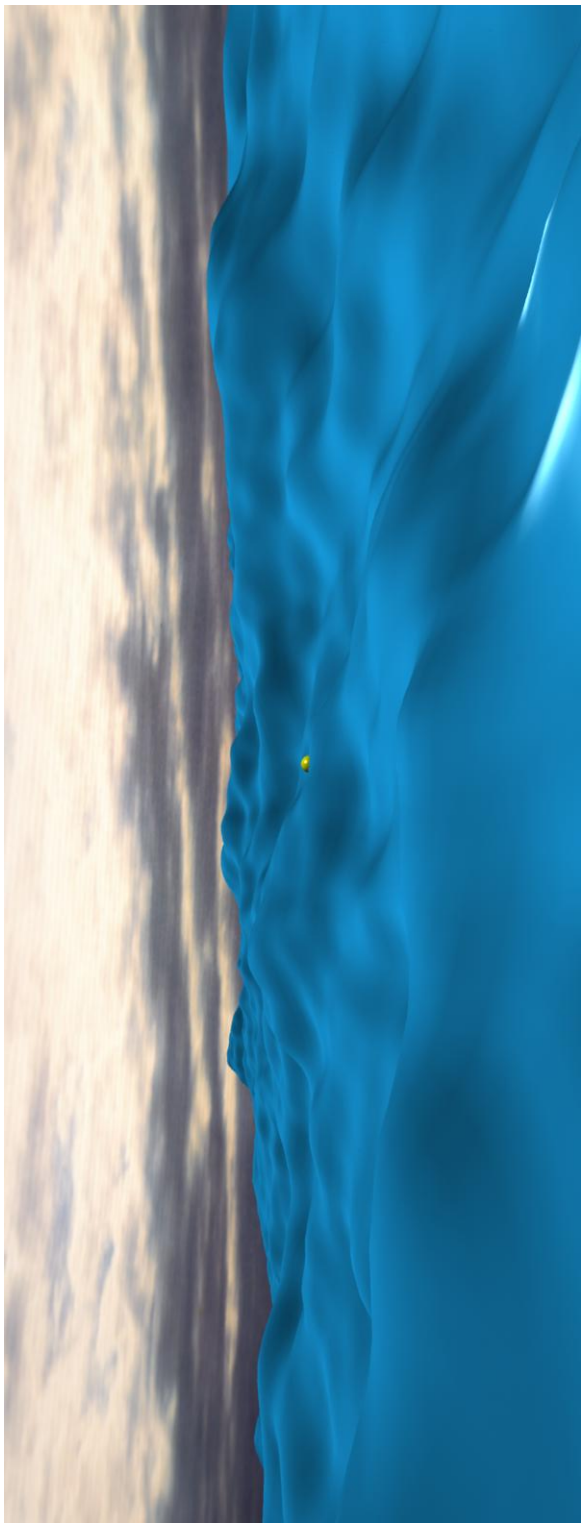


Figure 23: Simulated water surface with a generic plastic surface shader. Rendered with BMRT.

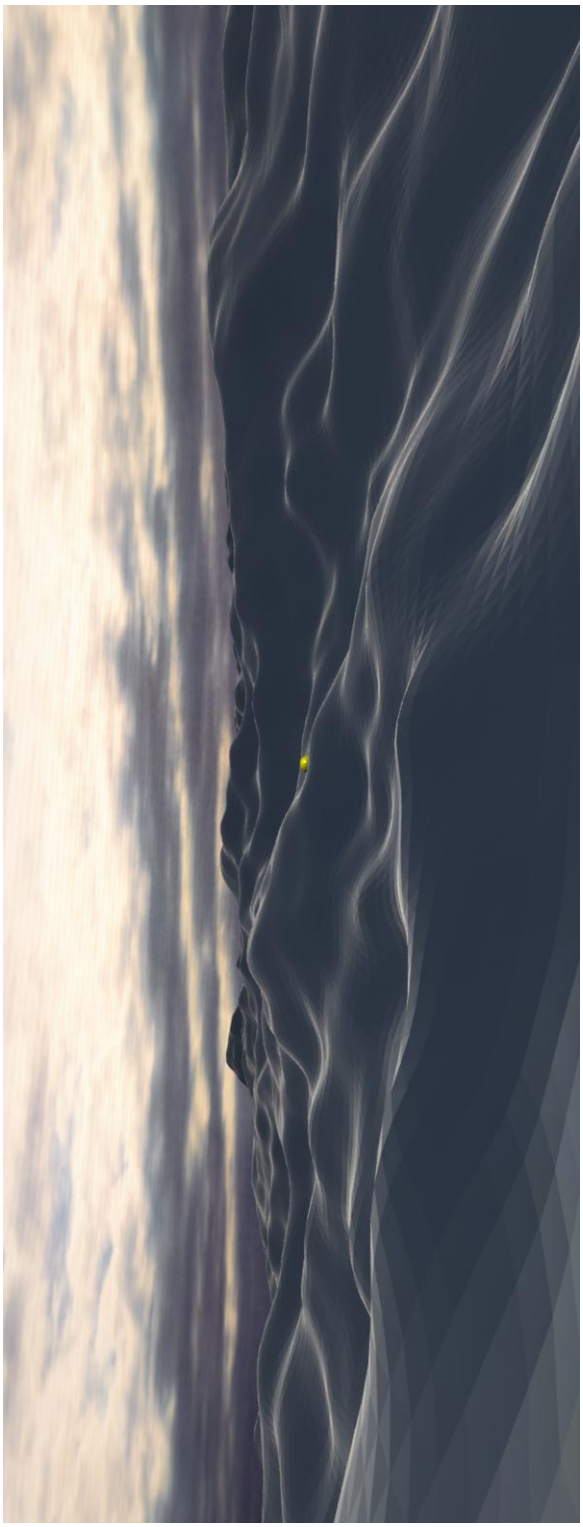


Figure 24: Simulated water surface with a realistic surface shader. Rendered with BMRT.

This shader was used to render the image in figure 24 using the BMRT raytrace renderer. For reference, the exact same image has been rendered in 23 with a generic plastic shader. Note that the realistic water shader tends to highlight the tops of the the waves, where the angle of incidence is nearly 90 degrees grazing and the reflectivity is high, while the sides of the waves are dark, where angle of incidence is nearly 0 that the reflectivity is low.

5 Water Volume Effects

The previous section was devoted to a discussion of the optical behavior of the surface of the ocean. In this section we focus on the optical behavior of the water volume below the surface. We begin with a discussion of the major optical effects the water volume has on light, followed by an introduction to color models researchers have built to try to connect the ocean color on any given day to underlying biological and physical processes. These models are built upon many years of in-situ measures off of ships and peers. Finally, we discuss two important effects, caustics and sunbeams, that sometimes are hard to grasp, and which produce beautiful images when properly simulated.

5.1 Scattering, Transmission, and Reflection by the Water Volume

In the open ocean, light is both scattering and absorbed by the volume of the water. The sources for these events are of three types: water molecules, living and dead organic matter, and non-organic matter. In most oceans around the world, away from the shore lines, absorption is a fairly even mixture of water molecules and organic matter. Scattering is dominated by organic matter however.

To simulate the processes of volumetric absorption and scattering, there are five quantities that are of interest: absorption coefficient, scattering coefficient, extinction coefficient, diffuse extinction coefficient, and bulk reflectivity. All of these coefficients have units of inverse length, and represent the exponential rate of attenuation of light with distance through the medium. The absorption coefficient a is the rate of absorption of light with distance, the scattering coefficient b is the rate of scattering with length, the extinction coefficient c is the sum of the two previous ones $c = a + b$, and the diffuse extinction coefficient K describes the rate of loss of intensity of light with distance after taking into account both absorption and scattering processes. The connection between K and the other parameters is not completely understood, in part because there are a variety of ways to define K in terms of operational measurements. Different ways change the details of the dependence. However, there is a condition called the *asymptotic* limit at very deep depths in the water, at which all operational definitions of K converge to a single value. This asymptotic value of K has been modeled in a variety of ways. There is a mathematically precise result that the ratio K/c depends only on b/c, the single scatter albedo, and some details of the angular distribution of individual scattering distributions. Figure 25 is an example of a model of K/c for reasonable water conditions. Models have been generated for the color dependence of K, most notably by Jerlov. In 1990, Austin and Petzold performed a revised analysis of spectral models, including new data, to produce refined models of K as a function of color. For typical visible light conditions in the ocean, K ranges in value from 0.03/meter to 0.1/meter. It is generally true that $a < K < c$.

One way to interpret these quantities for a simulation of water volume effects is as follows:

1. A ray of sunlight enters the water with intensity I (after losing some intensity to Fresnel transmission). Along a path underwater of a length s, the intensity at the end of the path is $I \exp(-cs)$, i.e. the ray of direct sunlight is attenuated as fast a possible.

2. Along the path through the water, a fraction of the ray is scattered into a distribution of directions. The strength of the scattering per unit length of the ray is b, so the intensity is proportional to $bI \exp(-cs)$.

3. The light that is scattered out of the ray goes through potentially many more scattering events. It would be nearly impossible to track all of them. However, the sum whole outcome of this process is to attenuate the ray along the path from the original path to the camera as $bI \exp(-cs) \exp(-K s_c)$, where s_c is the distance from the scatter point in the ocean to the camera.

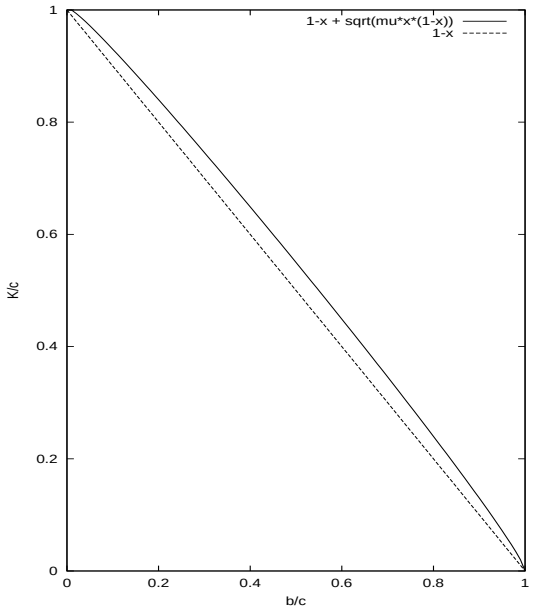


Figure 25: Dependence of the Diffuse Extinction Coefficient on the Single Scatter Albedo, normalized to the extinction.

A fifth quantity of interest for simulation is the bulk reflectivity of the water volume. This is a quantity that is intended to allow us to ignore the details of what is going on, treat the volume as a Lambertian reflector, and compute a value for bulk reflectivity. That number is sensitive to many factors, including wave surface conditions, sun angle, water optical properties, and details of the angular spread. Nevertheless, values of reflectivity around 0.04 seem to agree well with data.

5.2 The Underwater POV: Refracted Skylight, Caustics, and Sunbeams

Now that we have underwater optical properties at hand, we can look at two important phenomena in the ocean: caustics and sunbeams.

5.2.1 Caustics

Caustics, in this context, are a light pattern that is formed on surfaces underwater. Because the water surface is not flat, groups of

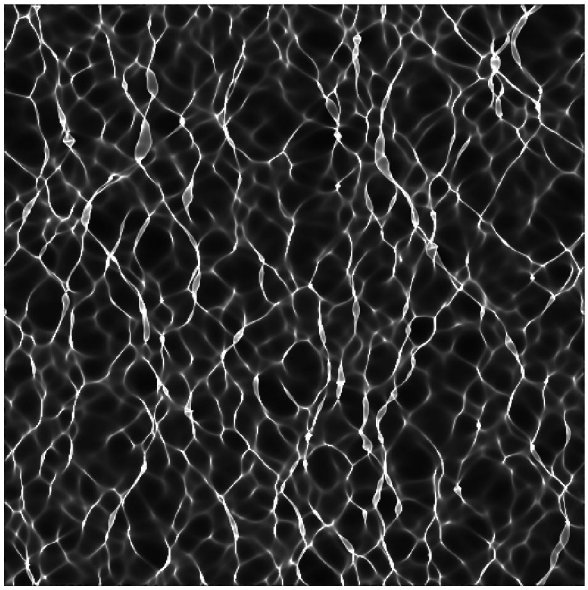


Figure 26: Rendering of a caustic pattern at a shallow depth (5 meters) below the surface.

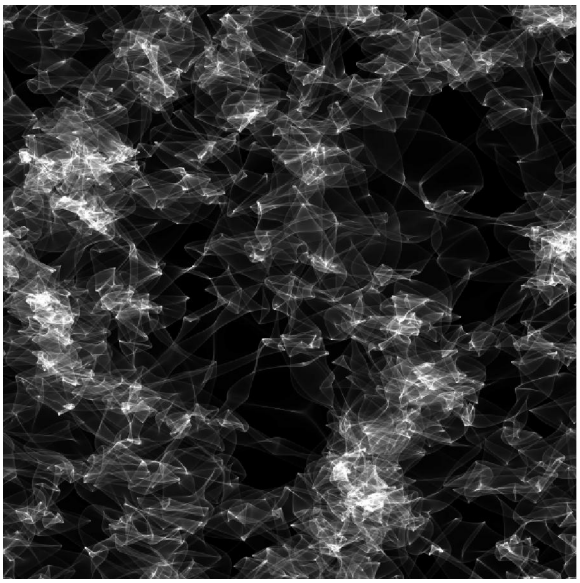


Figure 27: Rendering of a caustic pattern at great depth (100 meters) below the surface.

light rays incident on the surface are either focussed or defocussed. As a result, a point on a fictitious plane some depth below the ocean surface receives direct sunlight from several different positions on the surface. The intensity of light varies due to the depth, original contrast, and other factors. For now, lets write the intensity of the pattern as $I = Ref\ I_0$, with I_0 as the light intensity just above the water surface. The quantity Ref is the scaling factor that varies with position on the fictitious plane due to focussing and defocussing of waves, and is called a *caustic pattern*. Figure 26 shows an example of the caustic pattern Ref. Notice that the caustic pattern exhibits filaments and ring-like structure. At a very deep depth, the caustic pattern is even more striking, as shown in figure 27.

One of the important properties of underwater light that produce caustic patterns is conservation of flux. This is actually a simple idea: suppose a small area on the ocean surface has sunlight passing through it into the water, with intensity I at the surface. As we map that area to greater depths, the amount of area within it grows or shrinks, but most likely grows depending on whether the area is focussed or defocussed. The intensity at any depth within the water is proportional to inverse of the area of the projected region. Another way of saying this is that if a bundle of light rays diverges, their intensities are reduced to keep the product of intensity time area fixed.

Simulated caustic patterns can actually be compared (roughly) with real-world data. In a series of papers published throughout the 1970's, 1980's, and into the 1990's, Dera and others collected high-speed time series of light intensity[17]. As part of this data collection and analysis project, the data was used to generate a probability distribution function (PDF) for the light intensity. Figure 28 shows two PDFs taken from one of Dera's papers. The two PDF's were collected for different surface roughness conditions: rougher water tended to suppress more of the high magnitude fluctuations in intensity.

Figure 29 shows the pdf at two depths from a simulation of the ocean surface. These two sets do not match Dera's measurements because of many factors, but most importantly because we have not simulated the environmental conditions and instrumentation in Dera's experiments. Nevertheless, the similarity of figure 29 with

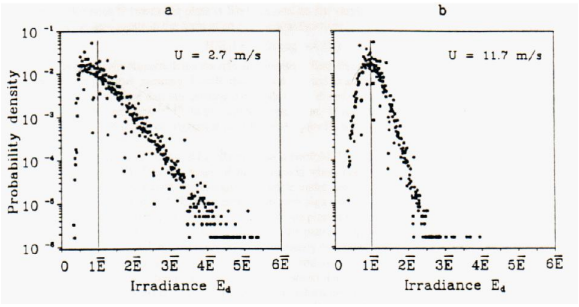


Figure 28: PDF's as measured by Dera in reference [17].

Dera's data is an encouraging point of information for the realism of the simulation.

5.2.2 Godrays

Underwater sunbeams, also called godrays, have a very similar origin to caustics. Direct sunlight passes into the water volume, focussed and defocussed at different points across the surface. As the rays of light pass down through the volume, some of the light is scattered in other directions, and a fraction arrives at the camera. The accumulated pattern of scattered light apparent to the camera are the godrays. So, while caustics are the pattern of direct sunlight that penetrates down to the floor of a water volume, sunbeams are scattered light coming from those shafts of direct sunlight in the water volume. Figure 30 demonstrates sunbeams as seen by a camera looking up at it.

References

[1] Jeff Odien, "On the Waterfront", Cinefex, No. 64, p 96, (1995)

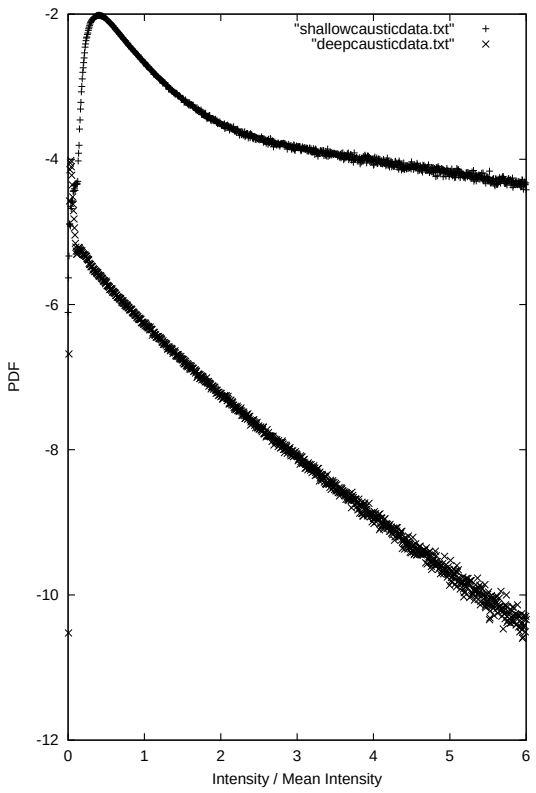


Figure 29: Computed Probability Density Function for light intensity fluctuations in caustics. (upper curve) shallow depth of 2 meters; (lower curve) deep depth of 10 meters.

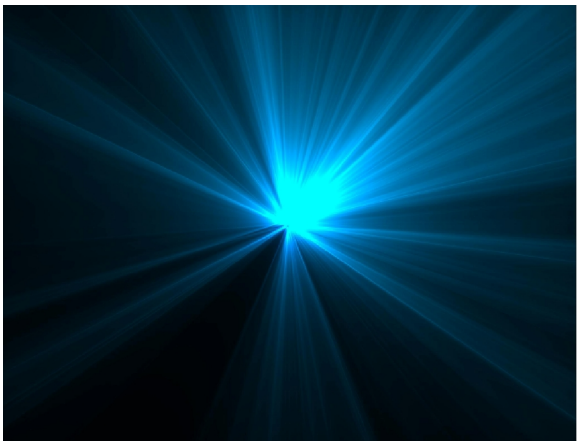


Figure 30: Rendering of sunbeams, or godrays, as seen looking straight up at the light source.

[2] Ted Elrick, “Elemental Images”, Cinefex, No. 70, p 114, (1997)

[3] Kevin H. Martin, “Close Contact”, Cinefex, No. 71, p 114, (1997)

[4] Don Shay, “Ship of Dreams”, Cinefex, No. 72, p 82, (1997)

[5] Kevin H. Martin, “Virus: Building a Better Borg”, Cinefex, No. 76, p 55, (1999)

[6] Simon Premože and Michael Ashikhmin, “Rendering Natural Waters,” Eighth Pacific Conference on Computer Graphics and Applications, October 2000.

[7] Gary A. Mastin, Peter A. Watterger, and John F. Mareda, “Fourier Synthesis of Ocean Scenes”, IEEE CG&A, March 1987, p 16-23.

[8] Alain Fournier and William T. Reeves, “A Simple Model of Ocean Waves”, Computer Graphics, Vol. 20, No. 4, 1986, p 75-84.

[9] Darwyn Peachey, “Modeling Waves and Surf”, Computer Graphics, Vol. 20, No. 4, 1986, p 65-74.

[10] Blair Kinsman, *Wind Waves, Their Generation and Propagation on the Ocean Surface*, Dover Publications, 1984.

[11] S. Gran, *A Course in Ocean Engineering*, *Developments in Marine Technology No. 8*, Elsevier Science Publishers B.V. 1992. See also `http://www.dnv.no/ocean/bk/grand.htm`

[12] Dennis B. Creamer, Frank Henyey, Roy Schult, and Jon Wright, “Improved Linear Representation of Ocean Surface Waves.” J. Fluid Mech, **205**, pp. 135-161, (1989).

[13] Seibert Q. Duntley, “Light in the Sea,” J. Opt. Soc. Am., **53**,2, pg 214-233, 1963.

[14] Curtis D. Mobley, *Light and Water: Radiative Transfer in Natural Waters*, Academic Press, 1994.

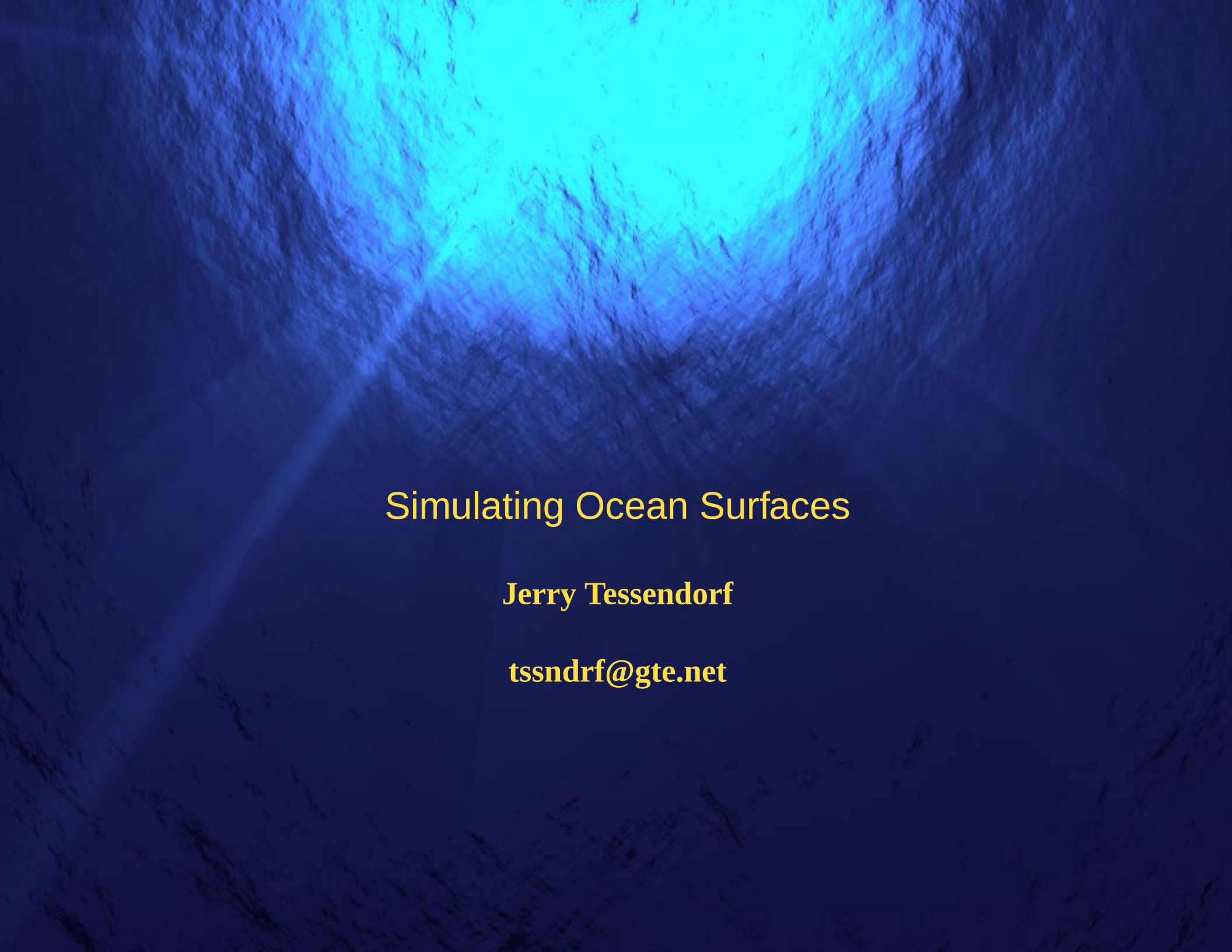
[15] *Selected Papers on Multiple Scattering in Plane-Parallel Atmospheres and Oceans: Methods*, ed. by George W. Kattawar, SPIE Milestones Series, **MS 42**, SPIE Opt. Eng. Press., 1991.

[16] R.W. Austin and T. Petzold, “Spectral Dependence of the Diffuse Attenuation Coefficient of Light in Ocean Waters: A Reexamination Using New Data,” *Ocean Optics X*, Richard W. Spinrad, ed., SPIE **1302**, 1990.

[17] Jerzy Dera, Slawomir Sagan, Dariusz Stramski, “Focusing of Sunlight by Sea Surface Waves: New Measurement Results from the Black Sea,” *Ocean Optics XI*, SPIE Proceedings, 1992.

Contents

- Where we stand
- objectives
- Examples in movies of cg water
- Navier Stokes, potential flow, and approximations
- FFT solution
- Oceanography
- Random surface generation
- High resoluton example
- Video Experiment
- Continuous Loops
- Hamiltonian approach
- Choppy waves from the FFT solution
- Spray Algorithm
- References

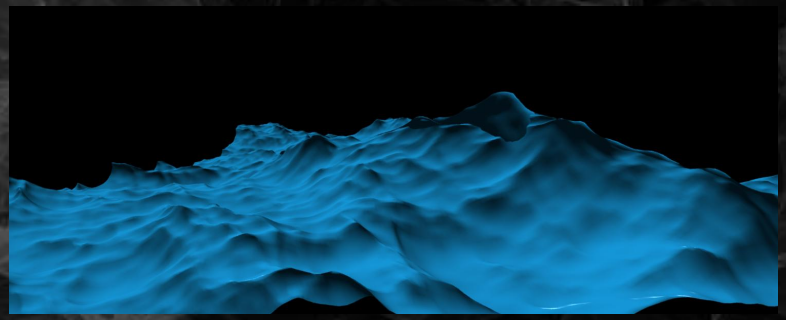


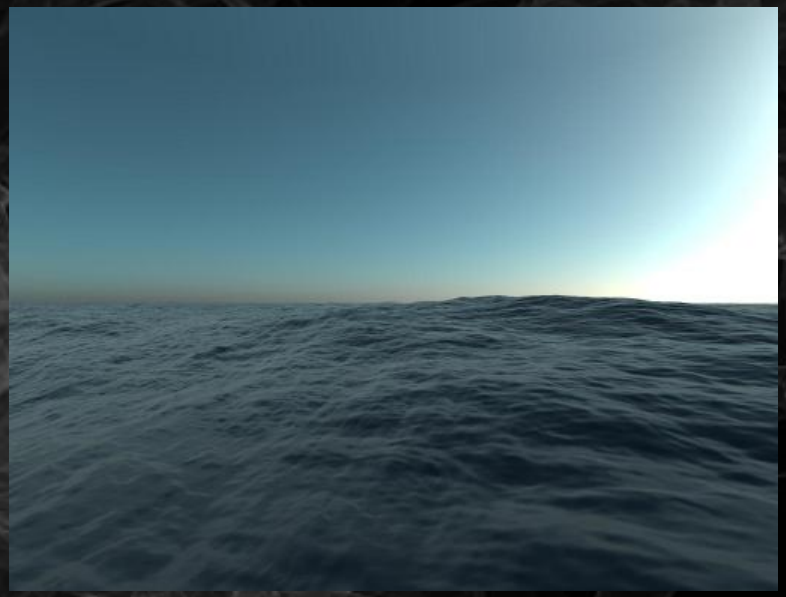
Simulating Ocean Surfaces

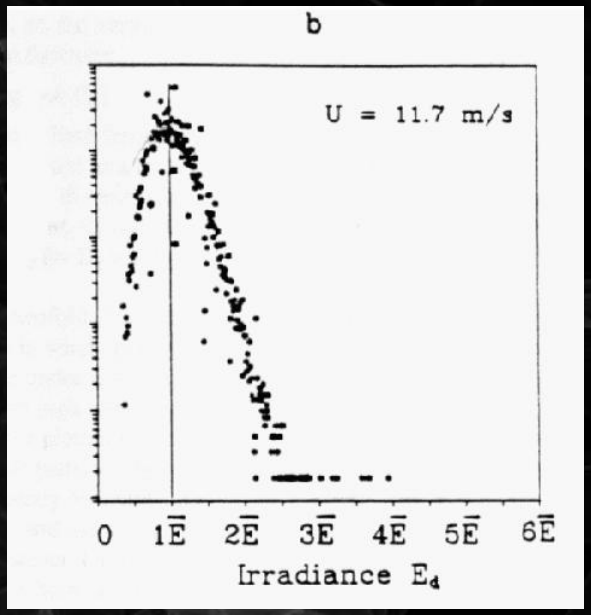
Jerry Tessendorf

tssndrf@gte.net

Objectives







- *Oceanography concepts*
- *Random wave math*
- *Hints for realistic look*
- *Advanced things*

$$h(x,z,t) = \int_{-\infty}^{\infty} dk_x\, dk_z\, \tilde{h}(\mathbf{k},t) \exp\{i(k_x x + k_z z)\}$$

$$\tilde{h}(\mathbf{k},t) = \tilde{h}_0(\mathbf{k}) \exp\{-i\omega_0(\mathbf{k})t\} + \tilde{h}_0^*(-\mathbf{k}) \exp\{i\omega_0(\mathbf{k})t\}$$

Waterworld	13th Warrior	Fifth Element
Truman Show	Titanic	Double Jeopardy
Hard Rain	Deep Blue Sea	Devil's Advocate
Contact	Virus	20k Leagues Under the Sea
Cast Away	World Is Not Enough	13 Days

Navier-Stokes Fluid Dynamics

Force Equation

$$\frac{\partial \mathbf{u}(\mathbf{x},t)}{\partial t} + \mathbf{u}(\mathbf{x},t) \cdot \nabla \mathbf{u}(\mathbf{x},t) + \nabla p(\mathbf{x},t)/\rho = -g\hat{\mathbf{y}} + \mathbf{F}$$

Mass Conservation

$$\nabla \cdot \mathbf{u}(\mathbf{x},t) = 0$$

Solve for functions of space and time:

- 3 velocity components
- pressure p
- density ρ distribution

Boundary conditions are important constraints

Very hard - Many scientitic careers built on this

Potential Flow

Special Substitution $\mathbf{u} = \nabla\phi(\mathbf{x}, t)$

Transforms the equations into

$$\frac{\partial\phi(\mathbf{x},t)}{\partial t} + \frac{1}{2}|\nabla\phi(\mathbf{x},t)|^2 + \frac{p(\mathbf{x},t)}{\rho} + g\mathbf{x}\cdot\hat{\mathbf{y}} = 0$$

$$\nabla^2\phi(\mathbf{x},t) = 0$$

This problem is MUCH simpler computationally and mathematically.

Free Surface Potential Flow

In the water volume, mass conservation is enforced via

$$\phi(\mathbf{x}) = \int_{\partial V} dA' \left\{ \frac{\partial \phi(\mathbf{x}')}{\partial n'} G(\mathbf{x}, \mathbf{x}') - \phi(\mathbf{x}') \frac{\partial G(\mathbf{x}, \mathbf{x}')}{\partial n'} \right\}$$

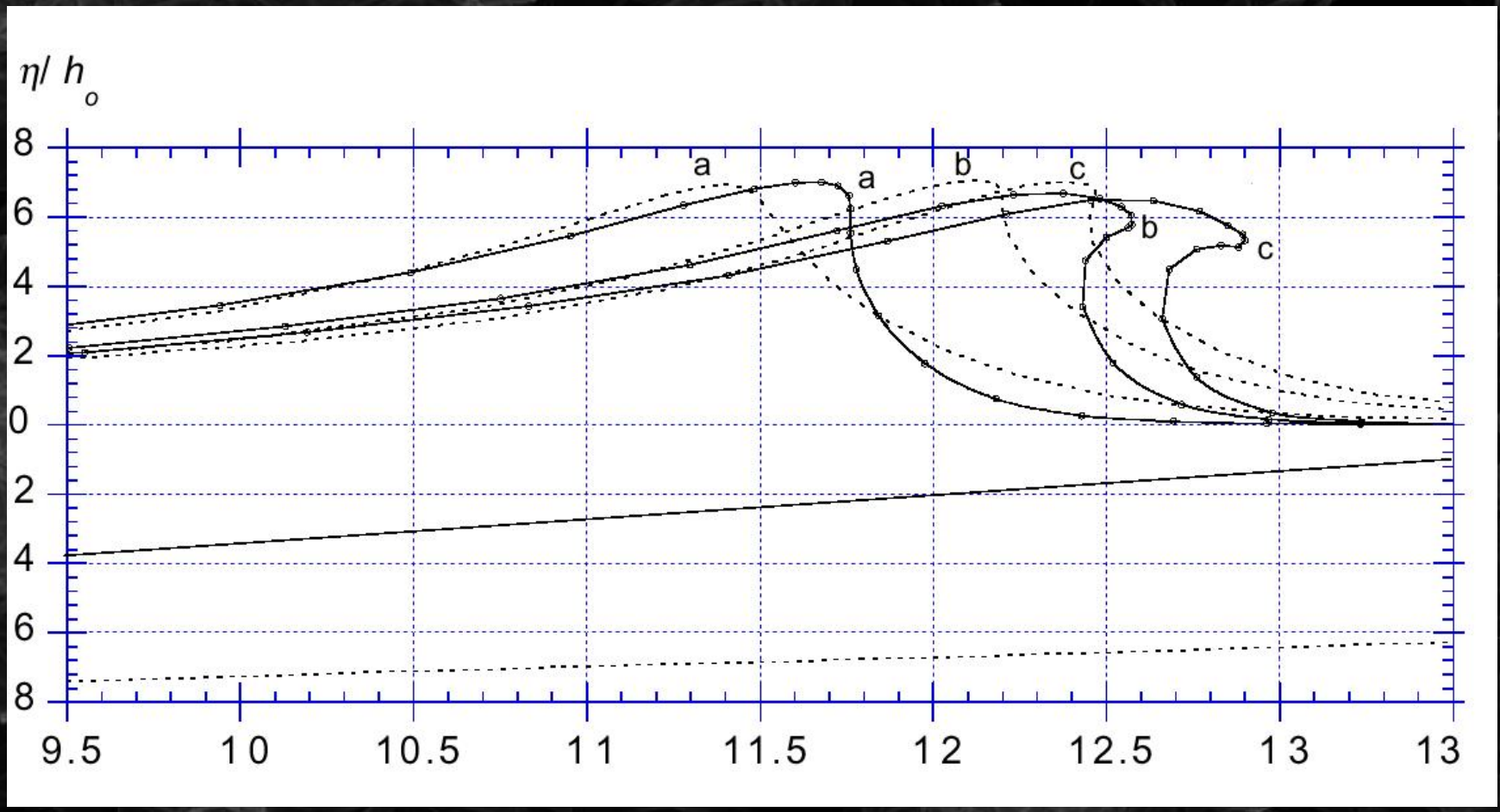
At points $\mathbf{r}$ on the surface

$$\frac{\partial \phi(\mathbf{r}, t)}{\partial t} + \frac{1}{2} |\nabla \phi(\mathbf{r}, t)|^2 + \frac{p(\mathbf{r}, t)}{\rho} + g\mathbf{r} \cdot \hat{\mathbf{y}} = 0$$

Dynamics of surface points:

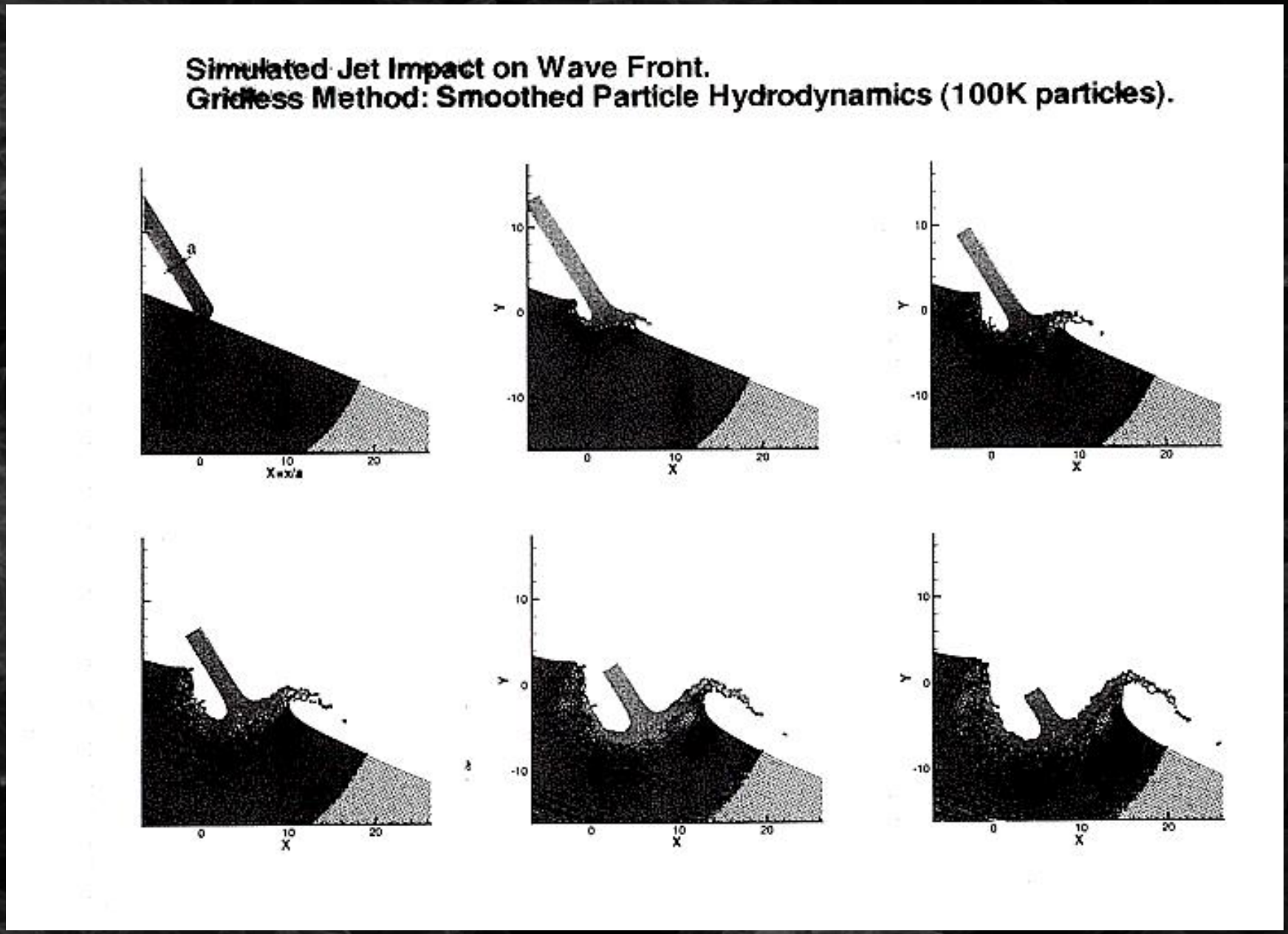
$$\frac{d\mathbf{r}(t)}{dt} = \nabla \phi(\mathbf{r}, t)$$

Numerical Wave Tank Simulation



Grilli, Guyenne, Dias (2000)

Plunging Break and Splash Simulation



Tulin (1999)

Simplifying the Problem

Road to practicality - ocean surface:

- Simplify equations for relatively mild conditions
- Fill in gaps with oceanography.

Original dynamical equation at 3D points in volume

$$\frac{\partial \phi(\mathbf{r}, t)}{\partial t} + \frac{1}{2}|\nabla \phi(\mathbf{r}, t)|^2 + \frac{p(\mathbf{r}, t)}{\rho} + g\mathbf{r} \cdot \hat{\mathbf{y}} = 0$$

Equation at 2D points (x, z) on surface with height h

$$\frac{\partial \phi(x, z, t)}{\partial t} = -gh(x, z, t)$$

Simplifying the Problem: Mass Conservation

Vertical component of velocity

$$\frac{\partial h(x,z,t)}{\partial t} = \hat{\mathbf{y}} \cdot \nabla \phi(x,z,t)$$

Use mass conservation condition

$$\hat{\mathbf{y}} \cdot \nabla \phi(x,z,t) \sim \left(\sqrt{-\nabla_H^2}\right) \phi = \left(\sqrt{-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial z^2}}\right) \phi$$

Linearized Surface Waves

$$\frac{\partial h(x,z,t)}{\partial t} = \left(\sqrt{-\nabla_H^2}\right)\phi(x,z,t)$$

$$\frac{\partial \phi(x,z,t)}{\partial t} = -gh(x,z,t)$$

General solution easily computed in terms of Fourier Transforms

Solution for Linearized Surface Waves

General solution in terms of Fourier Transform

$$h(x,z,t) = \int_{-\infty}^{\infty} dk_x\, dk_z\, \tilde{h}(\mathbf{k},t)\ \exp\{i(k_x x + k_z z)\}$$

with the amplitude depending on the *dispersion relationship*

$$\omega_0(\mathbf{k}) = \sqrt{g\,|\mathbf{k}|}$$

$$\tilde{h}(\mathbf{k},t) = \tilde{h}_0(\mathbf{k}) \exp\{-i\omega_0(\mathbf{k})t\} + \tilde{h}_0^*(-\mathbf{k}) \exp\{i\omega_0(\mathbf{k})t\}$$

The complex amplitude $\tilde{h}_0(\mathbf{k})$ is arbitrary.

Oceanography

- Think of the heights of the waves as a kind of random process
- Decades of detailed measurements support a statistical description of ocean waves.
- The wave height has a spectrum

$$\left\langle \left| \tilde{h}_0(\mathbf{k}) \right|^2 \right\rangle = P_0(\mathbf{k})$$

- Oceanographic models tie P_0 to environmental parameters like wind velocity, temperature, salinity, etc.

Models of Spectrum

- Wind speed V
- Wind direction vector $\hat{\mathbf{V}}$ (horizontal only)
- Length scale of biggest waves $L = V^2/g$ (g=gravitational constant)

Phillips Spectrum

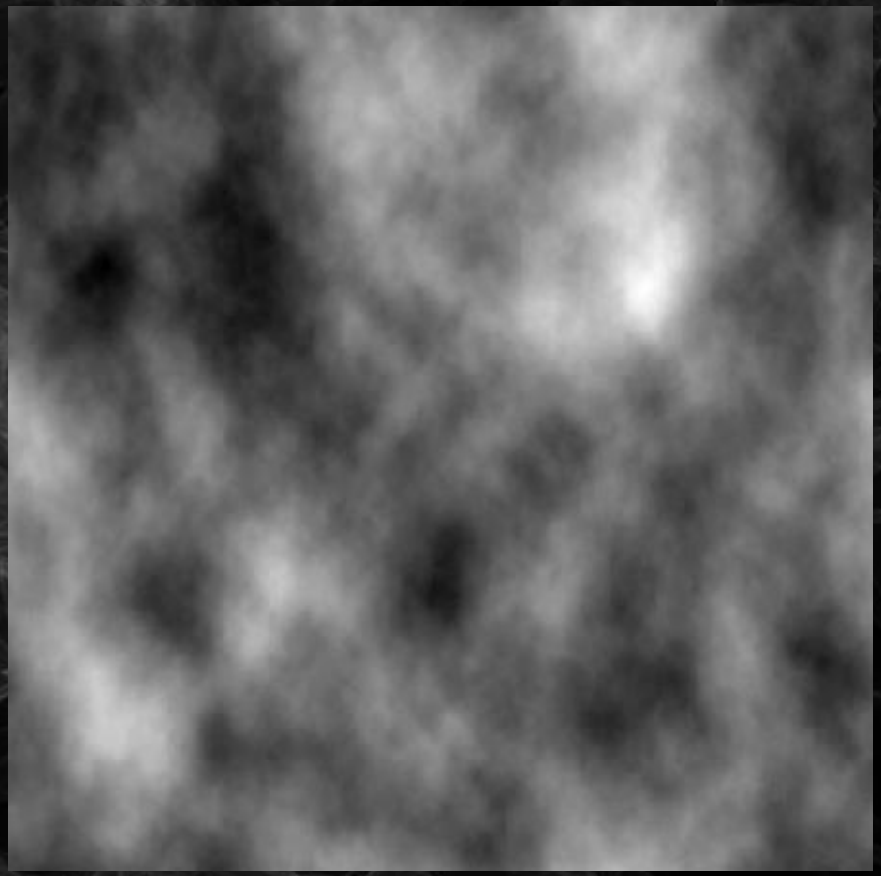
$$P_0(\mathbf{k}) = \left|\hat{\mathbf{k}} \cdot \hat{\mathbf{V}}\right|^2 \frac{\exp(-1/k^2L^2)}{k^4}$$

JONSWAP Frequency Spectrum

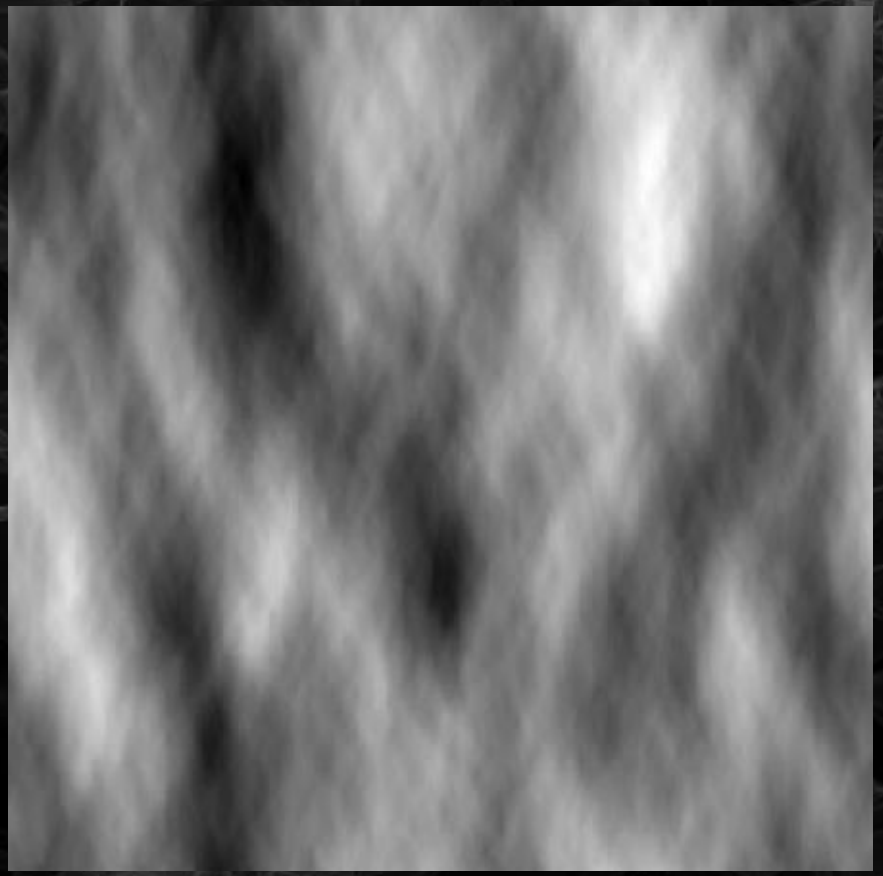
$$P_0(\omega) = \frac{\exp\left\{-\frac{5}{4}\left(\frac{\omega}{\Omega}\right)^{-4} + e^{-(\omega-\Omega)^2/2(\sigma\Omega)^2}\ln\gamma\right\}}{\omega^5}$$

Variation in Wave Height Field

Pure Phillips Spectrum



Modified Phillips Spectrum



Simulation of a Random Surface

Generate a set of “random” amplitudes on a grid

$$\tilde{h}_0(\mathbf{k}) = \xi e^{i\theta}\sqrt{P_0(\mathbf{k})}$$

ξ = Gaussian random number, mean 0 & std dev 1

θ = Uniform random number [0,2π].

$$k_x = \frac{2\pi}{\Delta x}\frac{n}{N} \quad (n = -N/2, \ldots, (N-1)/2)$$

$$k_z = \frac{2\pi}{\Delta z}\frac{m}{M} \quad (m = -M/2, \ldots, (M-1)/2)$$

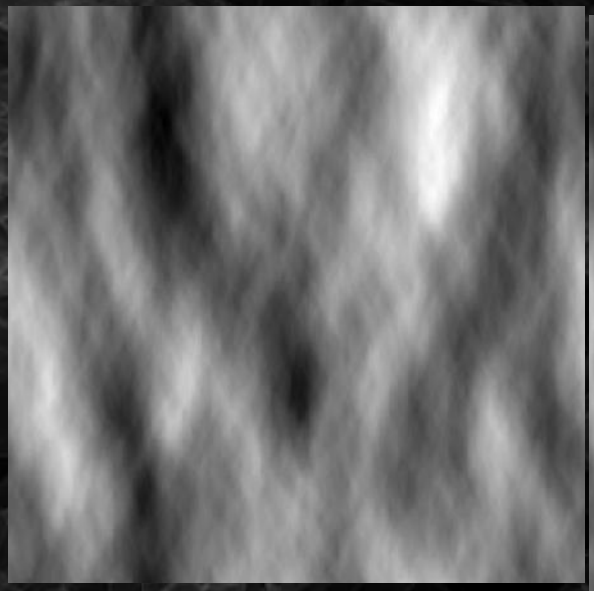
FFT of Random Amplitudes

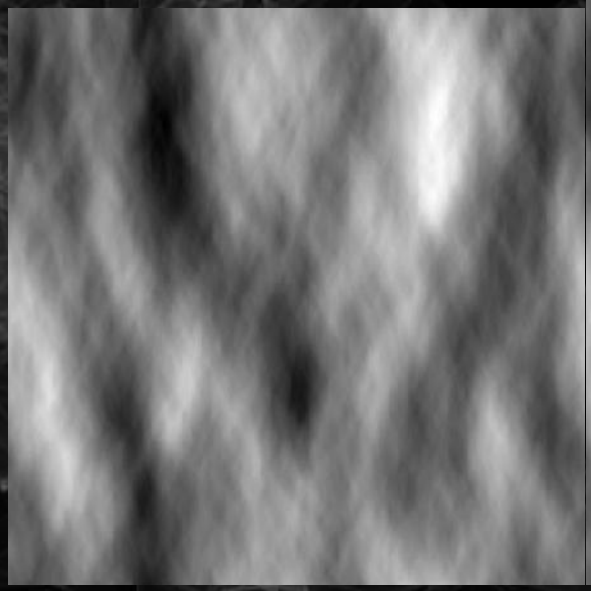
Use the Fast Fourier Transform (FFT) on the amplitudes to obtain the wave height realization $h(x, z, t)$

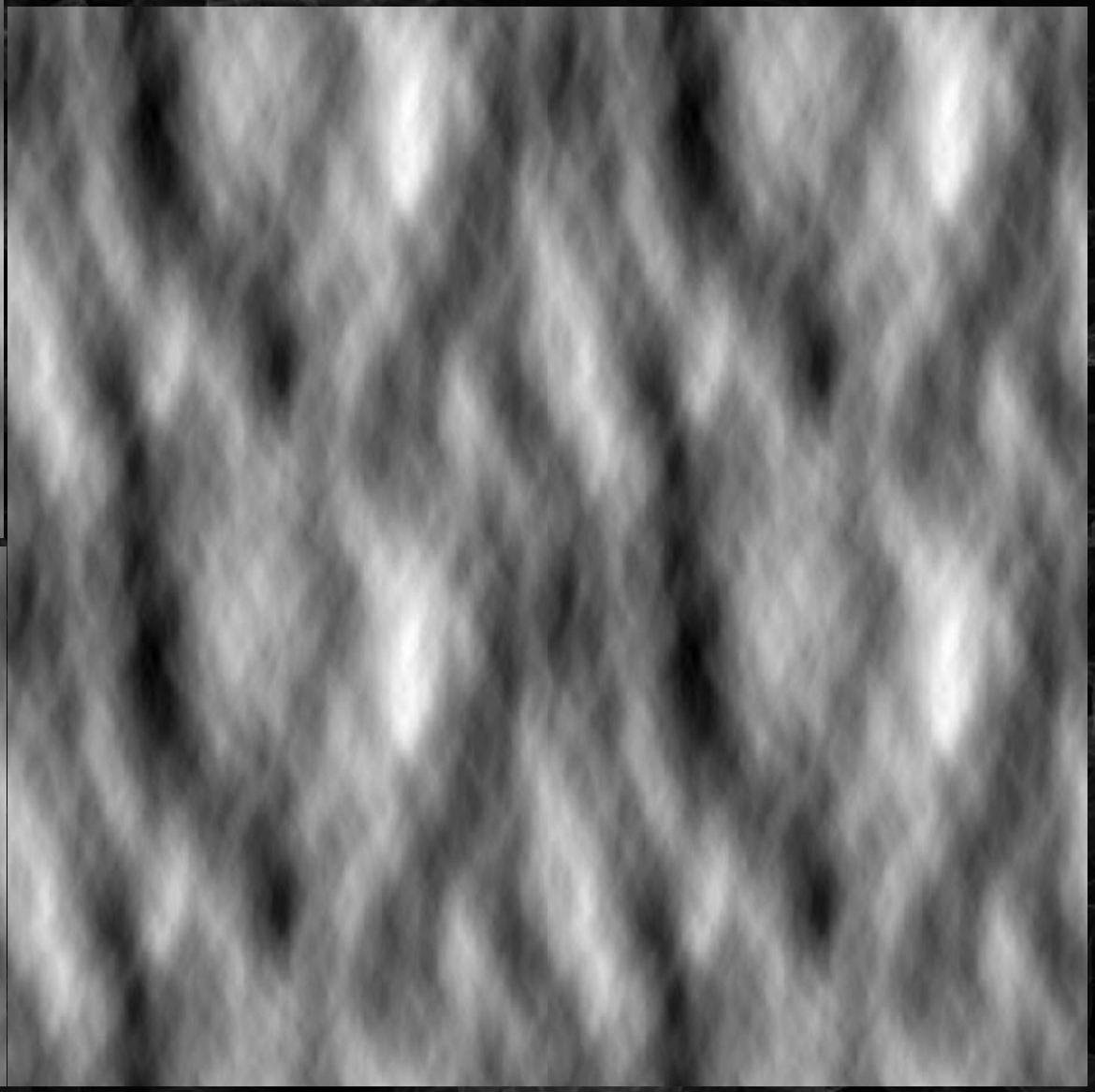
Wave height realization exists on a regular, periodic grid of points.

$$x = n\Delta x \quad (n = -N/2, \ldots, (N-1)/2)$$
$$z = m\Delta z \quad (m = -M/2, \ldots, (M-1)/2)$$

The realization tiles seamlessly. This can sometimes show up as repetitive waves in a render.





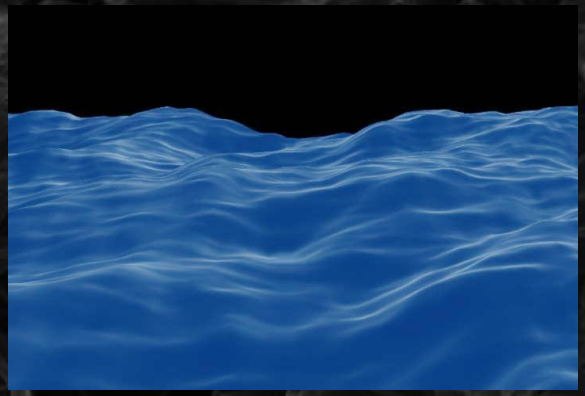


High Resolution Rendering

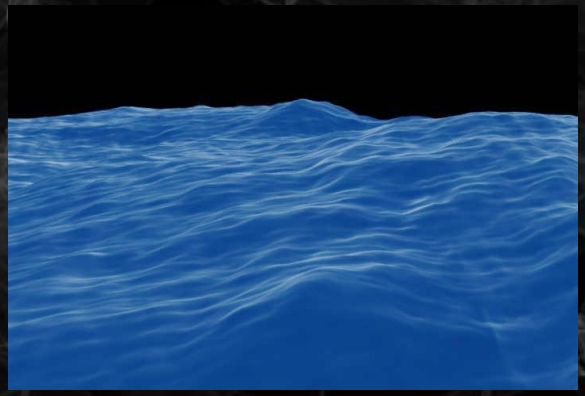
Sky reflection, upwelling light, sun glitter

1 inch facets, 1 kilometer range

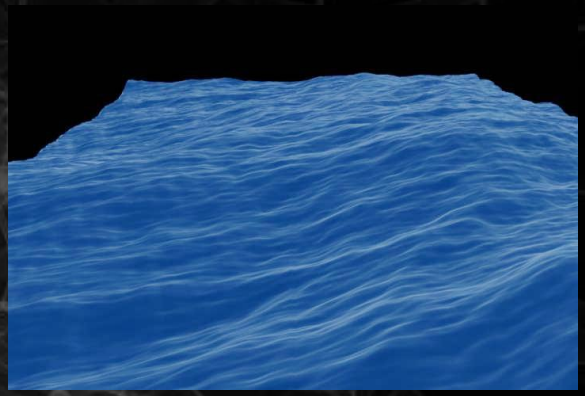
Effect of Resolution



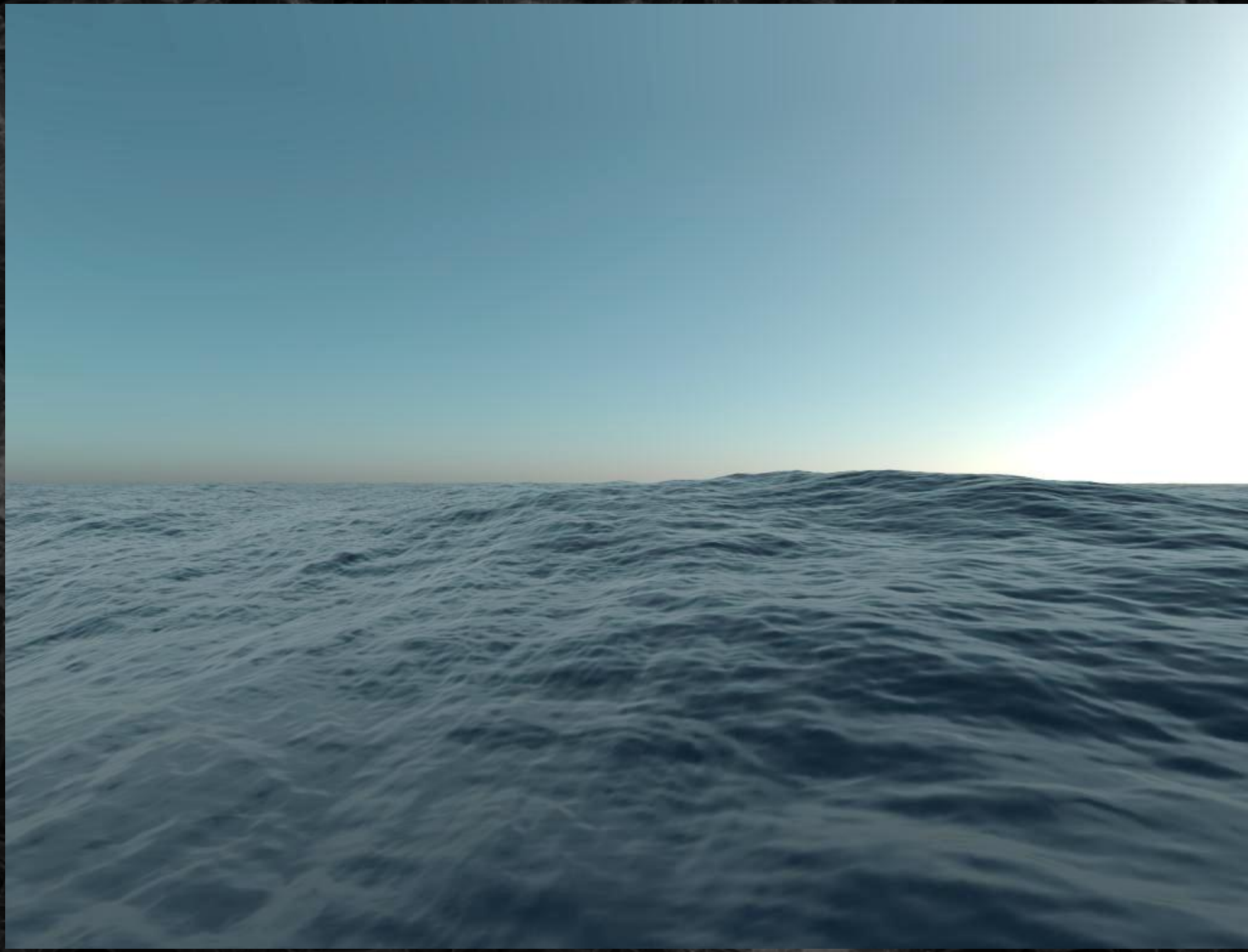
Low : 100 cm facets



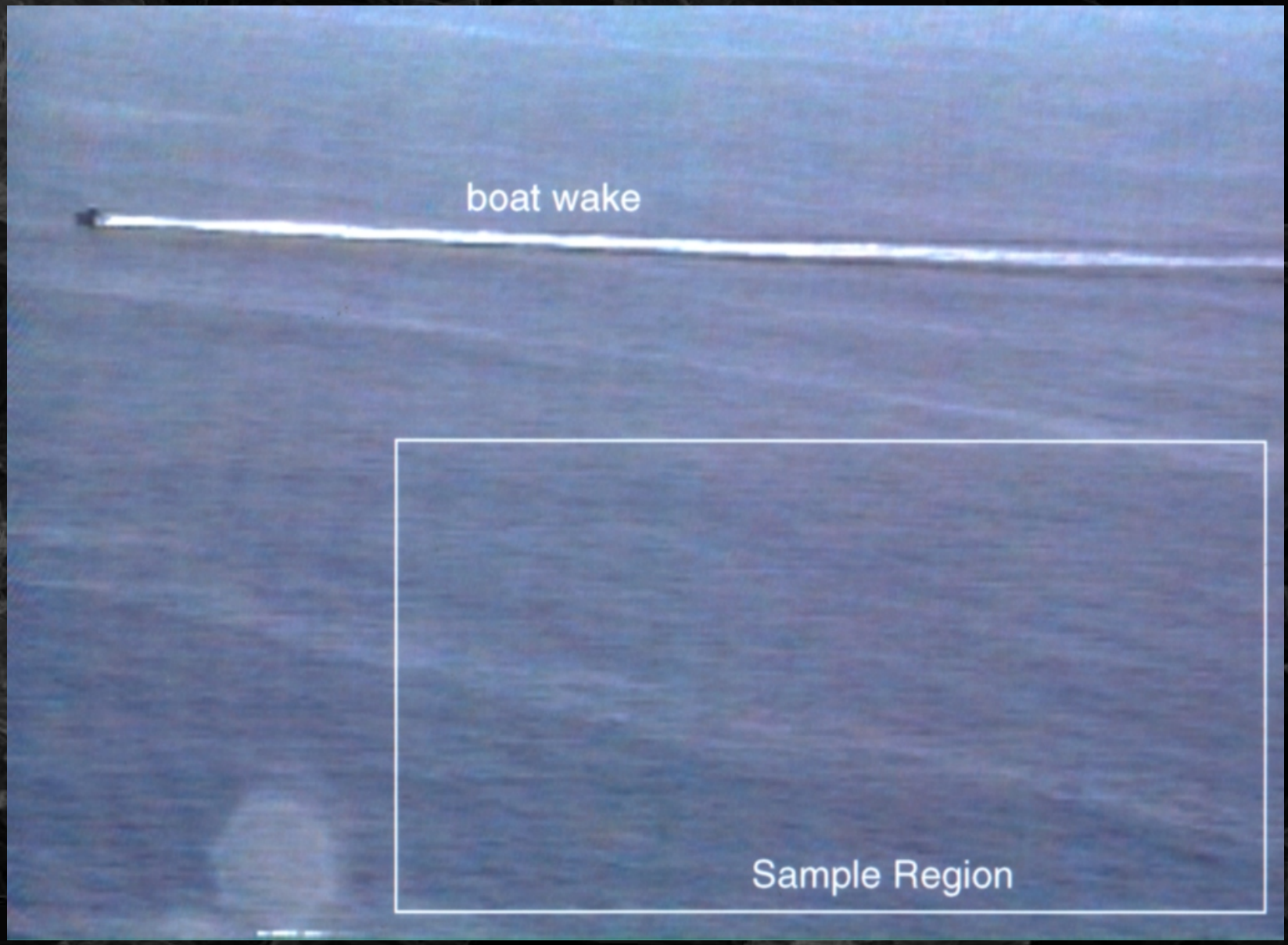
Medium : 10 cm facets



High : 1 cm facets



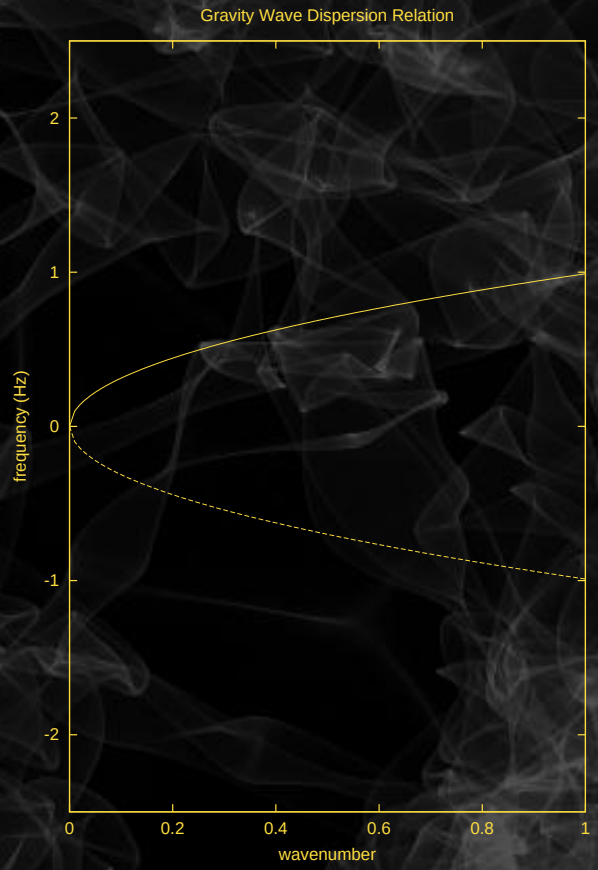
Simple Demonstration of Dispersion



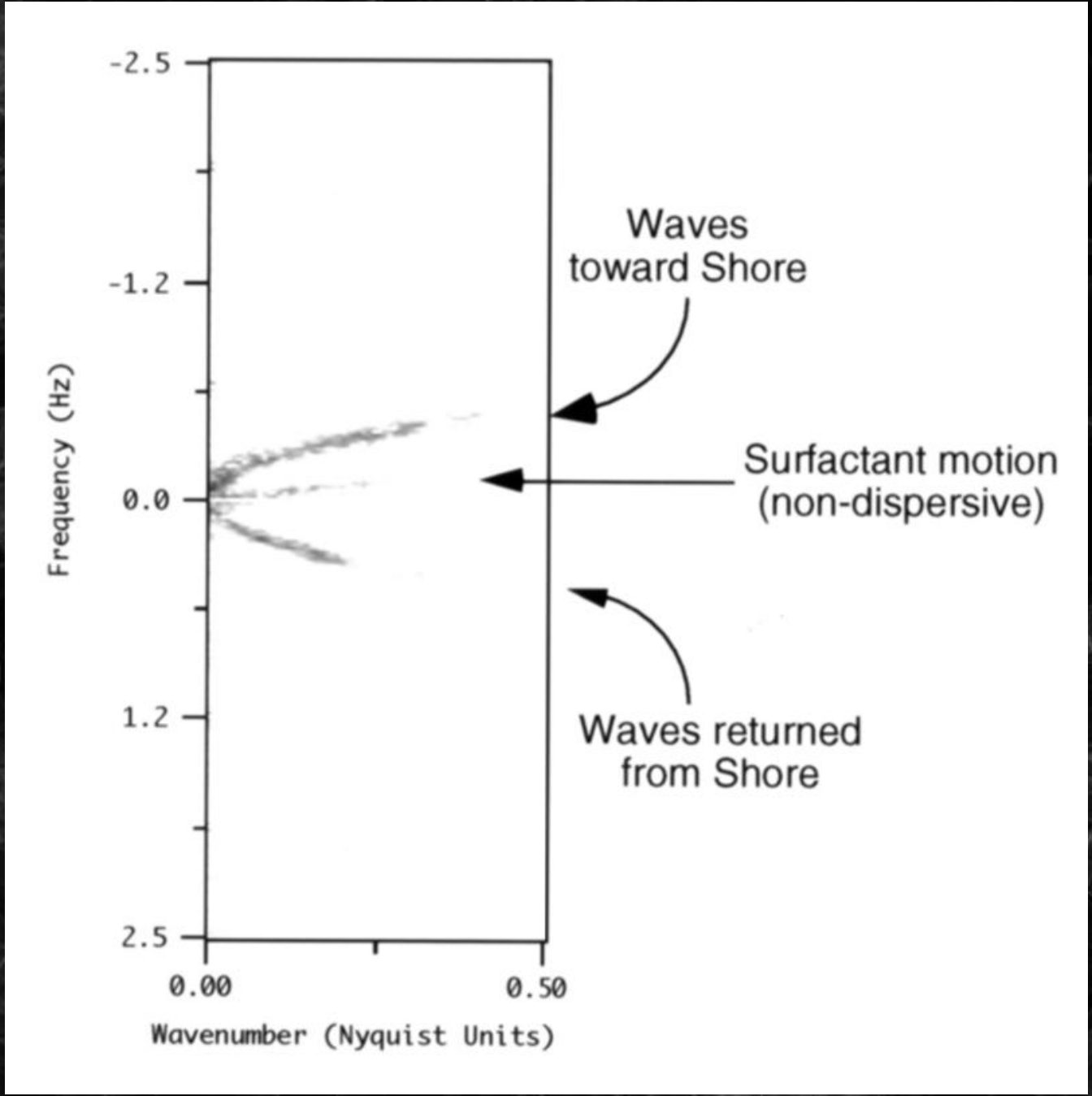
256 frames, 256×128 region

Data Processing

- Fourier transform in both time and space: $\tilde{h}(\mathbf{k}, \omega)$
- Form Power Spectral Density $P(\mathbf{k}, \omega) = \left\langle \left| \tilde{h}(\mathbf{k}, \omega) \right|^2 \right\rangle$
- If the waves follow dispersion relationship, then P is strongest at frequencies $\omega = \omega(k)$.



Processing Results



Looping in Time – Continuous Loops

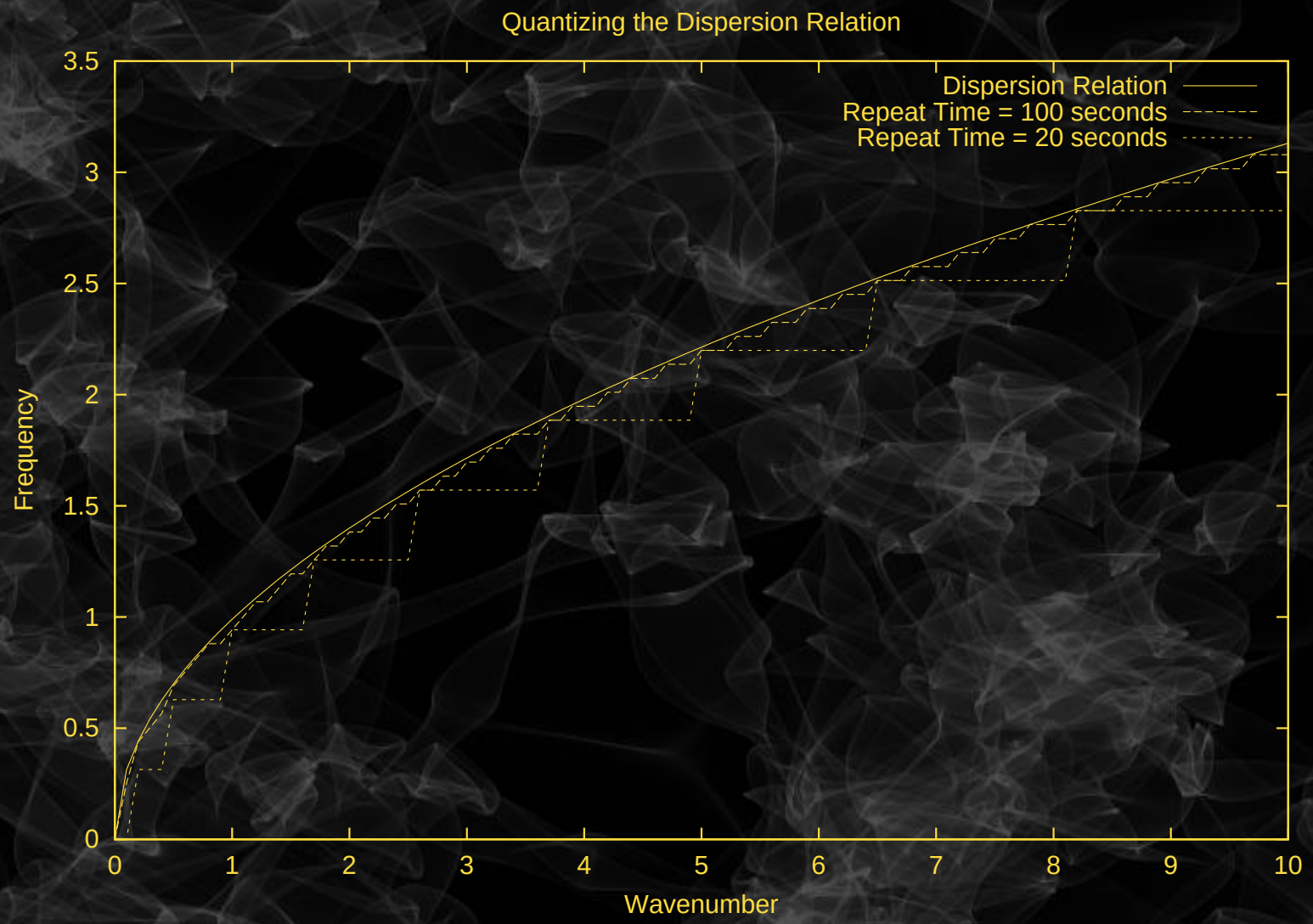
- Continuous loops can't be made because dispersion doesn't have a fundamental frequency.
- Loops can be made by modifying the dispersion relationship.

Repeat time T

Fundamental Frequency $\omega_0 = \frac{2\pi}{T}$

New dispersion relation $\tilde{\omega} = \text{integer}\left(\frac{\omega(k)}{\omega_0}\right) \omega_0$

Quantized Dispersion Relation



Hamiltonian Approach for Surface Waves

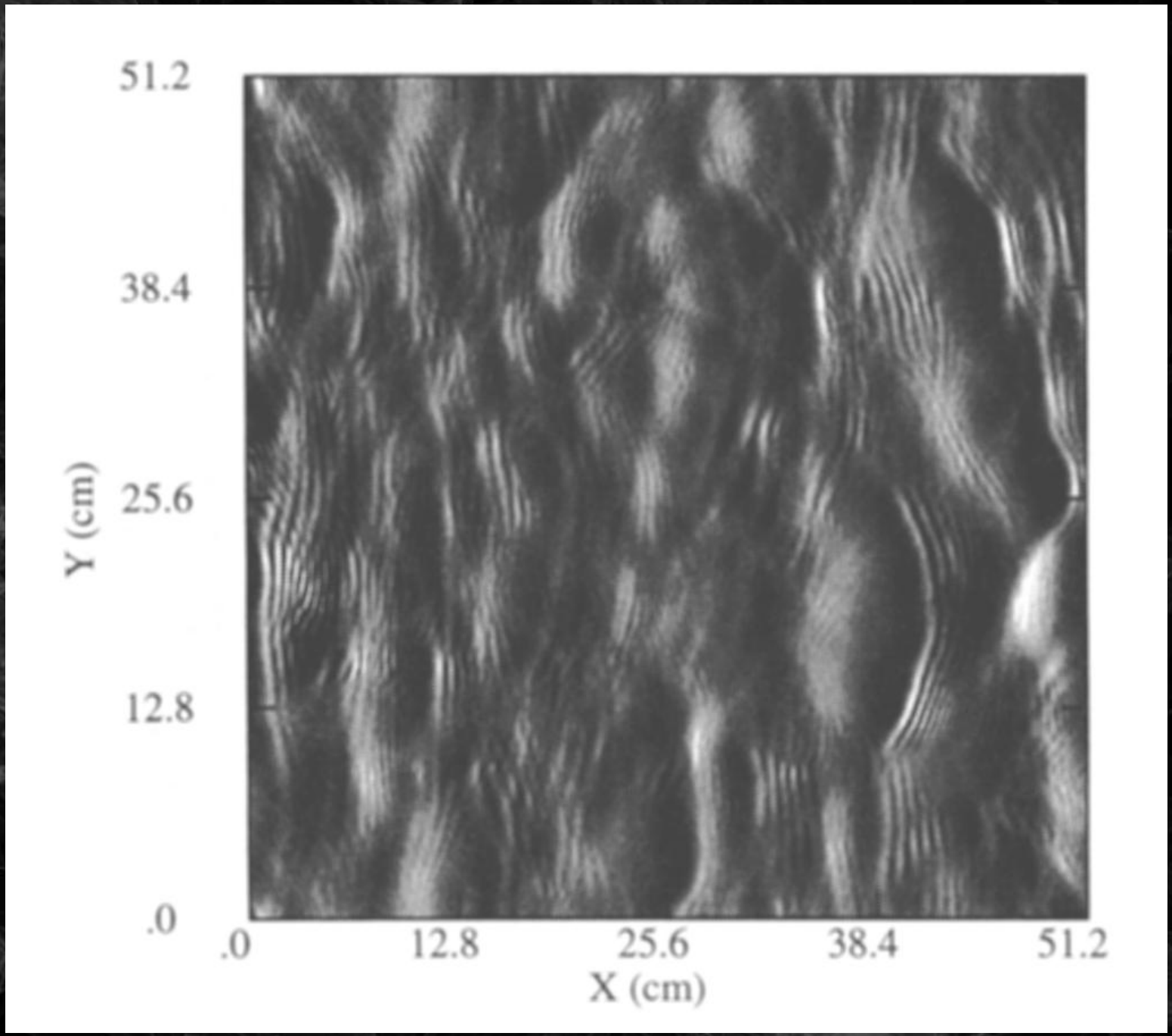
Miles, Milder, Henyey, . . .

- If a crazy-looking surface operator like $\sqrt{-\nabla_H^2}$ is ok, the exact problem can be recast as a *canonical problem* with momentum ϕ and coordinate h in 2D.

- Milder has demonstrated numerically:
  - The onset of wave breaking
  - Accurate capillary wave interaction

- Henyey *et al.* introduced *Canonical Lie Transformations*:
  - Start with the solution of the linearized problem - (ϕ_0, h_0)
  - Introduce a continuous set of transformed fields - (ϕ_q, h_q)
  - The exact solution for surface waves is for $q = 1$.

Surface Wave Simulation (Milder, 1990)



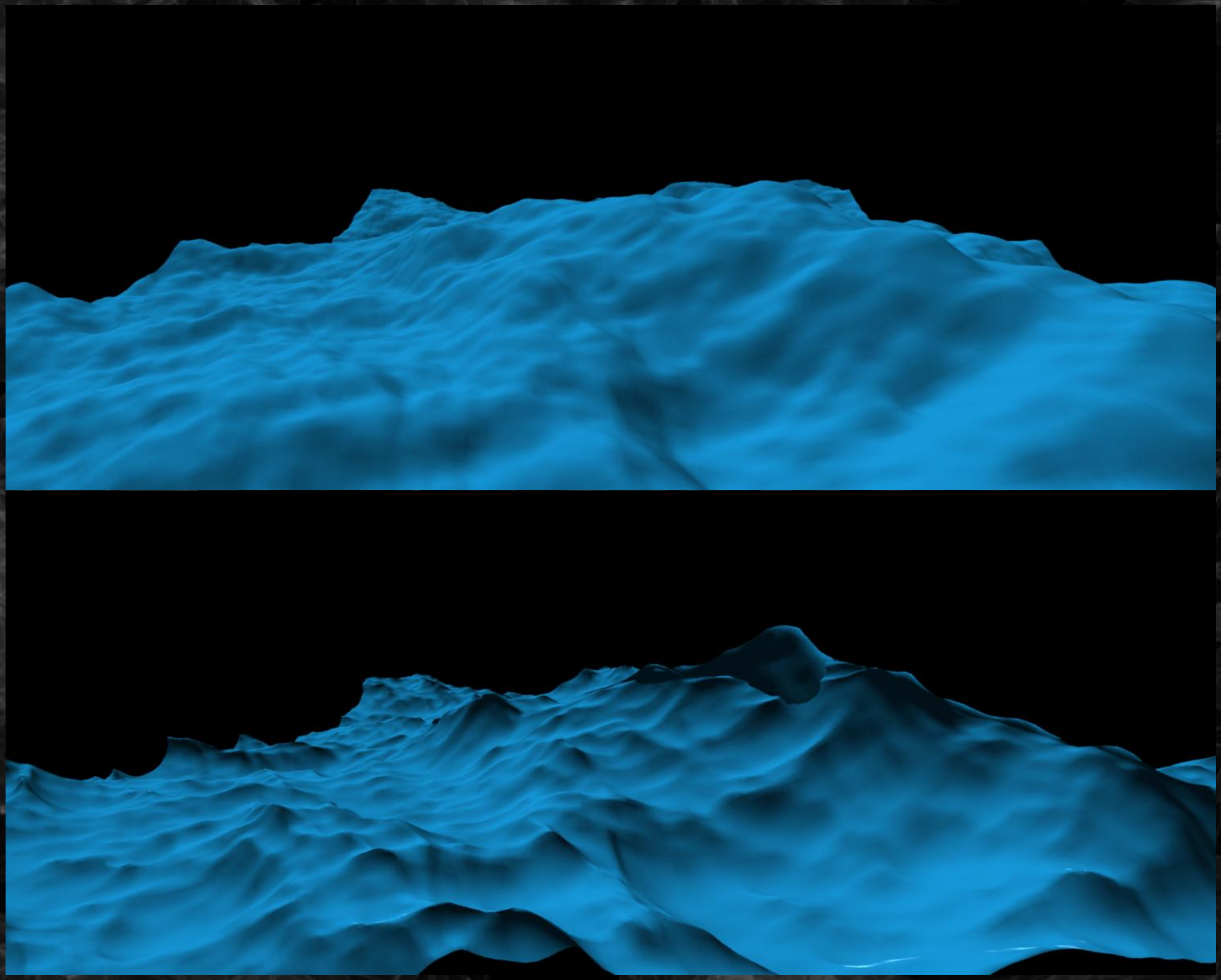
Choppy, Near-Breaking Waves

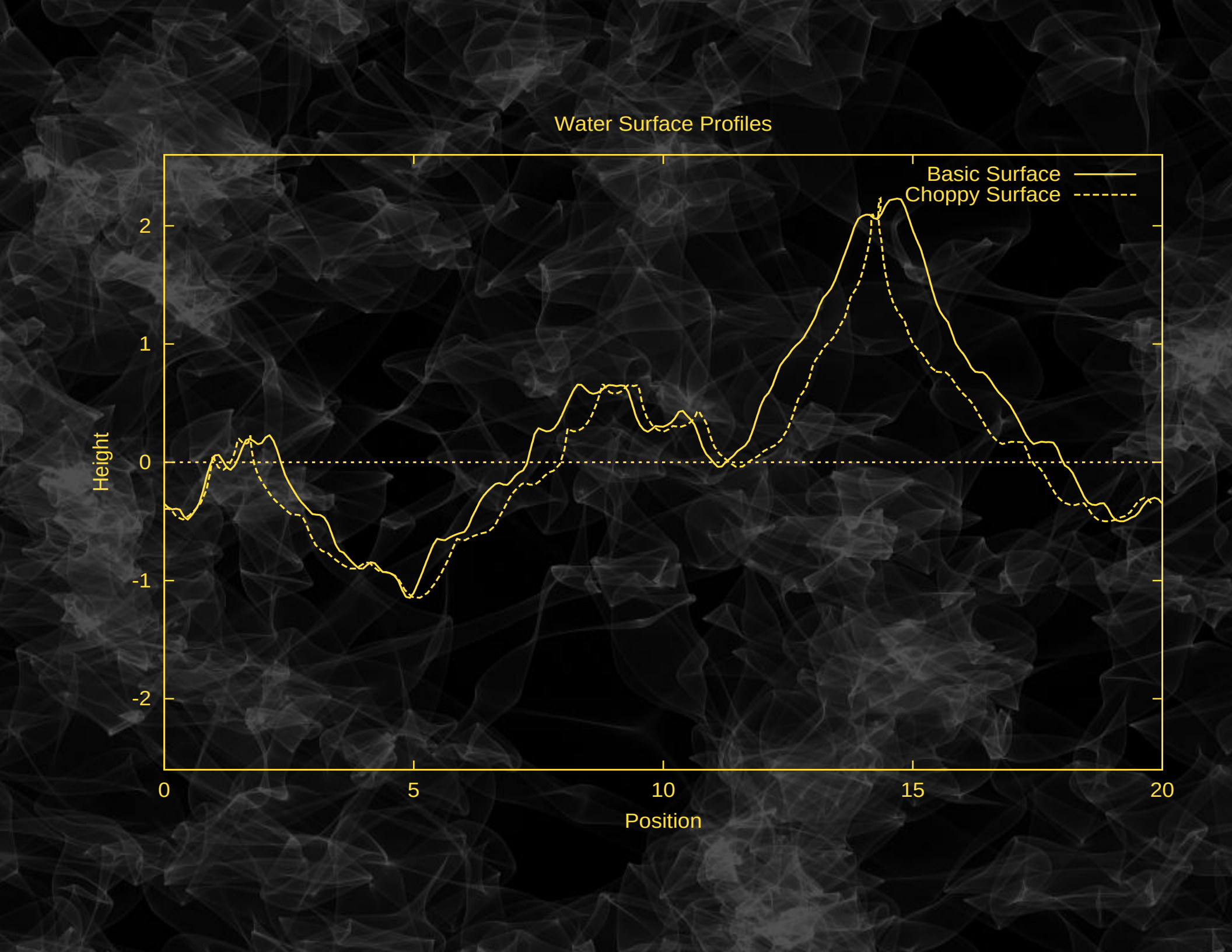
Horizontal velocity becomes important for distorting wave.

Wave at $\mathbf{x}$ morphs horizontally to the position $\mathbf{x} + \mathbf{D}(\mathbf{x}, t)$

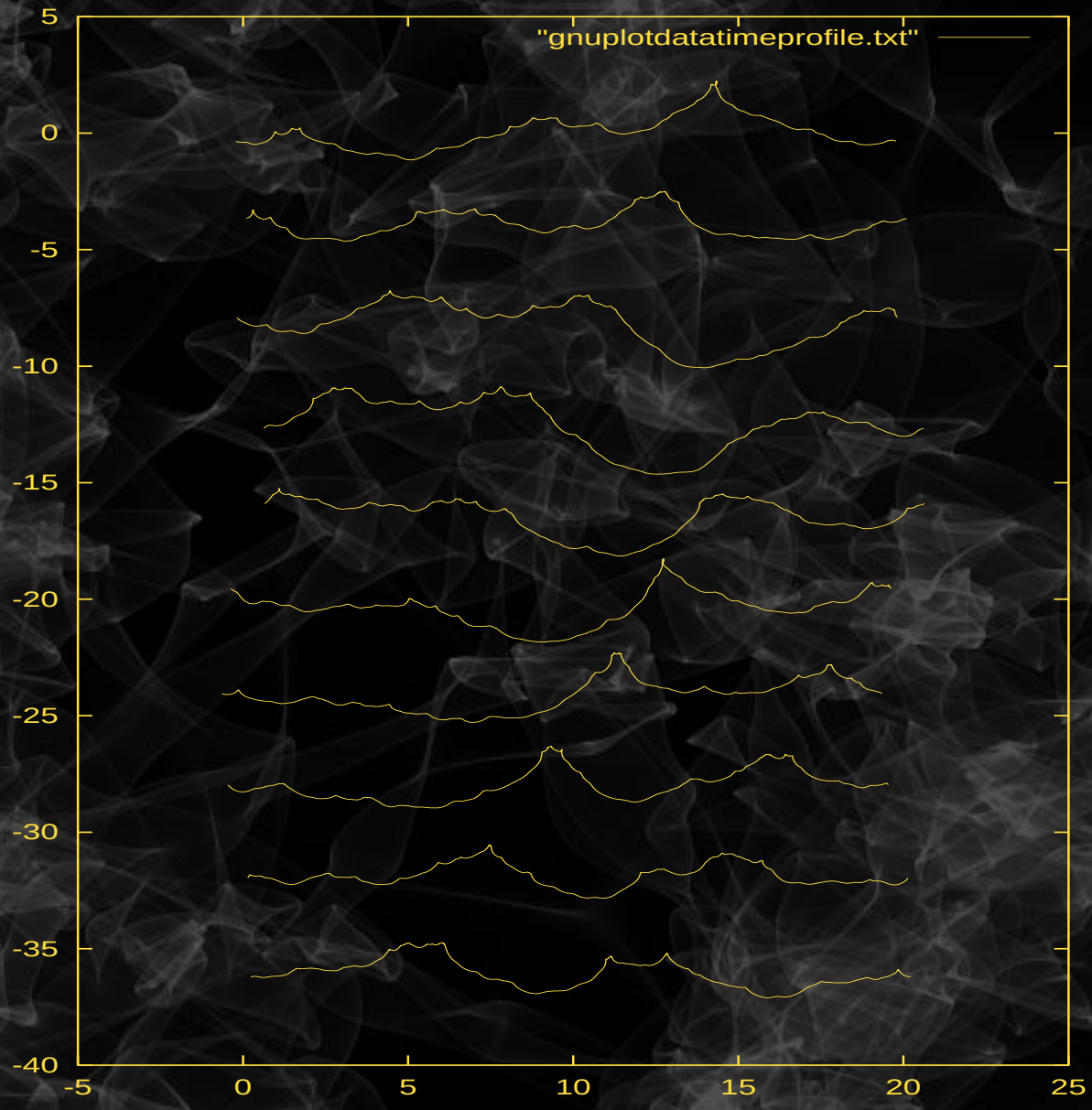
$$\mathbf{D}(\mathbf{x}, t) = -\lambda \int d^2k \, \frac{i\mathbf{k}}{|\mathbf{k}|} \tilde{h}(\mathbf{k}, t) \, \exp\{i(k_x x + k_z z)\}$$

The factor λ allows artistic control over the magnitude of the morph.




Water Surface Profiles
Basic Surface
Choppy Surface
Height
2
1
0
-1
-2
0
5
10
15
20
Position

Time Sequence of Choppy Waves



Choppy Waves: Detecting Overlap

$$\mathbf{x} \rightarrow \mathbf{X}(\mathbf{x}, t) = \mathbf{x} + \mathbf{D}(\mathbf{x}, t)$$

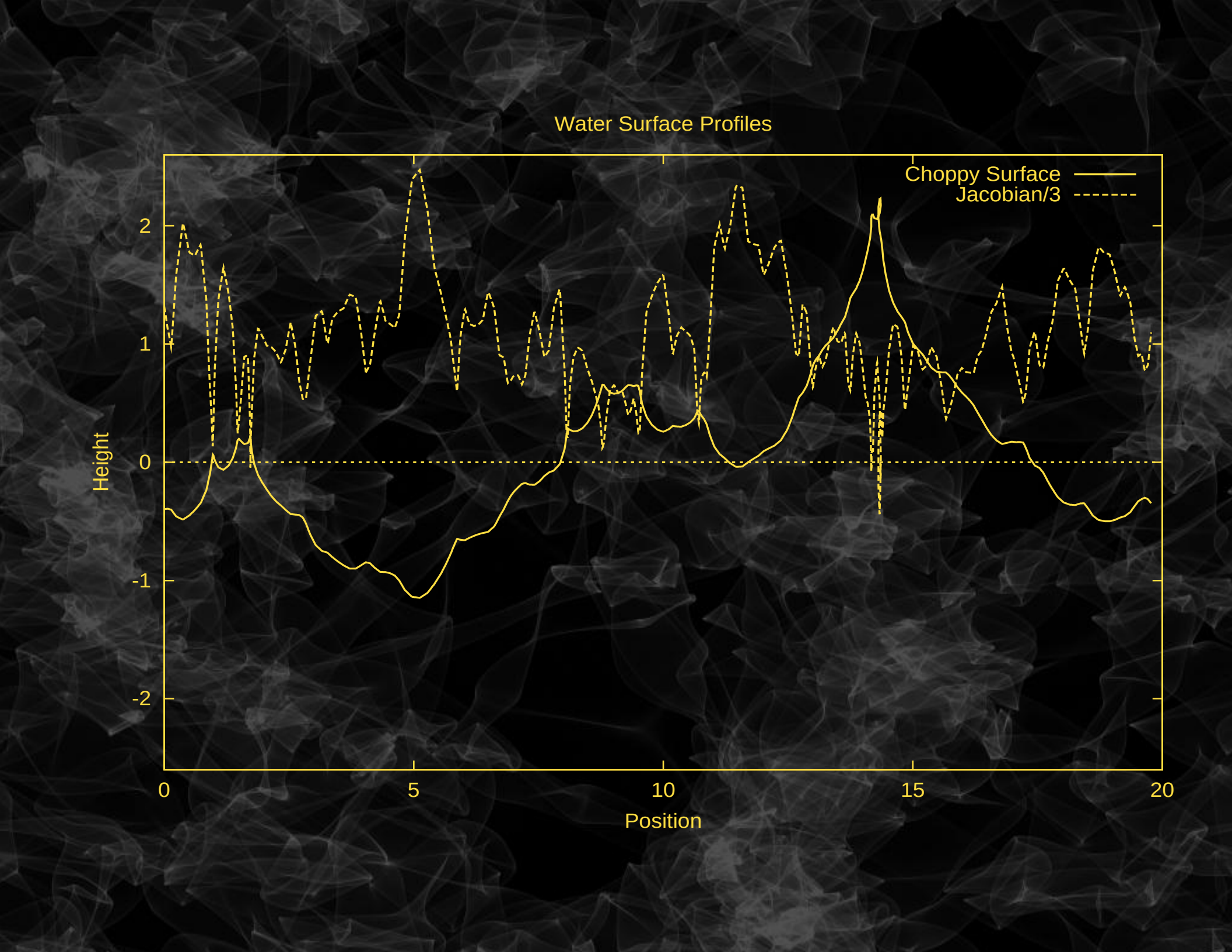
is unique and invertible as long as the surface does not intersect itself.

When the mapping intersects itself, it is not unique. The quantitative measure of this is the *Jacobian* matrix

$$J(\mathbf{x}, t) = \begin{bmatrix} \partial \mathbf{X}_x / \partial x & \partial \mathbf{X}_x / \partial z \\ \partial \mathbf{X}_z / \partial x & \partial \mathbf{X}_z / \partial z \end{bmatrix}$$

The signal that the surface intersects itself is

$$\det(J) \leq 0$$


Water Surface Profiles
Choppy Surface
Jacobian/3
Height
2
1
0
-1
-2
0
5
10
15
20
Position

Learning More About Overlap

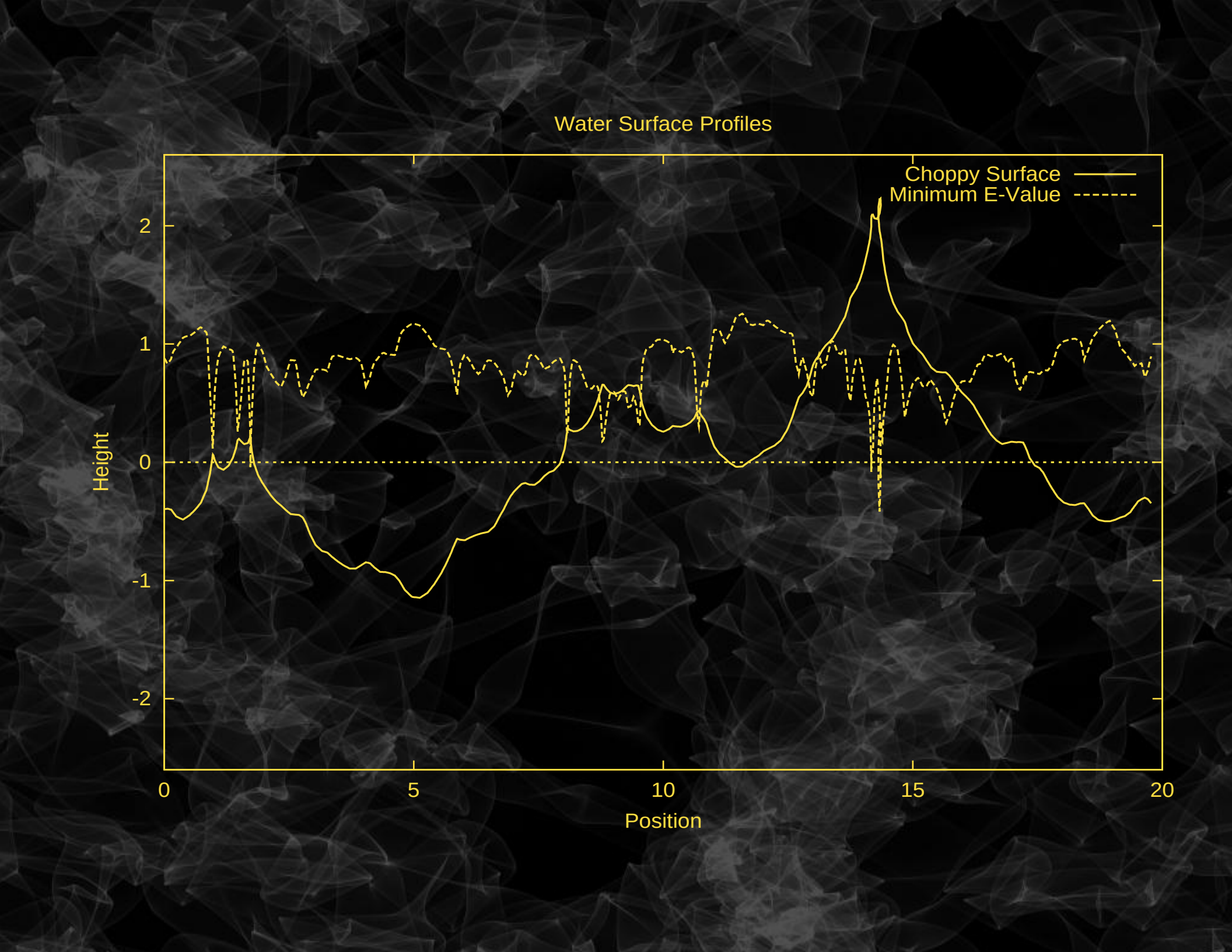
Two *eigenvalues*, $J_- \leq J_+$, and *eigenvectors* $\hat{\mathbf{e}}_-$, $\hat{\mathbf{e}}_+$

$$J = J_- \hat{\mathbf{e}}_- \hat{\mathbf{e}}_- \; + \; J_+ \hat{\mathbf{e}}_+ \hat{\mathbf{e}}_+$$

$$\det(J) = J_- J_+$$

For no chop, $J_- = J_+ = 1$. As the displacement magnitude increases, J_+ stays positive while J_- becomes negative at the location of overlap.

At overlap, $J_- < 0$, the alignment of the overlap is parallel to the eigenvalue $\hat{\mathbf{e}}_-$.


Water Surface Profiles
Choppy Surface
Minimum E-Value
Height
2
1
0
-1
-2
0
5
10
15
20
Position

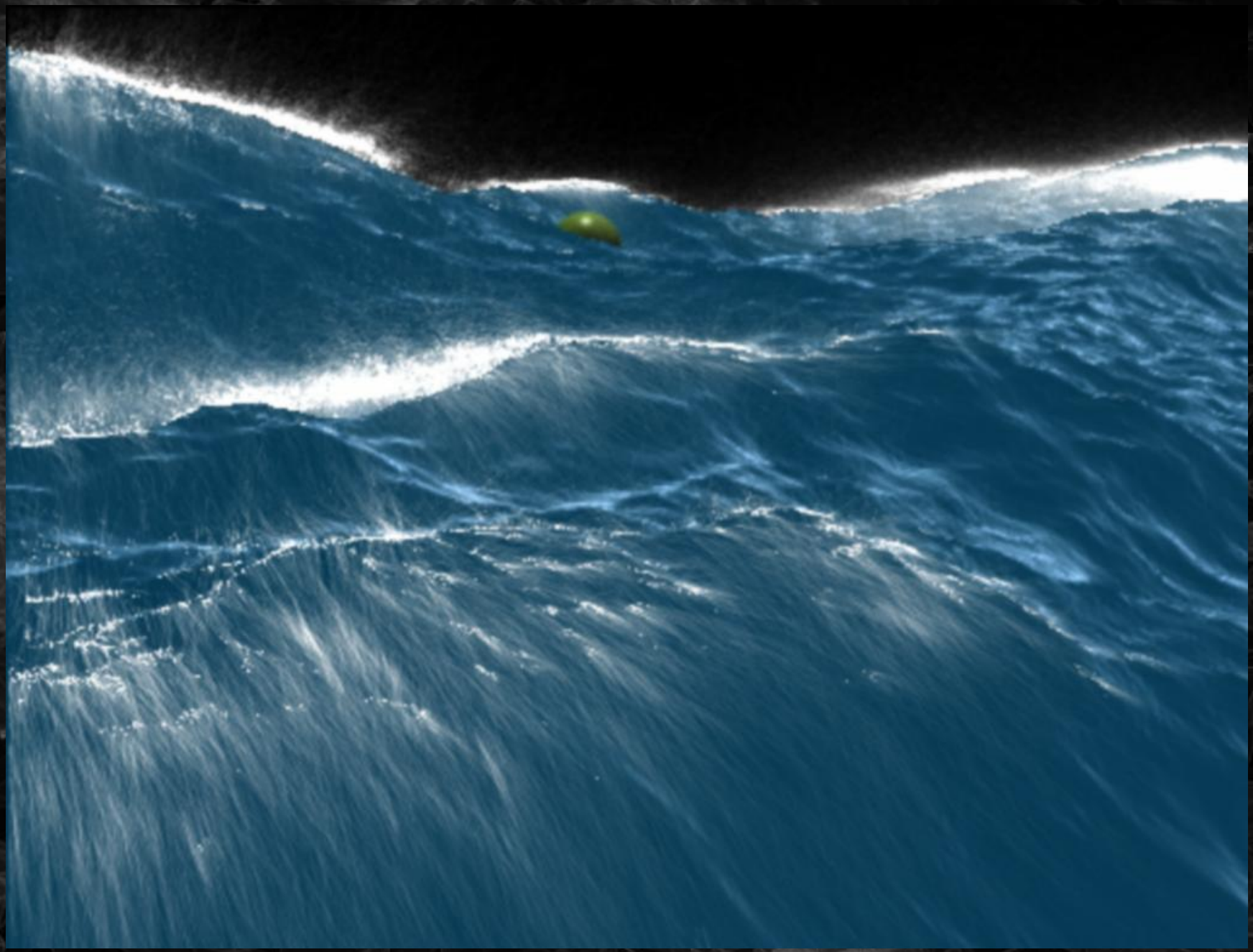
Simple Spray Algorithm

- Pick a point on the surface at random
- Emit a spray particle if $J_- < J_T$ threshold
- Particle initial direction ($\hat{\mathbf{n}}$ = surface normal)

$$\hat{\mathbf{v}} = \frac{(J_T - J_-)\hat{\mathbf{e}}_- + \hat{\mathbf{n}}}{\sqrt{1 + (J_T - J_-)^2}}$$

- Particle initial speed from a half-gaussian distribution with mean proportional to $J_T - J_-$.
- Simple particle dynamics: gravity and wind drag

Surface and Spray Render



Summary

- FFT-based random ocean surfaces are fast to build, realistic, and flexible.
- Based on a mixture of theory and experimental phenomenology.
- Used alot in professional productions.
- Real-time capable for games
- Lots of room for more complex behaviors.

Latest version of course notes and slides:

http://home1.gte.net/tssndrf/index.html

References

- *Ivan Aivazovsky Artist of the Ocean*, by Nikolai Novouspensky, Parkstone/Aurora, Bournemouth, England, 1995.
- Jeff Odien, “On the Waterfront”, Cinefex, No. 64, p 96, (1995)
- Ted Elrick, “Elemental Images”, Cinefex, No. 70, p 114, (1997)
- Kevin H. Martin, “Close Contact”, Cinefex, No. 71, p 114, (1997)
- Don Shay, “Ship of Dreams”, Cinefex, No. 72, p 82, (1997)
- Kevin H. Martin, “Virus: Building a Better Borg”, Cinefex, No. 76, p 55, (1999)
- Grilli, S.T., Guyenne, P., Dias, F., “Modeling of Overturning Waves Over Arbitrary Bottom in a 3D Numerical Wave Tank,” *Proceedings 10th Offshore and Polar Enging. Conf.* (ISOPE00, Seattle, USA, May 2000), Vol. **III**, 221-228.
- Marshall Tulin, “Breaking Waves in the Ocean,” *Program on Physics of Hydrodynamic Turbulence*, (Institute for Theoretical Phyics, Feb 7, 2000), http://online.itp.ucsb.edu/online/hydrot00/si-index.html
- Dennis B. Creamer, Frank Henyey, Roy Schult, and Jon Wright, “Improved Linear Representation of Ocean Surface Waves.” J. Fluid Mech, **205**, pp. 135-161, (1989).
- Milder, D.M., “The Effects of Truncation on Surface Wave Hamiltonians,” J. Fluid Mech., *217*, 249-262, 1990.

Stable Fluids

Jos Stam*

Alias | wavefront

Abstract

Building animation tools for fluid-like motions is an important and challenging problem with many applications in computer graphics. The use of physics-based models for fluid flow can greatly assist in creating such tools. Physical models, unlike key frame or procedural based techniques, permit an animator to almost effortlessly create interesting, swirling fluid-like behaviors. Also, the interaction of flows with objects and virtual forces is handled elegantly. Until recently, it was believed that physical fluid models were too expensive to allow real-time interaction. This was largely due to the fact that previous models used unstable schemes to solve the physical equations governing a fluid. In this paper, for the first time, we propose an unconditionally stable model which still produces complex fluid-like flows. As well, our method is very easy to implement. The stability of our model allows us to take larger time steps and therefore achieve faster simulations. We have used our model in conjuction with advecting solid textures to create many fluid-like animations interactively in two- and three-dimensions.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

Keywords: animation of fluids, Navier-Stokes, stable solvers, implicit elliptic PDE solvers, interactive modeling, gaseous phenomena, advected textures

1 Introduction

One of the most intriguing problems in computer graphics is the simulation of fluid-like behavior. A good fluid solver is of great importance in many different areas. In the special effects industry there is a high demand to convincingly mimic the appearance and behavior of fluids such as smoke, water and fire. Paint programs can also benefit from fluid solvers to emulate traditional techniques such as watercolor and oil paint. Texture synthesis is another possible application. Indeed, many textures result from fluid-like processes, such as erosion. The modeling and simulation of fluids is, of course, also of prime importance in most scientific disciplines and in engineering. Fluid mechanics is used as the standard mathematical framework on which these simulations are based. There is a consensus among scientists that the *Navier-Stokes* equations are a very good model for fluid flow. Thousands of books and articles have been published in various areas on how to compute these equations numerically. Which solver to use in practice depends largely on the problem at hand and on the computing power available. Most engineering tasks require that the simulation provide accurate bounds on the physical quantities involved to answer questions related to safety, performance, etc. The visual appearance (shape) of the flow is of secondary importance in these applications. In computer graphics, on the other hand, the shape and the behavior of the fluid are of primary interest, while physical accuracy is secondary or in some cases irrelevant. Fluid solvers, for computer graphics, should ideally provide a user with a tool that enables her to achieve fluid-like effects in real-time. These factors are more important than strict physical accuracy, which would require too much computational power.

In fact, most previous models in computer graphics were driven by visual appearance and not by physical accuracy. Early flow models were built from simple primitives. Various combinations of these primitives allowed the animation of particles systems [15, 17] or simple geometries such as leaves [23]. The complexity of the flows was greatly improved with the introduction of random turbulences [16, 20]. These turbulences are mass conserving and, therefore, automatically exhibit rotational motion. Also the turbulence is periodic in space and time, which is ideal for motion "texture mapping" [19]. Flows built up from a superposition of flow primitives all have the disadvantage that they do not respond dynamically to user-applied external forces. Dynamical models of fluids based on the Navier-Stokes equations were first implemented in two-dimensions. Both Yaeger and Upson and Gamito et al. used a vortex method coupled with a Poisson solver to create two-dimensional animations of fluids [24, 8]. Later, Chen et al. animated water surfaces from the pressure term given by a two-dimensional simulation of the Navier-Stokes equations [2]. Their method unlike ours is both limited to two-dimensions and is unstable. Kass and Miller linearize the shallow water equations to simulate liquids [12]. The simplifications do not, however, capture the interesting rotational motions characteristic of fluids. More recently, Foster and Metaxas clearly show the advantages of using the full three-dimensional Navier-Stokes equations in creating fluid-like animations [7]. Many effects which are hard to key frame manually such as swirling motion and flows past objects are obtained automatically. Their algorithm is based mainly on the work of Harlow and Welch in computational fluid dynamics, which dates back to 1965 [11]. Since then many other techniques which Foster and Metaxas could have used have been developed. However, their model has the advantage of being simple to code, since it is based on a finite differencing of the Navier-Stokes equations and an explicit time solver. Similar solvers and their source code are also available from the book of Griebel et al. [9]. The main problem with explicit solvers is that the numerical scheme can become unstable for large time-steps. Instability leads to numerical simulations that "blow-up" and therefore have to be restarted with a smaller time-step. The instability of these explicit algorithms sets serious limits on speed and interactivity. Ideally, a user should be able to interact in real-time with a fluid solver without having to worry about possible "blow ups".

In this paper, for the first time, we propose a stable algorithm that solves the full Navier-Stokes equations. Our algorithm is very easy to implement and allows a user to interact in real-time with three-dimensional fluids on a graphics workstation. We achieve this by using time-steps much larger than the ones used by Foster and Metaxas. To obtain a stable solver we depart from Foster and Metaxas' method of solution. Instead of their explicit Eulerian schemes, we use both Lagrangian and implicit methods to solve the Navier-Stokes equations. Our method cannot be found in the computational fluids literature, since it is custom made for computer graphics applications. The model would not be accurate enough for most engineering applications. Indeed, it suffers from too much "numerical dissipation", i.e., the flow tends to dampen too rapidly

*Alias | wavefront, 1218 Third Ave, 8th Floor, Seattle, WA 98101, U.S.A. jstam@aw.sgi.com

as compared to actual experiments. In a computer graphical application, on the other hand, this is not so bad, especially in an interactive system where the flow is "kept alive" by an animator applying external forces. In fact, a flow which does not dampen at all might be too chaotic and difficult to control. As our results demonstrate we are able to produce nice swirling flows despite the numerical dissipation.

In this paper we apply our flows mainly to the simulation of gaseous-like phenomena. We employ our solver to update both the flow and the motion of densities within the flow. To further increase the complexity of our animations we advect texture coordinates along with the density [13]. In this manner we are able to synthesize highly detailed "wispy" gaseous flows even with low resolution grids. We believe that the combination of physics-based fluid solvers and solid textures is the most promising method of achieving highly complex flows in computer graphics.

The next section presents the Navier-Stokes equations and the derivation which leads to our method of solution. That section contains all the fundamental ideas and techniques needed to obtain a stable fluids solver. Since it relies on sophisticated mathematical techniques, it is written in a mathematical physics jargon which should be familiar to most computer graphics researchers working in physics-based modeling. The application oriented reader who wishes only to implement our solver can skip Section 2 entirely and go straight to Section 3. There we describe our implementation of the solver, providing sufficient information to code our technique. Section 4 is devoted to several applications that demonstrate the power of our new solver. Finally, in Section 5 we conclude and discuss future research. To keep this within the confines of a short paper, we have decided not to include a "tutorial-type" section on fluid dynamics, since there are many excellent textbooks which provide the necessary background and intuition. Readers who do not have a background in fluid dynamics and who wish to fully understand the method in this paper should therefore consult such a text. Mathematically inclined readers may wish to start with the excellent book by Chorin and Marsden [3]. Readers with an engineering bent on the other hand can consult the didactic book by Abbott [1]. Also, Foster and Metaxas' paper does a good job of introducing the concepts from fluid dynamics to the computer graphics community.

2 Stable Navier-Stokes

2.1 Basic Equations

In this section we present the Navier-Stokes equations along with the manipulations that lead to our stable solver. A fluid whose density and temperature are nearly constant is described by a velocity field $\mathbf{u}$ and a pressure field p. These quantities generally vary both in space and in time and depend on the boundaries surrounding the fluid. We will denote the spatial coordinate by $\mathbf{x}$, which for two-dimensional fluids is $\mathbf{x} = (x, y)$ and three-dimensional fluids is equal to (x, y, z). We have decided not to specialize our results for a particular dimension. All results are thus valid for both two-dimensional and three-dimensional flows unless stated otherwise. Given that the velocity and the pressure are known for some initial time $t = 0$, then the evolution of these quantities over time is given by the Navier-Stokes equations [3]:

$$\nabla \cdot \mathbf{u} = 0 \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \tag{2}$$

where ν is the kinematic viscosity of the fluid, ρ is its density and $\mathbf{f}$ is an external force. Some readers might be unfamiliar with this compact version of the Navier-Stokes equations. Eq. 2 is a vector equation for the three (two in two-dimensions) components of the velocity field. The "$\cdot$" denotes a dot product between vectors, while the symbol ∇ is the vector of spatial partial derivatives. More precisely, $\nabla = (\partial/\partial x, \partial/\partial y)$ in two-dimensions and $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$ in three-dimensions. We have also used the shorthand notation $\nabla^2 = \nabla \cdot \nabla$. The Navier-Stokes equations are obtained by imposing that the fluid conserves both mass (Eq. 1) and momentum (Eq. 2). We refer the reader to any standard text on fluid mechanics for the actual derivation. These equations also have to be supplemented with boundary conditions. In this paper we will consider two types of boundary conditions which are useful in practical applications: *periodic* boundary conditions and *fixed* boundary conditions. In the case of periodic boundaries the fluid is defined on an n-dimensional torus ($n = 2, 3$). In this case there are no walls, just a fluid which wraps around. Although such fluids are not encountered in practice, they are very useful in creating evolving texture maps. Also, these boundary conditions lead to a very elegant implementation that uses the fast Fourier transform as shown below. The second type of boundary condition that we consider is when the fluid lies in some bounded domain D. In that case, the boundary conditions are given by a function $\mathbf{u}_D$ defined on the boundary ∂D of the domain. See Foster and Metaxas' work for a good discussion of these boundary conditions in the case of a hot fluid [7]. In any case, the boundary conditions should be such that the normal component of the velocity field is zero at the boundary; no matter should traverse walls.

The pressure and the velocity fields which appear in the Navier-Stokes equations are in fact related. A single equation for the velocity can be obtained by combining Eq. 1 and Eq. 2. We briefly outline the steps that lead to that equation, since it is fundamental to our algorithm. We follow Chorin and Marsden's treatment of the subject (p. 36ff, [3]). A mathematical result, known as the *Helmholtz-Hodge Decomposition*, states that any vector field $\mathbf{w}$ can uniquely be decomposed into the form:

$$\mathbf{w} = \mathbf{u} + \nabla q, \tag{3}$$

where $\mathbf{u}$ has zero divergence: $\nabla \cdot \mathbf{u} = 0$ and q is a scalar field. Any vector field is the sum of a mass conserving field and a gradient field. This result allows us to define an operator $\mathbf{P}$ which projects any vector field $\mathbf{w}$ onto its divergence free part $\mathbf{u} = \mathbf{P}\mathbf{w}$. The operator is in fact defined implicitly by multiplying both sides of Eq. 3 by "∇":

$$\nabla \cdot \mathbf{w} = \nabla^2 q. \tag{4}$$

This is a Poisson equation for the scalar field q with the Neumann boundary condition $\frac{\partial q}{\partial n} = 0$ on ∂D. A solution to this equation is used to compute the projection $\mathbf{u}$:

$$\mathbf{u} = \mathbf{P}\mathbf{w} = \mathbf{w} - \nabla q.$$

If we apply this projection operator on both sides of Eq. 2 we obtain a single equation for the velocity:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}\left(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}\right), \tag{5}$$

where we have used the fact that $\mathbf{P}\mathbf{u} = \mathbf{u}$ and $\mathbf{P}\nabla p = 0$. This is our fundamental equation from which we will develop a stable fluid solver.

2.2 Method of Solution

Eq. 5 is solved from an initial state $\mathbf{u}_0 = \mathbf{u}(\mathbf{x}, 0)$ by marching through time with a time step Δt. Let us assume that the field has been resolved at a time t and that we wish to compute the field at a later time $t + \Delta t$. We resolve Eq. 5 over the time span Δt in four steps. We start from the solution $\mathbf{w}_0(\mathbf{x}) = \mathbf{u}(\mathbf{x}, t)$ of the previous time step and then sequentially resolve each term on the right hand

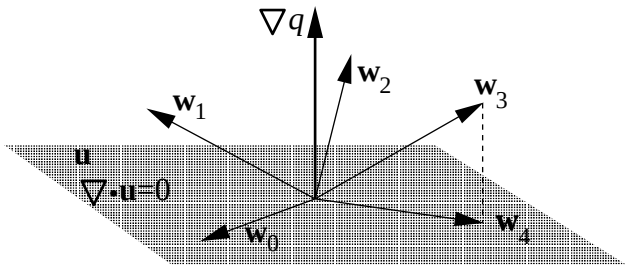


Figure 1: One simulation step of our solver is composed of steps. The first three steps may take the field out of the space of divergent free fields. The last projection step ensures that the field is divergent free after the entire simulation step.

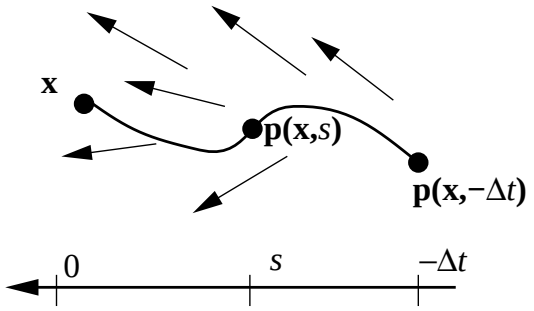


Figure 2: To solve for the advection part, we trace each point of the field backward in time. The new velocity at $\mathbf{x}$ is therefore the velocity that the particle had a time Δt ago at the old location $\mathbf{p}(\mathbf{x}, -\Delta t)$.

side of Eq. 5, followed by a projection onto the divergent free fields. The general procedure is illustrated in Figure 1. The steps are:

$$\mathbf{w}_0(\mathbf{x}) \overset{\text{add force}}{\longrightarrow} \mathbf{w}_1(\mathbf{x}) \overset{\text{advect}}{\longrightarrow} \mathbf{w}_2(\mathbf{x}) \overset{\text{diffuse}}{\longrightarrow} \mathbf{w}_3(\mathbf{x}) \overset{\text{project}}{\longrightarrow} \mathbf{w}_4(\mathbf{x}).$$

The solution at time $t + \Delta t$ is then given by the last velocity field: $\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{w}_4(\mathbf{x})$. A simulation is obtained by iterating these steps. We now explain how each step is computed in more detail.

The easiest term to solve is the addition of the external force $\mathbf{f}$. If we assume that the force does not vary considerably during the time step, then

$$\mathbf{w}_1(\mathbf{x}) = \mathbf{w}_0(\mathbf{x}) + \Delta t\ \mathbf{f}(\mathbf{x}, t)$$

is a good approximation of the effect of the force on the field over the time step Δt. In an interactive system this is a good approximation, since forces are only applied at the beginning of each time step.

The next step accounts for the effect of advection (or convection) of the fluid on itself. A disturbance somewhere in the fluid propagates according to the expression $-(\mathbf{u} \cdot \nabla)\mathbf{u}$. This term makes the Navier-Stokes equations non-linear. Foster and Metaxas resolved this component using finite differencing. Their method is stable only when the time step is sufficiently small such that $\Delta t < \Delta\tau/|\mathbf{u}|$, where $\Delta\tau$ is the spacing of their computational grid. Therefore, for small separations and/or large velocities, very small time steps have to be taken. On the other hand, we use a totally different approach which results in an unconditionally stable solver. No matter how big the time step is, our simulations will never "blow up". Our method is based on a technique to solve partial differential equations known as the *method of characteristics*. Since this method is of crucial importance in obtaining our stable solver, we provide all the mathematical details in Appendix A. The method, however, can be understood intuitively. At each time step all the fluid particles are moved by the velocity of the fluid itself. Therefore, to obtain the velocity at a point $\mathbf{x}$ at the new time $t+\Delta t$, we backtrace the point $\mathbf{x}$ through the velocity field $\mathbf{w}_1$ over a time Δt. This defines a path $\mathbf{p}(\mathbf{x}, s)$ corresponding to a partial streamline of the velocity field. The new velocity at the point $\mathbf{x}$ is then set to the velocity that the particle, now at $\mathbf{x}$, had at its previous location a time Δt ago:

$$\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{p}(\mathbf{x}, -\Delta t)).$$

Figure 2 illustrates the above. This method has several advantages. Most importantly it is unconditionally stable. Indeed, from the above equation we observe that the maximum value of the new field is never larger than the largest value of the previous field. Secondly, the method is very easy to implement. All that is required in practice is a particle tracer and a linear interpolator (see next Section). This method is therefore both stable and simple to implement, two highly desirable properties of any computer graphics fluid solver. We employed a similar scheme to move densities through user-defined velocity fields [19]. Versions of the method of characteristics were also used by other researchers. The application was either employed in visualizing flow fields [13, 18] or improving the rendering of gas simulations [21, 5]. Our application of the technique is fundamentally different, since we use it to update the velocity field, which previous researchers did not dynamically animate.

The third step solves for the effect of viscosity and is equivalent to a diffusion equation:

$$\frac{\partial \mathbf{w}_2}{\partial t} = \nu \nabla^2 \mathbf{w}_2.$$

This is a standard equation for which many numerical procedures have been developed. The most straightforward way of solving this equation is to discretize the diffusion operator ∇^2 and then to do an explicit time step as Foster and Metaxas did [7]. However, this method is unstable when the viscosity is large. We prefer, therefore, to use an implicit method:

$$\left(\mathbf{I} - \nu \Delta t \nabla^2\right) \mathbf{w}_3(\mathbf{x}) = \mathbf{w}_2(\mathbf{x}),$$

where $\mathbf{I}$ is the identity operator. When the diffusion operator is discretized, this leads to a sparse linear system for the unknown field $\mathbf{w}_3$. Solving such a system can be done efficiently, however (see below).

The fourth step involves the projection step, which makes the resulting field divergence free. As pointed out in the previous subsection this involves the resolution of the Poisson problem defined by Eq. 4:

$$\nabla^2 q = \nabla \cdot \mathbf{w}_3 \qquad \mathbf{w}_4 = \mathbf{w}_3 - \nabla q.$$

The projection step, therefore, requires a good Poisson solver. Foster and Metaxas solved a similar equation using a relaxation scheme. Relaxation schemes, though, have poor convergence and usually require many iterations. Foster and Metaxas reported that they obtained good results even after a very small number of relaxation steps. However, since we are using a different method to resolve for the advection step, we must use a more accurate method. Indeed, the method of characteristics is more precise when the field is close to divergent free. More importantly from a visual point of view, the projection step forces the fields to have vortices which result in more swirling-like motions. For these reasons we have used a more accurate solver for the projection step.

The Poisson equation, when spatially discretized, becomes a sparse linear system. Therefore, both the projection and the viscosity steps involve the solution of a large sparse system of equations.

Multigrid methods, for example, can solve sparse linear systems in linear time [10]. Since our advection solver is also linear in time, the complexity of our proposed algorithm is of complexity $O(N)$. Foster and Metaxas' solver has the same complexity. This performance is theoretically optimal since for a complicated fluid, any algorithm has to consult at least each cell of the computational grid.

2.3 Periodic Boundaries and the FFT

When we consider a domain with periodic boundary conditions, our algorithm takes a particularly simple form. The periodicity allows us to transform the velocity into the Fourier domain:

$$\mathbf{u}(\mathbf{x}, t) \longrightarrow \hat{\mathbf{u}}(\mathbf{k}, t).$$

In the Fourier domain the gradient operator "∇" is equivalent to the multiplication by $i\mathbf{k}$, where $i = \sqrt{-1}$. Consequently, both the diffusion step and the projection step are much simpler to solve. Indeed the diffusion operator and the projection operators in the Fourier domain are

$$\mathbf{I} - \nu\Delta t\nabla^2 \longrightarrow 1 + \nu\Delta t k^2 \quad \text{and}$$
$$\mathbf{P}\mathbf{w} \longrightarrow \hat{\mathbf{P}}\hat{\mathbf{w}}(\mathbf{k}) = \hat{\mathbf{w}}(\mathbf{k}) - \frac{1}{k^2}(\mathbf{k} \cdot \hat{\mathbf{w}}(\mathbf{k}))\mathbf{k},$$

where $k = |\mathbf{k}|$. The operator $\hat{\mathbf{P}}$ projects the vector $\hat{\mathbf{w}}(\mathbf{k})$ onto the plane which is normal to the wave number $\mathbf{k}$. The Fourier transform of the velocity of a divergent free field is therefore always perpendicular to its wavenumbers. The diffusion can be interpreted as a low pass filter whose decay is proportional to both the time step and the viscosity. These simple results demonstrate the power of the Fourier transform. Indeed, we are able to completely transcribe our solver in only a couple of lines. All that is required is a particle tracer and a fast Fourier transform (FFT).

FourierStep($\mathbf{w}_0$,$\mathbf{w}_4$,Δt):
add force: $\mathbf{w}_1 = \mathbf{w}_0 + \Delta t\ \mathbf{f}$
advect: $\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{p}(\mathbf{x}, -\Delta t))$
transform: $\hat{\mathbf{w}}_2 = \mathrm{FFT}\{\mathbf{w}_2\}$
diffuse: $\hat{\mathbf{w}}_3(\mathbf{k}) = \hat{\mathbf{w}}_2(\mathbf{k})/(1 + \nu\Delta t k^2)$
project: $\hat{\mathbf{w}}_4 = \hat{\mathbf{P}}\hat{\mathbf{w}}_3$
transform: $\mathbf{w}_4 = \mathrm{FFT}^{-1}\{\hat{\mathbf{w}}_4\}$

Since the Fourier transform is of complexity $O(N \log N)$, this method is theoretically slightly more expensive than a method of solution relying on multi-grid solvers. However, this method is very easy to implement. We have used this algorithm to generate the "liquid textures" of Section 4.

2.4 Moving Substances through the Fluid

A non-reactive substance which is injected into the fluid will be advected by it while diffusing at the same time. Common examples of this phenomenon include the patterns created by milk stirred in coffee or the smoke rising from a cigarette. Let a be any scalar quantity which is moved through the fluid. Examples of this quantity include the density of dust, smoke or cloud droplets, the temperature of a fluid and a texture coordinate. The evolution of this scalar field is conveniently described by an advection diffusion type equation:

$$\frac{\partial a}{\partial t} = -\mathbf{u} \cdot \nabla a + \kappa_a \nabla^2 - \alpha_a a + S_a,$$

where κ_a is a diffusion constant, α_a is a dissipation rate and S_a is a source term. This equation is very similar in form to the Navier-Stokes equation. Indeed, it includes an advection term, a diffusion term and a "force term" S_a. All these terms can be resolved exactly

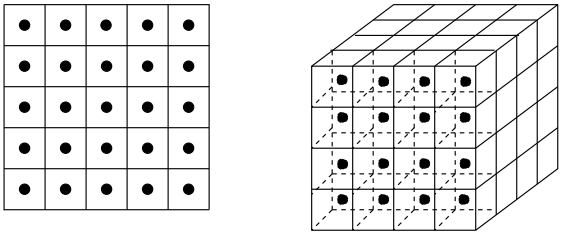


Figure 3: The values of the discretized fields are defined at the center of the grid cells.

in the same way as the velocity of the fluid. The dissipation term not present in the Navier-Stokes equation is solved as follows over a time-step:

$$(1 + \Delta t\alpha_a)a(\mathbf{x}, t + \Delta t) = a(\mathbf{x}, t).$$

Similar equations were used by Stam and Fiume to simulate fire and other gaseous phenomena [21]. However, their velocity fields were not computed dynamically.

We hope that the material in this section has convinced the reader that our stable solver is indeed based on the full Navier-Stokes equations. Also, we have pointed to the numerical techniques which should be used at each step of our solver. We now proceed to describe the implementation of our model in more detail.

3 Our Solver

3.1 Setup

Our implementation handles both the motion of fluids and the propagation by the fluid of any number of substances like mass-density, temperature or texture coordinates. Each quantity is defined on either a two-dimensional (`NDIM=2`) or three-dimensional (`NDIM=3`) grid, depending on the application. The grid is defined by its physical dimensions: origin `O[NDIM]` and length `L[NDIM]` of each side, and by its number of cells `N[NDIM]` in each coordinate. This in turn determines the size of each voxel `D[i]=L[i]/N[i]`. The definition of the grid is an input to our program which is specified by the animator. The velocity field is defined at the center of each cell as shown in Figure 3. Notice that previous researchers, e.g., [7], defined the velocity at the boundaries of the cells. We prefer the cell-centered grid since it is more straightforward to implement. We allocate two grids for each component of the velocity: `U0[NDIM]` and `U1[NDIM]`. At each time step of our simulation one grid corresponds to the solution obtained in the previous step. We store the new solution in the second grid. After each step, the grids are swapped. We also allocate two grids to hold a scalar field corresponding to a substance transported by the flow. Although our implementation can handle any number of substances, for the sake of clarity we present only the algorithm for one field in this section. This scalar quantity is stored in the grids `S0` and `S1`. The speed of interactivity is controlled by a single time step `dt`, which can be as large as the animator wishes, since our algorithm is stable.

The physical properties of the fluid are a function of its viscosity `visc` alone. By varying the viscosity, an animator can simulate a wide range of substances ranging from glue-like matter to highly turbulent flows. The properties of the substance are modeled by a diffusion constant `kS` and a dissipation rate `aS`. Along with these parameters, the animator also must specify the values of these fields on the boundary of the grid. There are basically two types: periodic or fixed. The boundary conditions can be of a different type for each coordinate. When periodic boundary conditions are chosen, the fluid wraps around. This means that a piece of fluid which

leaves the grid on one side reenters the grid on the opposite side. In the case of fixed boundaries, the value of each physical quantity must be specified at the boundary of the grid. The simplest method is to set the field to zero at the boundary. We refer the reader to Foster and Metaxas' paper for an excellent description of different boundary conditions and their resulting effects [7]. In the results section we describe the boundary conditions chosen for each animation. For the special case when the boundary conditions are periodic in each coordinate, a very elegant solver based on the fast Fourier transform can be employed. This algorithm is described in Section 2.3. We do not repeat it here since the solver in this section is more general and can handle both types of boundary conditions.

The fluid is set into motion by applying external forces to it. We have written an animation system in which an animator with a mouse can apply directional forces to the fluid. The forces can also be a function of other substances in the fluid. For example, a temperature field moving through the fluid can produce buoyant and turbulent forces. In our system we allow the user to create all sorts of dependencies between the various fields, some of which are described in the results section of this paper. We do not describe our animation system in great detail since its functionality should be evident from the examples of the next section. Instead we focus on our simulator, which takes the forces and parameters set by the animator as an input.

3.2 The Simulator

Once we worked out the mathematics underlying the Navier-Stokes equations in Section 2, our implementation became straightforward. We wish to emphasize that the theoretical developments of Section 2 are in no way gratuitous but are immensely useful in coding compact solvers. In particular, casting the problem into a mathematical setting has allowed us to take advantage of the large body of work done in the numerical analysis of partial differential equations. We have written the solver as a separate library of routines that are called by the interactive animation system. The entire library consists of only roughly 500 lines of C code. The two main routines of this library update either the velocity field `Vstep` or a scalar field `Sstep` over a given time step. We assume that the external force is given by an array of vectors `F[NDIM]` and that the source is given by an array `Ssource` for the scalar field. The general structure of our simulator looks like

```
while ( simulating ) {
  /* handle display and user interaction */
  /* get forces F and sources Ssource from the UI */
  Swap(U1,U0); Swap(S1,S0);
  Vstep ( U1, U0, visc, F, dt );
  Sstep ( S1, S0, kS, aS, U1, Ssource, dt );
}
```

The velocity solver is composed of four steps: the forces are added to the field, the field is advected by itself, the field diffuses due to viscous friction within the fluid, and in the final step the velocity is forced to conserve mass. The general structure of this routine is:

```
Vstep ( U1, U0, visc, F, dt )
 for(i=0;i<NDIM;i++)
   addForce ( U0[i], F[i], dt );
 for(i=0;i<NDIM;i++)
   Transport ( U1[i], U0[i], U0, dt );
 for(i=0;i<NDIM;i++)
   Diffuse ( U0[i], U1[i], visc, dt );
 Project ( U1, U0, dt );
```

The general structure of the scalar field solver is very similar to the above. It involves four steps: add the source, transport the field by the velocity, diffuse and finally dissipate the field. The scalar field solver shares some of the routines called by the velocity solver:

```
Sstep ( S1, S0, k, a, U, source, dt )
 addForce ( S0, source, dt );
 Transport ( S1, S0, U, dt );
 Diffuse ( S0, S1, k, dt );
 Dissipate ( S1, S0, a, dt );
```

The `addForce` routine adds the force field multiplied by the time step to each value of the field. The dissipation routine `Dissipate` divides each element of the first array by `1+dt*a` and stores it in the new array. The `Transport` routine is a key step in our simulation. It accounts for the movement of the substance due to the velocity field. More importantly it is used to resolve the nonlinearity of the Navier-Stokes equations. The general structure of this routine (in three-dimensions) is

```
Transport ( S1, S0, U, dt )
 for each cell (i,j,k) do
   X = O+(i+0.5,j+0.5,k+0.5)*D;
   TraceParticle ( X, U, -dt, X0 );
   S1[i,j,k] = LinInterp ( X0, S0 );
 end
```

The routine `TraceParticle` traces a path starting at `X` through the field `U` over a time `-dt`. The endpoint of this path is the new point `X0`. We use both a simple second order Runge-Kutta (RK2) method for the particle trace [14] and an adaptive particle tracer, which subsamples the time step only in regions of high velocity gradients, such as near object boundaries. The routine `LinInterp` linearly interpolates the value of the scalar field `S` at the location `X0`. We note that we did not use a higher order interpolation, since this might lead to instabilities due to the oscillations and overshoots inherent in such interpolants. On the other hand, higher order spline approximants may be used, though these tend to smooth out the resulting flows.

To solve for the diffusion (`Diffuse`) and to perform the projection (`Project`) we need a sparse linear solver `SolveLin`. The best theoretical choice is the multi-grid algorithm [10]. However, we used a solver from the FISHPAK library since it was very easy to incorporate into our code and gave good results [22][1]. In practice, it turned out to be faster than our implementation of the multi-grid algorithm. In Appendix B, we show exactly how these routines are used to perform both the `Diffuse` step and the `Project` step. These routines are ideal for domains with no internal boundaries. When complex boundaries or objects are within the flow, one can either use a sophisticated multi-grid solver or a good relaxation routine [9]. In any case, our simulator can easily accomodate new solvers.

4 Results

Our Navier-Stokes solver can be used in many applications requiring fluid-like motions. We have implemented both the two- and the three-dimensional solvers in an interactive modeler that allows a user to interact with the fluids in real-time. The motion is modeled by either adding density into the fluid or by applying forces. The evolution of the velocity and the density is then computed using our solver. To further increase the visual complexity of the flows, we add textural detail to the density. By moving the texture coordinates using the scalar solver as well, we achieve highly detailed flows. To compensate for the high distortions that the texture maps undergo,

[1] FISHPAK is available from `http://www.netlib.org`.

we use three sets of texture coordinates which are periodically reset to their initial (unperturbed) values. At every moment the resulting texture map is the superposition of these three texture maps. This idea was first suggested by Max et al. [13].

Figure 4.(a) shows a sequence of frames from an animation where the user interacts with one of our liquid textures. The figure on the backcover of the SIGGRAPH 1999 proceedings is another frame of a similar sequence with a larger grid size (100^2).

Figures 4.(b) through 4.(g) show frames from various animations that we generated using our three-dimensional solver. In each case the animations were created by allowing the animator to place density and apply forces in real-time. The gases are volume rendered using the three-dimensional hardware texture mapping capabilities of our SGI Octane workstation. We also added a single pass that computes self-shadowing effects from a directional light source in a fixed position. It should be evident that the quality of the renderings could be further improved using more sophisticated rendering hardware or software. Our grid sizes ranged from 16^3 to 30^3 with frame rates fast enough to monitor the animations while being able to control their behavior. In most of these animations we added a "noise" term which is proportional to the amount of density (the factor of proportionality being a user defined parameter). This produced nice billowing motions in some of our animations. In Figures 4.(d)-(e) we used a fractal texture map, while in Figure 4.(g) we used a texture map consisting of evenly spaced lines.

All of our animations were created on an SGI Octane workstation with a R10K processor and 192 Mbytes of memory.

5 Conclusions

The motivation of this paper was to create a general software system that allows an animator to design fluid-like motions in real time. Our initial intention was to base our system on Foster and Metaxas' work. However, the instabilities inherent in their method forced us to develop a new algorithm. Our solver has the property of being unconditionally stable and it can handle a wide variety of fluids in both two- and three-dimensions. The results that accompany this paper clearly demonstrate that our solver is powerful enough to allow an animator to achieve many fluid-like effects. We therefore believe that our solver is a substantial improvement over previous work in this area. The work presented here does not, however, discredit previous, more visually oriented models. In particular, we believe that the combination of our fluid solvers with solid textures, for example, may be a promising area of future research [4]. Our fluid solvers can be used to generate the overall motion, while the solid texture can add additional detail for higher quality animations.

Also we have not addressed the problem of simulating fluids with free boundaries, such as water [6]. This problem is considerably more difficult, since the geometry of the boundary evolves dynamically over time. We hope, however, that our stable solvers may be applied to this problem as well. Also, we wish to extend our solver to finite element boundary-fitted meshes. We are currently investigating such extensions.

Acknowledgments

I would like to thank Marcus Grote for his informed input on fluid dynamics and for pointing me to reference [3]. Thanks to Duncan Brinsmead for his constructive criticisms all along. Thanks also to Pamela Jackson for carefully proofreading the paper and to Brad Clarkson for his help with creating the videos.

A Method of Characteristics

The method of characteristics can be used to solve advection equations of the type

$$\frac{\partial a(\mathbf{x},t)}{\partial t} = -\mathbf{v}(\mathbf{x}) \cdot \nabla a(\mathbf{x},t) \quad \text{and} \quad a(\mathbf{x},0) = a_0(\mathbf{x}),$$

where a is a scalar field, $\mathbf{v}$ is a steady vector field and a_0 is the field at time $t = 0$. Let $\mathbf{p}(\mathbf{x}_0, t)$ denote the *characteristics* of the vector field $\mathbf{v}$ which flow through the point $\mathbf{x}_0$ at $t = 0$:

$$\frac{d}{dt}\mathbf{p}(\mathbf{x}_0,t) = \mathbf{v}(\mathbf{p}(\mathbf{x}_0,t)) \quad \text{and} \quad \mathbf{p}(\mathbf{x}_0,0) = \mathbf{x}_0.$$

Now let $\bar{a}(\mathbf{x}_0,t) = a(\mathbf{p}(\mathbf{x}_0,t),t)$ be the value of the field along the characteristic passing through the point $\mathbf{x}_0$ at $t = 0$. The variation of this quantity over time can be computed using the chain rule of differentiation:

$$\frac{d\bar{a}}{dt} = \frac{\partial a}{\partial t} + \mathbf{v} \cdot \nabla a = 0.$$

This shows that the value of the scalar does not vary along the streamlines. In particular, we have $\bar{a}(\mathbf{x}_0,t) = \bar{a}(\mathbf{x}_0,0) = a_0(\mathbf{x}_0)$. Therefore, the initial field and the characteristics entirely define the solution to the advection problem. The field for a given time t and location $\mathbf{x}$ is computed by first tracing the location $\mathbf{x}$ back in time along the characteristic to get the point $\mathbf{x}_0$, and then evaluating the initial field at that point:

$$a(\mathbf{p}(\mathbf{x}_0,t),t) = a_0(\mathbf{x}_0).$$

We use this method to solve the advection equation over a time interval $[t, t+\Delta t]$ for the fluid. In this case, $\mathbf{v} = \mathbf{u}(\mathbf{x},t)$ and a_0 is any of the components of the fluid's velocity at time t.

B FISHPAK Routines

The linear solver `POIS3D` from FISHPAK is designed to solve a general system of finite difference equations of the type:

```
K1*(S[i-1,j,k]-2*S[i,j,k]+S[i+1,j,k]) +
K2*(S[i,j-1,k]-2*S[i,j,k]+S[i,j+1,k]) +
A[k]*S[i,j,k-1]+B[k]*S[i,j,k]+
C[k]*S[i,j,k+1] = F[i,j,k].
```

For the diffusion solver, with diffusion constant `kappa`, the values of the constants on the left hand side are:

`K1 = -dt*kappa/(D[0]*D[0])`,
`K2 = -dt*kappa/(D[1]*D[1])`,
`A[k] = C[k] = -dt*kappa/(D[2]*D[2])` and
`B[k] = 1+2*dt*kappa/(D[2]*D[2])`,

while the right hand side is equal to the grid containing the previous solution: `F=S0`. In the projection step these constants are equal to

`K1 = 1/(D[0]*D[0])`, `K2 = 1/(D[1]*D[1])`,
`A[k] = C[k] = 1/(D[2]*D[2])` and
`B[k] = -2/(D[2]*D[2])`,

while the right hand side is equal to the divergence of the velocity field:

```
F[i,j,k] = 0.5*(
    (U[0][i+1,j,k]-U[0][i-1,j,k])/D[0]+
    (U[1][i,j+1,k]-U[1][i,j-1,k])/D[1]+
    (U[2][i,j,k+1]-U[2][i,j,k-1])/D[2]).
```

The gradient of the solution is then subtracted from the previous solution:

```
U1[0][i,j,k] = U0[0][i,j,k] -
   0.5*(S[i+1,j,k]-S[i-1,j,k])/D[0],
U1[1][i,j,k] = U0[1][i,j,k] -
   0.5*(S[i,j+1,k]-S[i,j-1,k])/D[1],
U1[2][i,j,k] = U0[2][i,j,k] -
   0.5*(S[i,j,k+1]-S[i,j,k-1])/D[2].
```

The FISHPAK routine is also able to handle different types of boundary conditions, both periodic and fixed.

References

[1] M. B. Abbott. *Computational Fluid Dynamics: An Introduction for Engineers*. Wiley, New York, 1989.

[2] J. X. Chen, N. da Vittoria Lobo, C. E. Hughes, and J. M. Moshell. Real-Time Fluid Simulation in a Dynamic Virtual Environment. *IEEE Computer Graphics and Applications*, pages 52–61, May-June 1997.

[3] A. J. Chorin and J. E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer-Verlag. Texts in Applied Mathematics 4. Second Edition., New York, 1990.

[4] D. Ebert, K. Musgrave, D. Peachy, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. AP Professional, 1994.

[5] D. S. Ebert, W. E. Carlson, and R. E. Parent. Solid Spaces and Inverse Particle Systems for Controlling the Animation of Gases and Fluids. *The Visual Computer*, 10:471–483, 1994.

[6] N. Foster and D. Metaxas. Realistic Animation of Liquids. *Graphical Models and Image Processing*, 58(5):471–483, 1996.

[7] N. Foster and D. Metaxas. Modeling the Motion of a Hot, Turbulent Gas. In *Computer Graphics Proceedings, Annual Conference Series, 1997*, pages 181–188, August 1997.

[8] M. N. Gamito, P. F. Lopes, and M. R. Gomes. Two-dimensional Simulation of Gaseous Phenomena Using Vortex Particles. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 3–15. Springer-Verlag, 1995.

[9] M. Griebel, T. Dornseifer, and T. Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. SIAM, Philadelphia, 1998.

[10] W. Hackbusch. *Multi-grid Methods and Applications*. Springer Verlag, Berlin, 1985.

[11] F. H. Harlow and J. E. Welch. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *The Physics of Fluids*, 8:2182–2189, December 1965.

[12] M. Kass and G. Miller. Rapid, Stable Fluid Dynamics for Computer Graphics. *ACM Computer Graphics (SIGGRAPH '90)*, 24(4):49–57, August 1990.

[13] N. Max, R. Crawfis, and D. Williams. Visualizing Wind Velocities by Advecting Cloud Textures. In *Proceedings of Visualization '92*, pages 179–183, Los Alamitos CA, October 1992. IEEE CS Press.

[14] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C. The Art of Scientific Computing*. Cambridge University Press, Cambridge, 1988.

[15] W. T. Reeves. Particle Systems. A Technique for Modeling a Class of Fuzzy Objects. *ACM Computer Graphics (SIGGRAPH '83)*, 17(3):359–376, July 1983.

[16] M. Shinya and A. Fournier. Stochastic Motion - Motion Under the Influence of Wind. In *Proceedings of Eurographics '92*, pages 119–128, September 1992.

[17] K. Sims. Particle Animation and Rendering Using Data Parallel Computation. *ACM Computer Graphics (SIGGRAPH '90)*, 24(4):405–413, August 1990.

[18] K. Sims. Choreographed Image Flow. *The Journal Of Visualization And Computer Animation*, 3:31–43, 1992.

[19] J. Stam. A General Animation Framework for Gaseous Phenomena. *ERCIM Research Report*, R047, January 1997. http://www.ercim.org/publications/technical_reports/047-abstract.html.

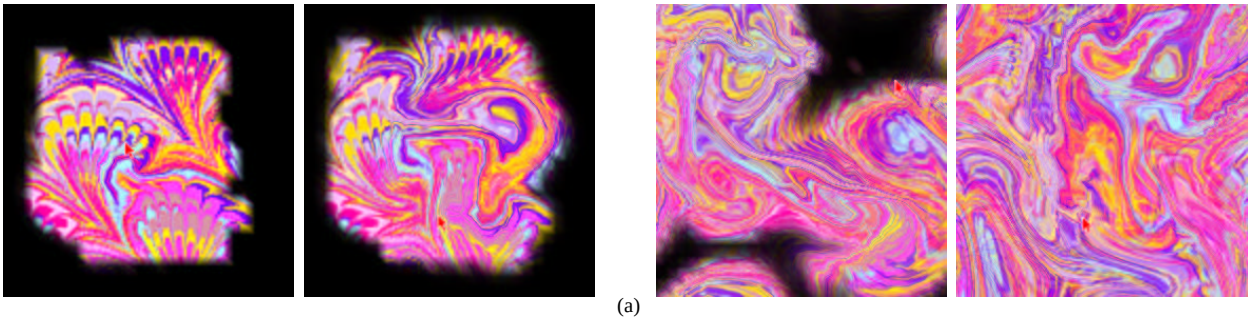
[20] J. Stam and E. Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *SIGGRAPH 93 Conference Proceedings, Annual Conference Series*, pages 369–376, August 1993.

[21] J. Stam and E. Fiume. Depicting Fire and Other Gaseous Phenomena Using Diffusion Processes. In *Proceedings of SIGGRAPH '95*, pages 129–136. Addison-Wesley Publishing Company, August 1995.

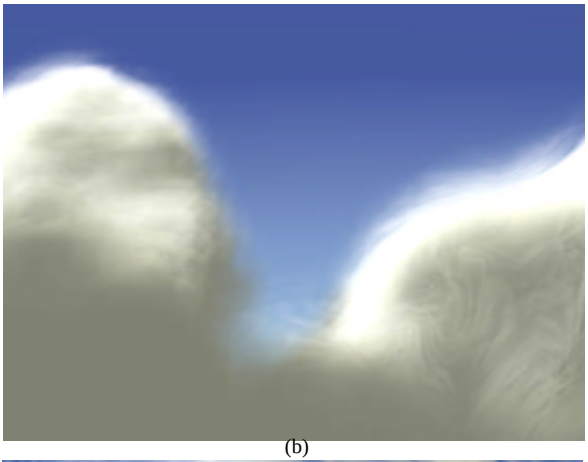
[22] P. N. Swarztrauber and R. A. Sweet. Efficient Fortran Subprograms for the Solution of Separable Elliptic Partial Differential Equations. *ACM Transactions on Mathematical Software*, 5(3):352–364, September 1979.

[23] J. Wejchert and D. Haumann. Animation Aerodynamics. *ACM Computer Graphics (SIGGRAPH '91)*, 25(4):19–22, July 1991.

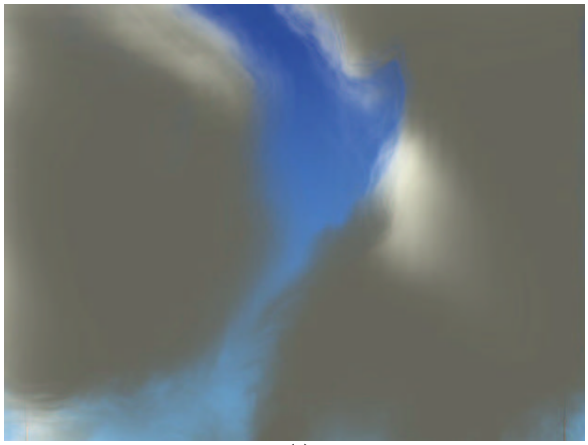
[24] L. Yaeger and C. Upson. Combining Physical and Visual Simulation. Creation of the Planet Jupiter for the Film 2010. *ACM Computer Graphics (SIGGRAPH '86)*, 20(4):85–93, August 1986.



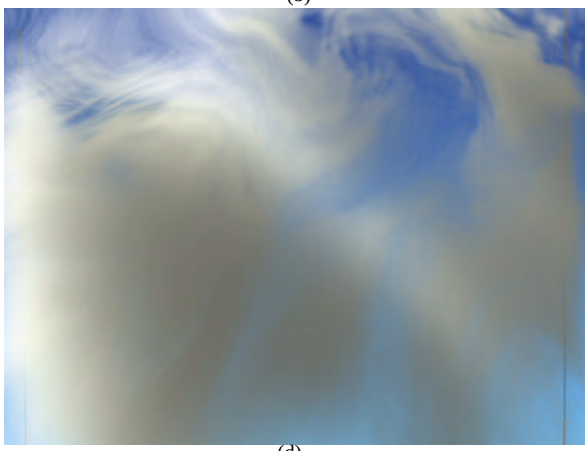
(a)



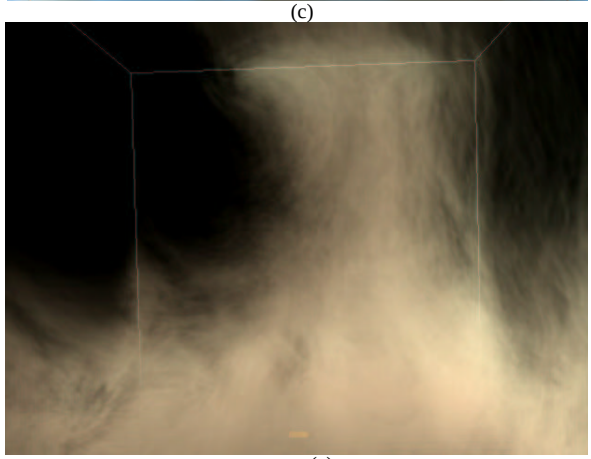
(b)



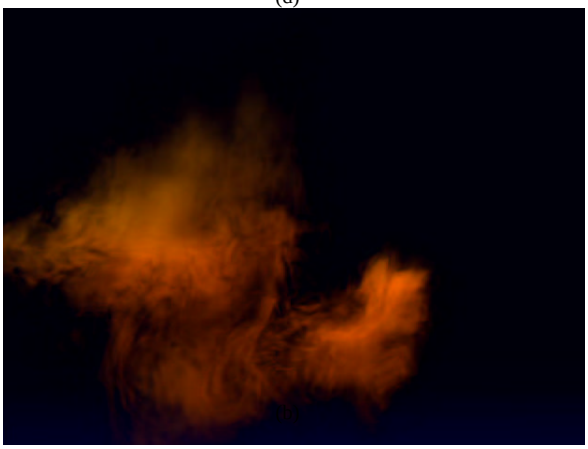
(c)



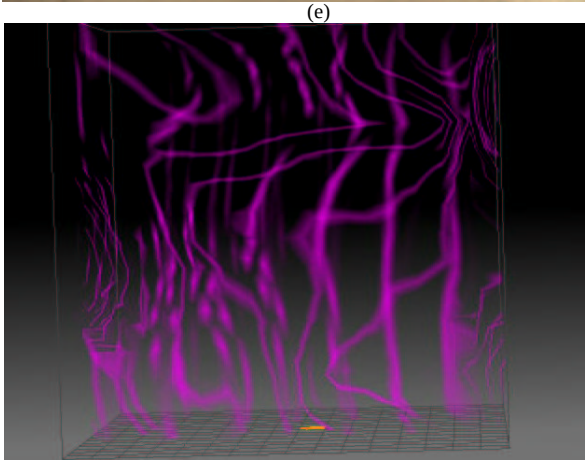
(d)



(e)



(f)



(g)

Figure 4: Snapshots from our interactive fluid solver.

A Simple Fluid Solver based on the FFT

Jos Stam*

Alias | wavefront

Abstract

This paper presents a very simple implementation of a fluid solver. Our solver is consistent with the equations of fluid flow and produces velocity fields that contain incompressible rotational structures and that dynamically react to user-supplied forces. The solver is specialized for a fluid which wraps around in space. This allows us to take advantage of the Fourier transform which greatly simplifies many aspects of the solver. Indeed, given a Fast Fourier Transform, our solver can be implemented in roughly one page of readable C code. We believe that our solver is a good starting point for anyone interested in coding a basic fluid solver. The fluid solver presented is also useful as a basic motion primitive that can be used for many different applications in computer graphics.

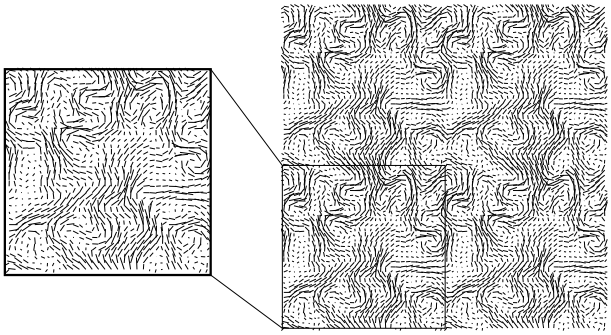


Figure 1: The fluids defined in this paper are periodic. The patch of fluid on the left can be used to seamlessly tile the entire plane.

1 Introduction

Simulating fluids is one of the most challenging problems in engineering. In fact, over the last 50 years much effort has been devoted to writing large fluid simulators that usually run on expensive supercomputers. Simulating fluids is also important to computer graphics, where there is a need to add visually convincing flows to virtual environments. However, while large production houses may be able to afford expensive hardware and large fluid solvers, most computer graphics practitioners require a simple solution that runs on a desktop PC. In this paper we introduce a fluid solver that can be written in roughly one page of C code. The algorithm is based on our Stable Fluid solver [5]. We specialize the algorithm for the case of a periodic domain, where the fluid's boundaries wrap around. In this case, we can use the machinery of Fourier transforms to efficiently solve the equations of fluid flow. Although periodic flows do not exist in Nature, we think that they can be useful in many applications. For example, the resulting fluid flow can be used as a "motion texture map." Similar texture maps based on the FFT have been used in computer graphics to model fractal terrains [9], ocean waves [2, 8], turbulent wind fields [3, 7] and the stochastic motion of flexible structures [4], for example. Also, our solver is a good starting point for someone new to fluid dynamics, since the computational fluid dynamics literature and the complexity of most existing solvers can be quite intimidating to a novice. We warn the reader, however, that our method sacrifices accuracy for speed and stability: our simulations are damped more rapidly than an accurate flow. On the other hand, our simulations do converge to the solutions of the equations of fluid flow in the limit when both the grid spacing and the time step go to zero.

This paper was first published in the journal of graphics tools [6] and is reproduced by permission of the publisher A. K. Peters.

2 Basic Structure of the Solver

The class of fluids that we will simulate in this paper is entirely described by a velocity field that evolves under the action of external forces. The velocity field is defined on a regular grid of N^d voxels, where d is the dimension of the domain. To each voxel's center we assign a velocity. Typically, we consider only two-dimensional and three-dimensional flows. However, our algorithm also works for flows living in an arbitrary d-dimensional space. For the sake of clarity, we describe our algorithm in a two-dimensional space where the underlying concepts are easier to visualize. We assume that our grid has the topology of a torus: the fluid is continuous across adjacent boundaries. Figure 1 illustrates this situation. Because the velocity is periodic, it can be used to tile the entire plane seamlessly. Although such fluids do not occur in Nature, they can be useful in computer graphics as a motion texture map defined everywhere in space.

Our solver updates the grid's velocities over a given time step `dt`. Consequently the simulation is a set of snapshots of the fluid's velocity over time. The simulation is driven by a user-supplied grid of external forces and a viscosity `visc`. The evolution of the fluid is entirely determined by these parameters. Subsequently, at each time step the fluid's grid is updated in four steps:

- Add force field
- Advect velocity
- Diffuse velocity
- Force velocity to conserve mass

The basic idea behind our solver is to perform the first two steps in the spatial domain and the last two steps in the Fourier domain. Before describing each step in more detail, we will say more about the Fourier transform applied to velocity fields.

3 A Fluid in the Fourier domain

The main reason to introduce the Fourier transform is that many operations become very easy to perform in the Fourier domain. This fact is well known in image and signal processing, where convolution becomes a simple multiplication in the Fourier domain. Before introducing Fourier transforms of vector fields we outline the basic

*Alias | wavefront, 1218 Third Ave, 8th Floor, Seattle, WA 98101, U.S.A. jstam@aw.sgi.com

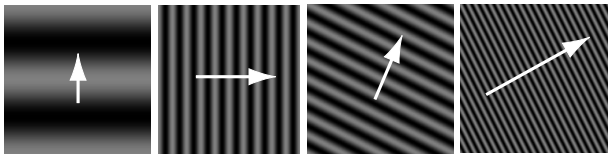


Figure 2: Four planar waves and their corresponding wavenumbers (white arrows).

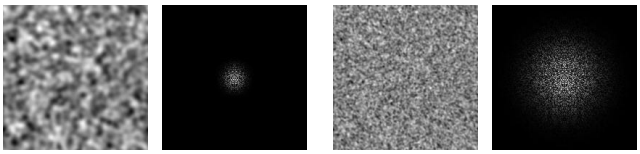


Figure 3: Two examples of a two-dimensional function and its corresponding FFT. The pair on the left is smoother than the one on the right.

ideas for single-valued scalar functions. For a more thorough treatment of the Fourier transform the reader is referred to any of the standard textbooks, e.g., [1]. Informally, a two-dimensional function can be written as a weighted sum of planar waves with different wavenumbers. In Figure 2 we depict four planar waves with their corresponding wavenumbers $\mathbf{k} = (k_x, k_y)$ (shown in white). The direction of the wavenumber is perpendicular to the crests of the wave, while the magnitude of the wavenumber $k = \sqrt{k_x^2 + k_y^2}$ is inversely proportional to the spacing between the waves. Larger wavenumbers therefore correspond to higher spatial frequencies. Any periodic two-dimensional function can be represented as a weighted sum of these simple planar waves. The Fourier transform of the signal is defined as a function that assigns the corresponding weight to each wavenumber $\mathbf{k}$. In Figure 3 we depict two pairs of functions with their corresponding Fourier transform. Notice that the smoother function on the left has a Fourier transform with weights that decay more rapidly for higher spatial frequencies than the one on the right.

To take the Fourier transform of a two-dimensional vector field, we independently take the Fourier transformations of the u components of the vectors and the v components of the vectors, each considered as a two-dimensional scalar field. Figure 4 shows a two-dimensional velocity field in the spatial domain (left) and the corresponding Fourier transform (right). The wavenumbers take values in the range $[-N/2, N/2] \times [-N/2, N/2]$, so that the origin is at the center. Since the velocity field is relatively smooth, we observe that the vectors are smaller for large wavenumbers, which is a common feature of most velocity fields. This observation directly suggests a method to account for the effects of viscosity which tend

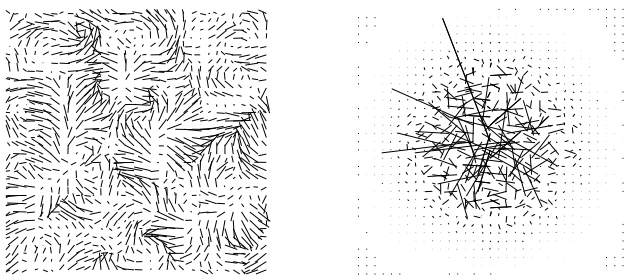


Figure 4: A velocity field (left) and the corresponding Fourier transform (right).

to smooth out the velocity field. Simply multiply the Fourier transform by a filter which decays along with the spatial frequency, the viscosity and the time step (see below).

As mentioned in the previous section not all operations are best performed in the Fourier domain. Therefore, we need a mechanism to switch between the spatial and the Fourier domain. Fortunately this can be done very efficiently using a Fast Fourier Transform (FFT). We will not describe an implementation of the FFT in this paper because very efficient public domain "black box" solvers are readily available. The one used in this paper is MIT's FFTW, the "Fastest Fourier Transform in the West."[1]

4 The Solver

We now describe in more detail the four steps of our solver. In Figure 7 we provide the corresponding implementation in C. Numbers between brackets in this section refer to the corresponding lines in the C-code. The input to our solver is the velocity of the previous time step `(u,v)`, the time step `dt`, a force field defined on a grid `(u0,v0)` and the viscosity of the fluid `visc` `[17-18]`. Because of the slight overhead in memory associated with the FFT, the arrays `u0` and `v0` should be of size `n*(n+2)`. Prior to calling the solver the routine `init_FFT` should be called at least once.

4.1 Adding Forces

This is the easiest step of our solver: we simply add the force grids multiplied by the time step to the velocity grid `[23-26]`. Consequently, in regions where the force field is large the fluid will move more rapidly.

4.2 Self-Advection

This step accounts for the non-linearities present in a fluid flow which makes them unpredictable and chaotic. Fluids such as air can advect (transport) many substances such as dust particles. In the latter case the particle's velocity is equal to that of the fluid at the particle's position. On the other hand, a fluid is also made up of matter. The particles that constitute this matter are also advected by the fluid's velocity. Over a time step these particles will move and transport the fluid's velocity to another location. This effect is called "self-advection": an auto-feedback mechanism within the fluid. To solve for this effect we use a very elegant technique known as a "semi-Lagrangian" scheme in the computational fluid dynamics literature `[28-38]`. Here is how it works. For each voxel of the velocity grid we trace its midpoint backwards through the velocity field over a time step `dt`. This point will end up somewhere else in the grid. We then linearly interpolate the velocity at that point from the neighboring voxels and transfer this velocity back to the departure voxel. This requires two grids, one for the interpolation `(u0,v0)` and one to store the new interpolated values `(u,v)`. To perform the particle trace, we use a simple linear approximation of the exact path. The interpolation is very easy to implement because our grids are periodic. Points that leave the grid simply reenter the grid from the opposite side.

4.3 Viscosity

We already alluded to this step in Section 3. We first transform our velocity field in the Fourier domain `[44]`. We then filter out the higher spatial frequencies by multiplying the Fourier transform by a filter whose decay depends on the magnitude of wavenumber, the viscosity and the time step `[46-60]`. A filter with all these properties which is consistent with the equations of fluid flow is given

[1] Available at `http://www.fftw.org`.

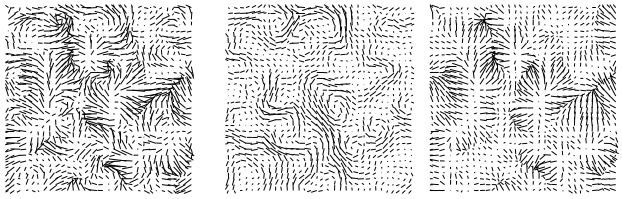


Figure 5: Any velocity field is the sum of a mass conserving field and a gradient field.

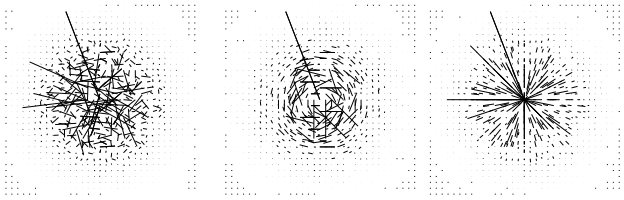


Figure 6: The corresponding decomposition of Fig. 5 in the Fourier domain.

by `exp(-kˆ2*visc*dt)`. Since we are after visual accuracy, other possibly cheaper filters could be used.

4.4 Conservation of Mass

Most fluids encountered in practice conserve mass. Looking at a small piece of fluid, we expect the flow coming in to be equal to the flow coming out. However, after the previous three steps this is rarely the case. We'll correct this in a final step. As for viscosity, it turns out that this step is best performed in the Fourier domain. To solve for this step we use a mathematical result known as the "Helmholtz decomposition" of a velocity field: every velocity field is the sum of a mass conserving field and a gradient field. A gradient field is one that comes from a scalar field and indicates at each point in which direction the scalar field increases most. In two-dimensions the scalar field can be visualized as a height field. In this case, the gradient field simply points to the steepest upward slope. Figure 5 depicts the decomposition for an arbitrary field shown on the left. The first field on the right hand side is the mass-conserving field while the second one is the gradient field. Notice the amount of swirliness in the mass conserving field. Visually, this is clearly the type of behavior that we are seeking in computer graphics. The gradient field, on the other hand, is the worst case scenario: at almost every point the flow is either completely in-flowing or completely out-flowing.

In Figure 6 we show the corresponding fields in the Fourier domain. Notice how these fields have a very simple structure. The velocity of the mass conserving field is always perpendicular to the corresponding wavenumber, while the gradient field is always parallel to its wavenumber. Therefore, in the Fourier domain we force our field to be mass conserving by simply projecting each velocity vector onto the line (or plane in three dimensions) perpendicular to the wavenumber `[53-58]`. This is a very simple operation. The mathematical reason why the mass-conserving and gradient fields are so simple in the Fourier domain follows from the fact that differentiation in the spatial domain corresponds to a multiplication by the wavenumber in the Fourier domain.

Finally after this step we transform the velocity field back into the spatial domain `[62]` and normalize it `[64-67]`.

4.5 Parameters

Typical values for the parameters of the fluid solver are `dt=1`, `visc=0.001` and force values having a magnitude of roughly 10. These values should give nice animations of fluids.

5 Conclusions

We have shown that a simple fluid solver can be coded in roughly one page of C code. The solver as in [5] can be advanced at any speed and is unconditionally stable. This is highly desirable in computer graphics applications. Although the code is only given for a two-dimensional fluid, it can easily be extended to handle three-dimensional or even higher dimensional fluids. Applications of fluid dynamics to dimensions higher than three are unknown to the author. Perhaps the velocity (rate of change) of a set of d parameters might be governed by a fluid equation. We leave this as a topic for future research.

Web Information

The C-code in this paper is available at `http://www.acm.org/jgt/papers/Stam01`.

References

[1] R. N. Bracewell. *The Fourier Transform and its Applications (2nd Edition).* McGrawHill, New York, 1978.

[2] G. A. Mastin, P. A. Watterberg, and J. F. Mareda. Fourier Synthesis of Ocean Scenes. *IEEE Computer Graphics and Applications*, 7(3):16–23, March 1987.

[3] M. Shinya and A. Fournier. Stochastic Motion - Motion Under the Influence of Wind. In *Proceedings of Eurographics '92*, pages 119–128, September 1992.

[4] J. Stam. Stochastic Dynamics: Simulating the Effects of Turbulence on Flexible Structures. *Computer Graphics Forum (EUROGRAPHICS'97 Proceedings)*, 16(3):159–164, 1997.

[5] J. Stam. Stable Fluids. In *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*, pages 121–128, August 1999.

[6] J. Stam. A simple fluid solver based on the FFT. *Journal of graphics tools*, 6(2):43–52, 2001.

[7] J. Stam and E. Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *SIGGRAPH 93 Conference Proceedings, Annual Conference Series*, pages 369–376, August 1993.

[8] J. Tessendorf. Simulating Ocean Water. In *SIGGRAPH'2000: Course Notes 25: Simulating Nature: From Theory to Practice.* 2000.

[9] R. F. Voss. Fractals in nature: From characterization to simulation. In *The Science of Fractal Images*, pages 21–70. Springer-Verlag, New York Berlin Heidelberg, 1988.

```c
#include <srfftw.h>

static rfftwnd_plan plan_rc, plan_cr;

void init_FFT ( int n )
{
   plan_rc = rfftw2d_create_plan ( n, n, FFTW_REAL_TO_COMPLEX, FFTW_IN_PLACE );
   plan_cr = rfftw2d_create_plan ( n, n, FFTW_COMPLEX_TO_REAL, FFTW_IN_PLACE );
}

#define FFT(s,u)\
if (s==1) rfftwnd_one_real_to_complex ( plan_rc, (fftw_real *)u, (fftw_complex *)u );\
else rfftwnd_one_complex_to_real ( plan_cr, (fftw_complex *)u, (fftw_real *)u )

#define floor(x) ((x)>=0.0?((int)(x)):(-((int)(1-(x)))))

void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ ) {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n ) {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n ) {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                       s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                       s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }

   FFT(1,u0); FFT(1,v0);

   for ( i=0 ; i<=n ; i+=2 ) {
      x = 0.5*i;
      for ( j=0 ; j<n ; j++ ) {
         y = j<=n/2 ?  j :  j-n;
         r = x*x+y*y;
         if ( r==0.0 ) continue;
         f = exp(-r*dt*visc);
         U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
         U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
         u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
         u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
         v0[i+  (n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
         v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
      }
   }

   FFT(-1,u0); FFT(-1,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j]; }
}
```

Figure 7: Our fluid solver coded in C.

Visual Simulation of Smoke

Ronald Fedkiw* Jos Stam† Henrik Wann Jensen‡

Stanford University Alias|wavefront Stanford University

Abstract

In this paper, we propose a new approach to numerical smoke simulation for computer graphics applications. The method proposed here exploits physics unique to smoke in order to design a numerical method that is both fast and efficient on the relatively coarse grids traditionally used in computer graphics applications (as compared to the much finer grids used in the computational fluid dynamics literature). We use the inviscid Euler equations in our model, since they are usually more appropriate for gas modeling and less computationally intensive than the viscous Navier-Stokes equations used by others. In addition, we introduce a physically consistent vorticity confinement term to model the small scale rolling features characteristic of smoke that are absent on most coarse grid simulations. Our model also correctly handles the interaction of smoke with moving objects.

1 Introduction

The modeling of natural phenomena such as smoke remains a challenging problem in computer graphics (CG). This is not surprising since the motion of gases such as smoke is highly complex and turbulent. Visual smoke models have many obvious applications in the industry including special effects and interactive games. Ideally, a good CG smoke model should both be easy to use and produce highly realistic results.

Obviously the modeling of smoke and gases is of importance to other engineering fields as well. More generally, the field of computational fluid dynamics (CFD) is devoted to the simulation of gases and other fluids such as water. Only recently have researchers in computer graphics started to excavate the abundant CFD literature for algorithms that can be adopted and modified for computer graphics applications. Unfortunately, current CG smoke models are either too slow or suffer from too much numerical dissipation. In this paper we adapt techniques from the CFD literature specific to the animation of gases such as smoke. We propose a model which is stable, rapid and doesn't suffer from excessive numerical dissipation. This allows us to produce animations of complex rolling smoke even on relatively coarse grids (as compared to the ones used in CFD).

1.1 Previous Work

The modeling of smoke and other gaseous phenomena has received a lot of attention from the computer graphics community over the last two decades. Early models focused on a particular phenomenon and animated the smoke's density directly without modeling its velocity [10, 15, 5, 16]. Additional detail was added using solid textures whose parameters were animated over time. Subsequently, random velocity fields based on a Kolmogoroff spectrum were used to model the complex motion characteristic of smoke [18]. A common trait shared by all of these early models is that they lack any dynamical feedback. Creating a convincing dynamic smoke simulation is a time consuming task if left to the animator.

A more natural way to model the motion of smoke is to simulate the equations of fluid dynamics directly. Kajiya and Von Herzen were the first in CG to do this [13]. Unfortunately, the computer power available at the time (1984) only allowed them to produce results on very coarse grids. Except for some models specific to two-dimensions [21, 9], no progress was made in this direction until the work of Foster and Metaxas [7, 6]. Their simulations used relatively coarse grids but produced nice swirling smoke motions in three-dimensions. Because their model uses an explicit integration scheme, their simulations are only stable if the time step is chosen small enough. This makes their simulations relatively slow, especially when the fluid velocity is large *anywhere* in the domain of interest. To alleviate this problem Stam introduced a model which is unconditionally stable and consequently could be run at any speed [17]. This was achieved using a combination of a semi-Lagrangian advection schemes and implicit solvers. Because a first order integration scheme was used, the simulations suffered from too much numerical dissipation. Although the overall motion looks fluid-like, small scale vortices typical of smoke vanish too rapidly.

Recently, Yngve et al. proposed solving the compressible version of the equations of fluid flow to model explosions [22]. While the compressible equations are useful for modeling shock waves and other compressible phenomena, they introduce a very strict time step restriction associated with the acoustic waves. Most CFD practitioners avoid this strict condition by using the incompressible equations whenever possible. For that reason, we do not consider the compressible flow equations. Another interesting alternative which we do not pursue in this paper is the use of lattice gas solvers based on cellular automata [4].

1.2 Our Model

Our model was designed specifically to simulate gases such as smoke. We model the smoke's velocity with the incompressible Euler equations. These equations are solved using a semi-Lagrangian integration scheme followed by a pressure-Poisson equation as in [17]. This guarantees that our model is stable for any choice of the time step. However, one of our main contributions is a method to reduce the numerical dissipation inherent in semi-Lagrangian schemes. We achieve this by using a technique from the CFD literature known as "vorticity confinement" [20]. The basic idea is to inject the energy lost due to numerical dissipation back into the fluid using a forcing term. This force is designed specifically to increase the vorticity of the flow. Visually this keeps the smoke alive over time. This forcing term is completely consistent with the Euler equations in the sense that it disappears as the number of grid cells is increased. In CFD this technique was applied to the numerical computation of complex turbulent flow fields around helicopters where it is not possible to add enough grid points to accurately resolve the flow field. The computation of the force only adds a small computational overhead. Consequently our simulations are almost as fast as the one's obtained from the basic Stable Fluids algorithm

*Stanford University, Gates Computer Science Bldg., Stanford, CA 94305-9020, fedkiw@cs.stanford.edu

†Alias|wavefront, 1218 Third Ave, 8th Floor, Seattle, WA 98101, U.S.A. jstam@aw.sgi.com

‡Stanford University, Gates Computer Science Bldg., Stanford, CA 94305-9020, henrik@graphics.stanford.edu

[17]. Our model remains stable as long as the magnitude of the forcing term is kept below a certain threshold. Within this range, our time steps are still orders of magnitude higher than the ones used in explicit schemes.

Semi-Lagrangian schemes are very popular in the atmospheric sciences community for modeling large scale flows dominated by constant advection where large time steps are desired, see e.g. [19] for a review. We borrow from this literature a higher order interpolation technique that further increases the quality of the flows. This technique is especially effective when moving densities and temperatures through the velocity field.

Finally, our model, like Foster and Metaxas' [6], is able to handle boundaries inside the computational domain. Therefore, we are able to simulate smoke swirling around objects such as a virtual actor.

The rest of the paper is organized as follows. In the next section we derive our model from the equations of fluid flow, and in section 3 we discuss vorticity confinement. In section 4, we outline our implementation. In section 5, we present both an interactive and a high quality photon map based renderer to depict our smoke simulations. Subsequently, in section 6, we present some results, while section 7 concludes and discusses future work.

2 The Equations of Fluid Flow

At the outset, we assume that our gases can be modeled as inviscid, incompressible, constant density fluids. The effects of viscosity are negligible in gases especially on coarse grids where numerical dissipation dominates physical viscosity and molecular diffusion. When the smoke's velocity is well below the speed of sound the compressibility effects are negligible as well, and the assumption of incompressibility greatly simplifies the numerical methods. Consequently, the equations that model the smoke's velocity, denoted by $\mathbf{u} = (u, v, w)$, are given by the incompressible Euler equations [14]

$$\nabla \cdot \mathbf{u} = 0 \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \mathbf{f}. \tag{2}$$

These two equations state that the velocity should conserve both mass (Equation 1) and momentum (Equation 2). The quantity p is the pressure of the gas and $\mathbf{f}$ accounts for external forces. Also we have arbitrarily set the constant density of the fluid to one.

As in [7, 6, 17] we solve these equations in two steps. First we compute an intermediate velocity field $\mathbf{u}^*$ by solving Equation 2 over a time step Δt without the pressure term

$$\frac{\mathbf{u}^* - \mathbf{u}}{\Delta t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \mathbf{f}. \tag{3}$$

After this step we force the field $\mathbf{u}^*$ to be incompressible using a projection method [3]. This is equivalent to computing the pressure from the following Poisson equation

$$\nabla^2 p = \frac{1}{\Delta t} \nabla \cdot \mathbf{u}^* \tag{4}$$

with pure Neumann boundary condition, i.e., $\frac{\partial p}{\partial \mathbf{n}} = 0$ at a boundary point with normal $\mathbf{n}$. (Note that it is also straightforward to impose Dirichlet boundary conditions where the pressure is specified directly as opposed to specifying its normal derivative.) The intermediate velocity field is then made incompressible by subtracting the gradient of the pressure from it

$$\mathbf{u} = \mathbf{u}^* - \Delta t \nabla p. \tag{5}$$

We also need equations for the evolution of both the temperature T and the smoke's density ρ. We assume that these two scalar quantities are simply moved (advected) along the smoke's velocity

$$\frac{\partial T}{\partial t} = -(\mathbf{u} \cdot \nabla)T, \tag{6}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho. \tag{7}$$

Both the density and the temperature affect the fluid's velocity. Heavy smoke tends to fall downwards due to gravity while hot gases tend to rise due to buoyancy. We use a simple model to account for these effects by defining external forces that are directly proportional to the density and the temperature

$$\mathbf{f}_{\text{buoy}} = -\alpha \rho \mathbf{z} + \beta(T - T_{\text{amb}})\mathbf{z}, \tag{8}$$

where $\mathbf{z} = (0, 0, 1)$ points in the upward vertical direction, T_{amb} is the ambient temperature of the air and α and β are two positive constants with appropriate units such that Equation 8 is physically meaningful. Note that when $\rho = 0$ and $T = T_{\text{amb}}$, this force is zero.

Equations 2, 6 and 7 all contain the advection operator $-(\mathbf{u} \cdot \nabla)$. As in [17], we solve this term using a semi-Lagrangian method [19]. We solve the Poisson equation (Equation 4) for the pressure using an iterative solver. We show in Section 4 how these solvers can also handle bodies immersed in the fluid.

3 Vorticity Confinement

Usually smoke and air mixtures contain velocity fields with large spatial deviations accompanied by a significant amount of rotational and turbulent structure on a variety of scales. Nonphysical numerical dissipation damps out these interesting flow features, and the goal of our new approach is to add them back on the coarse grid. One way of adding them back would be to create a random or pseudo-random small scale perturbation of the flow field using either a heuristic or physically based model. For example, one could generate a divergence free velocity field using a Kolmogorov spectrum and add this to the computed flow field to represent the missing small scale structure (see [18] for some CG applications of the Kolmogorov spectrum). While this provides small scale detail to the flow, it does not place the small scale details in the physically correct locations within the flow field where the small scale details are missing. Instead, the details are added in a haphazard fashion and the smoke can appear to be "alive", rolling and curling in a nonphysical fashion. The key to realistic animation of smoke is to make it look like a passive natural phenomena as opposed to a "living" creature made out of smoke.

Our method looks for the locations within the flow field where small scale features should be generated and adds the small scale features in these locations in a physically based fashion that promotes the passive rolling of smoke that gives it the realistic turbulent look on a coarse CG grid. With unlimited computing power, any consistent numerical method could be used to obtain acceptable results simply by increasing the number of grid points until the desired limiting behavior is observed. However, in practice, computational resources are limited, grids are fairly coarse (even coarser in CG than in CFD), and the discrete difference equations may not be asymptotically close enough to the continuous equations for a particular simulation to behave in the desired physically correct fashion. Our key idea is to design a consistent numerical method that behaves in an interesting and physically plausible fashion on a coarse grid. In general, this is very difficult to do, but luckily a vorticity confinement method was recently invented by Steinhoff, see e.g. [20], for the numerical computation of complex

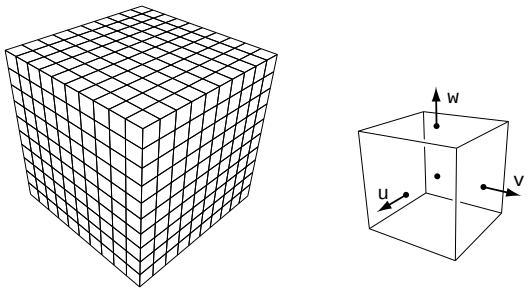


Figure 1: Discretization of the computational domain into identical voxels (left). The components of the velocity are defined on the faces of each voxel (right).

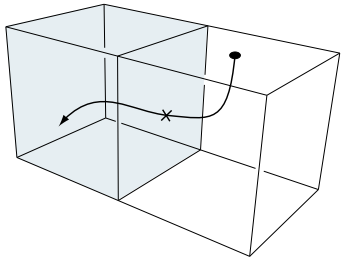


Figure 2: Semi-Lagrangian paths that end up in a boundary voxel are clipped against the boundaries' face.

turbulent flow fields around helicopters where it is not possible to add enough grid points to accurately resolve the flow.

The first step in generating the small scale detail is to identify where it comes from. In incompressible flow, the vorticity

$$\boldsymbol{\omega} = \nabla \times \mathbf{u} \tag{9}$$

provides the small scale structure. Each small piece of vorticity can be thought of as a paddle wheel trying to spin the flow field in a particular direction. Artificial numerical dissipation damps out the effect of these paddle wheels, and the key idea is to simply add it back. First normalized vorticity location vectors

$$\mathbf{N} = \frac{\boldsymbol{\eta}}{|\boldsymbol{\eta}|} \qquad (\boldsymbol{\eta} = \nabla|\boldsymbol{\omega}|) \tag{10}$$

that point from lower vorticity concentrations to higher vorticity concentrations are computed. Then the magnitude and direction of the paddle wheel force is computed as

$$\mathbf{f}_{\mathrm{conf}} = \epsilon\, h\, (\mathbf{N} \times \boldsymbol{\omega}) \tag{11}$$

where $\epsilon > 0$ is used to control the amount of small scale detail added back into the flow field and the dependence on the spatial discretization h guarantees that as the mesh is refined the physically correct solution is still obtained.

This technique was invented by Steinhoff about 10 years ago with a form similar to Equation 11 without the dependence on h, see for example [20]. This method has been used successfully as an engineering model for very complex flow fields, such as those associated with rotorcraft, where one cannot computationally afford to add enough grid points to resolve the important small scale features of the flow.

4 Implementation

We use a finite volume spatial discretization to numerically solve the equations of fluid flow. As shown in Figure 1, we dice up the computational domain into identical voxels. The temperature, the smoke's density and the external forces are defined at the center of each voxel while the velocity is defined on the appropriate voxel faces (see Figure 1, right). Notice that this arrangement is identical to that of Foster and Metaxas [6] but differs from the one used by Stam [17] where the velocity was defined at the voxel centers as well. Our staggered grid arrangement of the velocity field gives improved results for numerical methods with less artificial dissipation. See appendix A for more details on our discretization.

To handle boundaries immersed in the fluid we tag all voxels that intersect an object as being occupied. All occupied voxel cell faces have their velocity set to that of the object. Similarly, the temperature at the center of the occupied voxels is set to the object's temperature. Consequently an animator can create many interesting effects by simply moving or heating up an object. The smoke's density is of course equal to zero inside the object. However, to avoid a sudden drop-off of the density near the object's boundary, we set the density at boundary voxels equal to the density of the closest unoccupied voxel.

Our solver requires two voxel grids for all physical quantities. We advance our simulation by updating one grid from the other over a fixed time step Δt. At the end of each time step we swap these grids. The grid may initially contain some user provided data, but in most cases the grids are simply empty. We first update the velocity components of the fluid. This is done in three steps. First, we add the force fields to the velocity grid. The forces include user supplied fields, the buoyancy force defined by Equation 8 and the new confinement force defined by Equation 11. This is done by simply multiplying each force by the time step and adding it to the velocity (see appendix A). Next we solve for the advection term in Equation 3. We do this using a semi-Lagrangian scheme, see [19] for a review and [17] for its first application in computer graphics.

The semi-Lagrangian algorithm builds a new grid of velocities from the ones already computed by tracing the midpoints of each voxel face through the velocity field. New velocities are then interpolated at these points and their values are transferred to the face cells they originated from. It is possible that the point ends up in one of the occupied voxels. In this case we simply clip the path against the voxel boundary as shown in Figure 2. This guarantees that the point always lies in the unoccupied fluid. Simple linear interpolation is easy to implement and combined with our new confinement force gives satisfactory results. It is also unconditionally stable. Higher order interpolation schemes are, however, desirable in some cases for high quality animations. The tricky part with higher order schemes is that they usually overshoot the data which results in instabilities. In appendix B we provide a cubic interpolator which does not overshoot the data.

Finally we force the velocity field to conserve mass. As already stated in section 2, this involves the solution of a Poisson equation for the pressure (Equation 4). The discretization of this equation results in a sparse linear system of equations. We impose free Neumann boundary conditions at the occupied voxels by setting the normal pressure gradient equal to zero at the occupied boundary faces. The system of equations is symmetric, and the most natural linear solver in this case is the conjugate gradient method. This method is easy to implement and has much better convergence properties

than simple relaxation methods. To improve the convergence we used an incomplete Choleski preconditioner. These techniques are all quite standard and we refer the reader to the standard text [11] for more details. In practice we found that only about 20 iterations of this solver gave us visually acceptable results. After the pressure is computed, we subtract its gradient from the velocity. See appendix A for the exact discretization of the operators involved.

After the velocity is updated, we advect both the temperature and the smoke's density. We solve these equations using again a semi-Lagrangian scheme. In this case, however, we trace back the centers of each voxel. The interpolation scheme is similar to the velocity case.

5 Rendering

For every time step, our simulator outputs a grid that contains the smoke's density ρ. In this section we present algorithms to realistically render the smoke under various lighting conditions. We have implemented both a rapid hardware based renderer as in [17] and a high quality global illumination renderer based on the photon map [12]. The hardware based renderer provides rapid feedback and allows an animator to get the smoke to "look right". The more expensive physics-based renderer is used at the end of the animation pipeline to get production quality animations of smoke.

We first briefly recall the additional physical quantities needed to characterize the interaction of light with smoke. The amount of interaction is modeled by the inverse of the mean free path of a photon before it collides with the smoke and is called the extinction coefficient σ_t. The extinction coefficient is directly related to the density of the smoke through an extinction cross-section C_{ext}: $\sigma_t = C_{\text{ext}}\rho$. At each interaction with the smoke a photon is either scattered or absorbed. The probability of scattering is called the albedo Ω. A value of the albedo near zero corresponds to very dark smoke, while a value near unity models bright gases such as steam and clouds.

In general the scattering of light in smoke is mostly focused in the forward direction. The distribution of scattered light is modeled through a phase function $p(\theta)$ which gives the probability that an incident photon is deflected by an angle θ. A convenient model for the phase function is the Henyey-Greenstein function

$$p(\theta) = \frac{1-g^2}{(1+g^2-2g\cos\theta)^{3/2}}, \tag{12}$$

where the dimensionless parameter $g < 1$ models the anisotropy of the scattering. Values near unity of this parameter correspond to gases which scatter mostly in the forward direction. We mention that this phase function is quite arbitrary and that other choices are possible [1].

5.1 Hardware-Based Renderer

In our implementation of the hardware-based renderer, we follow the algorithm outlined in [17]. In a first pass, we compute the amount of light that directly reaches each voxel of the grid. This is achieved using a fast Bresenham line drawing voxel traversal algorithm [8]. Initially the transparencies of each ray are set to one ($T_{\text{ray}} = 1$). Then, each time a voxel is hit, the transparency is computed from the voxel's density: $T_{\text{vox}} = \exp(-C_{\text{ext}}h)$, where h is the grid spacing. Then the voxel's radiance is set to

$$L_{\text{vox}} = \Omega\, L_{\text{light}}\, (1 - T_{\text{vox}})\, T_{\text{ray}},$$

while the transparency of the ray is simply multiplied by the voxel's transparency: $T_{\text{ray}} = T_{\text{ray}} T_{\text{vox}}$. Since the transparency of the ray diminishes as it traverses the smoke's density this pass correctly mimics the effects of self-shadowing.

In a second pass we render the voxel grid from front to back. We decompose the voxel grid into a set of two-dimensional grid-slices along the coordinate axis most aligned with the viewing direction. The vertices of this grid-slice correspond to the voxel centers. Each slice is then rendered as a set of transparent quads. The color and opacity at each vertex of a quad correspond to the radiance L_{vox} and the opacity $1 - T_{\text{vox}}$, respectively, of the corresponding voxel. The blending between the different grid slices when rendered from front to back is handled by the graphics hardware.

5.2 Photon Map Renderer

Realistic rendering of smoke with a high albedo (such as water vapor) requires a full simulation of multiple scattering of light inside the smoke. This involves solving the full volume rendering equation [2] describing the steady-state of light in the presence of participating media. For this purpose we use the photon mapping algorithm for participating media as introduced in [12]. This is a two pass algorithm in which the first pass consists of building a volume photon map by emitting photons towards the medium and storing these as they interact with the medium. We only store the photons corresponding to indirect illumination.

In the rendering pass we use a forward ray marching algorithm. We have found this to be superior to the backward ray marching algorithm proposed in [12]. The forward ray marching algorithm allows for a more efficient culling of computations in smoke that is obscured by other smoke. In addition it enables a more efficient use of the photon map by allowing us to use less photons in the query as the ray marcher gets deeper into the smoke. Our forward ray marcher has the form

$$L_n(x_n, \vec{\omega}) = L_{n-1}(x_{n-1}, \vec{\omega}) + e^{-\tau(x_n)} \Delta x_n L_s(x'_n, \vec{\omega}) \tag{13}$$

where $\tau(x_n) = \int_{x_0}^{x_n} \sigma_t\, dx$ is the optical depth, L_s is the fraction of the inscattered radiance that is scattered in direction $\vec{\omega}$, $\Delta x_n > 0$ is the size of the nth step, $x_{n+1} = x_n + \Delta x_n$ and x'_n is a randomly chosen location in the nth segment. The factor $e^{-\tau(x_n)}$ can be considered the weight of the nth segment, and we use this value to adjust the required accuracy of the computation.

L_s is given by

$$L_s(x, \vec{\omega}) = \Omega\, \sigma_t(x) \int_{4\pi} L_i(x, \vec{\omega}'_s) p(x'_n, \vec{\omega}', \vec{\omega})\, d\omega' \tag{14}$$

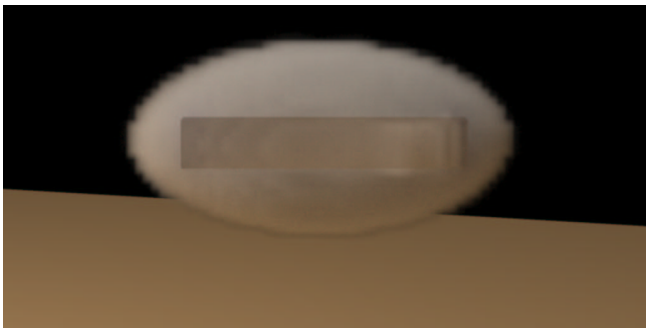
where L_i is the inscattered radiance, and p is the phase function describing the local distribution of the scattered light. We split the inscattered radiance into a single scattering term, L_d, and a multiple scattering term, L_m. The single scattering term is computed using standard ray tracing, and the multiple scattering term is computed using the volume radiance estimate from the photon map by locating the n_p nearest photons from which we get

$$L_m(x, \vec{\omega}) = \frac{1}{\sigma_s} \sum_{1}^{n_p} \frac{\Phi_p(\vec{\omega}') p(x, \vec{\omega}', \vec{\omega})}{\frac{4}{3}\pi r^3}. \tag{15}$$

Here Φ_p is the power of the pth photon and r is the smallest sphere enclosing the n_p photons.

6 Results

This section contains several examples of smoke simulations. We have run most of the simulations including the rendering on a dual-Pentium3-800 or comparable machine. The images in Figures 3-6 have been rendered at a width of 1024 pixels using 4 samples per pixel. These photon map renderings were done using 1-2 million



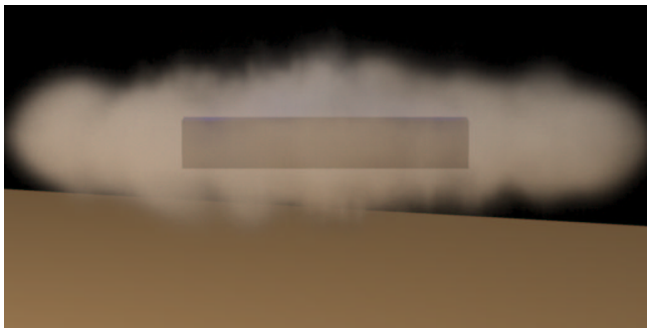


Figure 6: Two stills from the rotor animation. A box is rotating inside the smoke cloud causing it to disperse. Notice how the smoke is sucked in vertically towards the box as it is pushed outwards horizontally. The simulation time for a 120x60x120 grid was roughly 60 seconds/frame.

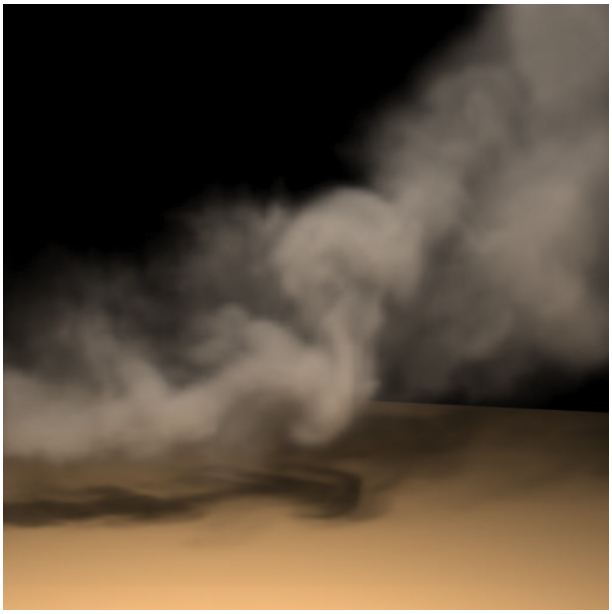


Figure 3: Rising smoke. Notice how the vorticies are preserved in the smoke. The simulation time for a 100x100x40 grid was roughly 30 seconds/frame.

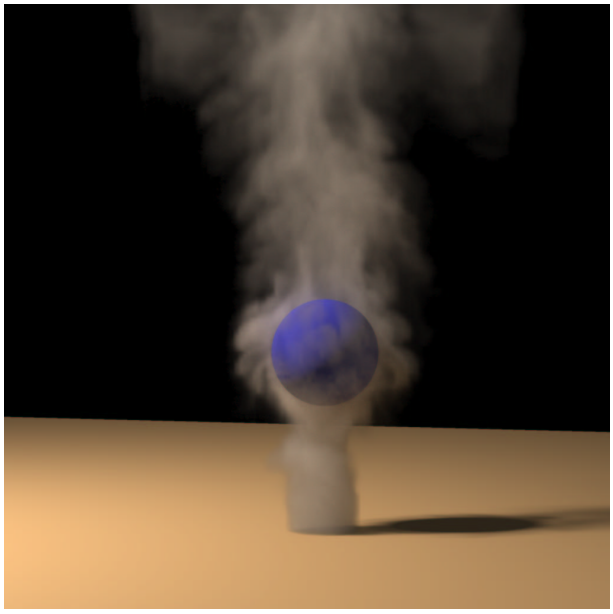


Figure 5: Rising smoke swirling around a sphere. Notice how the smoke correctly moves around the sphere. The simulation time for a 90x135x90 grid was roughly 75 seconds/frame.

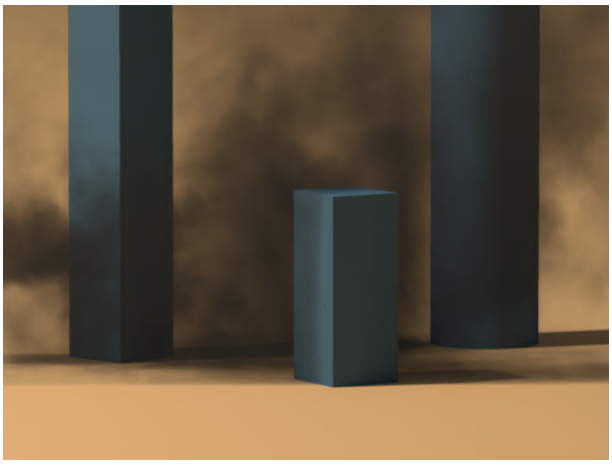


Figure 4: Low albedo smoke passing through several objects. Each object interacts with the smoke and causes local turbulence and vorticity. The simulation time for a 160x80x80 grid was roughly 75 seconds/frame.

Figure 7: Six frames rendered using our interactive hardware renderer of the smoke. The simulation time for a 40x40x40 grid was roughly 1 second/frame.

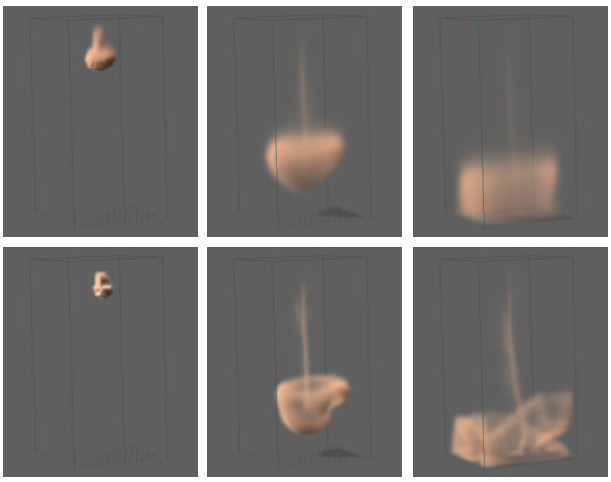


Figure 8: Comparison of linear interpolation (top) and our new monotonic cubic interpolation (bottom). The simulation time for a 20x20x40 grid was roughly 0.1 second/frame (linear) and 1.8 seconds/frame (third order).

photons in the volume photon map and the rendering times for all the photon map images are 20-45 minutes.

Figure 3 is a simple demonstration of smoke rising. The only external force on the smoke is the natural boyancy of the smoke causing it to rise. Notice how even this simple case is enough to create a realistic and swirly apperance of the smoke. Figures 4 and 5 demonstrate that our solver correctly handles the interaction with objects immersed in the smoke. These objects need not be at rest. Figure 6 shows two stills from an animation where a rotating cube is inside a smoke cloud. The rotation of the cube causes the smoke to be pushed out horizontally and sucked in vertically. The grid resolutions and the cost of each time step are reported in the figure captions.

Figure 7 shows six frames of an animation rendered using our interactive renderer. The rendering time for each frame was less than a second on a nVidia Quadro graphics card. The speed, while not real-time, allowed an animator to interactively place densities and heat sources in the scene and watch the smoke raise and billow.

Finally, Figure 8 demonstrates the benefits of using a higher order interpolant in the semi-Lagrangian scheme. The three pictures on the top show the appearance of falling smoke using a linear interpolant, while the pictures on the bottom show the same smoke using our new monotonic cubic interpolant. Clearly the new interpolation reduces the amount of numerical dissipation and produces smoke simulations with more fine detail.

7 Conclusions

In this paper we proposed a new smoke model which is both stable and does not suffer from numerical dissipation. We achieved this through the use of a new forcing term that adds the lost energy back exactly where it is needed. We also included the interaction of objects with our smoke. We believe that our model is ideal for CG applications where visual detail and speed are crucial.

We think that vorticity confinement is a very elegant and powerful technique. We are investigating variants of this technique custom tailored for other phenomena such as fire. We are also investigating techniques to improve the interaction of the smoke with objects. In our current model objects may sometimes be too coarsely sampled on the grid.

8 Acknowledgements

We would like to thank John Steinhoff (Flow Analysis Inc. and UTSI) and Pat Hanrahan (Stanford University) for many helpful discussions. The work of the first author was supported in part by ONR N00014-97-1-0027.

A Discretization

We assume a uniform discretization of space into N^3 voxels with uniform spacing h. The temperature and the smoke's density are both defined at the voxel centers and denoted by

$$T_{i,j,k} \quad \text{and} \quad \rho_{i,j,k}, \qquad i,j,k = 1,\cdots,N,$$

respectively. The velocity on the other hand is defined at the cell faces. It is usual in the CFD literature to use half-way index notation for this

$$\begin{aligned} &u_{i+1/2,j,k}, \quad i = 0,\cdots,N, \quad j,k = 1,\cdots,N, \\ &v_{i,j+1/2,k}, \quad j = 0,\cdots,N, \quad i,k = 1,\cdots,N, \\ &w_{i,j,k+1/2}, \quad k = 0,\cdots,N, \quad i,j = 1,\cdots,N. \end{aligned}$$

Using these notations we can now define some discrete operators. The divergence is defined as

$$\begin{aligned} (\nabla \cdot \mathbf{u})_{i,j,k} \quad = \quad & (u_{i+1/2,j,k} - u_{i-1/2,j,k} + \\ & v_{i,j+1/2,k} - v_{i,j-1/2,k} + \\ & w_{i,j,k+1/2} - w_{i,j,k-1/2})/h \end{aligned}$$

while the discrete gradients are (note $\nabla p = (p_x, p_y, p_z)$)

$$\begin{aligned} (p_x)_{i+1/2,j,k} &= (p_{i+1,j,k} - p_{i,j,k})/h, \\ (p_y)_{i,j+1/2,k} &= (p_{i,j+1,k} - p_{i,j,k})/h, \\ (p_z)_{i,j,k+1/2} &= (p_{i,j,k+1} - p_{i,j,k})/h. \end{aligned}$$

The discrete Laplacian is simply the combination of the divergence and the gradient operators. The discrete version of the vorticity $\boldsymbol{\omega} = (\omega^1, \omega^2, \omega^3)$ is defined as follows. First we compute the cell-centered velocities through averaging

$$\begin{aligned} \bar{u}_{i,j,k} &= (u_{i-1/2,j,k} + u_{i+1/2,j,k})/2, \\ \bar{v}_{i,j,k} &= (v_{i,j-1/2,j} + v_{i,j+1/2,j})/2, \\ \bar{w}_{i,j,k} &= (w_{i,j,k-1/2} + w_{i,j,k+1/2})/2. \end{aligned}$$

Then

$$\begin{aligned} \omega^1_{i,j,k} &= (\bar{w}_{i,j+1,k} - \bar{w}_{i,j-1,k} - \bar{v}_{i,j,k+1} + \bar{v}_{i,j,k-1})/2h, \\ \omega^2_{i,j,k} &= (\bar{u}_{i,j,k+1} - \bar{u}_{i,j,k-1} - \bar{w}_{i+1,j,k} + \bar{w}_{i-1,j,k})/2h, \\ \omega^3_{i,j,k} &= (\bar{v}_{i+1,j,k} - \bar{v}_{i-1,j,k} - \bar{u}_{i,j+1,k} + \bar{u}_{i,j-1,k})/2h. \end{aligned}$$

All of our force fields are defined at the center of the grid voxels. To get values at the faces we simply average again. If the force field $\mathbf{f} = (f^1, f^2, f^3)$, then the velocity is updated as

$$\begin{aligned} u_{i+1/2,j,k} &+= \Delta t\ (f^1_{i,j,k} + f^1_{i+1,j,k})/2, \\ v_{i,j+1/2,k} &+= \Delta t\ (f^2_{i,j,k} + f^2_{i,j+1,k})/2, \\ w_{i,j,k+1/2} &+= \Delta t\ (f^3_{i,j,k} + f^3_{i,j,k+1})/2. \end{aligned}$$

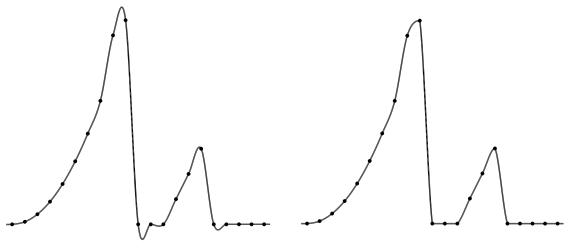


Figure 9: Standard cubic Hermite interpolation (left) produces overshoots while our modified interpolation scheme (right) guarantees that no overshoots occur.

B Monotonic Cubic Interpolation

In this appendix we present a cubic interpolation scheme which does not overshoot the data. Since our voxel grids are regular the three-dimensional interpolation can be broken down into a sequence of one-dimensional interpolations along each coordinate axis. Therefore, it is sufficient to describe the one-dimensional case only. The data consists of a set of values f_k defined at the locations $k = 0, \cdots, N$. A value at a point $t \in [t_k, t_{k+1}]$ can be interpolated using a Hermite interpolant as follows [8]

$$f(t) = a_3(t - t_k)^3 + a_2(t - t_k)^2 + a_1(t - t_k) + a_0,$$

where

$$\begin{aligned} a_3 &= d_k + d_{k+1} - \Delta_k \\ a_2 &= 3\Delta_k - 2d_k - d_{k+1} \\ a_1 &= d_k \\ a_0 &= f_k \end{aligned}$$

and

$$d_k = (f_{k+1} - f_{k-1})/2, \quad \Delta_k = f_{k+1} - f_k.$$

However, this interpolant usually overshoots the data as we show on the left hand side of Figure 9. We want to avoid this, since monotone interpolation guarantees stability. One solution is to simply clip the interpolation against the data, but this results in sharp discontinuities. Another remedy is to force the interpolant to be monotonic over each interval $[t_k, t_{k+1}]$. A necessary condition for this to be the case is that

$$\begin{cases} sign(d_k) = sign(d_{k+1}) = sign(\Delta_k) & \Delta_k \neq 0 \\ d_k = d_{k+1} = 0 & \Delta_k = 0 \end{cases}.$$

In our implementation we first compute Δ_k and then set the slopes to zero whenever they have a sign different from Δ_k. On the right hand side of Figure 9, we show the our new interpolant applied to the same data. Clearly the overshooting problem is fixed.

References

[1] P. Blasi, B. Le Saec, and C. Schlick. A Rendering Algorithm for Discrete Volume Density Objects. *Computer Graphics Forum (EUROGRAPHICS 93 Conference Proceedings)*, 12(3):201–210, 1993.

[2] S. Chandrasekhar. *Radiative Transfer*. Dover, New York, 1960.

[3] A. Chorin. A Numerical Method for Solving Incompressible Viscous Flow Problems. *Journal of Computational Physics*, 2:12–26, 1967.

[4] Y. Dobashi, K. Kaneda, T. Okita, and T. Nishita. A Simple, Efficient Method for Realistic Animation of Clouds. In *SIGGRAPH 2000 Conference Proceedings, Annual Conference Series*, pages 19–28, July 2000.

[5] D. S. Ebert and R. E. Parent. Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques. *Computer Graphics (SIGGRAPH 90 Conference Proceedings)*, 24(4):357–366, August 1990.

[6] N. Foster and D. Metaxas. Realistic Animation of Liquids. *Graphical Models and Image Processing*, 58(5):471–483, 1996.

[7] N. Foster and D. Metaxas. Modeling the Motion of a Hot, Turbulent Gas. In *SIGGRAPH 97 Conference Proceedings, Annual Conference Series*, pages 181–188, August 1997.

[8] J. D. Fowley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice. Second Edition*. Addison-Wesley, Reading, MA, 1990.

[9] M. N. Gamito, P. F. Lopes, and M. R. Gomes. Two-dimensional Simulation of Gaseous Phenomena Using Vortex Particles. In *Proceedings of the 6th Eurographics Workshop on Computer Animation and Simulation*, pages 3–15. Springer-Verlag, 1995.

[10] G. Y. Gardner. Visual Simulation of Clouds. *Computer Graphics (SIGGRAPH 85 Conference Proceedings)*, 19(3):297–384, July 1985.

[11] G. Golub and C. Van Loan. *Matrix Computations.* The John Hopkins University Press, Baltimore, 1989.

[12] H. W. Jensen and P. H. Christensen. Efficient Simulation of Light Transport in Scenes with Participating Media using Photon Maps. In *SIGGRAPH 98 Conference Proceedings, Annual Conference Series*, pages 311–320, July 1998.

[13] J. T. Kajiya and B. P. von Herzen. Ray Tracing Volume Densities. *Computer Graphics (SIGGRAPH 84 Conference Proceedings)*, 18(3):165–174, July 1984.

[14] L. D. Landau and E. M. Lifshitz. *Fluid Mechanics, 2nd edition*. Butterworth-Heinemann, Oxford, 1998.

[15] K. Perlin. An Image Synthesizer. *Computer Graphics (SIGGRAPH 85 Conference Proceedings)*, 19(3):287–296, July 1985.

[16] G. Sakas. Fast Rendering of Arbitrary Distributed Volume Densities. In F. H. Post and W. Barth, editors, *Proceedings of EUROGRAPHICS '90*, pages 519–530. Elsevier Science Publishers B.V. (North-Holland), September 1990.

[17] J. Stam. Stable Fluids. In *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*, pages 121–128, August 1999.

[18] J. Stam and E. Fiume. Turbulent Wind Fields for Gaseous Phenomena. In *SIGGRAPH 93 Conference Proceedings, Annual Conference Series*, pages 369–376, August 1993.

[19] A. Staniforth and J. Cote. Semi-lagrangian integration schemes for atmospheric models: A review. *Monthly Weather Review*, 119:2206–2223, 1991.

[20] J. Steinhoff and D. Underhill. Modification of the euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. *Physics of Fluids*, 6(8):2738–2744, 1994.

[21] L. Yaeger and C. Upson. Combining Physical and Visual Simulation. Creation of the Planet Jupiter for the Film 2010. *Computer Graphics (SIGGRAPH 86 Conference Proceedings)*, 20(4):85–93, August 1986.

[22] G. Yngve, J. O'Brien, and J. Hodgins. Animating explosions. In *SIGGRAPH 2000 Conference Proceedings, Annual Conference Series*, pages 29–36, July 2000.

Physically Based Modeling and Animation of Fire

Duc Quang Nguyen
Stanford University
Industrial Light & Magic
dqnguyen@stanford.edu

Ronald Fedkiw
Stanford University
Industrial Light & Magic
fedkiw@cs.stanford.edu

Henrik Wann Jensen
Stanford University
henrik@graphics.stanford.edu

Abstract

We present a physically based method for modeling and animating fire. Our method is suitable for both smooth (laminar) and turbulent flames, and it can be used to animate the burning of either solid or gas fuels. We use the incompressible Navier-Stokes equations to independently model both vaporized fuel and hot gaseous products. We develop a physically based model for the expansion that takes place when a vaporized fuel reacts to form hot gaseous products, and a related model for the similar expansion that takes place when a solid fuel is vaporized into a gaseous state. The hot gaseous products, smoke and soot rise under the influence of buoyancy and are rendered using a blackbody radiation model. We also model and render the blue core that results from radicals in the chemical reaction zone where fuel is converted into products. Our method allows the fire and smoke to interact with objects, and flammable objects can catch on fire.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically based modeling; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Ray Tracing;

Keywords: flames, fire, smoke, chemical reaction, blackbody radiation, implicit surface, incompressible flow, stable fluids, vorticity confinement

1 Introduction

The modeling of natural phenomena such as fire and flames remains a challenging problem in computer graphics. Simulations of fluid behavior are in demand for special effects depicting smoke, water, fire and other natural phenomena. Fire effects are especially in demand due to the dangerous nature of this phenomenon. Fire simulations are also of interest for virtual reality effects, for example to help train fire fighters or to determine proper placement of exit signs in smoke filled rooms (i.e. so they can be seen). The interested reader is referred to [Rushmeier 1994].

Combustion processes can be loosely classified into two rather distinct types of phenomena: detonations and deflagrations. In both of these processes, chemical reactions convert fuel into hot gaseous products. Deflagrations are low speed events such as the fire and flames we address in this paper, while detonations are high speed events such as explosions where shock waves and other compressible effects are important, see e.g. [Yngve et al. 2000] and [Neff and Fiume 1999]. As low speed events, deflagrations can be modeled using the equations for incompressible flow (as opposed to those for compressible flow). Furthermore, since viscous effects are small, we use the incompressible inviscid Euler equations similar to [Fedkiw et al. 2001]. As noted therein, these equations can be solved efficiently using a semi-Lagrangian stable fluid approach, see e.g. [Staniforth and Cote 1991] and [Stam 1999].

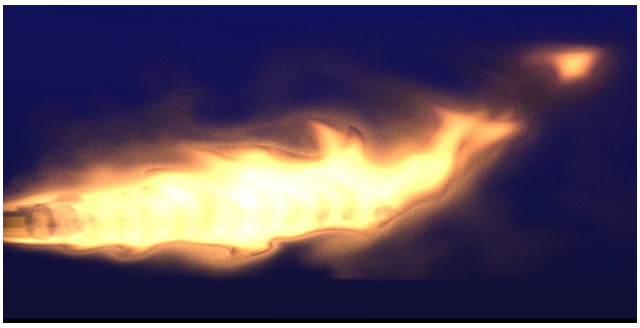


Figure 1: **A turbulent gas flame model of a flamethrower.**

An important, often neglected aspect of fire and flame modeling concerns the expansion of the fuel as it reacts to form hot gaseous products. This expansion is the reason for the visual fullness observed in many flames and is partly responsible for the visual turbulence as well. Since the incompressible equations do not account for expansion, we propose a simple thin flame model for capturing these effects. This is accomplished by using an implicit surface to represent the reaction zone where the gaseous fuel is converted into hot gaseous products. Although real reaction zones have a nonzero (but small) thickness, the thin flame approximation works well for visual modeling and has been used by scientists as well, see for example [Markstein 1964] who first proposed this methodology.

Our implementation of the thin flame model is as follows. First, a dynamic implicit surface is used to track the reaction zone where the gaseous fuel is converted into hot gaseous products. Then both the gaseous fuel and the hot gaseous products are separately modeled using independent sets of incompressible flow equations. Finally, these incompressible flow equations are updated together in a coupled fashion using the fact that both mass and momentum must be conserved as the gas reacts at the interface. While this gives rather pleasant looking laminar (smooth) flames, we include a vorticity confinement term, see [Steinhoff and Underhill 1994] and [Fedkiw et al. 2001], to model the larger scale turbulent flame structures that are difficult to capture on the relatively coarse grids used for efficiency reasons in computer graphics simulations. We also include other features important for visual simulation, such as the buoyancy effects generated by hot gases and the interaction of fire with flammable and nonflammable objects. We render the fire as a participating medium with blackbody radiation using a stochastic ray marching algorithm. In our rendering we pay careful attention

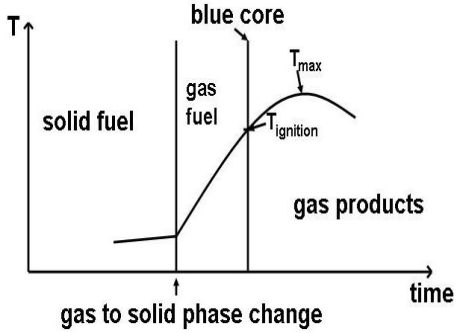


Figure 2: **Flame temperature profile for a solid (or gaseous) fuel.**

to the chromatic adaptation of the observer in order to get the correct colors of the fire.

2 Previous Work

A simple laminar flame was texture mapped onto a flame-like implicit primitive and then volume-traced by [Inakage 1989]. [Perry and Picard 1994] applied a velocity spread model from combustion science to propagate flames. [Chiba et al. 1994] computed the exchange of heat between objects by projecting the environment onto a plane. The spread of flame was a function of both the temperature and the concentration of fuel. [Stam and Fiume 1995] present a similar model in three spatial dimensions for the creation, extinguishing and spread of fire. The spread of the fire is controlled by the amount of fuel available, the geometry of the environment and the initial conditions. Their velocity field is predefined, and then the temperature and density fields are advected using an advection-diffusion type equation. They render the fire using a diffusion approximation which takes into account multiple scattering. [Bukowski and Sequin 1997] integrated the Berkeley Architectural Walkthrough Program with the National Institute of Standards and Technology's CFAST fire simulator. The integrated system creates a simulation based design environment for building fire safety systems.

Although we do not consider high-speed combustion phenomena such as detonations in this paper, there has been some notable work on this subject. [Musgrave 1997] concentrated on the explosive cloud portion of the explosion event using a fractal noise approach. [Neff and Fiume 1999] model and visualize the blast wave portion of an explosion based on a blast curve approach. [Mazarak et al. 1999] discuss the elementary blast wave equations, which were used to model exploding objects. They also show how to incorporate the blast wave model with a rigid body motion simulator to produce realistic animation of flying debris. Most recently, [Yngve et al. 2000] model the propagation of an explosion through the surrounding air using a computational fluid dynamics based approach to solve the equations for compressible, viscous flow. Their system includes two way coupling between solid objects and surrounding fluid, and uses the spectacular brittle fracture technology of [O'Brien and Hodgins 1999]. While the compressible flow equations are useful for modeling shock waves and other compressible phenomena, they introduce a very strict time step restriction associated with the acoustic waves. We use the incompressible flow equations instead to avoid this restriction making our method more computationally efficient.

3 Physically Based Model

We consider three distinct visual phenomena associated with flames. The first of these is the blue or bluish-green core seen in many flames. These colors are emission lines from intermediate chemical species, such as carbon radicals, produced during the chemical reaction. In the thin flame model, this thin blue core is located adjacent to the implicit surface. Therefore, in order to track this blue core, we need to track the movement of the implicit surface. The second visual phenomenon is the blackbody radiation emitted by the hot gaseous products, in particular the carbon soot. This is characterized by the yellowish-orange color we associate with fire. In order to model this with visual accuracy we need to track the temperatures associated with a flame as depicted in figure 2 (read from left to right). If the fuel is solid or liquid, the first step is the heating of the solid until it undergoes a phase change to the gaseous state. (Obviously, for gas fuels, we start in this gaseous state region in the figure.) Then the gas heats up until it reaches its ignition temperature corresponding to our implicit surface and the beginning of the thin blue core region. The temperature continues to increase as the reaction proceeds reaching a maximum before radiative cooling and mixing effects cause the temperature to decrease. As the temperature decreases, the blackbody radiation falls off until the yellowish-orange color is no longer visible. The third and final visual effect we address is the smoke or soot that is apparent in some flames after the temperature cools to the point where the blackbody radiation is no longer visible. We model this effect by carrying along a density variable in a fashion similar to the temperature. One could easily add particles to represent small pieces of soot, but our focus in this paper is the fire, not the smoke. For more details on smoke, see [Foster and Metaxas 1997], [Stam 1999] and [Fedkiw et al. 2001]. Figure 3 shows smoke coupled to our gas flame.

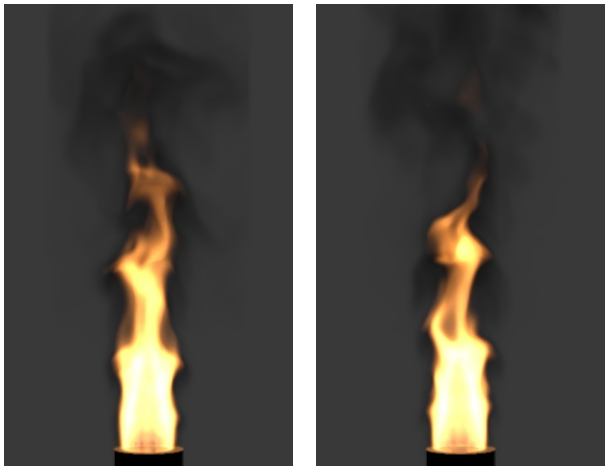


Figure 3: **The hot gaseous products and soot emit blackbody radiation that illuminates the smoke.**

3.1 Blue Core

Our implicit surface separates this gaseous fuel from the hot gaseous products and surrounding air. Consider for example the injection of gaseous fuel from a cylindrically shaped tube. If the fuel were not burning, then the implicit surface would simply move at the same velocity as the gaseous fuel being injected. However, when the fuel is reacting, the implicit surface moves at the velocity of the unreacted fuel plus a flame speed S that indicates how fast fuel is being converted into gaseous products. S indicates how fast the unreacted gaseous fuel is crossing over the implicit surface turning into hot gaseous products. The approximate surface area of the blue core, A_S, can be estimated with the following equation

$$v_f A_f = S A_S, \tag{1}$$

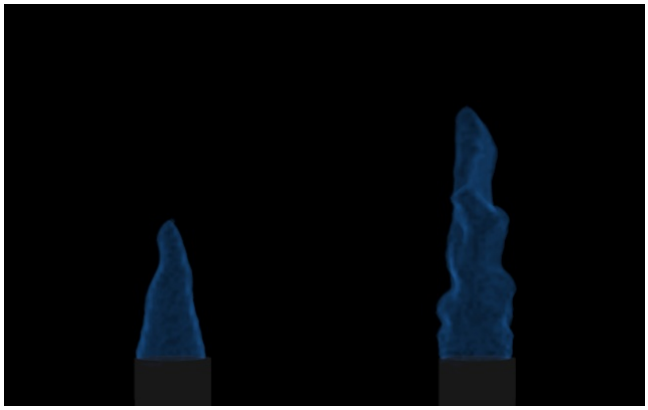


Figure 4: **Blue reaction zone cores for large (left) and small (right) values of the flame reaction speed S. Note the increased turbulence on the right.**

where v_f is the speed the fuel is injected across the injection surface with area A_f, e.g. A_f is the cross section of the cylindrical tube. This equation results from canceling out the density in the equation for conservation of mass. The left hand side is the fuel being injected into the region bounded by the implicit surface, and the right hand side is the fuel leaving this region crossing over the implicit surface as it turns into gaseous products. From this equation, we see that injecting more (less) gas is equivalent to increasing (decreasing) v_f resulting in a larger (smaller) blue core. Similarly, increasing (decreasing) the reaction speed S results in a smaller (larger) blue core. While we can turn the velocity up or down on our cylindrical jet, the reaction speed S is a property of the fuel. For example, S is approximately $.44m/s$ for a propane fuel that has been suitably premixed with oxidizer [Turns 1996]. (We use $S = .5m/s$ for most of our examples.) Figure 4 shows the effect of varying the parameter S. The smaller value of S gives a blue core with more surface area as shown in the figure.

This thin flame approximation is fairly accurate for premixed flames where the fuel and oxidizer are premixed so that the injected gas is ready for combustion. Non-premixed flames, commonly referred to as diffusion flames, behave somewhat differently. In a diffusion flame, the injected fuel has to mix with a surrounding oxidizer before it can combust. Figure 5 shows the injection of fuel out of a cylindrically shaped pipe. The cone shaped curve is the predicted location of the blue core for a premixed flame while the larger rounded curve is the predicted location of the blue core for a diffusion flame. As can be seen in the figure, diffusion flames tend to have larger cores since it takes a while for the injected fuel and surrounding oxidizer to mix. This small-scale molecular diffusion process is governed by a second order partial differential equation that is computationally costly model. Thus for visual purposes,

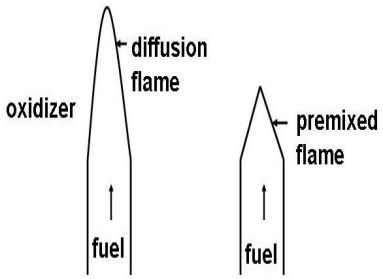


Figure 5: **Location of the blue reaction zone core for a premixed flame versus a diffusion (non-premixed) flame**

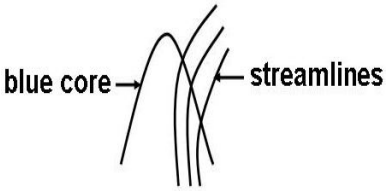


Figure 6: **Streamlines illustrating the path of individual fluid elements as they across the blue reaction zone core. The curved path is caused by the expansion of the gas as it reacts.**

we model diffusion flames with larger blue cores simply by using a smaller value of S than that used for a corresponding premixed flame.

3.2 Hot Gaseous Products

In order to get the proper visual look for our flames, it is important to track individual elements of the flow and follow them through their temperature histories given by figure 2. This is particularly difficult because the gas expands as it undergoes reaction from fuel to hot gaseous products. This expansion is important to model since it changes the trajectories of the gas and the subsequent look and feel of the flame as individual elements go through their temperature profile. Figure 3.2 shows some sample trajectories of individual elements as they cross over the reaction front. Note that individual elements do not go straight up as they pass through the reaction front, but instead turn outward due to the effects of expansion. It is difficult to obtain visually full turbulent flames without modeling this expansion effect. In fact, many practitioners resort to a number of low level hacks (and lots of random numbers) in an attempt to sculpt this behavior, while we obtain the behavior intrinsically by using the appropriate model. The expansion parameter is usually given as a ratio of densities, ρ_f/ρ_h where ρ_f is the density of the gaseous fuel and ρ_h is the density of the hot gaseous products. Figure 7 shows three flames side by side with increasing amounts of expansion from left to right. Note how increasing the expansion makes the flames appear fuller. We used $\rho_f = 1kg/m^3$ (about the density of air) for all three flames with $\rho_h = .2kg/m^3$, $.1kg/m^3$ and $.05kg/m^3$ from left to right.

We use one set of incompressible flow equations to model the fuel and a separate set of incompressible flow equations to model the hot gaseous products and surrounding airflow. We require a model for coupling these two sets of incompressible flow equations together across the interface in a manner that models the expansion that takes place across the reaction front. Given that mass and mo-

Figure 7: **Comparison of flame shapes for differing degrees of gaseous expansion. The amount of expansion increases from left to right making the flame appear fuller and more turbulent.**

mentum are conserved we can derive the following equations for the coupling across the thin flame front:

$$\rho_h(V_h - D) = \rho_f(V_f - D), \quad (2)$$

$$\rho_h(V_h - D)^2 + p_h = \rho_f(V_f - D)^2 + p_f, \quad (3)$$

where V_f and V_h are the normal velocities of the fuel and the hot gaseous products, and p_f and p_h are their pressures. Here, $D = V_f - S$ is the speed of the implicit surface in the normal direction. These equations indicate that both the velocity and the pressure are discontinuous across the flame front. Thus, we will need to exercise caution when taking derivatives of these quantities as is required when solving the incompressible flow equations. (Note that the tangential velocities are continuous across the flame front.)

3.3 Solid Fuels

When considering solid fuels, there are two expansions that need to be accounted for. Besides the expansion across the flame front, a similar expansion takes place when the solid is converted to a gas. However, S is usually relatively small for this reaction (most solids burn slowly in a visual sense), so we can use the boundary of the solid fuel as the reaction front. Since we do not currently model the pressure in solids, only equation 2 applies. We rewrite this equation as

$$\rho_f(V_f - D) = \rho_s(V_s - D), \quad (4)$$

where ρ_s and V_s are the density and the normal velocity of the solid fuel. Substituting $D = V_s - S$ and solving for V_f gives

$$V_f = V_s + \left(\rho_s/\rho_f - 1\right) S \quad (5)$$

indicating that the gasified solid fuel moves at the velocity of the solid fuel plus a correction that accounts for the expansion. We model this phase change by injecting gas out of the solid fuel at the appropriate velocity. This can be used to set arbitrary shaped solid objects on fire as long as they can be voxelized with a suitable surface normal assigned to each voxel indicating the direction of gaseous injection.

In figure 8, we simulate a campfire using two cylindrically shaped logs as solid fuel injecting gas out of the logs in a direction consistent with the local unit surface normal. Note the realistic rolling of the fire up from the base of the log. The ability to inject (or not inject) gaseous fuel out of individual voxels on the surface of a complex solid object allows us to animate objects catching on fire, burn different parts of an object at different rates or not at all (by using spatially varying injection velocities), and extinguish solid fuels simply by turning off the injection velocity. While building an animation system that allows the user to hand paint temporally and spatially varying injection velocities on the surface of solid objects is beyond the scope of this paper, it is the subject of future work.

4 Implementation

We use a uniform discretization of space into N^3 voxels with uniform spacing h. The implicit surface, temperature, density and pressure are defined at the voxel centers and are denoted $\phi_{i,j,k}$, $T_{i,j,k}$, $\rho_{i,j,k}$ and $p_{i,j,k}$ where $i,j,k = 1,\cdots,N$. The velocities are defined at the cell faces and we use half-way index notation: $u_{i+1/2,j,k}$ where $i = 0,\cdots,N$ and $j,k = 1,\cdots,N$; $v_{i,j+1/2,k}$ where $j = 0,\cdots,N$ and $i,k = 1,\cdots,N$; $w_{i,j,k+1/2}$ where $k = 0,\cdots,N$ and $i,j = 1,\cdots,N$.

Figure 8: **Two burning logs are placed on the ground and used to emit fuel. The crossways log on top is not lit so the flame is forced to flow around it.**

4.1 Level Set Equation

We track our reaction zone (blue core) using the level set method of [Osher and Sethian 1988] to track the moving implicit surface. We define ϕ to be positive in the region of space filled with fuel, negative elsewhere and zero at the reaction zone.

The implicit surface moves with velocity $\mathbf{w} = \mathbf{u}_f + S\mathbf{n}$ where $\mathbf{u}_f$ is the velocity of the gaseous fuel and the $S\mathbf{n}$ term governs the conversion of fuel into gaseous products. The local unit normal, $\mathbf{n} = \nabla\phi/|\nabla\phi|$ is defined at the center of each voxel using central differencing to approximate the necessary derivatives, e.g. $\phi_x \approx (\phi_{i+1,j,k} - \phi_{i-1,j,k})/2h$. Standard averaging of voxel face values is used to define $\mathbf{u}_f$ at the voxel centers, e.g. $u_{i,j,k} = (u_{i-1/2,j,k} + u_{i+1/2,j,k})/2$. The motion of the implicit surface is defined through

$$\phi_t = -\mathbf{w}\cdot\nabla\phi \quad (6)$$

and solved at each grid point using

$$\phi^{new} = \phi^{old} - \Delta t\left(w_1\phi_x + w_2\phi_y + w_3\phi_z\right) \quad (7)$$

and an upwind differencing approach to estimate the spatial derivatives. For example, if $w_1 > 0$, $\phi_x \approx (\phi_{i,j,k} - \phi_{i-1,j,k})/h$. Otherwise if $w_1 < 0$, $\phi_x \approx (\phi_{i+1,j,k} - \phi_{i,j,k})/h$. This simple approach is efficient and produces visually appealing blue cores.

To keep the implicit surface well conditioned, we occasionally adjust the values of ϕ in order to keep ϕ a signed distance function with $|\nabla\phi| = 1$. First, interpolation is used to reset the values of ϕ at voxels adjacent to the $\phi = 0$ isocontour (which we don't want to

move since it is the visual location of the blue core). Then we march out from the zero isocontour adjusting the values of ϕ at the other grid points as we cross them. [Tsitsiklis 1995] showed that this could be accomplished in an accurate, optimal and efficient manner solving quadratic equations and sorting points with a binary heap data structure. Later, [Sethian 1996] proposed the finite difference formulation of this algorithm that we currently use.

4.2 Incompressible Flow

We model the flow of the gaseous fuel and the hot gaseous products using a separate set of incompressible Euler equations for each. Incompressibility is enforced through conservation of mass (or volume), i.e. $\nabla \cdot \mathbf{u} = 0$ where $\mathbf{u} = (u, v, w)$ is the velocity field. The equations for the velocity

$$\mathbf{u}_t = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p/\rho + \mathbf{f} \tag{8}$$

are solved for in two parts. First, we use this equation to compute an intermediate velocity $\mathbf{u}^*$ ignoring the pressure term, and then we add the pressure (correction) term using

$$\mathbf{u} = \mathbf{u}^* - \Delta t \nabla p/\rho. \tag{9}$$

The key idea to this splitting method is illustrated by taking the divergence of equation 9 to obtain

$$\nabla \cdot \mathbf{u} = \nabla \cdot \mathbf{u}^* - \Delta t \nabla \cdot (\nabla p/\rho) \tag{10}$$

and then realizing that we want $\nabla \cdot \mathbf{u} = 0$ to enforce mass conservation. Thus the left hand side of equation 10 should vanish leaving a Poisson equation of the form

$$\nabla \cdot (\nabla p/\rho) = \nabla \cdot \mathbf{u}^*/\Delta t \tag{11}$$

that can be solved to find the pressure needed for updating equation 9.

We use a semi-Lagrangian stable fluids approach for finding the intermediate velocity $\mathbf{u}^*$ and refer the reader to [Stam 1999] and [Fedkiw et al. 2001] for the details. Since we use two sets of incompressible flow equations, we need to address the stable fluid update when a characteristic traced back from one set of incompressible flow equations crosses the implicit surface and queries the velocities from the other set of incompressible flow equations. Since the normal velocity is discontinuous across the interface, the straightforward stable fluids approach fails to work. Instead, we need to use the balance equation 2 for conservation of mass to correctly interpolate a velocity.

Suppose we are solving for the hot gaseous products and we interpolate across the interface into a region where a velocity from the gaseous fuel might incorrectly be used. Instead of using this value, we compute a ghost value as follows. First, we compute the normal velocity of the fuel, $V_f = \mathbf{u}_f \cdot \mathbf{n}$. Then we use the balance equation 2 to find a ghost value for V_h^G as

$$V_h^G = V_f + (\rho_f/\rho_h - 1)\, S. \tag{12}$$

Since the tangential velocities are continuous across the implicit surface, we combine this new normal velocity with the existing tangential velocity to obtain

$$\mathbf{u}_h^G = V_h^G \mathbf{n} + \mathbf{u}_f - (\mathbf{u}_f \cdot \mathbf{n})\mathbf{n} \tag{13}$$

as a ghost value for the velocity of the hot gaseous products in the region where only the fuel is defined. This ghost velocity can then be used to correctly carry out the stable fluids update. Since both $\mathbf{n}$ and $\mathbf{u}_f$ are defined throughout the region occupied by the fuel, and ρ_f, ρ_h and S are known constants, a ghost cell value for the hot gaseous products, $\mathbf{u}_h^G$, can be found anywhere in the fuel region (even quite far from the interface) by simply algebraically evaluating the right hand side of equation 13. [Nguyen et al. 2001] showed that this ghost fluid method, invented in [Fedkiw et al. 1999], could be used to compute physically accurate engineering simulations of deflagrations.

After computing the intermediate velocity $\mathbf{u}^*$ for both sets of incompressible flow equations, we solve equation 11 for the pressure and finally use equation 9 to find our new velocity field. Equation 11 is solved by assembling and solving a linear system of equations for the pressure as discussed in more detail in [Foster and Fedkiw 2001] and [Fedkiw et al. 2001]. Once again, we need to exercise caution here since the pressure is discontinuous across the interface. Using the ghost fluid method and equation 3, we can obtain and solve a slightly modified linear system incorporating this jump in pressure. We refer the reader to [Nguyen et al. 2001] for explicit details and a demonstration of the physical accuracy of this approach in the context of deflagration waves.

The temperature affects the fluid velocity as hot gases tend to rise due to buoyancy. We use a simple model to account for these effects by defining external forces that are directly proportional to the temperature

$$\mathbf{f}_{buoy} = \alpha (T - T_{air})\, \mathbf{z}, \tag{14}$$

where $\mathbf{z} = (0, 0, 1)$ points in the upward vertical direction, T_{air} is the ambient temperature of the air and α is positive constant with the appropriate units.

Fire, smoke and air mixtures contain velocity fields with large spatial deviations accompanied by a significant amount of rotational and turbulent structure on a variety of scales. Nonphysical numerical dissipation damps out these interesting flow features, so we aim to add them back on the coarse grid. We use the vorticity confinement technique invented by Steinhoff (see e.g. [Steinhoff and Underhill 1994]) and used by [Fedkiw et al. 2001] to generate the swirling effects for smoke. The first step in generating the small scale detail is to identify the vorticity $\boldsymbol{\omega} = \nabla \times \mathbf{u}$ as the source of this small scale structure. Each small piece of vorticity can be thought of as a paddle wheel trying to spin the flow field in a particular direction. Normalized vorticity location vectors, $\mathbf{N} = \nabla|\boldsymbol{\omega}|/|\nabla|\boldsymbol{\omega}||$ simply point from lower concentrations of vorticity to higher concentrations. Using these, the magnitude and direction of the vorticity confinement (paddle wheel) force is computed as

$$\mathbf{f}_{conf} = \varepsilon h (\mathbf{N} \times \boldsymbol{\omega}), \tag{15}$$

where $\varepsilon > 0$ and is used to control the amount of small scale detail added back into the flow field. The dependence on h guarantees that as the mesh is refined the physically correct solution is still obtained. All these quantities can be evaluated in a straightforward fashion as outlined in [Fedkiw et al. 2001].

Usually a standard CFL time step restriction dictates that the time step $\triangle t$ should be limited by $\triangle t < h/|\mathbf{u}|_{max}$ where $|\mathbf{u}|_{max}$ is the maximum velocity in the flow field. While this is true for our level set equation 6 with $\mathbf{u}$ replaced by $\mathbf{w}$, the combination of the semi-Lagrangian discretization and the ghost fluid method allows us to take a much larger time step for the incompressible flow equations. We choose our incompressible flow time step to be about five times bigger than that dictated by applying the CFL condition to the level set equation, and then stably update ϕ using substeps. This reduces the number of times one needs to solve for the pressure, which is the most expensive part of the calculation, by a factor of five.

4.3 Temperature and Density

The temperature profile has great effect on how we visually perceive flames, and we need to generate a temperature time history for fluid elements that behaves as shown in figure 2. Since this figure depicts a time history of the temperature of fluid elements, we

need a way to track individual fluid elements as they cross over the blue core and rise upward due to buoyancy. In particular, we need to know how much time has elapsed since a fluid element has passed through the blue core so that we can assign an appropriate temperature to it. This is easily accomplished using a reaction coordinate variable Y governed by the equation

$$Y_t = -(\mathbf{u}\cdot\nabla)Y - k, \tag{16}$$

where k is a positive constant which we take to be 1 (larger or smaller values can be used to get a good numerical variation of Y in the flame). Ignoring the convection term, $Y_t = -1$ can be solved exactly to obtain $Y(t) = -t + Y(0)$. If we set $Y(0) = 1$ in the region of space occupied by the gaseous fuel and solve equation 16 for Y, then the local value of $1-Y$ is equal to the total time elapsed since a fluid element crossed over the blue reaction core.

We solve equation 16 using the semi-Lagrangian stable fluids method to first update the convection term obtaining an intermediate value Y^*. Then we separately integrate the source term analytically so it too is stable for large time steps, i.e. $Y^{new} = -k\Delta t + Y^*$.

We can now use the values of Y to assign temperature values to the flow. Since $T_{ignition}$ is usually below the visual blackbody emission threshold, the temperature we set inside the blue core is usually not important. Therefore, we can set $T = T_{ignition}$ for the points inside the blue core. The region between the blue core and the maximum temperature in figure 2 is important since it models the rise in temperature due to the progress of a complex chemical reaction (which we do not model for the sake of efficiency). Here the animator has a lot of freedom to sculpt temperature rise curves and adjust how the mapping corresponds to the local Y values. For example, one could use $T = T_{ignition}$ at $Y = 1$, $T = T_{max}$ at $Y = .9$ and use a linear temperature function for the in between values of $Y \in (.9, 1)$. For large flames, this temperature rise interval will be compressed too close to the blue core for our grid to resolve. In these instances we use the ghost fluid method to set $T = T_{max}$ for any characteristic that looks across the blue core into the gaseous fuel region. The blue core then "spits" out gas at the maximum temperature that immediately starts to cool off, i.e. there is no temperature rise region. In fact, we did not find it necessary to use the temperature rise region in our examples as we are interested in larger scale flames, but this temperature rise region would be useful, for example, when modeling candle.

The animator can also sculpt the temperature falloff region to the right of figure 2. However, there is a physically correct, viable (i.e. computationally cheap) alternative. For the values of Y in the temperature falloff region, we simply solve

$$T_t = -(\mathbf{u}\cdot\nabla)T - c_T\left(\frac{T - T_{air}}{T_{max} - T_{air}}\right)^4 \tag{17}$$

which is derived from conservation of energy. Similar to equation 16, we solve this equation by first using the semi-Lagrangian stable fluids method to solve for the convection term. Then we integrate the fourth power term analytically to cool down the flame at a rate governed by the cooling constant c_T.

Similar to the temperature curve in figure 2, the animator can sculpt a density curve for smoke and soot formation. The density should start low and increase as the reaction proceeds. In the temperature falloff region, the animator can switch from the density curve to a physically correct equation

$$\rho_t = -(\mathbf{u}\cdot\nabla)\rho \tag{18}$$

that can (once again) be solved using the semi-Lagrangian stable fluids method. Again, we did not find it necessary to sculpt densities for our particular examples.

5 Rendering of Fire

Fire is a participating medium. It is more complex than the types of participating media (e.g. smoke and fog) that are typically encountered in computer graphics since fire emits light. The region that creates the light-energy typically has a complex shape, which makes it difficult to sample. Another complication with fire is that the fire is bright enough that our eyes adapt to its color. This chromatic adaptation is important to account for when displaying fire on a monitor. In this section, we will first describe how we simulate the scattering of light within a fire-medium. Then, we will detail how to properly integrate the spectral distribution of power in the fire and account for chromatic adaptation.

5.1 Light Scattering in a Fire Medium

Fire is a blackbody radiator and a participating medium. The properties of a participating medium are described by the scattering, absorption and emission properties. Specifically, we have the scattering coefficient, σ_s, the absorption coefficient, σ_a, and the extinction coefficient, $\sigma_t = \sigma_a + \sigma_s$. These coefficients specify the amount of scattering, absorption and extinction per unit-distance for a beam of light moving through the medium. The spherical distribution of the scattered light at a location is specified by a phase-function, p. We use the Henyey-Greenstein phase-function [Henyey and Greenstein 1941]

$$p(\vec{\omega}\cdot\vec{\omega}') = \frac{1-g^2}{4\pi(1+g^2-2g\vec{\omega}\cdot\vec{\omega}')^{1.5}}. \tag{19}$$

Here, $g \in [-1, 1]$ is the scattering anisotropy of the medium, $g > 0$ is forward scattering, $g < 0$ is backward scattering, while $g = 0$ is isotropic scattering. Note that the distribution of the scattered light only depends on the angle between the incoming direction, $\vec{\omega}$, and the outgoing direction, $\vec{\omega}'$.

Light transport in participating media is described by an integro-differential equation, the radiative transport equation [Siegel and Howell 1981]:

$$\begin{aligned}(\vec{\omega}\cdot\nabla)L_\lambda(x,\vec{\omega}) \quad = \quad & -\sigma_t(x)L_\lambda(x,\vec{\omega}) + \\ & \sigma_s(x)\int_{4\pi} p(\vec{\omega},\vec{\omega}')L_\lambda(x,\vec{\omega}')d\vec{\omega}' + \\ & \sigma_a(x)L_{e,\lambda}(x,\vec{\omega}).\end{aligned} \tag{20}$$

Here, L_λ is the spectral radiance, and $L_{e,\lambda}$ is the emitted spectral radiance. Note that σ_s, σ_a, and σ_t vary throughout the medium and therefore depend on the position x.

We solve Equation 20 to estimate the radiance distribution in the medium by using a stochastic adaptive ray marching algorithm which recursively samples multiple scattering. In highly scattering media this approach is costly; however, we are concerned about fire which is a blackbody radiator (no scattering, only absorption) that creates a low-albedo smoke (the only scattering part of the fire-medium). This makes the Monte Carlo ray tracing approach practical.

To estimate the radiance along a ray traversing the medium, we split the ray into short segments. For a given segment, n, the scattering properties of the medium are assumed constant, and the radiance, L_n, at the start of the segment is computed as:

$$\begin{aligned}L_{n,\lambda}(x,\vec{\omega}) \quad = \quad & e^{-\sigma_t\Delta x}L_{(n-1),\lambda}(x+\Delta x,\vec{\omega}) + \\ & L_\lambda(x,\vec{\omega}')p(\vec{\omega}\cdot\vec{\omega}')\sigma_s\Delta x + \\ & \sigma_a L_{e,\lambda}(x)\Delta x.\end{aligned} \tag{21}$$

This equation is evaluated recursively to compute the total radiance at the origin of the ray. Δx is the length of the segment, L_{n-1} is the radiance at the beginning of the next segment, and $\vec{\omega}'$ is a sample

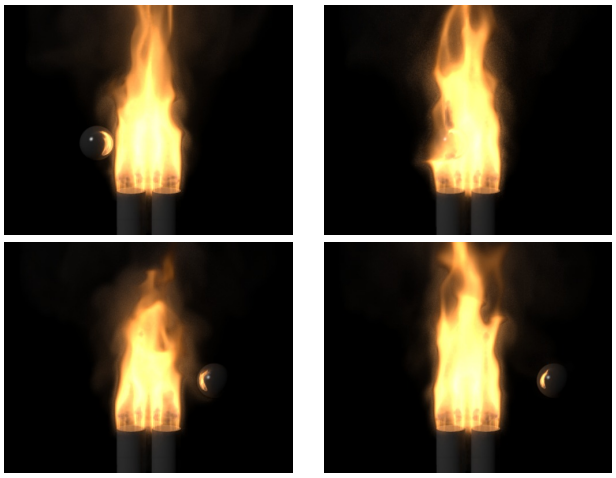


Figure 9: **A metal ball passes through and interacts with a gas flame.**

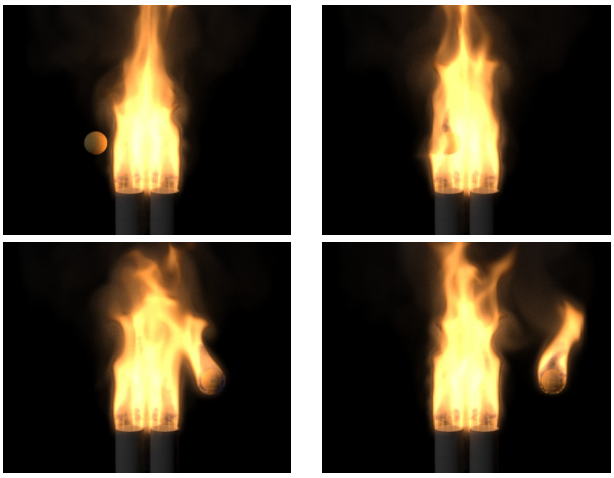


Figure 10: **A flammable ball passes through a gas flame and catches on fire.**

direction for a new ray that evaluates the indirect illumination in a given direction for the segment. We find the sample direction by importance sampling the Henyey-Greenstein phase function. Note that we do not explicitly sample the fire volume; instead we rely on the Monte Carlo sampling to pick up energy as sample rays hit the fire. This strategy is reasonably efficient in the presence of the low-albedo smoke generated by the fire.

The emitted radiance is normally ignored in graphics, but for fire it is an essential component. For a blackbody we can compute the emitted spectral radiance using Planck's formula:

$$L_{e,\lambda}(x) = \frac{2C_1}{\lambda^5(e^{C_2/(\lambda T)} - 1)}, \tag{22}$$

where T is the temperature, $C_1 \approx 3.7418 \cdot 10^{-16} W m^2$, and $C_2 \approx 1.4388 \cdot 10^{-2} m^{\circ} K$ [Siegel and Howell 1981]. In the next section, we will describe how to properly render fire taking this spectral distribution of emitted radiance into account.

5.2 Reproducing the Color of Fire

Accurately reproducing the colors of fire is critical for a realistic fire rendering. The full spectral distribution can be obtained directly by using Planck's formula for spectral radiance when performing the ray marching. This spectrum can then be converted to RGB before being displayed on a monitor. To get the right colors of fire out of this process it is necessary to take into account the fact that our eyes adapt to the spectrum of the fire.

To compute the chromatic adaptation for fire, we use a von Kries transformation [Fairchild 1998]. We assume that the eye is adapted to the color of the spectrum for the maximum temperature present in the fire. We map the spectrum of this white point to the LMS cone responsivities (L_w, M_w, S_w). This enables us to map a spectrum to the monitor as follows. We first integrate the spectrum to find the raw XYZ tristimulus values (X_r, Y_r, Z_r). We then find the adapted XYZ tristimulus values (X_a, Y_a, Z_a) as:

$$\begin{bmatrix} X_a \\ Y_a \\ Z_a \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} 1/L_w & 0 & 0 \\ 0 & 1/M_w & 0 \\ 0 & 0 & 1/S_w \end{bmatrix} \mathbf{M} \begin{bmatrix} X_r \\ Y_r \\ Z_r \end{bmatrix}. \tag{23}$$

Here, $\mathbf{M}$ maps the XYZ colors to LMS (consult [Fairchild 1998] for the details). Finally, we map the adapted XYZ tristimulus values to the monitor RGB space using the monitor white point.

In our implementation, we integrate the spectrum of the blackbody at the source (e.g. when emitted radiance is computed); we then map this spectrum to RGB before using it in the ray marcher. This is much faster than doing a full spectral participating media simulation, and we found that it is sufficiently accurate, since we already assume that the fire is the dominating light source in the scene when doing the von Kries transformation.

6 Results

Figure 1 shows a frame from a simulation of a flamethrower. We used a domain that was 8 meters long with 160 grid cells in the horizontal direction ($h = .05$). The flame was injected at $30m/s$ out of a cylindrical pipe with diameter $.4m$. We used $S = .1m/s$, $rho_f = 1kg/m^3$, $rho_h = .01kg/m^3$, $c_T = 3000K/s$ and $\alpha = .15m/(Ks^2)$. The vorticity confinement parameter was set to $\varepsilon = 16$ for the gaseous fuel and to $\varepsilon = 60$ for the hot gaseous products. The simulation cost was approximately 3 minutes per frame using a Pentium IV.

Solid objects are treated by first tagging all the voxels inside the object as occupied. Then all the occupied voxel cell faces have their velocity set to that of the object. The temperature at the center of occupied voxels is set to the object's temperature and the (smoke) density is set to zero. Figure 9 shows a metal sphere as it passes through and interacts with a gas fire. Note the reflection of the fire on the surface of the sphere. For more details on object interactions with liquids and gases see [Foster and Fedkiw 2001] and [Fedkiw et al. 2001].

Since we have high temperatures (i.e. fire) in our flow field, we allow our objects to heat up if their temperature is lower than that of their surroundings. We use a simple conduction model where we increase the local temperature of an object depending on the surrounding air temperature and object temperature as well as the time step Δt. Normally, the value of the implicit surface is set to a negative value of h at the center of all voxels occupied by objects indicating that there is no available fuel. However, we can easily model ignition for objects we designate as flammable. Once the temperature of a voxel inside an object increases above a predefined threshold indicating ignition, we change the value of the implicit surface in that voxel from $-h$ to h indicating that it contains fuel. In addition, those voxel's faces have their velocities augmented above the object velocity by an increment in the direction normal to the object surface indicating that gaseous fuel is being injected according to the phase change addressed earlier for solid fuels. In figure 10, we illustrate this technique with a spherical ball that heats up and subsequently catches on fire as it passes through the flame. Both this flammable ball and the metal ball were com-

puted on a $120 \times 120 \times 120$ grid at approximately 5 minutes per frame.

7 Conclusion

We have presented a physically based model for animating and rendering fire and flames. We demonstrated that this model could be used to produce realistic looking turbulent flames from both solid and gaseous fuels. We showed plausible interaction of our fire and smoke with objects, including ignition of objects by the flames.

8 Acknowledgment

Research supported in part by an ONR YIP and PECASE award N00014-01-1-0620, NSF DMS-0106694, NSF ACI-0121288, and the DOE ASCI Academic Strategic Alliances Program (LLNL contract B341491).

References

BUKOWSKI, R., AND SEQUIN, C. 1997. Interactive Simulation of Fire in Virtual Building Environments. In *Proceedings of SIGGRAPH 1997*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 35–44.

CHIBA, N., MURAOKA, K., TAKAHASHI, H., AND MIURA, M. 1994. Two dimensional Visul Simulation of Flames, Smoke and the Spread of Fire. *The Journal of Visualization and Computer Animation 5*, 37–53.

FAIRCHILD, M. 1998. *Color Appearance Models*. Addison Wesley Longman, Inc.

FEDKIW, R., ASLAM, T., MERRIMAN, B., AND OSHER, S. 1999. A Non-oscillatory Eulerian Approach to Interfaces in Multimaterial Flows (The Ghost Fluid Method). *J. Comput. Phys. 152*, 457.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual Simulation of Smoke. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 15–22.

FOSTER, N., AND FEDKIW, R. 2001. Practical Animation of Liquids. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 23–30.

FOSTER, N., AND METAXAS, D. 1997. Modeling the Motion of a Hot, Turbulent Gas. In *Proceedings of SIGGRAPH 97*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 181–188.

HENYEY, L., AND GREENSTEIN, J. 1941. Diffuse radiation in the galaxy. *Astrophysics Journal 93*, 70–83.

INAKAGE, M. 1989. A Simple Model of Flames. In *Proceedings of Computer Graphics International 89*, Springer-Verlag, 71–81.

MARKSTEIN, G. H. 1964. *Nonsteady Flame Propagation*. Pergamon, Oxford.

MAZARAK, O., MARTINS, C., AND AMANATIDES, J. 1999. Animating Exploding Objects. In *Proceedings of Graphics Interface 99*, 211–218.

MUSGRAVE, F. K. 1997. Great Balls of Fire. In *SIGGRAPH 97 Animation Sketches, Visual Procedings*, ACM SIGGRAPH, 259–268.

NEFF, M., AND FIUME, E. 1999. A Visual Model for Blast Waves and Fracture. In *Proceedings of Graphics Interface 99*, 193–202.

NGUYEN, D., FEDKIW, R., AND KANG, M. 2001. A Boundary Condition Capturing Method for Incompressible Flame Discontinuities. *J. Comput. Phys. 172*, 71–98.

O'BRIEN, J. F., AND HODGINS, J. K. 1999. Graphical Modeling and Animation of Brittle Fracture. In *Proceedings of SIGGRAPH 1999*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 137–146.

OSHER, S., AND SETHIAN, J. A. 1988. Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton-Jacobi Formualtions. *J. Comput. Phys. 79*, 12.

PERRY, C., AND PICARD, R. 1994. Synthesizing Flames and their Spread. *SIGGRAPH 94 Technical Sketches Notes* (July).

RUSHMEIER, H. 1994. Rendering Participating Media: Problems and Solutions from Application Areas. In *Proceedings of the 5th Eurographics Workshop on Rendering*, 35–56.

SETHIAN, J. 1996. A Fast Marching Level Set Method for Monotonically Advancing Fronts. *Proc. Nat. Acad. Sci. 93*, 1591–1595.

SIEGEL, R., AND HOWELL, J. 1981. *Thermal Radiation Heat Transfer*. Hemisphere Publishing Corp., Washington, DC.

STAM, J., AND FIUME, E. 1995. Depicting Fire and Other Gaseous Phenomena Using Diffusion Process. In *Proceedings of SIGGRAPH 95*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 129–136.

STAM, J. 1999. Stable Fluids. In *SIGGRAPH 99 Conference Proceedings, Annual Conference Series*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 121–128.

STANIFORTH, A., AND COTE, J. 1991. Semi-Lagrangian Integration Schemes for Atmospheric Models: A Review. *Monthly Weather Review 119*, 2206–2223.

STEINHOFF, J., AND UNDERHILL, D. 1994. Modification of the Euler Equations for "Vorticity Confinement": Application to the Computation of Interacting Vortex Rings. *Physics of Fluids 6*, 8, 2738–2744.

TSITSIKLIS, J. 1995. Efficient Algorithms for Globally Optimal Trajectories. *IEEE Transactions on Automatic Control 40*, 1528–1538.

TURNS, S. R. 1996. *An Introduction to Combustion*. McGraw-Hill, Inc.

YNGVE, G. D., O'BRIEN, J. F., AND HODGINS, J. K. 2000. Animating Explosions. In *Proceedings of SIGGRAPH 2000*, ACM Press / ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, ACM, 29–36.

Practical Animation of Liquids

Nick Foster*
PDI/DreamWorks

Ronald Fedkiw**
Stanford University

Abstract

We present a general method for modeling and animating liquids. The system is specifically designed for computer animation and handles viscous liquids as they move in a 3D environment and interact with graphics primitives such as parametric curves and moving polygons. We combine an appropriately modified semi-Lagrangian method with a new approach to calculating fluid flow around objects. This allows us to efficiently solve the equations of motion for a liquid while retaining enough detail to obtain realistic looking behavior. The object interaction mechanism is extended to provide control over the liquid's 3D motion. A high quality surface is obtained from the resulting velocity field using a novel adaptive technique for evolving an implicit surface.

Keywords: animation, computational fluid dynamics, implicit surface, level set, liquids, natural phenomena, Navier-Stokes, particles, semi-Lagrangian.

1. Introduction

The desire for improved physics-based animation tools has grown hand in hand with the advances made in computer animation on the whole. It is natural then, that established engineering techniques for simulating and modeling the real world have been modified and applied to computer graphics more frequently over the last few years. One group of methods that have resisted this transition are those used to model the behavior of liquids from the field of computational fluid dynamics (CFD). Not only are such techniques generally complex and computationally intensive, but they are also not readily adaptable to what could be considered the basic requirements of a computer animation system.

One of the key difficulties encountered when using these methods for animation directly characterizes the trade off between simulation and control. Physics-based animations usually rely on direct numerical simulation (DNS) to achieve realism. In engineering terms, this means that initial conditions and boundary conditions are specified and the process is left to run freely with only minor influence on the part of the animator. The majority of engineering techniques for liquid simulation assume this model.

From an animation viewpoint, we are interested in using numerical techniques to obtain behaviors that would be prohibitive to model by hand. At the same time we want control over the global, low frequency motion so we can match it to the behavior we are trying to create. This then becomes the goal when transitioning between engineering and computer animation; preserve as much of the realistic behavior as feasible while allowing for control over motion on both a local and global scale. This has to be achieved without compromising the overall requirement of a visually coherent and realistic look.

* nickf@pdi.com, ** fedkiw@cs.stanford.edu

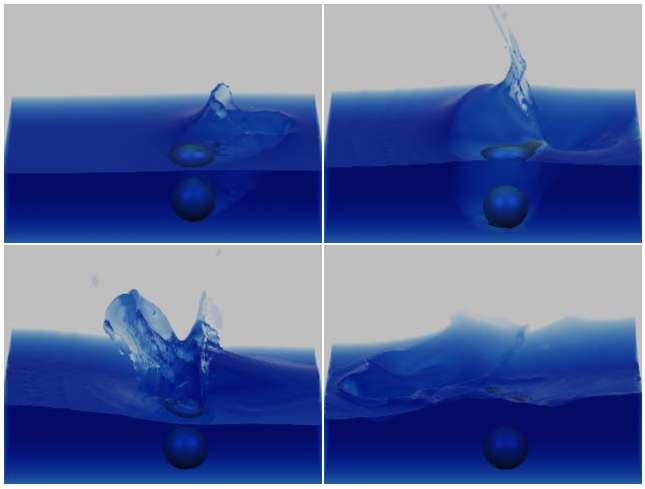


***Figure 1**: A ball splashes into a tank of water.*

This paper specifically addresses these issues for liquid animation. The method presented is for animating viscous liquids ranging from water to thick mud. These liquids can freely mix, move arbitrarily within a fixed three-dimensional grid and interact realistically with stationary or moving polygonal objects. This is achieved for animation by trading off engineering correctness for computational efficiency.

We start with the Navier-Stokes equations for incompressible flow and solve for liquid motion using an adaptation of a semi-Lagrangian method introduced recently to graphics for solving fluid flows [25]. These methods usually result in mass dissipation. While not an issue for gas or smoke, this is visually unacceptable for modeling liquids. We correct for this by tracking the motion of the liquid surface using a novel hybrid combination of inertialess particles and an implicit surface called a level set. The level set prevents mass dissipation while the particles allow the liquid to still splash freely. A useful consequence is that this combined surface can be rendered in a highly believable way.

The next innovation involves taking account of the effects of moving polygonal objects within the liquid. We develop a new technique that, while not accurate in an engineering sense, satisfies the physics of object/liquid interactions and looks visually realistic. This method is efficient and robust, and we show that it can be adapted to provide low frequency directional control over the liquid volume. This allows us to efficiently calculate liquid behavior that would be impossible to get by hand,

while at the same time allowing us to "dial-in" specific motion components.

When the techniques described above are applied together, the result is a comprehensive system for modeling and animating liquids for computer graphics. The main contributions of the system are a numerical method that takes the minimal computational effort required for visual realism combined with tailor-made methods for handling moving objects and for maintaining a smooth, temporally coherent liquid surface.

2. Previous Work

The behavior of a volume of liquid can be described by a set of equations that were jointly developed by Navier and Stokes in the early eighteen hundreds (see next section). The last fifty years has seen an enormous amount of research by the CFD community into solving these equations for a variety of engineering applications. We direct the interested reader to Abbot and Basco [1] which covers some of the important principles without being too mathematically dense.

Early graphics work concentrated on modeling just the surface of a body of water as a parametric function that could be animated over time to simulate wave transport [12, 22, 23]. Kass and Miller [17] approximated the 2D shallow water equations to get a dynamic height field surface that interacted with a static ground "object". Chen and Lobo [4] extended the height field approach by using the pressure arising from a 2D solution of the Navier-Stokes equations to modulate surface elevation. O'Brien and Hodgins [20] simulated splashing liquids by combining a particle system and height field, while Miller and Pearce [19] used viscous springs between particles to achieve dynamic flow in 3D. Terzopoulos, Platt and Fleischer [27] simulated melting deformable solids using a molecular dynamics approach to simulate the particles in the liquid phase.

Surface or particle based methods are relatively fast, especially in the case of large bodies of water, but they don't address the full range of motion exhibited by liquids. Specifically, they don't take advantage of the realism inherent in a full solution to the Navier-Stokes equations. They are also not easily adapted to include interaction with moving objects. Foster and Metaxas [11] modified an original method by Harlow and Welch [15] (later improved by others, see e.g. [5]) to solve the full equations in 3D with arbitrary static objects and extended it to include simple control mechanisms [9]. Foster and Metaxas also applied a similar technique to model hot gases [10]. Stam [25] replaced their finite difference scheme with a semi-Lagrangian method to achieve significant performance improvements at the cost of increased rotational damping. Yngve et al. used a finite difference scheme to solve the compressible Navier-Stokes equations to model shock wave and convection effects generated by an explosion [28].

3. Method Outline

The Navier-Stokes equations for describing the motion of a liquid consist of two parts. The first, enforces incompressibility by saying that mass should always be conserved, i.e.

$$\nabla \cdot \mathbf{u} = 0 , \qquad (3.1)$$

where $\mathbf{u}$ is the liquid velocity field, and

$$\nabla = \left(\partial / \partial \mathrm{x}, \partial / \partial \mathrm{y}, \partial / \partial \mathrm{z}\right)$$

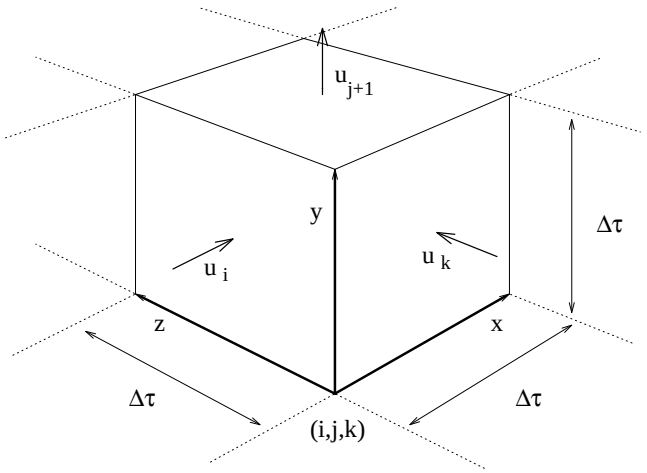


Figure 2: *A single grid cell with three of its six face velocities shown.*

is the gradient operator. The second equation couples the velocity and pressure fields and relates them through the conservation of momentum, i.e.

$$\mathbf{u}_t = \nu\nabla \cdot \left(\nabla \mathbf{u}\right) - \left(\mathbf{u} \cdot \nabla\right)\mathbf{u} - \frac{1}{\rho}\nabla \mathrm{p} + \mathbf{g} . \qquad (3.2)$$

This equation models the changes in the velocity field over time due to the effects of viscosity (ν), convection, density (ρ), pressure (p), and gravity (**g**). By solving (3.1) and (3.2) over time, we can model the behavior of a volume of liquid. The new algorithm we are proposing to do this consists of six straightforward steps.

I. Model the static environment as a voxel grid.

II. Model the liquid volume using a combination of particles and an implicit surface.

Then, for each simulation time step

III. Update the velocity field by solving (3.2) using finite differences combined with a semi-Lagrangian method.

IV. Apply velocity constraints due to moving objects.

V. Enforce incompressibility by solving a linear system built from (3.1).

VI. Update the position of the liquid volume (particles and implicit surface) using the new velocity field.

These steps are described in detail in the following sections. Steps IV and V are presented in reverse order for clarity.

4. Static Environment

Equations (3.1) and (3.2) model a liquid as two coupled dynamic fields, velocity and pressure. The motion of the liquid we are modeling will be determined by evolving these fields over time. We start by representing the environment that we want the liquid to move in as a rectangular grid of voxels with side length Δτ. The grid does not have to be rectangular, but the overhead of unused (non-liquid containing) cells will be low and so it is convenient. Each cell has a pressure variable at its center and shares a velocity variable with each of its adjacent neighbors (see figure 2). This velocity is defined at the center of the face shared

by the two neighboring cells and represents the magnitude of the flow normal to that face. This is the classic "staggered" MAC grid [15]. Each cell is then either tagged as being empty (available to be filled with liquid) or filled completely with an impermeable static object. Despite the crude voxelized approximation of both objects and the liquid volume itself, we'll show that we can still obtain and track a smooth, temporally coherent liquid surface.

5. Liquid Representation

The actual distribution of liquid in the environment is represented using an implicit surface. The implicit function is derived from a combination of inertialess particles and a dynamic isocontour. The isocontour provides a smooth surface to regions of liquid that are well resolved compared to our grid, whereas the particles provide detail where the surface starts to splash.

5.1 Particles

Particles are placed (or introduced via a source) into the grid according to some initial liquid distribution. Their positions then evolve over time by simple convection. Particle velocity is computed directly from the velocity grid using tri-linear interpolation and each particle is moved according to the inertialess equation $d\mathbf{x}_p/dt = \mathbf{v}_x$, where $\mathbf{v}_x$ is the fluid velocity at $\mathbf{x}_p$. Particles have a low computational overhead and smoothly integrate the changing liquid velocity field over time. The obvious drawback to using them, however, is that there is no straightforward way to extract a smooth polygonal (or parametric) description of the actual liquid surface. This surface is preferred because we want to render the liquid realistically using traditional computer graphics techniques. It is possible to identify it by connecting all the particles together into triangles, although deducing both the connectivity and set of surface triangles is difficult. In addition, since the particles do not generally form a smooth surface, the resulting polygonal mesh suffers from temporal aliasing as triangles "pop" in or out.

5.2 Isocontour

An alternative technique for representing the liquid surface is to generate it from an isocontour of an implicit function. The function is defined on a high resolution Eulerian sub-grid that sits inside the Navier-Stokes grid. Let each particle represent the center of an implicitly defined volumetric object (see Bloomenthal et al. [3] for a survey of implicit surfaces). Specifically, an implicit function centered at the particle location $\mathbf{x}_p$ with radius r is given by

$$\phi_p(\mathbf{x}) = \sqrt{(x_i - x_{pi})^2 + (x_j - x_{pj})^2 + (x_k - x_{pk})^2} - r .$$

The surface of that particle is defined as the spherical shell around $\mathbf{x}_p$ where $\phi_p(\mathbf{x})=0$. An implicit function, $\phi(\mathbf{x})$, is then defined over all the particles by taking the value of $\phi_p(\mathbf{x})$ from the particle closest to $\mathbf{x}$. If we sample $\phi(\mathbf{x})$ at each sub-grid point we can use a marching cubes algorithm [18] to tessellate the $\phi(\mathbf{x})=0$ isocontour with polygons. More sophisticated blend functions could be used to create an implicit function, however, we are going to temporally and spatially smooth $\phi(\mathbf{x})$ so it isn't necessary. We refer those interested in wrapping implicit surfaces around particles to the work of Desbrun and Cani-Gascuel [7].

The first step towards smoothing the surface is to normalize ϕ so that $|\phi(\mathbf{x})|$ equals the distance from $\mathbf{x}$ to the closest point on the zero isocontour. The sign of ϕ is set negative inside the liquid and positive outside. This signed distance function can be created quickly using the Fast Marching Method [24] starting from the initial guess of $\phi(\mathbf{x})$ defined by the particles as outlined above.

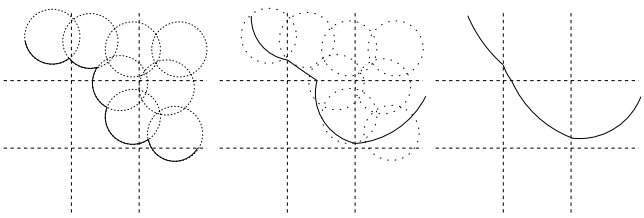


Figure 3: *The isocontour due to the implicit function around the particles, interpolated ϕ values, and smoothed ϕ values, respectively.*

In order to smooth out ϕ to reduce unnatural "folds" or "corners" in the surface (see figure 3), a smoothing equation of the form

$$\phi_\eta = -S\left(\phi^{\eta=0}\right)\left(\left|\nabla\phi\right| - 1\right), \qquad (5.1)$$

is used to modify values of ϕ close to the $\phi(\mathbf{x})=0$ isocontour. $S(\phi)$ is a smoothed sign function given by

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + \Delta\tau^2}} .$$

If applied for a few relaxation steps in fictitious time η (everything else remains constant), (5.1) smooths out the $\phi(\mathbf{x})=0$ isocontour while maintaining overall shape. Once smoothed, the isocontour can be ray traced directly using a root finding algorithm to identify the zero values of ϕ. A fast root finder can be built easily because at any sub-grid point the value of ϕ explicitly gives the minimum step to take to get closer to the surface. Note that the surface normal is given by $\mathbf{n} = \nabla\phi / \left|\nabla\phi\right|$.

By creating a smooth isocontour for each frame of animation, we get an improved surface representation compared to using particles alone. There are still drawbacks however. A high density of particles is required at the $\phi(\mathbf{x})=0$ isocontour before the surface looks believably flat. Particles are also required throughout the entire liquid volume even when it's clear that they make no contribution to the visible surface. The solution is to create ϕ once using the particles, and then track how it moves using the same velocity field that we're using to move the particles. This leads to a temporally smoothed dynamic isosurface known in the CFD literature as a level set.

5.3 Dynamic Level Set

An obvious way to track the evolution of the surface of a volume of liquid would be to attach particles directly to the surface in its initial position and then just move them around in the velocity field. This would require adding extra particles when the surface becomes too sparsely resolved, and removing them as the surface folds, or "splashes" back over itself. An alternative method which is intuitively similar, but that doesn't use particles, was developed by Osher and Sethian [21] and is called the level set method.

We want to evolve ϕ directly over time using the liquid velocity field $\mathbf{u}$. We have a smooth surface but need to conform, visually at least, to the physics of liquids. It has been shown [21] that the

equation to update ϕ under these circumstances has the following structure,

$$\phi_t + \mathbf{u} \cdot \nabla \phi = 0 \,. \qquad (5.2)$$

Using (5.2), the surface position is evolved over time by tracking $\phi(\mathbf{x})=0$. The $(\mathbf{u} \cdot \nabla \phi)$ term is a convection term similar to the $(\mathbf{u} \cdot \nabla)\mathbf{u}$ term in (3.2) implying that we could use a semi-Lagrangian method to solve this equation. However, since this equation represents the mass evolution of our liquid, the semi-Lagrangian method tends to be too inaccurate. Instead we use a higher order upwind differencing procedure [1] on the $(\mathbf{u} \cdot \nabla \phi)$ term. Fedkiw et al. [8] used this methodology to track a fluid surface and give explicit details on solving (5.2). This method can suffer from severe volume loss especially on the relatively coarse grids commonly used in computer graphics. This is clearly visible when regions of liquid break away during splashing and then disappear because they are too small to be resolved by the level set. We require visual coherency for this to be a useful graphics technique and so the level set method needs to be modified to preserve volume.

5.4 Hybrid Surface Model

Particle evolution is a fully Lagrangian approach to the mass motion while level set evolution is a fully Eulerian approach. Since they tend to have complementary strengths and weakness, a combined approach gives superior results under a wider variety of situations. Level set evolution suffers from volume loss near detailed features while particle evolution suffers from visual artifacts in the surface when the number of particles is small. Conversely, the level set is always smooth, and particles retain detail regardless of flow complexity. Therefore we suggest a novel combination of the two approaches.

At each time step we evolve the particles and the level set ϕ forward in time. Next, we use the updated value of the level set function to decide how to treat each particle. If a particle is more than a few grid cells away from, and inside the surface, as indicated by the locally interpolated value of ϕ, then that particle is deleted. This increases efficiency since particles are only needed locally near the surface of the liquid as opposed to throughout the entire liquid volume. In addition, for cells close to $\phi(\mathbf{x})=0$ that are sparsely populated, extra particles can be introduced "within" the isocontour. Thus, for a bounded number of particles, we get improved surface resolution.

Next, for each particle near the surface, the locally interpolated curvature of the interface, calculated as

$$k = \nabla \cdot \left(\nabla\phi / \left|\nabla\phi\right| \right),$$

is used to indicate whether or not the surface is smooth. Smooth regions have low curvature and the particles are ignored allowing the level set function to give a very smooth representation of the liquid surface. On the other hand, regions of high curvature indicate splashing. In these regions, the particles are a better indicator of the rough surface topology. Particles in these regions are allowed to modify the local values of ϕ. At grid points where the implicit basis function for the particle would give a smaller value of ϕ (i.e. a particle is "poking" out of the zero level set), this smaller value is used to replace the value obtained from the time evolution of ϕ.

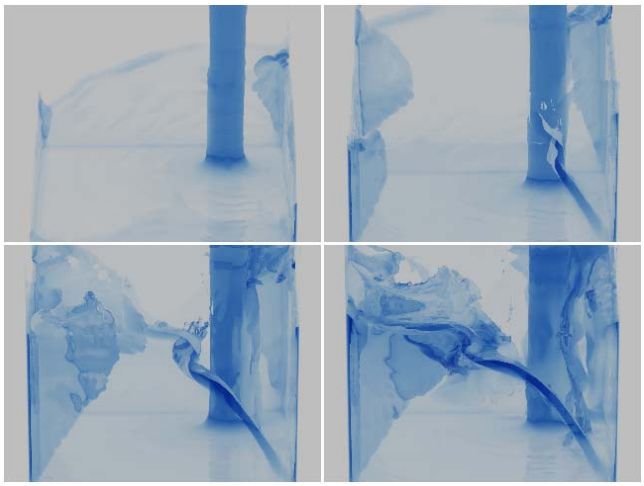


Figure 4: *Water pours into a container causing a complex surface to develop.*

Even with the tight coupling between the particles and the level set, some particles will escape the inside of the liquid layer since the grid is too coarse to represent them individually. These particles can be rendered directly as small liquid drops. In addition, these stray particles could be used as control particles to indicate the presence of fine spray or mist.

6. Updating the Velocity Field

We have a representation of the graphics environment and a way of tracking the surface of a volume of liquid. We can now apply (3.2) to the existing velocity field to advance it through an Euler-integration time step Δt. The equation is solved in three stages. First we compute Δt using the CFL condition (see Appendix A). Next, we update the convective component, i.e. $(\mathbf{u} \cdot \nabla)\mathbf{u}$, using a first order semi-Lagrangian method, as per Courant et al. [6] and by Stam [25]. We use the same formulation as Stam and refer readers to his description. Standard central differencing is then used on the viscous terms of (3.2) as described by Foster and Metaxas [11]. The results from this and the preceding calculation are added together to get an updated (though not mass conserving) velocity field for time $t+\Delta t$.

Semi-Lagrangian methods allow us to take large time steps without regard for the sometimes overly restrictive CFL condition [26]. Unfortunately, these large time steps come at the cost of added dissipation. This is visually acceptable for gases such as smoke where it appears realistic. For liquids however, mass dissipation ruins the visual effect. Therefore, even though we use a semi-Lagrangian method to update (3.2), the time step for evolving the particles and the level set still needs to be limited according to a plausible CFL condition. Updating the surface position isn't particularly expensive computationally, and so we alternate between a large time step for updating the Navier-Stokes equations and a series of small time steps (that add up to the large time step) for the particles and the level set. Our experience suggests that the velocity field time step can only be a few (around five) times bigger than that dictated by the usual CFL criterion. However, even this gives tremendous computational savings, since enforcing incompressibility (step V, discussed in section 8) is the most expensive part of the algorithm.

We caution the reader that using a particle only evolution with the semi-Lagrangian method introduces noise into the surface, and that using a level set only evolution with the semi-Lagrangian method gives noticeable volume loss. The key to making the semi-Lagrangian method work for liquids is the mixed Eulerian-Lagrangian formulation that uses *both* particles and level sets to evolve the surface position over time.

7. Boundary Conditions

When solving (3.2) within the grid, we need to specify pressure and velocity values for certain cells. We want stationary object cells to resist liquid motion and cells that represent the boundary between air and liquid to behave appropriately.

7.1.1 Non-liquid Cells

Cells in the grid that don't contain particles and aren't contained within the isosurface are either considered empty (open air) or are part of an object. If a cell is empty, its pressure is set to atmospheric pressure, and the velocity on each of its faces shared with another empty cell is set to zero. This assumes that air dynamics has a negligible effect. An object cell, on the other hand, can have velocities and pressures set using many different combinations to approximate liquid flowing into or out of the environment, or to approximate different object material properties. Foster and Metaxas [10] summarize and discuss methods to do this.

7.1.2 Liquid Surface

Other grid cells that require special attention are those that contain part of the $\phi(\mathbf{x})=0$ isocontour. Such cells represent what we know about the location of the liquid surface within the grid. The movement of the isocontour will determine how the surface evolves, but we need to set velocities on faces between empty and liquid cells so that normal and tangential stresses are zero. Intuitively, we need to make sure that the "air" doesn't mix with or inhibit the motion of the liquid, while allowing it to flow freely into empty cells. This is done by explicitly enforcing incompressibility within each cell that contains part of the liquid surface. Velocities adjacent to a liquid filled cell are left alone, whereas the others are set directly so (3.1) is satisfied for that cell. The pressure in a surface cell is set to atmospheric pressure.

8. Conservation of Mass

The velocity field generated after evolving the Navier-Stokes equations (steps III and IV) has rotation and convection components that are governed by (3.2) (excluding the pressure term). However, (3.1), conservation of mass, is only satisfied in surface cells where we have explicitly enforced it. The best we can do to preserve mass within our grid is to ensure that the incompressibility condition is satisfied for every grid cell (at least to some tolerance). Foster and Metaxas [11] achieved this using a technique called Successive Over Relaxation.

A more efficient method for enforcing incompressibility comes from solving the linear system of equations given by using the Laplacian operator to couple local pressure changes to the divergence in each cell. Specifically, this gives

$$\nabla^2 p = \rho \nabla \cdot \mathbf{u} / \Delta t \,, \qquad (8.1)$$

where $\nabla^2 p$ is the spatial variation (Laplacian) of the pressure and $\mathbf{u}$ is the velocity field obtained after solving (3.2). Applied at the center of a cell, (8.1) can be discretized as

$$\sum_{n=\{ijk\}} (p_{n+1} + p_{n-1}) - 6p = \rho \frac{\Delta \tau}{\Delta t} \sum_{n=\{ijk\}} (u_{n+1} - u_n), \qquad (8.2)$$

where $p_{n \pm 1}$ is the pressure from the cell ahead (+) or behind (-) in the n direction, and the u values are taken directly from the grid cell faces. Using (8.2), we form a linear system AP = b where P is the vector of unknown pressures needed to make the velocity field divergence free, b is the RHS of (8.2), and A has a regular but sparse structure. The diagonal coefficients of A, a_{ii}, are equal to the negative number of liquid cells adjacent to $cell_i$ (e.g., -6 for a fully "submerged" cell) while the off diagonal elements are simply $a_{ij}=a_{ji}=1$ for all liquid $cells_j$ adjacent to $cell_i$.

Conveniently, the system described above is symmetric and positive definite (as long as there is at least one surface cell as part of each volume). Static object and empty cells don't disrupt this structure. In that case pressure and velocity terms can disappear from both sides of (8.2), but the system remains symmetric. Because of this, it can be solved quickly and efficiently using a Preconditioned Conjugate Gradient (PCG) method. Further efficiency gains can be made by using an Incomplete Choleski preconditioner to accelerate convergence. There is a wealth of literature available regarding PCG techniques and we recommend any of the standard implementations, see Barret et al. [2] for some basic templates. Once the new pressures have been determined, the velocities in each cell are updated according to

$$u_{\{ijk\}}^{t+\Delta t} = u_{\{ijk\}} - \frac{\Delta t}{\rho \Delta \tau} (p_n - p_{n-1})$$

The resulting velocity field conserves mass (is divergence free) and satisfies the Navier-Stokes equations.

9. Moving Objects

Previous techniques proposed for liquid animation could deal with static objects that have roughly the same resolution as the grid, but they have difficulty dealing with moving objects. Unfortunately, the CFD literature has little to offer to help resolve the effects of moving objects on a liquid in terms of animation. There are sophisticated methods available for handling such interactions, but they typically require highly resolved computational grids or a grid mechanism that can adapt itself to the moving object surface. Neither approach is particularly well suited to animation because of the additional time complexity involved. Therefore, we propose the following method for handling interactions between moving objects and the liquid.

Consider an object (or part of an object) moving within a cell that contains liquid. There are two basic conditions that we want to enforce with respect to the computational grid, and an additional condition with respect to the surface tracking method. These are

1. Liquid should not flow into the object. At any point of contact, the relative velocity between the liquid and object along the object's surface normal should be greater than or equal to zero.
2. Tangential to the surface, the liquid should flow freely without interference.
3. Neither the particles nor the level set surface should pass through any part of the surface of the object.

The last of these is relatively straightforward. We know where the polygons that comprise the object surface are and in what direction they are moving. We simply move the particles so that they are always outside the surface of the object. As long as we accurately take account of the velocity field within the grid then the isocontour will remain in the correct position relative to the object.

To prevent liquid from flowing into the object we directly set the component of liquid velocity normal to the object. We know the object surface normal, $\mathbf{n}_s$, and can calculate the liquid velocity relative to that surface, $\mathbf{v}_r$, in a given cell. If $\mathbf{v}_r.\mathbf{n}_s < 0$ then liquid is flowing through the surface. In such cases we manipulate $\mathbf{u}$ in the cell so that $\mathbf{v}_r.\mathbf{n}_s = 0$ leaving the tangential ("slip") part of the velocity unchanged.

These velocities need to be applied without introducing visual artifacts into the flow. The following method solves for both normal and tangential velocity components. It's relatively intuitive, and it seems to work well in practice. The steps are

1. As a boundary condition, any cell within a solid object has its velocities set to that of the moving object.
2. The velocity field is updated using (3.2). No special consideration is given to cells containing an object, i.e. they are all allowed to change freely as if they contain liquid.
3. Each cell that intersects an object surface gets the component of the object velocity along its normal set explicitly as outlined above.
4. Cells internal to the object have their velocities set back to the object velocity.
5. During the mass conservation step (section 8) the velocity for any grid cell that intersects the object is held fixed.

The result of this approach is that liquid is both pushed along by an object while being allowed to flow freely around it, causing realistic-looking behavior in the mean time. Obviously it's only possible to accurately account for one polygon face per grid cell. Objects that are more detailed with respect to the grid can still be handled by determining an average object surface normal and velocity for each intersecting cell, but grid resolution remains the limiting factor.

10. Control

Animation is all about control. Having things behave according to some arbitrary aesthetic is the goal of most production software. The difficulty is in providing this level of control over a physics-based animation while still maintaining a realistic behavioral component. The nature of the governing equations of motion of liquids means that they will always swirl, mix, and splash unless the applied forces are identical everywhere. This necessarily limits the level of control that we can have over the final motion and comes with the territory of non-linear simulation.

Gates [13] has shown that mass conserving flow fields can be blended with calculated fields to get good non-dynamic results. The Navier-Stokes equations allow for the body force term, $\mathbf{g}$, to be manipulated directly [9] much like a traditional particle system. Forces aren't always a very intuitive way of getting motion that we want however. The moving object mechanism on the other hand, is well suited to this. Instead of moving polygons, we can explicitly set velocities anywhere in the grid by introducing "fake" surfaces (a single point even) that have normals and velocities pointing in the direction that we want the liquid to go. Setting the normal *and* tangential velocities in individual cells is also possible if it is done before the mass conservation calculation. This allows the solver to smooth out any lack of physical correctness in applied velocities before passing them into (3.2).

As a brief example, consider a set of 3D parametric space curves that define the desired path for the liquid to follow. We instance a set of points along each curve giving each point a parametric coordinate φ_p. A point's spatial position is then given simply by the curve definition, i.e. $\mathbf{x}_p=F(\varphi_p)$. The velocity of the point can then be described as

$$\mathbf{v}_p = C(t)\left(dF(\varphi_p)/d\varphi_p\right)$$

where $C(t)$ is a monotonic key framed scaling function. $C(t)$ is also used to update φ_p over time according to $d\varphi_p/dt = C(t)$. The "fake" surface normal of the point is then simply $\mathbf{n}_p = \mathbf{v}_p/|\mathbf{v}_p|$. By manipulating $\mathbf{x}_p$, $\mathbf{v}_p$, and $\mathbf{n}_p$ over time, we can "trap" small pockets of liquid and control them directly. The governing equations then make sure that neighboring liquid attempts to follow along.

This basic approach can be adapted to surfaces or even volumetric functions as long as they vary smoothly. While still not giving perfect direct control over the liquid motion, when combined with force fields it is good enough to make it a useful animation tool.

11. Results

The animation system described in the preceding sections was used to generate all of the examples in this paper. The basic Navier-Stokes solver and implicit surface are demonstrated by the container-filling example in figure 4. The combination of particles and level set make sure that the resulting surface stays smooth and behaves in a physically believable way. The splashing object examples in figures 1, 5 and 6 show close interaction between the liquid and moving objects. They also show how the hybrid surface can handle extreme splashing without either the particles or level set being apparent. The particles play a large role in both cases by allowing the liquid to "splash" at a higher resolution than would be possible with the level set alone. All of these images were rendered using a ray-tracing algorithm that marches through the implicit surface grid as outlined in section 5.

The final example, figure 7, makes use of just a spherical implicit function around each particle. It shows the interaction between a thick (high viscosity) liquid and a hand animated character. The character surface is sampled at each grid cell and the mechanisms described in section 9 take account of all the motion in the scene. This includes the character filling his mouth with mud. The mud is later ejected using a 3D space curve as a controller as outlined in section 10. The captions to each figure give the static grid size used during calculation along with computation times per frame (for motion, not rendering) on a PentiumII 500MHz.

12. Conclusion

We have presented a method for modeling and animating liquids that is a pragmatic compromise between the numerical care that needs to be taken when simulating non-linear physics and the interaction and control animators require. Where appropriate, we have drawn on techniques from computational fluid dynamics and combined them with recent computer graphics advances as well

as new methods for free surface evolution and interaction between moving objects and the liquid volume. The result is a technique that is very general, efficient, and offers flexible control mechanisms for specifying how the liquid should behave.

13. Acknowledgements

The work of the second author was supported in part by ONR N00014-97-1-0027.

A. Courant-Friedrichs-Levy (CFL) Condition

The CFL condition is a restriction on the size of the time step, Δt, that can be used together with a time-marching numerical simulation. It says that Δt must be less than the minimum time over which "something significant" can occur in the system for the simulation to remain numerically stable. The CFL condition depends both on the physical system being modeled as well as the specifics of the discretization method employed. In the context of the system described in this paper a good CFL condition is that a discrete element of liquid cannot "jump over" a cell in the computational grid, i.e. $\Delta t < \Delta\tau / |\mathbf{u}|$.

Note that the viscosity related terms also impose a CFL type restriction. This can be avoided by locally adjusting the magnitude of the viscosity in cells where the viscous terms would dictate the necessity for a smaller time step.

References

[1] Abbot, M. and Basco, D., "Computational Fluid Dynamics – An Introduction for Engineers", Longman, 1989.

[2] Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. and van der Vorst, H., "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", Society for Industrial and Applied Mathematics, 1993.

[3] Bloomenthal, J., Bajaj, C., Blinn, J., Cani-Gascual, M.-P., Rockwood, A., Wyvill, B. and Wyvill, G., "Introduction to Implicit Surfaces", Morgan Kaufmann Publishers Inc., San Francisco, 1997.

[4] Chen, J. and Lobo, N., "Toward Interactive-Rate Simulation of Fluids with Moving Obstacles Using the Navier-Stokes Equations", Graphical Models and Image Processing 57, 107-116 (1994).

[5] Chen, S., Johnson, D., Raad, P. and Fadda, D., "The Surface Marker and Micro Cell Method", Int. J. Numer. Methods in Fluids 25, 749-778 (1997).

[6] Courant, R., Issacson, E. and Rees, M., "On the Solution of Nonlinear Hyperbolic Differential Equations by Finite Differences", Comm. Pure and Applied Math 5, 243-255 (1952).

[7] Desbrun, M. and Cani-Gascuel, M.P., "Active Implicit Surface for Animation", Graphics Interface 98, 143-150 (1998).

[8] Fedkiw, R., Aslam, T., Merriman, B. and Osher, S., "A Non-Oscillatory Eulerian Approach to Interfaces in Multimaterial Flows (The Ghost Fluid Method)", J. Comput. Phys. 152, 457-492 (1999).

[9] Foster, N. and Metaxas, D., "Controlling Fluid Animation", Computer Graphics International 97, 178-188 (1997).

[10] Foster, N. and Metaxas, D., "Modeling the Motion of a Hot Turbulent Gas", ACM SIGGRAPH 97, 181-188 (1997).

[11] Foster, N. and Metaxas, D., "Realistic Animation of Liquids", Graphical Models and Image Processing 58, 471-483 (1996).

[12] Fournier, A. and Reeves, W.T., "A Simple Model of Ocean Waves", ACM SIGGRAPH 86, 75-84 (1986).

[13] Gates, W.F., "Interactive Flow Field Modeling for the Design and Control of Fluid Motion in Computer Animation", UBC CS Master's Thesis, 1994.

[14] Golub, G.H. and Van Loan, C.F., "Matrix Computations", The John Hopkins University Press, 1996.

[15] Harlow, F.H. and Welch, J.E., "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with a Free Surface", The Physics of Fluids 8, 2182-2189 (1965).

[16] Kang, M., Fedkiw, R. and Liu, X.-D., "A Boundary Condition Capturing Method For Multiphase Incompressible Flow", J. Sci. Comput. 15, 323-360 (2000).

[17] Kass, M. and Miller, G., "Rapid, Stable Fluid Dynamics for Computer Graphics", ACM SIGGRAPH 90, 49-57 (1990).

[18] Lorenson, W.E. and Cline, H.E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", Computer Graphics 21, 163-169 (1987).

[19] Miller, G. and Pearce, A., "Globular Dynamics: A Connected Particle System for Animating Viscous Fluids", Computers and Graphics 13, 305-309 (1989).

[20] O'Brien, J. and Hodgins, J., "Dynamic Simulation of Splashing Fluids", Computer Animation 95, 198-205 (1995).

[21] Osher, S. and Sethian, J.A., "Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations", J. Comput. Phys. 79, 12-49 (1988).

[22] Peachy, D., "Modeling Waves and Surf", ACM SIGGRAPH 86, 65-74 (1986).

[23] Schachter, B., "Long Crested Wave Models", Computer Graphics and Image Processing 12, 187-201 (1980).

[24] Sethian, J.A. "Level Set Methods and Fast Marching Methods", Cambridge University Press, Cambridge 1999.

[25] Stam, J., "Stable Fluids", ACM SIGGRAPH 99, 121-128 (1999).

[26] Staniforth, A. and Cote, J., "Semi-Lagrangian Integration Schemes for Atmospheric Models – A Review", Monthly Weather Review 119, 2206-2223 (1991).

[27] Terzopoulos, D., Platt, J. and Fleischer, K., "Heating and Melting Deformable Models (From Goop to Glop)", Graphics Interface 89, 219-226 (1995).

[28] Yngve, G., O'Brien, J. and Hodgins, J., "Animating Explosions", ACM SIGGRAPH 00, 29-36 (2000).

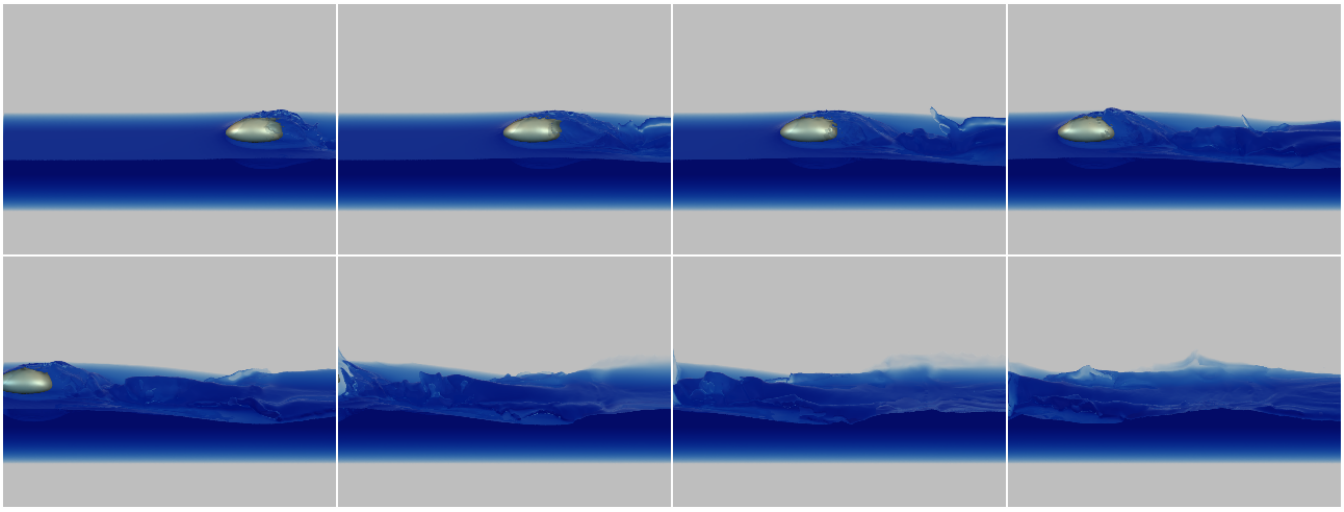


Figure 5: *An ellipsoid slips along through shallow water. The combination of particle and level set tracking allows water to flow over the object without any visual loss of volume. The environment for this example was 250x75x90 cells. It took approximately seven minutes to calculate the liquid motion (including surface evolution) per frame.*

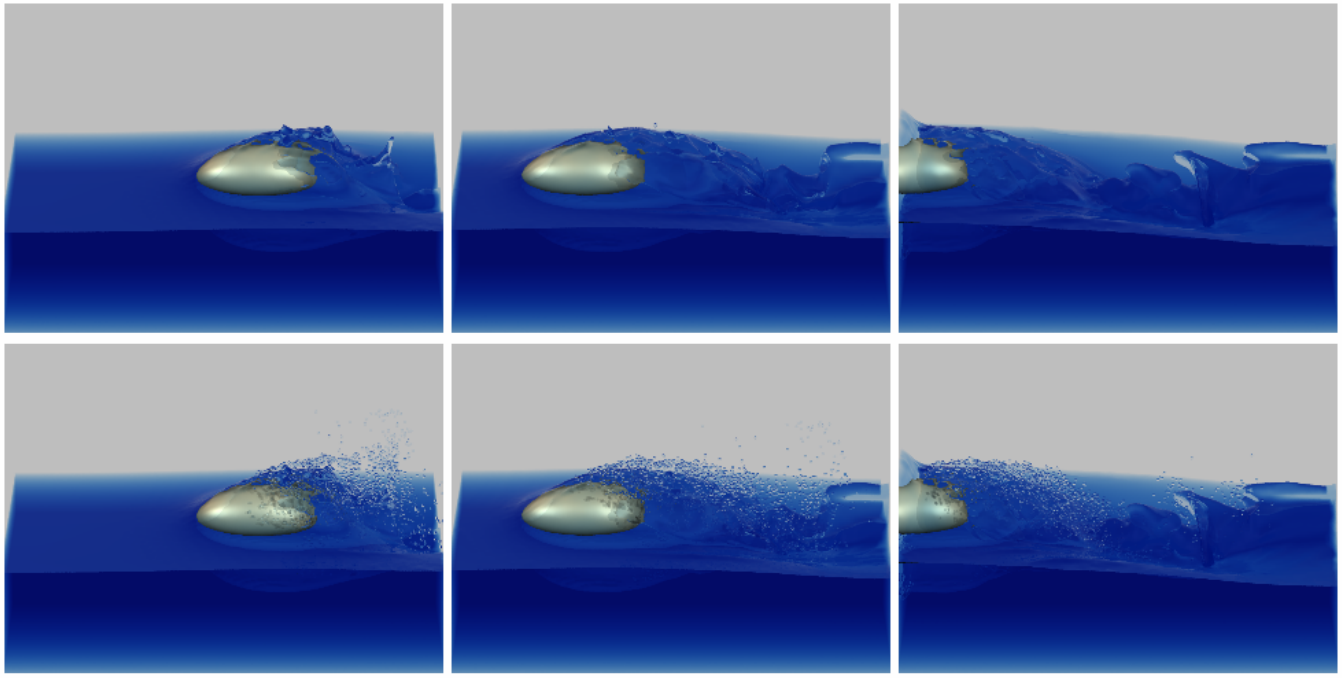


Figure 6: *A close up of the ellipsoid from figure 5 showing the implicit surface derived from combining the particle basis functions and level set (top), and with the addition of the freely splashing particles raytraced as small spheres (bottom). The environment for this example was 150x75x90 cells. Calculation times were approximately four minutes per frame.*

Figure 7: *A fully articulated animated character interacts with viscous mud. The environment surrounding the character is 150x200x150 cells. That resolution is sufficient to accurately model the character filling his mouth with mud. A 3D control curve is used to eject (spit) the mouthful of mud later in the sequence. This example runs at three minutes per frame.*

Animation and Rendering of Complex Water Surfaces

Douglas Enright
Stanford University
Industrial Light & Magic
enright@stanford.edu

Stephen Marschner
Stanford University
srm@graphics.stanford.edu

Ronald Fedkiw
Stanford University
Industrial Light & Magic
fedkiw@cs.stanford.edu

Abstract

We present a new method for the animation and rendering of *photorealistic* water effects. Our method is designed to produce visually plausible three dimensional effects, for example the pouring of water into a glass (see figure 1) and the breaking of an ocean wave, in a manner which can used in a computer animation environment. In order to better obtain photorealism in the behavior of the simulated water surface, we introduce a new "thickened" front tracking technique to accurately represent the water surface and a new velocity extrapolation method to move the surface in a smooth, water-like manner. The velocity extrapolation method allows us to provide a degree of control to the surface motion, e.g. to generate a wind-blown look or to force the water to settle quickly. To ensure that the photorealism of the simulation carries over to the final images, we have integrated our method with an advanced physically based rendering system.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: computational fluid dynamics, implicit surfaces, natural phenomena, physically based animation, rendering, volume rendering

1 Introduction

Water surrounds us in our everyday lives. Given the ubiquity of water and our constant interaction with it, the animation and rendering of water poses one of the greatest challenges in computer graphics. The difficulty of this challenge was underscored recently through the use of water effects in a variety of motion pictures including the recent feature film "Shrek" where water, mud, beer and milk effects were seen. In response to a question concerning what was the single hardest shot in "Shrek", DreamWorks SKG principal and producer of "Shrek", Jeffrey Katzenberg, stated, *It's the pouring of milk into a glass.* [Hiltzik and Pham 2001].

The above quote illustrates the need for *photorealistic* simulation and rendering of water (and other liquids such as milk), especially in the case of complex, three dimensional behavior as seen when water is poured into a glass as in figure 1. A key to achieving this goal is the visually accurate treatment of the surface separating the water from the air. The behavior of this surface provides the visual impression that water is being seen. If the numerical simulation method used to model this surface is not robust enough to capture the essence of water, then the effect is ruined for the viewer. A variety of new techniques for treatment of the surface are proposed in this paper in order to provide visually pleasing motion and photorealistic results.

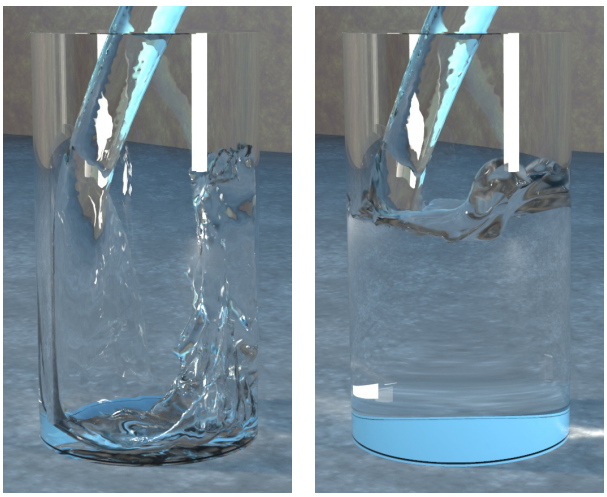


Figure 1: **Water being poured into a glass (55x120x55 grid cells).**

We propose a new "thickened" front tracking approach to modeling the surface, called the "particle level set method". It is a hybrid surface tracking method that uses massless marker particles combined with a dynamic implicit surface. This method was entirely inspired by the hybrid liquid volume model proposed in [Foster and Fedkiw 2001], but exhibits significantly more realistic surface behavior. This effect is achieved by focusing on modeling the surface as opposed to the liquid volume as was done by [Foster and Fedkiw 2001]. This shift in philosophy away from volume modeling and towards surface modeling, is the key idea behind our new techniques, resulting in better photorealistic behavior of the water surface.

We propose a new treatment of the velocity at the surface in order to obtain more visually realistic water surface behavior. The motion of both the massless marker particles and the implicit function representing the surface is dependent upon the velocities contained on the underlying computational mesh. By extrapolating velocities across the water surface and into the region occupied by the air, we obtain more accurate and better visual results. In the limit as the computational grid is refined, the resulting surface condition is identical to the traditional approach of making the velocity divergence free, but it gives more visually appealing and physically plausible results on coarse grids. Furthermore, this velocity extrapolation procedure alows us to add a degree of control to the behavior of the water surface. We can add dampening and/or churning effects forcing it to quiet down or splash up faster than would be

allowed by a straightforward physical simulation.

Our new advances can be easily incorporated into a pre-existing Navier-Stokes solver for water. In fact, we solve the Navier-Stokes equations for liquid water along the lines of [Foster and Fedkiw 2001], in particular using a semi-Lagrangian "stable fluid" approach introduced to the community by Stam [Stam 1999]. Our approach preserves as much of the realistic behavior of the water as possible, while at the same time providing a degree of control necessary for its use in a computer animation environment.

Photorealistic rendering is necessary in order to complete the computational illusion of real water. In some ways water is an easy material to render, because unlike many common materials its optical properties are well understood and are easy to describe. In all but the largest-scale scenes, surface tension prevents water surfaces from exhibiting the microscopic features that make reflection from many other materials so complicated. However, water invariably creates situations in which objects are illuminated through complex refracting surfaces, which means that the light transport problem that is so easy to state is difficult to solve. Most widely used rendering algorithms disregard this sort of illumination or handle it using simple approximations, but because water and its illumination effects are so familiar these approaches fail to achieve realism. There are several rendering algorithms that can properly account for all transport paths, including the illumination through the water surface; some examples are path tracing [Kajiya 1986], bidirectional path tracing [Heckbert 1990; Lafortune and Willems 1993; Veach and Guibas 1994], Metropolis light transport [Veach and Guibas 1997], and photon mapping [Jensen 1995]. In our renderings of clear water for this paper we have chosen photon mapping because it is simple and it makes it easy to avoid the distracting noise that often arises in pure path sampling algorithms from illumination through refracting surfaces.

2 Previous Work

Early (and continuing) work by the graphics community into the modeling of water phenomenon focused on reduced model representations of the water surface, ranging from Fourier synthesis methods [Masten et al. 1987] to parametric representations of the water surface [Schachter 1980; Fournier and Reeves 1986; Peachey 1986; Ts'o and Barsky 1987]. The last three references are notable in the way they attempt to model realistic wave behavior including the change in wave behavior as a function of the depth of the water. Fairly realistic two dimensional wave scenery can be developed using these methods including the *illusion* of breaking waves, but ultimately they are all constrained by the sinusoidal modeling assumption present in each of them. They are unable to easily deal with complex three dimensional behaviors such as flow around objects and dynamically changing boundaries. A summary of the above methods and their application to modeling and rendering of ocean wave environments can be found in [Tessendorf 2001].

In order to obtain water models which could potentially be used in a dynamic animation environment, researchers turned towards using two dimensional approximations to the full 3D Navier-Stokes equations. [Kass and Miller 1990] use a linearized form of the 2D shallow water equations to obtain a height field representation of the water surface. A pressure defined height field formulation was used by [Chen and Lobo 1994] in fluid simulations with moving obstacles. [O'Brien and Hodgins 1995] used a height model combined with a particle system in order to simulate splashing liquids. The use of a height field gives a three dimensional look to a two dimensional flow calculation, but it constrains the surface to be a function, i.e. the surface passes the vertical line test where for each (x,y) position there is at most one z value. The surface of a crashing wave or of water being poured into a glass does not satisfy the vertical line test. Use of particle systems permits the surface to become multivalued. A viscous spring particle representation of a liquid has been proposed by [Miller and Pearce 1989]. An alternative molecular dynamics approach to the simulation of particles in the liquid phase has been developed by [Terzopoulos et al. 1989]. Particle methods, while quite versatile, can pose difficulties when trying to reconstruct a smooth water surface from the locations of the particles alone.

The simulation of complex water effects using the full 3D Navier-Stokes equations has been based upon the large amount of research done by the computational fluid dynamics community over the past 50 years. [Foster and Metaxas 1996] utilized the work of [Harlow and Welch 1965] in developing a 3D Navier-Stokes methodology for the realistic animation of liquids. Further CFD enhancements to the traditional marker and cell method of Harlow and Welch which allow one to place particles only near the surface can be found in [Chen et al. 1997]. A semi-Lagrangian "stable fluids" treatment of the convection portion of the Navier-Stokes equations was introduced to the computer graphics community by [Stam 1999] in order to allow the use of significantly larger time steps without hindering stability. [Foster and Fedkiw 2001] made significant contributions to the simulation and control of three dimensional fluid simulations through the introduction of a hybrid liquid volume model combining implicit surfaces and massless marker particles; the formulation of plausible boundary conditions for moving objects in a liquid; the use of an efficient iterative method to solve for the pressure; and a time step subcycling scheme for the particle and implicit surface evolution equations in order to reduce the amount of visual error inherent to the large semi-Lagrangian "stable fluid" time step used for time evolving the fluid velocity and the pressure. The combination of all of the above advances in 3D fluid simulation technology along with ever increasing computational resources has set the stage for the inclusion of fully 3D fluid animation tools in a production environment.

Most work on simulating water at small scales has not specifically addressed rendering and has not used methods that correctly account for all significant light transport paths. Research on the rendering of illumination through water [Watt 1990; Nishita and Nakamae 1994] has used methods based on processing each polygon of a mesh that represents a fairly smooth water surface, so these methods cannot be used for the very complex implicit surfaces that result from our simulations. For the case of 2D wave fields in the open ocean, approaches motivated by physical correctness have produced excellent results [Premoze and Ashikhmin 2000; Tessendorf 2001].

3 Simulation Method

3.1 Motivation

[Foster and Fedkiw 2001] chose to define the liquid volume being simulated as one side of an isocontour of an implicit function, ϕ. The surface of the water was defined by the $\phi = 0$ isocontour with $\phi \leq 0$ representing the water and $\phi > 0$ representing the air. By using an implicit function representation of the liquid volume, they obtained a smooth, temporally coherent liquid surface. They rejected the use of particles alone to represent the liquid surface because it is difficult to a calculate a visually desirable smooth liquid surface from the discrete particles alone. The implicit surface was dynamically evolved in space and time according to the underlying liquid velocity $\vec{u}$. As shown in [Osher and Sethian 1988], the appropriate equation to do this is

$$\phi_t + \vec{u} \cdot \nabla \phi = 0 \tag{1}$$

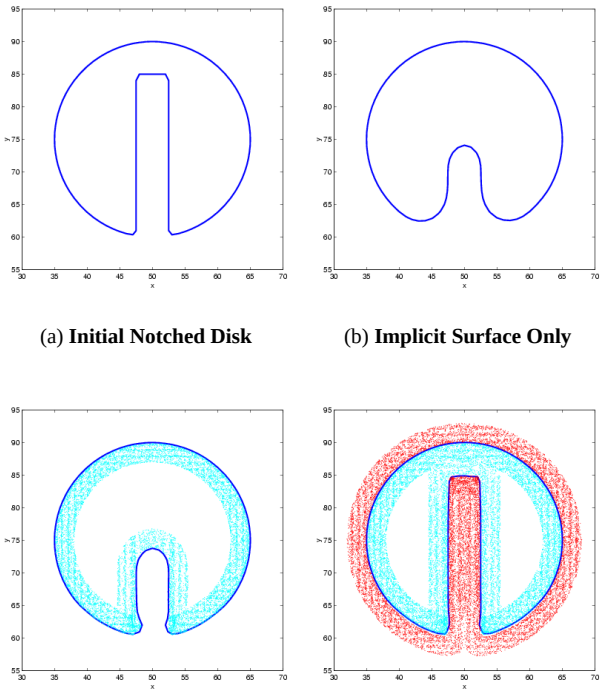
where ϕ_t is the partial derivative of ϕ with respect to time and $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$ is the gradient operator.

An implicit function only approach to modeling the surface will not yield realistic surface behavior due to an excessive amount of volume loss on coarse grids. A seminal advance of [Foster and Fedkiw 2001] in creating realistic liquids for computer animation is the hybridization of the visually pleasing smooth implicit function modeling of the liquid volume with particles that can maintain the liquid volume on coarse grids. The inclusion of particles provides a way for capturing the liveliness of a real liquid with spray and splashing effects. Curvature was used as an indicator for allowing particles to influence the implicit surface representation of the water. This is a natural choice since small droplets of water have very high curvature and dynamic implict surfaces have difficulty resolving features with sharp corners.

Figure 2(a) shows a notched disk that we rotate for one rigid body rotation about the point (50,50). The inside of the disk can be thought of as a volume of liquid. Figure 2(b) shows the result of using an implicit surface only approach (after one rotation) where both the higher and lower corners of the disk are erroneously shaved off causing both loss of visual features and an artificially viscous look for the liquid. This numerical result was obtained using a highly accurate fifth order WENO discretization of equation 1 (see e.g. [Jiang and Peng 2000; Osher and Fedkiw 2002]). For the sake of comparison, we note that [Sethian 1999] only proposes second order accurate methods for discretizing this equation. Figure 2(c) shows the result obtained with our implementation of the method from [Foster and Fedkiw 2001]. The particles inside the disk do not allow the implicit surface to cross over them and help to preserve the two corners near the bottom. However, there is little they can do to stop the implicit surface from drifting away from them near the top corners. This represents loss of air or bubbles as the method erroneously gains liquid volume. This is not desirable since many complex water motions such as wave phenomenon are due in part to differing heights of water columns adjacent to each other. Loss of air in a water column reduces the pressure forces from neighboring columns destroying many of the dynamic splashing effects as well as the overall visually stimulating liveliness of the liquid.

While the hybrid liquid volume model of Foster and Fedkiw attempts to maintain the volume of the liquid accurately, it fails to model the air or more generally the opposite side of the liquid surface. We shift the focus away from maintaining a liquid volume towards maintaining the liquid surface itself. An immediate advantage of this approach is that it leads to symmetry in particle placement. We place particles on both sides of the surface and use them to maintain an accurate representation of the surface itself regardless of what may be on one side or the other. The particles are not meant to track volume, they are intended to correct errors in the surface representation by the implicit function. In [Enright et al. 2002] we showed that this surface only approach leads to the most accurate 3D results for surface tracking ever achieved (in both CFD and CG). This was done for analytical "test" velocity fields. Figure 2(d) shows that this new method correctly computes the rigid body rotation for the notched disk preserving both the water and the air volumes so that more realistic water motion can be obtained.

In this current paper, we couple this new method to real velocity fields and fluid dynamics calculations for the first time. Representative results of this new method can be seen in figure 3. A ball is thrown into a tank of water with the same tank geometry, grid spacing and ball speed as seen in figure 4 (courtesy of [Foster and Fedkiw 2001]). The resulting splash after the ball impacts the surface of the water is dramatically different between the two figures. Our new method produces the well formed, thin sheet one would visually expect. Note that the distorted look of the ball in our figure is due to the correct calculation of the refraction of light when it passes through the surface of the water. To give an indication of the additional computational cost incurred using our new simulation method, figure 3 took about 11 minutes per frame, figure 4 took about 7 minutes per frame and a level set only solution takes about 3 minutes per frame. The rendering time for figure 3 was approximately 15 minutes per frame.



(a) **Initial Notched Disk** (b) **Implicit Surface Only**

(c) **Particles Inside Only** (d) **Our New Method**

Figure 2: **Rigid Body Rotation Test**

3.2 Particle Level Set Method

3.2.1 Initialization of Particles

Two sets of particles are randomly placed in a "thickened" surface region (we use three grid cells on each side of the surface) with *positive* particles in the $\phi > 0$ region and *negative* particles in the $\phi \leq 0$ region. There is no need to place particles far away from the surface since the sign of the level set function readily identifies these regions gaining large computational savings. The number of particles placed in each cell is an adjustable parameter that can be used to control the amount of resolution, e.g. we use 64 partlces per cell for most of our examples. Each particle possesses a radius, r_p, which is constrained to take a minimum and maximum value based upon the size of the underlying computational cells used in the simulation. A minimum radius of $.1\min(\Delta x, \Delta y, \Delta z)$ and maximum radius of $.5\min(\Delta x, \Delta y, \Delta z)$ appear to work well. The radius of a particle changes dynamically throughout the simulation, since a particle's location relative to the surface changes. The radius is set according to:

$$r_p = \begin{cases} r_{max} & \text{if } s_p\phi(\vec{x}_p) > r_{max} \\ s_p\phi(\vec{x}_p) & \text{if } r_{min} \leq s_p\phi(\vec{x}_p) \leq r_{max} \\ r_{min} & \text{if } s_p\phi(\vec{x}_p) < r_{min} \end{cases}, \qquad (2)$$

where s_p is the sign of the particle (+1 for positive particles and -1 for negative particles). This radius adjustment keeps the boundary of the spherical particle tangent to the surface whenever possible. This fact combined with the overlapping nature of the particle

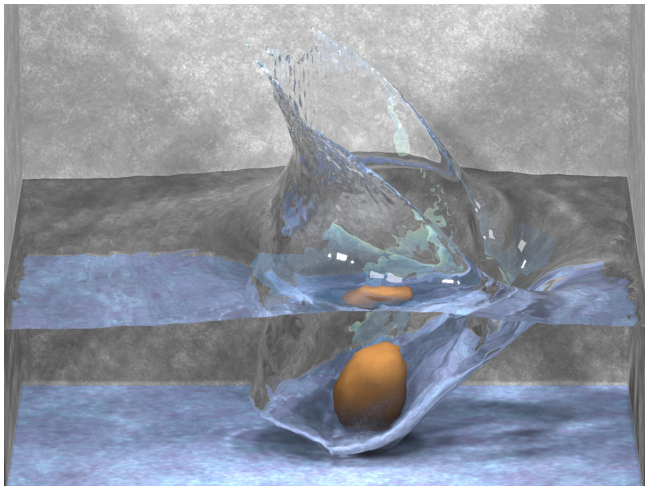


Figure 3: **Our New Method (140x110x90 grid cells).**

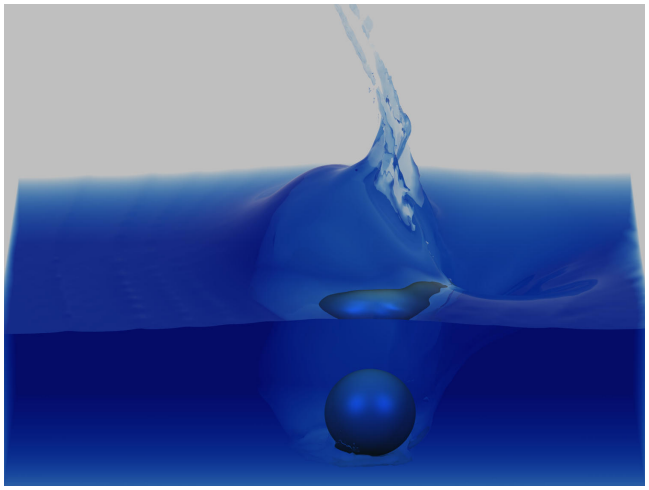


Figure 4: **Foster and Fedkiw 2001 (140x110x90 grid cells).**

spheres allows for an enhanced reconstruction capability of the liquid surface.

3.2.2 Time Integration

The marker particles and the implicit function are separately integrated forward in time using a forward Euler time integration scheme. The implicit function is integrated forward using equation 1, while the particles are passively advected with the flow using $d\vec{x}_p/dt = \vec{u}_p$, where $\vec{u}_p$ is the fluid velocity interpolated to the particle position $\vec{x}_p$.

3.2.3 Error Correction of the Implicit Surface

Identification of Error: The main role of the particles is to detect when the implicit surface has suffered inaccuracies due to the coarseness of the computational grid in regions with sharp features. Particles that are on the wrong side of the interface by more than their radius, as determined by a locally interpolated value of ϕ at the particle position $\vec{x}_p$, are considered to have *escaped* their side of the interface. This indicates errors in the implicit surface representation. In smooth, well resolved regions of the interface, our dynamic implicit surface is highly accurate and particles do not drift a non-trivial distance across the interface.

Quantification of Error: We associate a spherical implicit function , designated ϕ_p, with each particle p whose size is determined by the particle radius, i.e.

$$\phi_p(\vec{x}) = s_p(r_p - |\vec{x} - \vec{x}_p|). \tag{3}$$

Any difference in ϕ from ϕ_p indicates errors in the implicit function representation of the surface. That is, the implicit version of the surface and the particle version of the surface disagree.

Error Correction: We use escaped positive particles to rebuild the $\phi > 0$ region and escaped negative particles to rebuild the $\phi \leq 0$ region as defined by the implicit function. The reconstruction of the implicit surface occurs locally within the cell that each escaped particle currently occupies. Using equation 3, the ϕ_p values of escaped particles are calculated for the eight grid points on the boundary of the cell containing the particle. This value is compared to the current value of ϕ for each grid point and we take the smaller value (in magnitude) which is the value closest to the $\phi = 0$ isocontour defining the surface. We do this for all escaped positive and escaped negative particles. The result is an improved representation of the surface of the liquid.

3.2.4 When To Apply Error Correction

We apply the error correction method discussed above after any computational step in which ϕ has been modified in some way. This occurs when ϕ is integrated forward in time and when the implicit function is smoothed to obtain a visually pleasing surface. We smooth the implicit surface with an equation of the form

$$\phi_\tau = -S(\phi_{\tau=0})(|\nabla\phi| - 1), \tag{4}$$

where τ is a fictitious time and $S(\phi)$ is a smoothed signed distance function given by

$$S(\phi) = \frac{\phi}{\sqrt{\phi^2 + (\Delta x)^2}}. \tag{5}$$

More details on this are given in [Foster and Fedkiw 2001].

3.2.5 Particle Reseeding

In complex flows, a liquid interface can be stretched and torn in a dynamic fashion. The use of only an initial seeding of particles will not capture these effects well, as regions will form that lack a sufficient number of particles to adequately perform the error correction step. Periodically, e.g. every 20 frames, we randomly reseed particles about the "thickened" interface to avoid this dilema. This is done by randomly placing particles near the interface, and then using geometric information contained within the implicit function (e.g. the direction of the shortest possible path to the surface is given by $\vec{N} = \nabla\phi/|\nabla\phi|$) to move the particles to their respective domains, $\phi > 0$ or $\phi \leq 0$. The goal of this reseeding step is to preserve the initial particle resolution of the interface, e.g. 64 particles per cell. Thus, if a given cell has too few or too many particles, some can be added or deleted respectively.

3.2.6 A Note on Alternative Methods

If we felt that preserving the volume of the fluid was absolutely necessary in order to obtain visually pleasing fluid behavior, we would have chosen to use a volume of fluid (VOF) [Hirt and Nichols 1981] representation of the fluid. Although VOF methods explicitly conserve volume, they produce visually disturbing artifacts allowing thin liquid sheets to artificially break up and form "blobbies" and

"flotsam" of liquid. Also, a visually desirable smooth fluid interface is difficult to construct when using these methods.

Another alternative is to explicitly discretize the free surface with particles and maintain a connectivity list between particles, see e.g. [Unverdi and Tryggvason 1992]. This connectivity list is difficult to maintain when parts of the free surface break apart or merge together as is often seen in complex flows of water and other liquids. Our approach avoids the especially difficult issues associated with maintaining particle connectivity information.

3.3 Velocities at the Surface

Although the Navier-Stokes equations can be used to find the velocity within the liquid volume, boundary conditions are needed for the velocity on the air side near the free surface. These boundary condition velocities are used in updating the Navier-Stokes equations, moving the surface, and moving the particles placed near the surface. The velocity at the free surface of the water can be determined through the usual enforcement of the conservation of mass (volume) constraint, $\nabla \cdot \vec{u} = 0$, where $\vec{u} = (u, v, w)$ is the velocity of the liquid. This equation allows us to determine the velocities on all the faces of computational cells that contain the $\phi = 0$ isocontour. Unfortunately, the procedure for doing this is not unique when more than one face of a cell needs a boundary condition velocity. A variety of methods have been proposed, e.g. see [Chen et al. 1995] and [Foster and Metaxas 1996].

We propose a different approach altogether, the extrapolation of the velocity across the surface into the surrounding air. As the computational grid is refined, this method is equivalent to the usual method, but it gives a smoother and more visually pleasing motion of the surface on coarser (practical sized) grids. We extrapolate the velocity out a few grid cells into the air, obtaining boundary condition velocities in a band of cells on the air side of the surface. This allows us to use higher order accurate methods and obtain better results when moving the implicit surface using equation 1 and also provides velocities for updating the position of particles on the air side of the surface. Velocity extrapolation also assists in the implementation of the semi-Lagrangian "stable fluid" method, since there are times when characteristics generated by this approach look back across the interface (a number of cells) into the air region for valid velocities.

3.3.1 Extrapolation Method

The equation modeling this extrapolation for the x component of the velocity, u, is given by

$$\frac{\partial u}{\partial \tau} = -\vec{N} \cdot \nabla u, \tag{6}$$

where $\vec{N} = (n_x, n_y, n_z)$ is a unit vector perpendicular to the implicit surface and τ is fictitious time. A similar equation holds for the v and w components of velocity field. Since we use an implicit surface to describe the fluid, $\vec{N} = \nabla\phi / |\nabla\phi|$. Fast methods exist for solving this equation in $O(n \log n)$ time, where n is the number of grid points that one needs to extrapolate over, in our case a five grid cell thick band on the air side of the interface. The fast method is based upon the observation that information in equation 6 propagates in only one direction away from the surface. This implies that we do not have to revisit previously computed values of $\vec{u}_{ext}$ (the extrapolated velocity) if we perform the calculation in the correct order. The order is determined by the value of ϕ allowing us to do an $O(n \log n)$ sorting of the points before beginning the calculation. The value of u itself is determined by enforcing the condition at steady state, namely $\nabla\phi \cdot \nabla u = 0$ where the derivatives are determined using previously calculated values of ϕ and u. From this scalar equation, a new value of u can be determined, and then we proceed to the next point corresponding to the next smallest value of ϕ, etc. Further details of this method are discussed in [Adalsteinsson and Sethian 1999].

3.3.2 Velocity Advection

The momentum portion of the Navier-Stokes equations is:

$$\vec{u}_t = -\vec{u} \cdot \nabla\vec{u} + \nu \nabla \cdot (\nabla\vec{u}) - \frac{1}{\rho}\nabla p + \vec{g}, \tag{7}$$

where ν is the kinematic viscosity of the fluid, ρ is the density of the fluid, p is the pressure, and $\vec{g}$ is an externally applied gravity field. We use the semi-Lagrangian "stable fluids" method [Stam 1999] to update the convective portion of this equation, i.e. the $\vec{u} \cdot \nabla\vec{u}$ term. This method calculates the first term on the left hand side of equation 7 by following the fluid characteristics backwards in time to determine from which computational cell the volume of fluid came, and then taking an average of the appropriate velocities there. This allows one to stably take much larger time steps than would be allowed using other time advancement schemes for velocity advection. A consequence of the now allowed large semi-Lagrangian time step is that near the surface, we might look across the interface as many as a few grid cells into the air region to find velocities. In a standard approach, valid velocities are not defined in this region. However, our velocity extrapolation technique easily handles this case ensuring that physically plausible velocities exist a few grid cells into the air region. In fact, we extrapolate the velocity the maximum distance from the surface that would be allowed during a semi-Lagrangian "stable fluids" time step guaranteeing that a smooth, physically plausible and visually appealing velocity can be found there.

3.3.3 Control

Our velocity extrapolation method enables us to apply a newly devised method for controlling the nature of the surface motion. This is done simply by modifying the extrapolated velocities on the air side of the surface. For example, to model wind-blown water as a result of air drag, we take a convex combination of the extrapolated velocities with a pre-determined wind velocity field

$$\vec{u} = (1 - \alpha)\vec{u}_{ext} + \alpha\vec{u}_{wind}, \tag{8}$$

where $\vec{u}_{ext}$ is the extrapolated velocity, $\vec{u}_{wind}$ a desired air-like velocity, and $0 \leq \alpha \leq 1$ the mixing constant. This can be applied throughtout the surface or only locally in select portions of the computational domain as desired. Note that setting $\vec{u}_{wind} = 0$ forces churning water to settle down faster with the fastest settling resulting from $\alpha = 1$. All of the figures shown in the paper used $\alpha = 0$, but we demonstrate how α can be used to force a poured glass of water to settle more quickly in the accompanying video.

3.4 Summary

We divide up the computational domain into voxels with the components of $\vec{u}$ stored on the appropriate faces and p, ρ and ϕ stored at the center of each cell. This arrangement of computational variables is the classic staggered MAC-style arrangement [Harlow and Welch 1965]. The density of a given cell is given by the value of ϕ at the center of the cell. The evolution of $\vec{u}$, p, ρ and ϕ in a given time step is performed in a series of three steps as outlined below:

1. The current surface velocity is smoothly extrapolated across the surface into the air region as discussed in section 3.3.1. Appropriate control behavior modifications to the velocity field are made.

2. The water surface and marker particles are integrated forward in time via an explicit time step subcycling method with the appropriate corrections to ϕ as described in section 3.2.3.

3. The velocity field is updated with equation 7 using the updated values for ρ. This is done by first using the semi-Lagrangian "stable fluid" method to find an estimate for the velocity. This estimate is further augmented by the viscous and forcing terms. The spatial derivatives used in calculating these terms are calculated using a standard centered second order accurate finite difference scheme. Then a system of linear equations is assembled and solved for the pressure in order to make this intermediate velocity field divergence free. The newly calculated pressure is applied as a correction to the intermediate velocity in order to fully update the water's velocity field. Interaction of the liquid with objects, walls, etc. is treated here as well. For this step, we follow exactly the method of [Foster and Fedkiw 2001] and refer the reader there for more details.

This sequence of steps is repeated until a user defined stopping point is reached. The time step for each iteration of the above steps is determined using the water's velocity to calculate an appropriate CFL condition which is approximately five times larger than the traditional CFL condition used in fluid simulations (the semi-Lagrangian "stable fluids" method allows this). Also, any viscosity related CFL restrictions are locally dealt with by reducing the viscosity in the offending cells in order to allow for our larger time step.

3.5 Rendering

Our results are rendered using a physically based Monte Carlo ray tracer capable of handling all types of illumination using photon maps and irradiance caching [Jensen 2001]. To integrate our simulation system with the renderer we implemented a geometry primitive that intersects rays with the implicit surface directly by solving for the root of the signed distance along the ray. Depending on the accuracy required by a particular scene we use either a trilinear or a tricubic filter [Marschner and Lobb 1994] to reconstruct ϕ. The normal is computed using trilinearly interpolated central differences for the trilinear surface, or simply using the derivative of the reconstructed tricubic surface.

The properties of ϕ have two advantages in the rendering context. First, intersecting a ray with the surface is much more efficient than it would be for an isosurface of a general function. The signed distance function provides a lower bound on the distance to the intersection along the ray, allowing us to take large steps when the current point is far from the surface. Second, it is easy to provide for motion blur in the standard distribution ray tracing framework. To compute the surface at an intermediate time between two frames we simply interpolate between the two signed distance volumes and use the same intersection algorithm unchanged. For the small motions that occur between frames the special properties of ϕ are not significantly compromised.

4 Results

4.1 Pouring Water Into A Glass

Figures 1 and 7 show the high degree of complexity in the water surface. Note the liveliness of the surface of the water when the water is initially being poured. The ability to maintain the visually pleasing thin sheets of water during the turbulent mixing phase is a consequence of our new method. We do not loose any of the fine detail with regards to the air pockets formed, since we model both sides of the water surface. Even though the calculation was performed on a Cartesian computational grid, the glass was shaped as a cylinder on the grid, with the grid points outside the cylinder treated as an object which does not permit the fluid to interpenetrate it. The glass was modeled as smooth and clear in order to highlight the action of the water being poured into the glass. The computational cost was approximately 8 minutes per frame.

To our knowledge, the only other complex, three dimensional simulation of a liquid being poured into a glass for computer animation is from the Gingerbread Man torture scene in the feature film "Shrek". Milk lacks the transparency of water making it difficult to clearly view the dynamic behavior of the milk surface. Also, the modeling of a thick polygonal glass with a rough surface does not provide a clear view of the milk making a direct comparison difficult.

The scene used to render our result contains just a simple cylindrical glass, the simulated water surface, and a few texture mapped polygons for the background. Illumination comes from a physically based sky model and two area sources. Because water only reflects and refracts other objects, the scene including the sky is very important to the appearance. The glass and the water together create three dielectric interfaces: glass-air, water-air, and air-water, so the inside surface of the glass has two sets of material properties depending on whether it is inside or outside the implicitly defined surface (which is easy to determine by checking the sign of ϕ). The illumination in the shadow of the glass is provided by a photon map, and illumination from the sky on diffuse surfaces in the scene is computed using Monte Carlo integration. We also note that milk would not present as difficult a global illumination problem due to its lack of transparency. Motion blur was included for these renderings. The rendering time for the glass was approximately 15 minutes per frame.

4.2 Breaking Wave

As a second example, we have performed a breaking wave calculation in a numerical wave tank as shown in figure 8. To begin to model this pheonmenon, we needed to chose an initial condition for the wave. We chose to use the theoretical solution of a solitary wave of finite amplitude propagating without shape change [Radovitzky and Ortiz 1998]. The initial velocities u and v in the x and y directions respectively and surface height η are given by:

$$u = \sqrt{gd}\frac{H}{d}\operatorname{sech}^2\left[\sqrt{\frac{3H}{4d^3}}x\right] \quad (9)$$

$$v = \sqrt{3gd}\left(\frac{H}{d}\right)^{3/2}\frac{y}{d}\operatorname{sech}^2\left[\sqrt{\frac{3H}{4d^3}}x\right]\tanh\left[\sqrt{\frac{3H}{4d^3}}x\right] \quad (10)$$

$$\eta = d + H\operatorname{sech}^2\left[\sqrt{\frac{3H}{4d^3}}x\right], \quad (11)$$

where g is the gravitational constant 9.8 m/s. For our simulations, we set $H = 7$ and $d = 10$. We used the same initial conditions in three spatial dimensions, replicating the two dimensional initial conditions along the z-axis.

Next, we needed to introduce a model of a sloping underwater shelf in order cause the propagating pulse to actually pile up on itself and form a breaking wave. We performed a variety of two dimensional tests to determine the best underwater shelf geometry to generate a visually pleasing breaking wave. Figure 5 is a sequence of frames from one of the two dimensional tests we ran with the same x-y cross section as can be found in our 3D example. We found this prototyping technique to be a fast and easy way to explore possible breaking wave behaviors in a fraction of the time it would take to run a fully three dimensional test case.

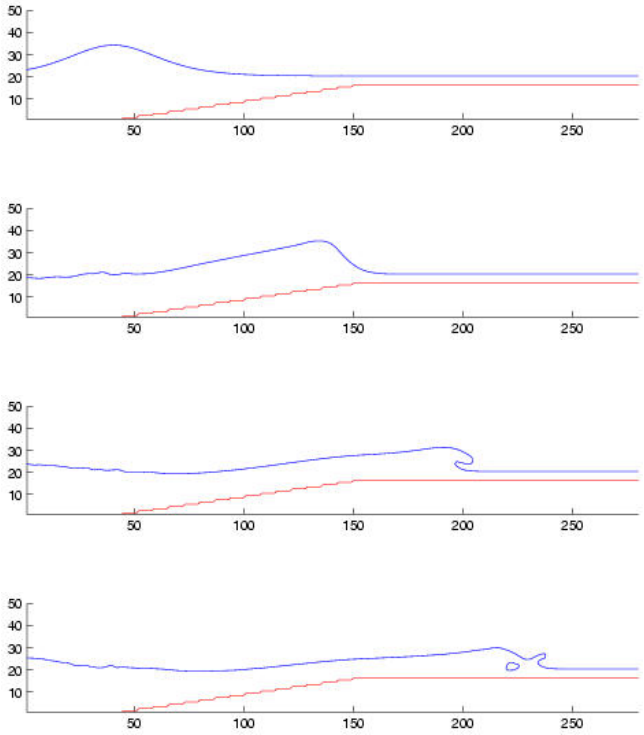


Figure 5: **Two Dimensional Breaking Wave**

After determining the best submerged shelf geometry to generate a breaking wave, we generated a slight tilt in the geometry in order to induce a curl in the break of the wave and enhance the three dimensional look. A sketch of the shelf geometry used in our three dimensional calculations is shown in figure 6. An incline of slope 1:7 rises up from the seabottom to a depth of 2 m below the surface of the water. Instead of allowing the incline to go all the way to the surface we chose to have it flatten out at this depth in order to illustrate the splash up effects after the initial breaking of the wave.

Next we ran a fully three dimensional simulation, the results of which can be seen in figures 8 and 9. The wave has the intended curling effect. The formation of a tube of air is clearly seen after the wave splashes down, with the "air" particles maintaining the tube even after the wave begins a secondary splash up. We observe some solely three dimensional effects including the fingering of the breaking wave. The computational cost was approximately 3 minutes per frame.

Because of the very different scale of this simulation as compared to the water glass, different optical effects are important. We model the water volume under the surface, which in surf is densely populated with scattering particles, as a gray diffuse reflector. The color of the water comes from a cloudy blue sky. Because the features that can be resolved are so fine compared to the grid resolution, we used cubic interpolation for this scene.

The simulation shown captures the basic phenomena of the breaking wave on a very coarse grid, but in real waves there are small-scale features that, while not important to the *behavior* of the large wave, are very important for its *appearance*. Coupling a 2D simulation for the small scale features to the wave simulation is a promising avenue for future work. Once those waves are incorporated it will become important to treat diffusion of light through the water more correctly. Also, no texture mapping was performed, e.g. use of a Philips spectrum [Tessendorf 2001], or bump mapping technique [Fournier and Reeves 1986]. Another challenge will be

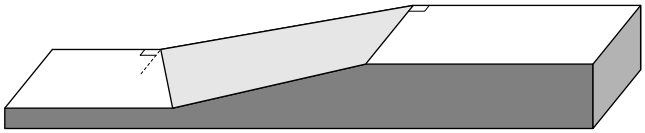


Figure 6: **Submerged Shelf**

to augment our method to naturally handle spray and foam.

5 Conclusion and Future Work

We have presented some novel computational methods for enhanced surface tracking and modeling of the surface motion. Combining these leads to the possibility of creating photorealistic behavior in 3D water simulations for the purpose of computer graphics animation. We discussed some advances in rendering the simulated photorealistic behavior in order to complete the illusion that real water is being seen. The computational methods presented can easily be included into an existing three dimensional fluid simulation animation tool. While we have not done any texture mapping of the water surfaces, we believe that our "thickened" front tracking approach should enable texture mapping of an implicitly defined fluid surface, and we will pursue this as future work.

6 Acknowledgment

Research supported in part by an ONR YIP and PECASE award N00014-01-1-0620, NSF DMS-0106694, NSF ACI-0121288, NSF IIS-0085864, and the DOE ASCI Academic Strategic Alliances Program (LLNL contract B341491).

The authors acknowledge Henrik Wann Jensen for the use of his *dali* rendering system and for his help with the extensions that were added to it for this paper.

The third author would like to acknowledge the fruitful collaboration with Nick Foster regarding the hybridization of particle and level set methods that was published in [Foster and Fedkiw 2001]. Without it, neither [Enright et al. 2002] nor this current work would have been possible.

References

ADALSTEINSSON, D., AND SETHIAN, J. 1999. The fast construction of extension velocities in level set methods. *J. Comp. Phys. 148*, 2–22.

CHEN, J., AND LOBO, N. 1994. Toward interactive-rate simulation of fluids with moving obstacles using the navier-stokes equations. *Computer Graphics and Image Processing 57*, 107–116.

CHEN, S., JOHNSON, D., AND RAAD, P. 1995. Velocity boundary conditions for the simulation of free surface fluid flow. *J. Comp. Phys. 116*, 262–276.

CHEN, S., JOHNSON, D., RAAD, P., AND FADDA, D. 1997. The surface marker and micro cell method. *Int. J. for Num. Meth. in Fluids 25*, 749–778.

ENRIGHT, D., FEDKIW, R., FERZIGER, J., AND MITCHELL, I. 2002. A hybrid particle level set method for improved interface capturing. *J. Comp. Phys.*.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. In *Proceedings of SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 23–30.

FOSTER, N., AND METAXAS, D. 1996. Realistic animation of liquids. *Graphical Models and Image Processing 58*, 471–483.

FOURNIER, A., AND REEVES, W. T. 1986. A simple model of ocean waves. In *Computer Graphics (Proceedings of SIGGRAPH 86), 20(4)*, ACM, 75–84.

HARLOW, F., AND WELCH, J. 1965. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Phys. Fluids 8*, 2182–2189.

HECKBERT, P. S. 1990. Adaptive radiosity textures for bidirectional ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, ACM, 145–154.

HILTZIK, M. A., AND PHAM, A. 2001. Synthetic actors guild. *Los Angeles Times*. May 8, 2001, natl. ed. : A1+.

HIRT, C., AND NICHOLS, B. 1981. Volume of fluid (VOF) method for the dynamics of free boundaries. *J. Comp. Phys. 39*, 201–225.

JENSEN, H. W. 1995. Importance driven path tracing using the photon map. In *Eurographics Rendering Workshop 1995*, 326–335.

JENSEN, H. W. 2001. *Realistic Image Synthesis Using Photon Maps*. A K Peters.

JIANG, G.-S., AND PENG, D. 2000. Weighted eno schemes for hamilton-jacobi equations. *SIAM J. Sci. Comput. 21*, 2126–2143.

KAJIYA, J. T. 1986. The rendering equation. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, vol. 20, 143–150.

KASS, M., AND MILLER, G. 1990. Rapid, stable fluid dynamics for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, ACM, 49–57.

LAFORTUNE, E. P., AND WILLEMS, Y. D. 1993. Bi-directional path tracing. In *Proceedings of Compugraphics '93*, 145–153.

MARSCHNER, S., AND LOBB, R. 1994. An evaluation of reconstruction filters for volume rendering. In *Proceedings of Visualization 94*, IEEE Comput. Soc. Press, 100–107.

MASTEN, G., WATTERBERG, P., AND MAREDA, I. 1987. Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Application 7*, 16–23.

MILLER, G., AND PEARCE, A. 1989. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics 13*, 3, 305–309.

NISHITA, T., AND NAKAMAE, E. 1994. Method of displaying optical effects within water using accumulation buffer. In *Proceedings of SIGGRAPH 94*, ACM SIGGRAPH / ACM Press, Computer Graphics Proceedings, Annual Conference Series, ACM, 373–381.

O'BRIEN, J., AND HODGINS, J. 1995. Dynamic simulation of splashing fluids. In *Proceedings of Computer Animation 95*, 198–205.

OSHER, S., AND FEDKIW, R. 2002. *The Level Set Method and Dynamic Implicit Surfaces*. Springer-Verlag, New York.

OSHER, S., AND SETHIAN, J. 1988. Fronts propagating with curvature dependent speed: Algorithms based on hamiliton-jacobi formulations. *J. Comp. Phys. 79*, 12–49.

PEACHEY, D. R. 1986. Modeling waves and surf. In *Computer Graphics (Proceedings of SIGGRAPH 86), 20(4)*, ACM, 65–74.

PREMOZE, S., AND ASHIKHMIN, M. 2000. Rendering natural waters. In *The proceedings of Pacific Graphics 2000*, 23–30.

RADOVITZKY, R., AND ORTIZ, M. 1998. Lagrangian finite element analysis of newtonian fluid flows. *Int. J. Numer. Meth. Engng. 43*, 607–619.

SCHACHTER, B. 1980. Long crested wave models. *Computer Graphics and Image Processing 12*, 187–201.

SETHIAN, J. 1999. *Level Set Methods and Fast Marching Methods*. Cambridge University Press.

STAM, J. 1999. Stable fluids. In *Proceedings of SIGGRAPH 99*, ACM SIGGRAPH / Addison Wesley Longman, Computer Graphics Proceedings, Annual Conference Series, ACM, 121–128.

TERZOPOULOS, D., PLATT, J., AND FLEISCHER, K. 1989. Heating and melting deformable models (from goop to glop). In *Graphics Interface '89*, 219–226.

TESSENDORF, J. 2001. Simulating ocean water. In *SIGGRAPH 2001 Course Notes*.

TS'O, P. Y., AND BARSKY, B. A. 1987. Modeling and rendering waves: Wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Transactions on Graphics 6*, 3, 191–214.

UNVERDI, S.-O., AND TRYGGVASON, G. 1992. A front-tracking method for viscous, incompressible, multi-fluid flows. *J. Comp. Phys. 100*, 25–37.

VEACH, E., AND GUIBAS, L. J. 1994. Bidirectional estimators for light transport. In *Eurographics Rendering Workshop 1994 Proceedings*, 147–162.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proceedings of SIGGRAPH 97*, ACM SIGGRAPH / Addison Wesley, Computer Graphics Proceedings, Annual Conference Series, ACM, 65–76.

WATT, M. 1990. Light-water interaction using backward beam tracing. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, ACM, 377–385.

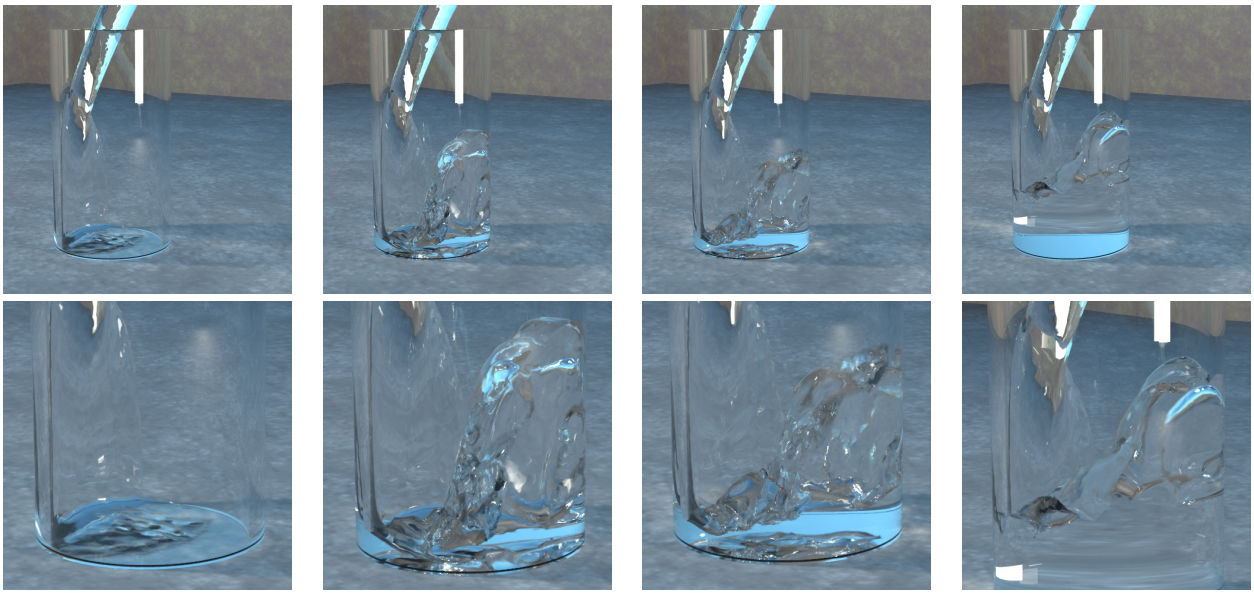


Figure 7: **Water being poured into a clear, cylindrical glass (55x55x120 grid cells). Our method makes possible the fine detail seen in the turbulent mixing of the water and air.**

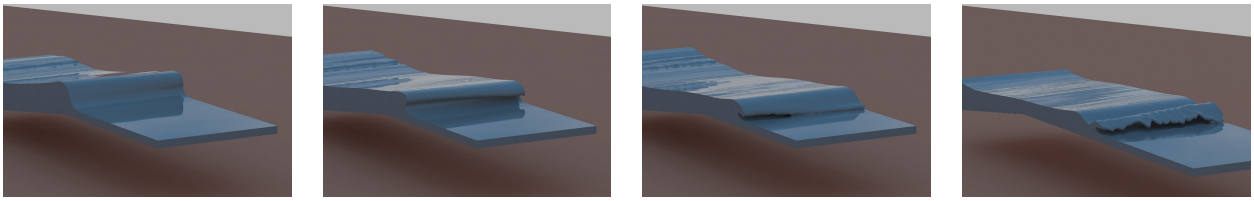


Figure 8: **Abstract view of wave breaking on a submerged shelf (360x50x80 grid cells). Note the ability to properly model the initial breaking (first three frames) and secondary splash up (last frame) phases.**

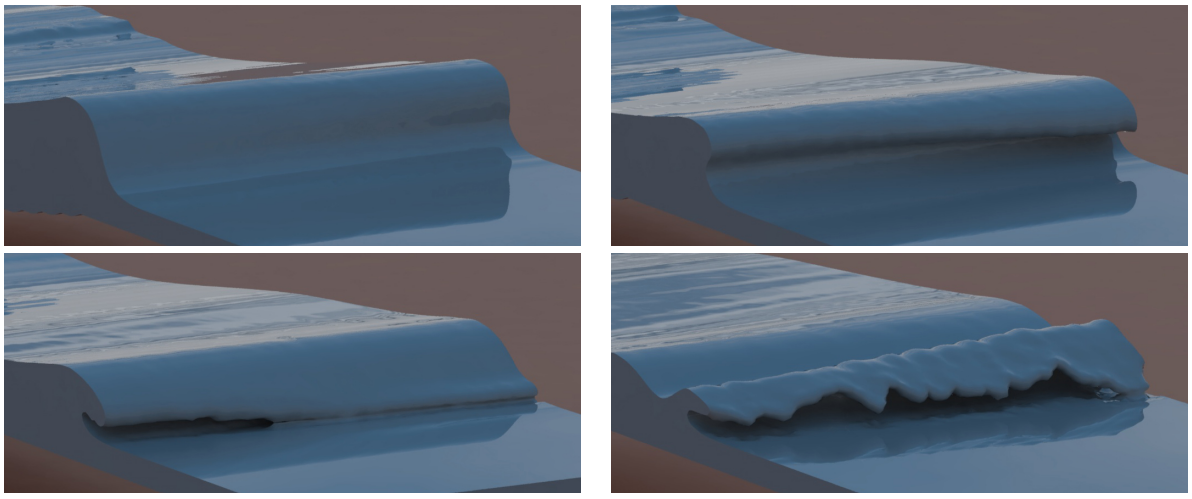


Figure 9: **Closeup of wave action seen in figure 8.**

The notes for these slides can be found on the course CDROM.

Stable Fluids

Jos Stam
Alias|wavefront
Seattle, USA

Fluids in Computer Graphics

- Fast.
- Looks good.
- Easy to code.

Fluid Mechanics

- Natural framework for fluid modeling
  Full Navier-Stokes Equations
- Has a long history
  reuse code/algorithms
- Equations are hard to solve
  non-linear

Previous Work (computer graphics)

Two dimensions:

- Yaeger & Upson 86 + Gamito et al. 95 (vortex blobs)
- Chen et al. 97 (explicit in time, finite differences)

Three-dimensions:

- Foster & Metaxas 97 (explicit in time, finite differences)

unstable

Inaccurate schemes can be useful

Main Contribution

Stable Navier-Stokes solver

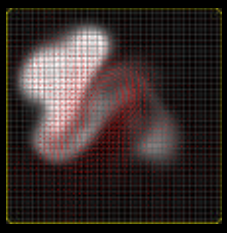
Any time step can be used

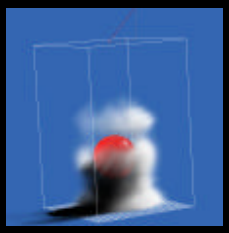
Bigger time steps = faster simulations

NOT accurate

Application

Use velocity to move densities:





Application

Use velocity to move densities:

While (simulating)
- Get force from UI
- Get density source from UI
- Update velocity
- Update density
- Display density

Equations

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

+ velocity should conserve mass

Equations very similar

Equations

Evolution of density (assume velocity known)

$$\boxed{\frac{\partial \rho}{\partial t}} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Over a time step...

Equations

Evolution of density (assume velocity known)

$$\frac{\partial \rho}{\partial t} = \boxed{-(\mathbf{u} \cdot \nabla)\rho} + \kappa \nabla^2 \rho + S$$

Density changes in the direction of the flow

Equations

Evolution of density (assume velocity known)

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \boxed{\kappa \nabla^2 \rho} + S$$

Density diffuses over time

Equations

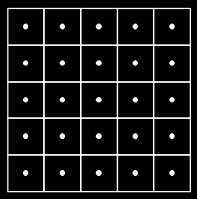
Evolution of density (assume velocity known)

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + \boxed{S}$$

Increases due to sources from the UI

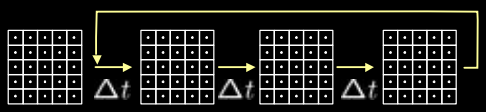
Algorithm

Subdivide space into voxels



Velocity + density defined in the center of each voxel

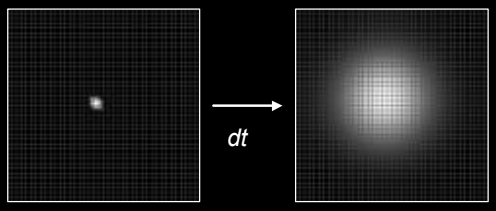
Algorithm



add source | diffuse | move

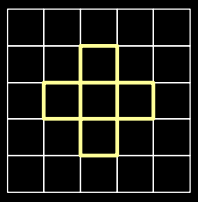
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Diffusing Densities



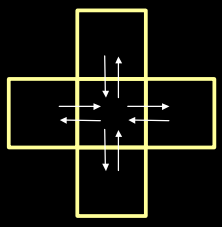
D^n D^{n+1}

Diffusing Densities



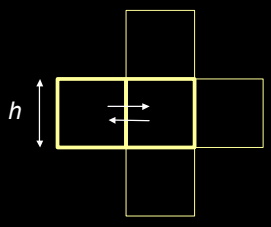
Exchange of density between neighbors

Diffusing Densities



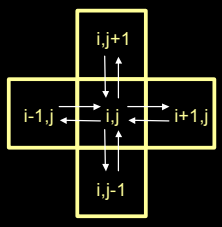
Exchange of density between neighbors

Diffusing Densities



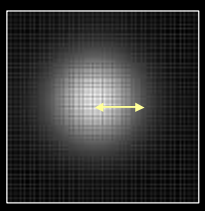
Change = density flux in - density flux out
= $k\,dt$ (neighbor - center) / h^2

Diffusing Densities



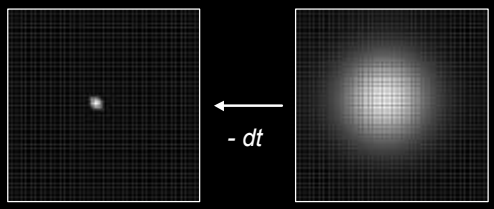
$D^{n+1}_{i,j} = D^n_{i,j} + k\,dt\,(D^n_{i-1,j} + D^n_{i+1,j} + D^n_{i,j-1} + D^n_{i,j+1} - 4D^n_{i,j})/h^2$

Diffusing Densities



Unstable when $k\,dt / h^2 > 1/2$

Diffusing Densities



Find densities which when diffused backward in time give the original densities.

Diffusing Densities

Linear system:

$D^{n+1}_{i,j} - k\,dt\,(D^{n+1}_{i-1,j}+D^{n+1}_{i+1,j}+D^{n+1}_{i,j-1}+D^{n+1}_{i,j+1}-4D^{n+1}_{i,j})/h^2 = D^n_{i,j}$

$A\,x = b$

A can be huge *but* is sparse

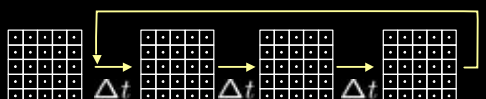
-> requires fast linear solver

Diffusing Densities

Linear solvers:

Name	Cost	Comments
Gaussian elimination	N^3	Use only for very small N (test code)
Jacobi/SOR relaxation	N^2	Easy to code but slow
FFT/cyclical reduction	N logN	Use when no internal boundaries
Conjugate gradient	$N^{1.5}$	Use when internal boundaries
Multi-grid	N	Slower than FFT in practice. Hard to code when internal boundaries present

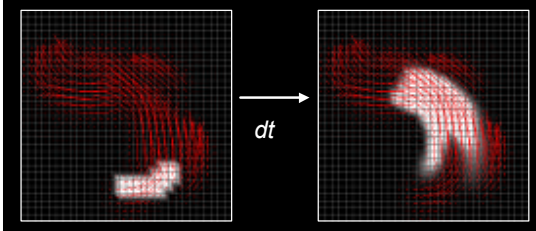
Algorithm



add source diffuse **move**

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u}\cdot\nabla)\,\rho + \kappa\nabla^2\rho + S$$

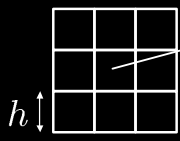
Moving Densities



Velocity known

Moving Densities

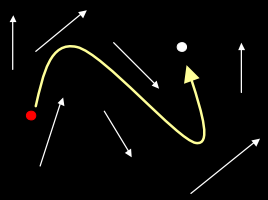
Finite Differences: transfer only between neighbors



Unstable when $\Delta t\ |\mathbf{u}| > h$

Moving Densities

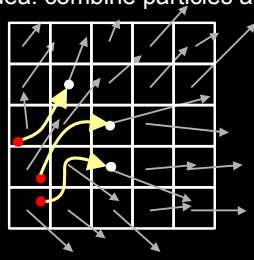
Easy if density defined on particles



Any time step ok

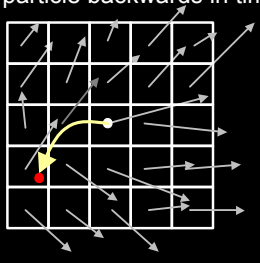
Moving Densities

Key Idea: combine particles and grids



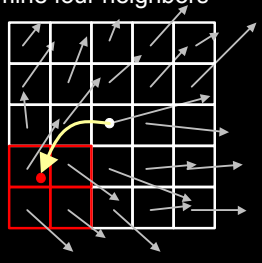
Moving Densities

Trace particle backwards in time



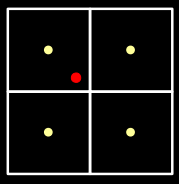
Moving Densities

Determine four neighbors



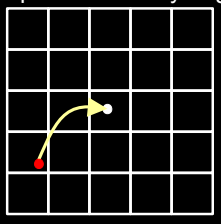
Moving Densities

Interpolate the density at new location



Moving Densities

Set interpolated density at grid location



Requires two grids

Moving Densities

This scheme is unconditionally stable:

$$\rho_{int} = (1-s)\rho_0 + s\rho_1$$

$$\rho_0, \rho_1 \leq \rho_{max}$$

$$\rho_{int} \leq (1-s+s)\rho_{max} \leq \rho_{max}$$

⟶ density is always bounded

Computing Velocities

Computing Velocities

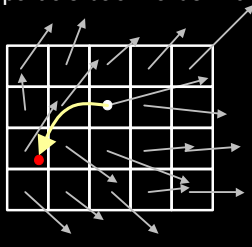
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u}\cdot\nabla)\rho + \kappa\nabla^2\rho + S$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{f}$$

Velocity is moved by itself

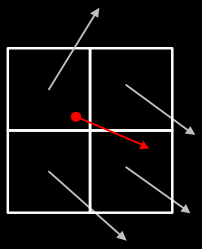
Moving Velocity

Trace particle backwards in time



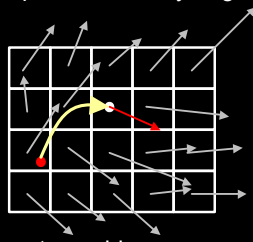
Moving Velocity

Interpolate the velocity at new location



Moving Velocity

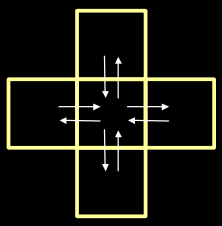
Set interpolated velocity at grid location



Requires two grids

Conservation of Mass

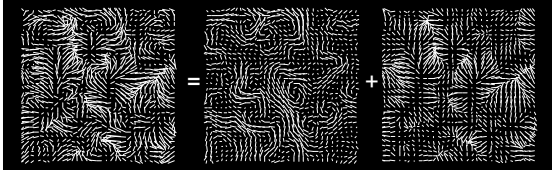
Conservation of Mass



Flow into cell = Flow out of the cell

$U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1} = 0$ not

Conservation of Mass



Our field = mass conserving + gradient

Hodge decomposition

Conservation of Mass

Mass conserving = our field - gradient

Conservation of Mass

Scalar field satisfies a Poisson Equation:

$$P_{i+1,j}+P_{i-1,j}+P_{i,j+1}+P_{i,j-1} - 4\,P_{i,j} = (U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1})\,h$$

Linear system

Summary

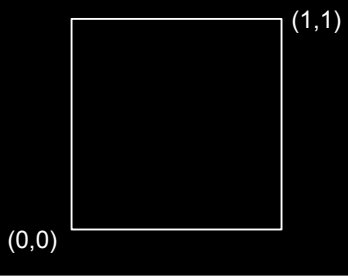
```
UpdateVelocity(U1,U0,F,visc,dt)
    AddForce(U1,U0,F,dt)
    Diffuse(U0,U1,visc,dt)
    Move(U1,U0,U0,dt)
    ConserveMass(U1,dt)
```

Very easy to code. Only need:
Particle tracer + grid interpolator
Linear solver (FISHPAK or CG)

Show 2D demo

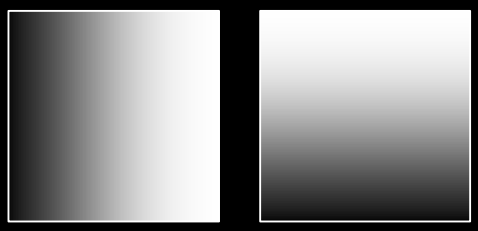
Liquid Textures

Animate texture coordinates



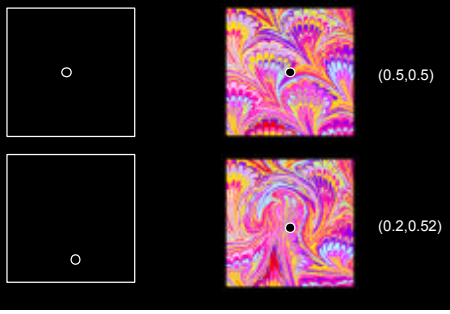
Liquid Textures

Treat texture coordinates as densities



U-coordinate V-coordinate

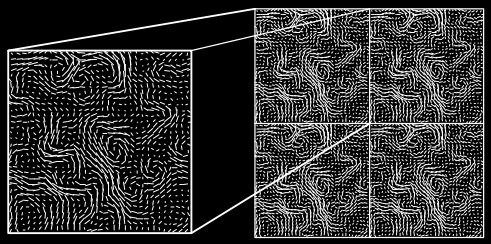
Liquid Textures



Show 2D texture demo

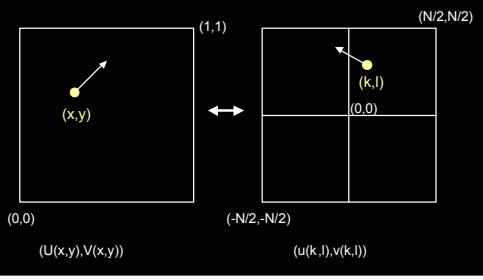
Simple Stable Fluid Solver

Periodic boundaries



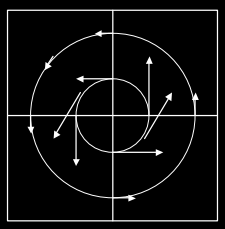
Simple Stable Fluid Solver

Fourier space



Simple Stable Fluid Solver

diffusion = low pass filter: exp(-(k^2+l^2)? ?t)
mass conservation = velocity perpendicular to the fourier directions



Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
    float x, y, x0, y0, f, r, U[2], V[2], s, t;
    int i, j, i0, j0, i1, j1;

    for ( i=0 ; i<n*n ; i++ )
    {
        u[i] += dt*u0[i]; u0[i] = u[i];
        v[i] += dt*v0[i]; v0[i] = v[i];
    }

    for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
    {
        for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
        {
            x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
            i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
            j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
            u[i+n*j] = (1-s)*(1-t)*u0[i0+n*j0]+s*(1-t)*u0[i1+n*j0]+
                       (1-s)*t*u0[i0+n*j1]+s*t*u0[i1+n*j1];
            v[i+n*j] = (1-s)*(1-t)*v0[i0+n*j0]+s*(1-t)*v0[i1+n*j0]+
                       (1-s)*t*v0[i0+n*j1]+s*t*v0[i1+n*j1];
        }
    }

    for ( i=0 ; i<n ; i++ )
        for ( j=0 ; j<n ; j++ )
            { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }

    FFT(1,n,u0); FFT(1,n,v0);

    for ( i=0 ; i<n ; i++ )
    {
        x = 0.5+i;
        for ( j=0 ; j<n ; j++ )
        {
            y = j<n/2 ? j : j-n;
            r = x*x+y*y;
            if ( r==0.0 ) continue;
            f = exp(-r*dt*visc);
            U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
            U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
            u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
            u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
            v0[i  +(n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
            v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
        }
    }

    FFT(-1,n,u0); FFT(-1,n,v0);

    f = 1.0/(n*n);
    for ( i=0 ; i<n ; i++ )
        for ( j=0 ; j<n ; j++ )
            { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j]; }

    return;
}
```

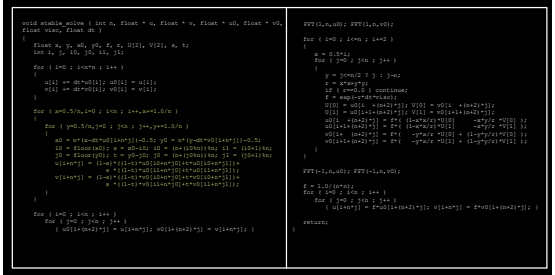
60 lines of (readable) C code

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
    float x, y, x0, y0, f, r, U[2], V[2], s, t;
    int i, j, i0, j0, i1, j1;

    for ( i=0 ; i<n*n ; i++ )
    {
        u[i] += dt*u0[i]; u0[i] = u[i];
        v[i] += dt*v0[i]; v0[i] = v[i];
    }

    for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
    {
        for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
        {
            x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
            i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
            j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
            u[i+n*j] = (1-s)*(1-t)*u0[i0+n*j0]+s*(1-t)*u0[i1+n*j0]+
                       (1-s)*t*u0[i0+n*j1]+s*t*u0[i1+n*j1];
            v[i+n*j] = (1-s)*(1-t)*v0[i0+n*j0]+s*(1-t)*v0[i1+n*j0]+
                       (1-s)*t*v0[i0+n*j1]+s*t*v0[i1+n*j1];
        }
    }

    for ( i=0 ; i<n ; i++ )
        for ( j=0 ; j<n ; j++ )
            { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }

    FFT(1,n,u0); FFT(1,n,v0);

    for ( i=0 ; i<n ; i++ )
    {
        x = 0.5+i;
        for ( j=0 ; j<n ; j++ )
        {
            y = j<n/2 ? j : j-n;
            r = x*x+y*y;
            if ( r==0.0 ) continue;
            f = exp(-r*dt*visc);
            U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
            U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
            u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
            u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
            v0[i  +(n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
            v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
        }
    }

    FFT(-1,n,u0); FFT(-1,n,v0);

    f = 1.0/(n*n);
    for ( i=0 ; i<n ; i++ )
        for ( j=0 ; j<n ; j++ )
            { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j]; }

    return;
}
```

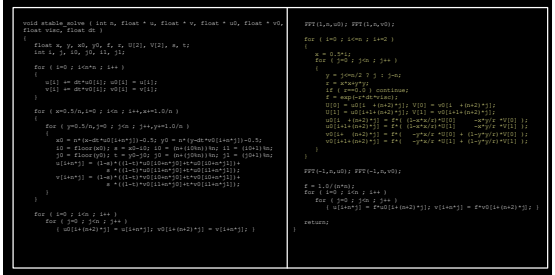
Add forces

Simple Stable Fluid Solver



Move velocity

Simple Stable Fluid Solver



Diffuse + project

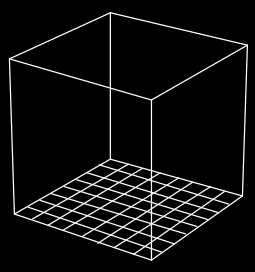
Show 2D simple demo

3D Demos

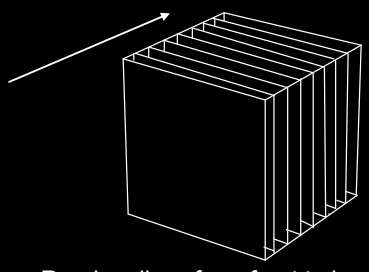
Move 3D solid texture coordinates

Interactive Volume Rendering

Volume Rendering



Volume Rendering



Render slices from front to back

Show 3D demo

Defeating Dissipation

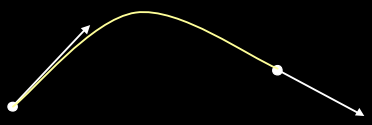
Work with

Ronald Fedkiw & Henrik Wann Jensen

Stanford University

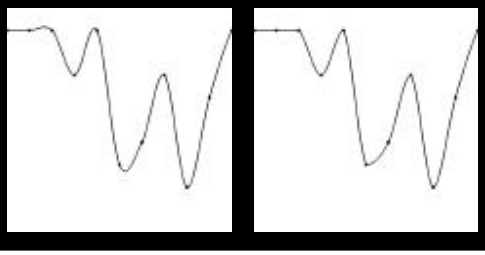
Higher Order Interpolation

Use Hermite interpolation instead of Linear interpolation.



Higher Order Interpolation

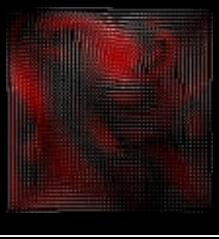
Force interpolant to be monotonic to avoid instabilities



Higher Order Interpolation

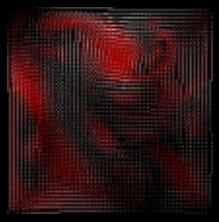
linear Hermite

Vorticity Confinement



$$\omega = \nabla \times \mathbf{u}$$

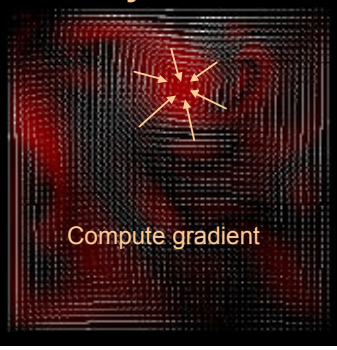
Vorticity Confinement



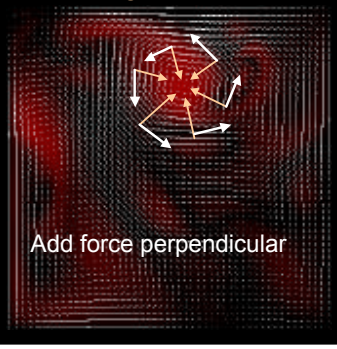
Basic idea :Increase vorticity

John Steinhoff 1987 (Flow Analysis)

Vorticity Confinement



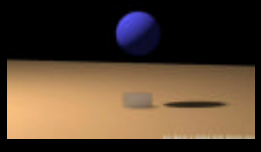
Vorticity Confinement



Show confinement demo

High Quality Renderings

High Quality Renderings

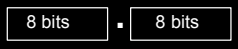


PocketPC demo

Show demo

PocketPC demo

Fixed point math:




```
#define freal short     // 16 bits

#define X1 (1<<8)
#define I2X(i) ((i)<<8)
#define X2I(x) ((x)>>8)
#define F2X(f) ((f)*X1)
#define X2F(x) ((float)(x)/(float)X1)
#define XM(x,y) ((freal)(((long)(x)*(long)(y))>>8))
#define XD(x,y) ((freal)(((long)(x))<<8)/(long)(y)))

x = a*(b/c)       x = XM(a,XD(b,c))
```

Future Work

- Handle free boundaries (water)
- Parallel implementation (in progress)
- Adaptive grids (in progress)
- “Smarter” texture maps

Stable Fluids

Jos Stam
Alias|wavefront
Seattle, USA

In this talk I will present some research I have been doing over the last four years on modeling fluids at Alias|wavefront.

Note on the notes. I typed these notes in one session as if I was giving the talk. Usually the talk takes about one hour. I haven't corrected the grammar at all so it is very informal, but hey this is a talk not a paper.

I have given this talk at the following places: University of Washington (May 2000), SIGGRAPH course in New Orleans (July 2000), INRIA Rocquencourt, (January 2001), Gamer Seminar at SFO (January 2001), IMA Workshop on Graphics in Minneapolis (May 2001), invited talk at the Spanish Conference on Computer Graphics in Girona (July 2001), SIGGRAPH course in LA (August 2001), invited talk at the Eurographics Workshop on Simulation and Animation in Manchester (September 2001), INRIA Grenoble (September 2001).

Fluids in Computer Graphics

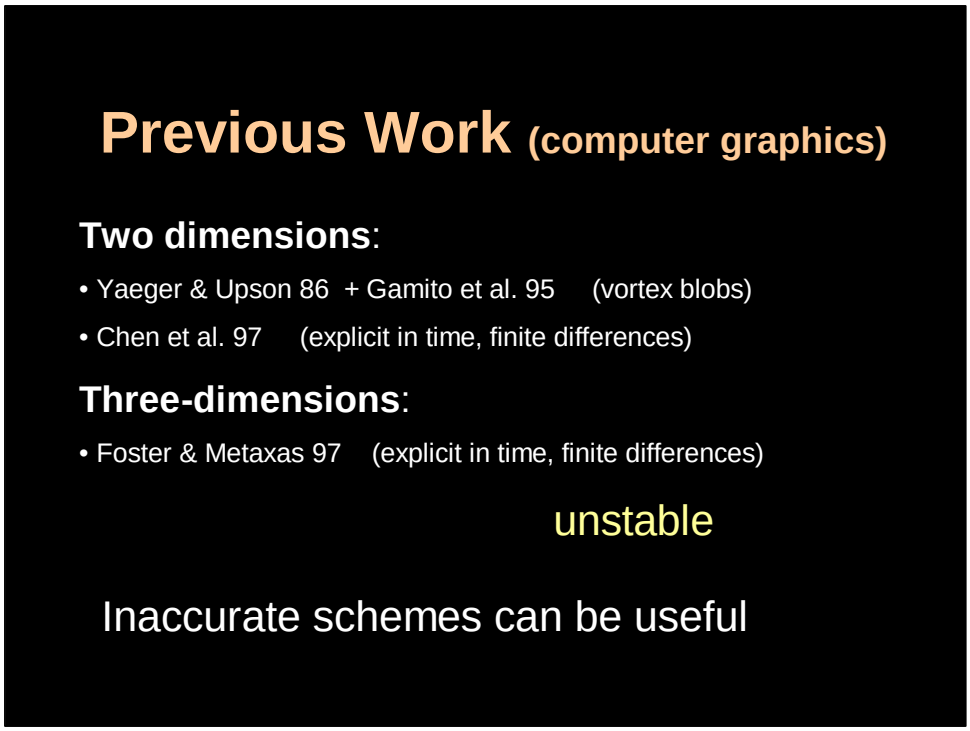
- Fast.
- Looks good.
- Easy to code.

The main goal in computer graphics is to have fluids that are both fast and look convincing. Ideally we want a user to be able to interact in real-time with a virtual fluid. In this manner effects can be orchestrated more rapidly. Real time performance is also important in games, for example. In addition I like solvers that aren't too hard to code. Later in this talk I will show you that a version of my solver can be coded in roughly 60 lines of C code.

Fluid Mechanics

- Natural framework for fluid modeling
  Full Navier-Stokes Equations
- Has a long history
  reuse code/algorithms
- Equations are hard to solve
  non-linear

To achieve these goals we can do whatever we want. Historically fluids in computer graphics have been modeled using a combination of simple primitives and the clever usage of texture maps. This is fine, but the dynamics of fluids are very hard to capture that way, and I am speaking from experience. A more natural way to model fluids is to use the physical equations that describe their motion. These equations are known as the Navier-Stokes equations and have been around for quite some time now. So potentially we can reuse the abundant literature in physics and engineering. Of course the reason that are so many articles published in this area is that these equations are very hard to solve because they are non-linear.



It is therefore not surprising that so little work has focused on solving these equations directly in computer graphics. Why should we be able to solve these equations if the experts at NASA can't ? But there has been some work. The early techniques were mostly restricted to 2D and used techniques such as vortex blobs that work mainly in 2D. The most important previous work, at least to me, was the paper by Foster and Metaxas. They clearly showed that very convincing flows could be obtained even on coarse grids. Their technique is also quite easy to implement. Unfortunately it is not fast. The problem is that they use explicit scheme which become unstable for large time steps or large velocities.

Main Contribution

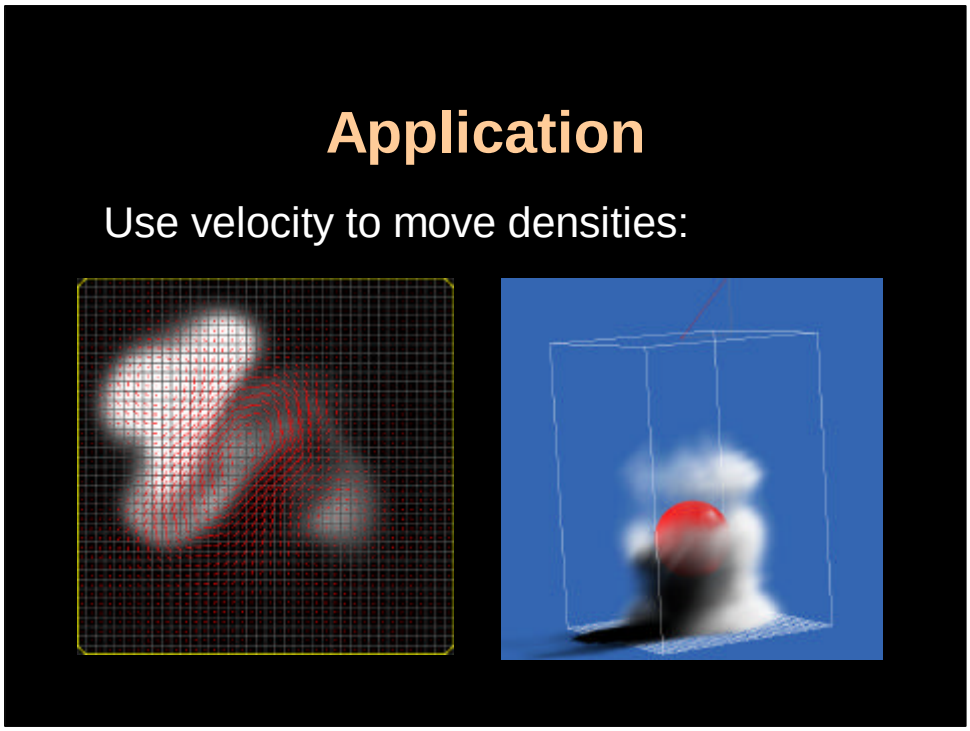
Stable Navier-Stokes solver

Any time step can be used

Bigger time steps = faster simulations

NOT accurate

So the motivation for this work was to extend Foster and Metaxas' work and make the solver stable for any time step. Large time steps means faster simulation. In fact you can make them as fast as you like. They might look strange but the simulation will not blow up. Of course the technique cannot be claimed to be accurate. But in graphics if it looks good it is good. And I will let you be the judge whether my simulations look good or not when I give the demos in a moment.



There are of course many applications for a fluid solver. In this talk I will use the fluid solver to move densities like smoke around in an environment. Here I show two snapshots from my interactive solver. The one on the left is from the 2D solver. The velocity field is shown in red and the density naturally follows the field. On the right is an example in 3D of a sphere interacting with the smoke density.

Application

Use velocity to move densities:

While (simulating)
- Get force from UI
- Get density source from UI
- Update velocity
- Update density
- Display density

The main structure of my solver is as follows. It is basically a single while loop. First I get some forces from the User Interface. In 2D these can be related to the movement of the mouse for example. Then I read in sources of densities from the UI. Then I update the velocity and the density using the solver and finally I display the density.

Equations

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

+ velocity should conserve mass

Equations very similar

To achieve this we need the physical equations for both the evolution of the density and the velocity. Here they are. What is immediately apparent is that these two equations look a lot the same. This should be obvious even to someone who has never seen these equations before. The top equation describes the evolution of the density denoted by rho. The velocity is denoted by the boldface vector u. The evolution of the velocity is given by the second equation which are the Navier-Stokes equations. The first equation is linear in rho and is much easier to solve than the second equation which is non-linear. The non-linear term is the second term on the right hand side where u appears twice. This term makes these equations hard to solve. Historically I first worked on a solver for the simpler equation and then applied to the harder equation. So in this talk I will first explain to you how to solve the first equation and then I will show you that the exact same techniques can be used to solve the second equation.

Equations

Evolution of density (assume velocity known)

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Over a time step...

The first equation tells us how the density evolves over time, that's what the symbol on the left means. And the change is due to three causes which correspond to the three terms on the right hand side of the equation.

Equations

Evolution of density (assume velocity known)

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Density changes in the direction of the flow

The first time says that the density should follow the velocity field. This makes sense since if I blow on smoke smoke it will naturally follow the direction of the wind field.

Equations

Evolution of density (assume velocity known)

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Density diffuses over time

The second term says that the density may diffuse at a rate kappa.

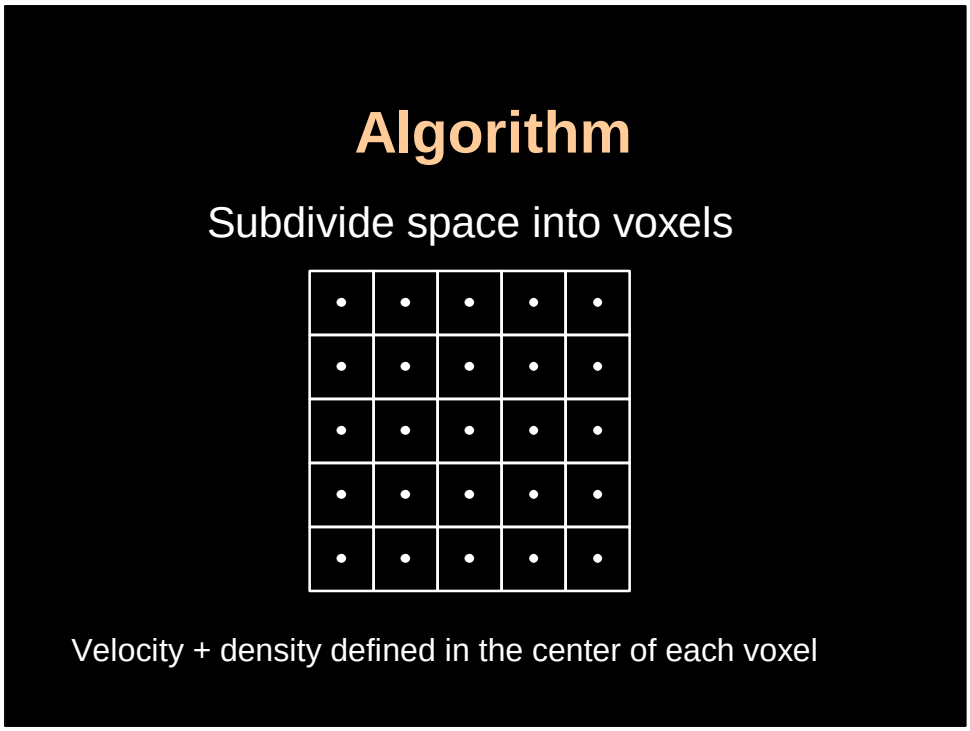
Equations

Evolution of density (assume velocity known)

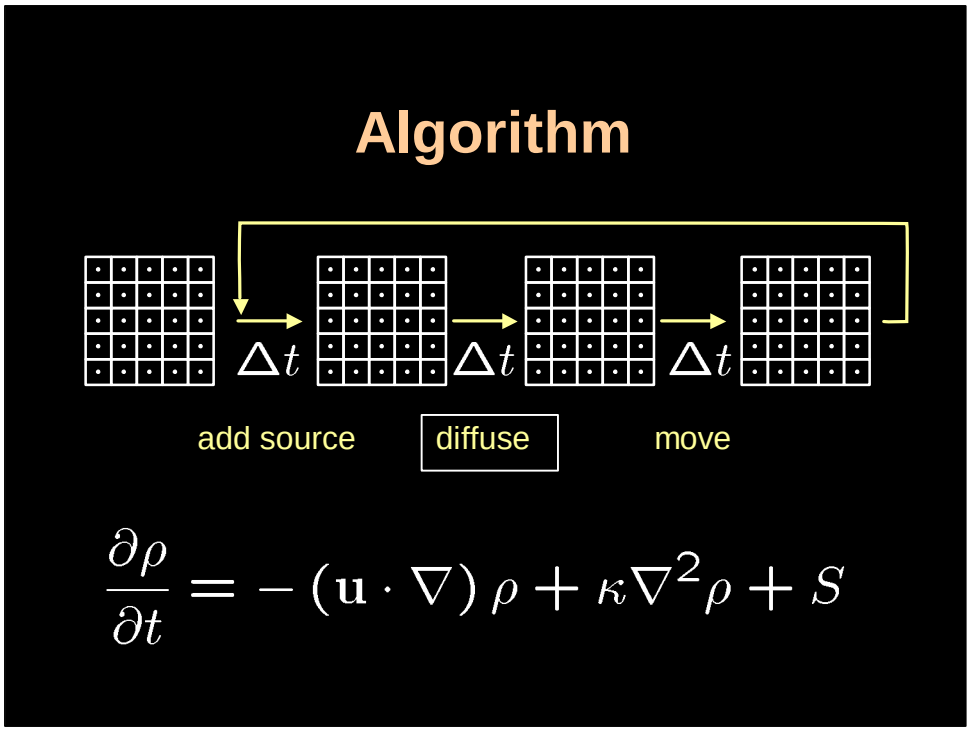
$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Increases due to sources from the UI

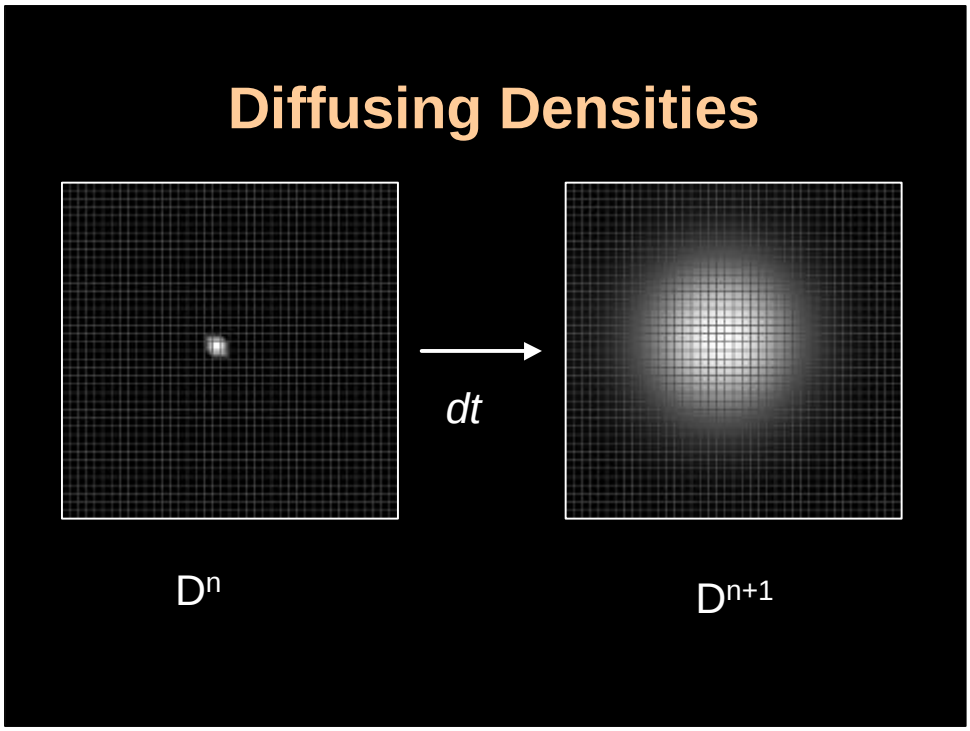
And finally the last term says that the density should increase due to external source. As I said before these sources are provided by the user through a suitable user interface.



Ok so how do we solve these equations. Well first we discretize the entire space into identical voxels with the density defined at the center of each voxel. In this talk I will present the 2D case just so everything is easier to visualize. However, nothing I say will be restricted to 2D. Everything I say can be easily extended to 3D by adding another index and or another for loop.

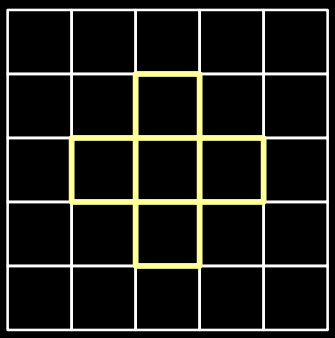


The main structure of the solver follows that of the equation. We first start with an initial grid of densities which is typically empty. Then we update this grid in three steps. Each step correspond to one of the terms of the equation. First we add the sources to the grid. This is really easy. In my implementation the sources are provided by a grid. So all I have to do is multiply this grid by the time step and add them to the density grid. Let me know explain how to solve for the diffusion step.



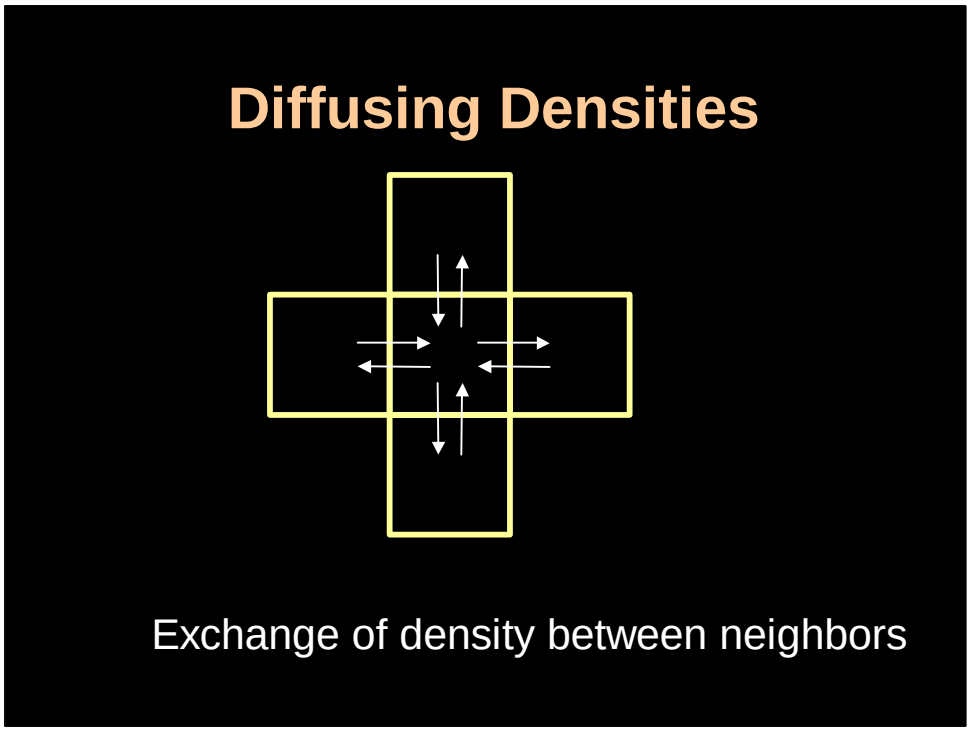
In this step we want to account for the effects of diffusion. This is shown in this slide. On the left is the density grid before diffusion and on the right after diffusion.

Diffusing Densities

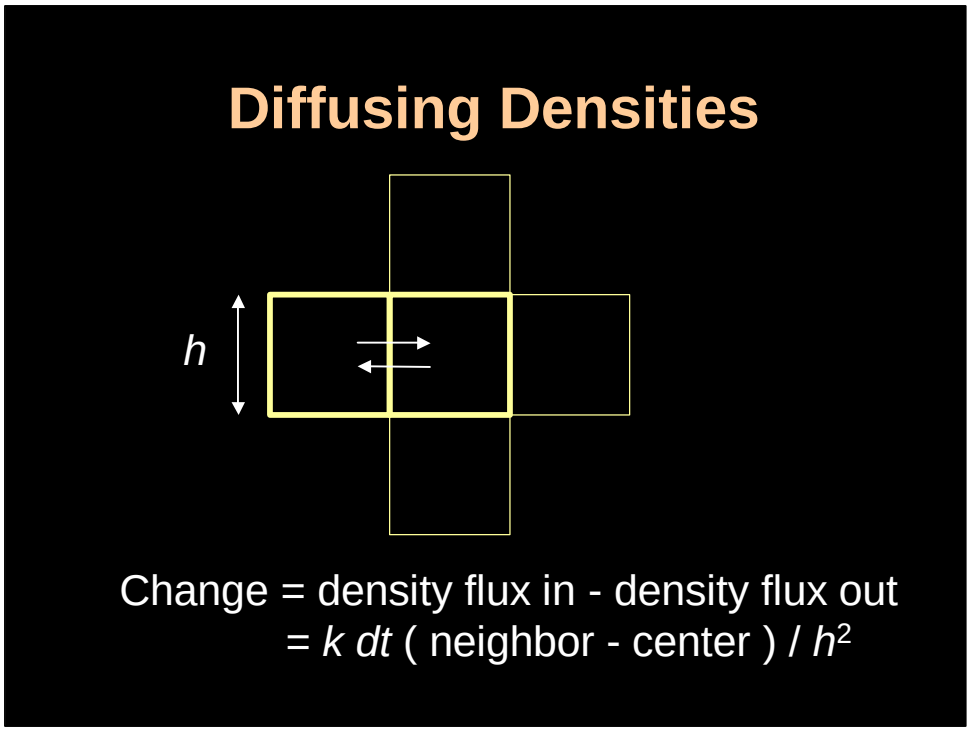


Exchange of density between neighbors

The basic idea is to look at the exchange of density between immediate neighbors only. They are highlighted here in yellow for the cell in the center.

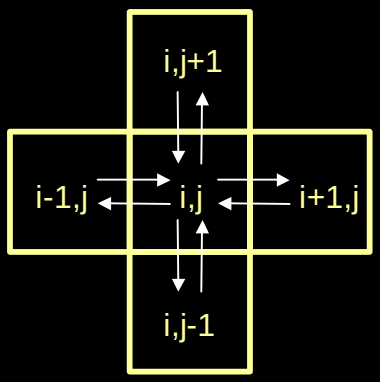


We assume that density is exchange out and into the cell through the adjacent faces (edges in 2D).



For a single face (edge) the exchange is equal to the density of flux coming in minus the density flux coming out. This is simply the difference in densities multiplied by the time step, the diffusion rate kappa and divided by the grid spacing squared. So the flux is higher for large time steps, large diffusion rate or small grid spacings.

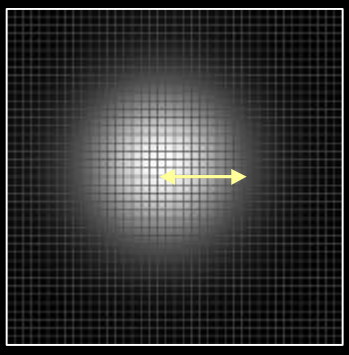
Diffusing Densities



$$D^{n+1}_{i,j} = D^{n}_{i,j} + k\,dt\,(D^{n}_{i-1,j} + D^{n}_{i+1,j} + D^{n}_{i,j-1} + D^{n}_{i,j+1} - 4D^{n}_{i,j})/h^2$$

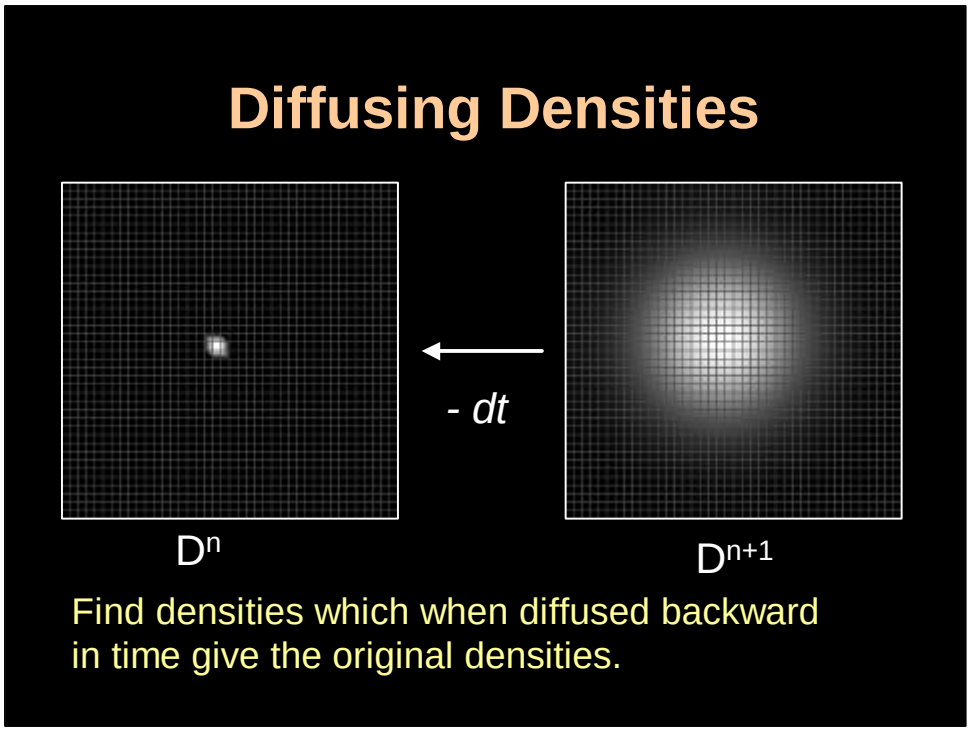
By summing up the fluxes for all the faces we end up with the following update rule. This is really easy to code simply add two for loops around this equation and you are done. Unfortunately it doesn't work.

Diffusing Densities



Unstable when $k\,dt / h^2 > 1/2$

The reason it doesn't work is that the it can become unstable when the diffusion rate is too high, time step too large or grid spacing too small. The problem is that this method breaks down when the density propagates further than just between neighbors.



So we have to consider an alternative. The alternative is to use implicit techniques. Intuitively we look for the densities which when diffuse backward in time give us the densities that we currently have. Implicit techniques always work this way by looking back in time.

Diffusing Densities

Linear system:

$$D^{n+1}_{i,j} - k\,dt\,(D^{n+1}_{i-1,j}+D^{n+1}_{i+1,j}+D^{n+1}_{i,j-1}+D^{n+1}_{i,j+1}-4D^{n+1}_{i,j})/h^2 = D^{n}_{i,j}$$

$$A\,x = b$$

A can be huge *but* is sparse

-> requires fast linear solver

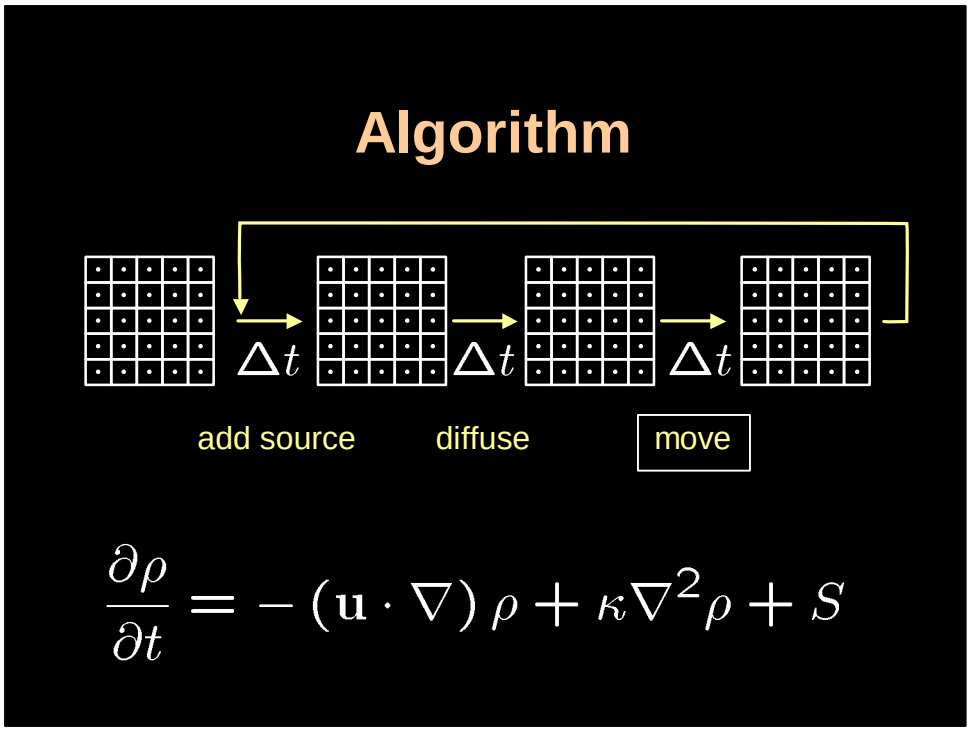
So now we can rewrite our update rule with the roles of Dn and Dn plus one exchanged and the time step reversed. The problem here is that all the terms on the left hand side are unknown. So we end up with a linear system that we have to solve. This seems like a crazy idea, the unstable solution is so simple and now we have to solve an entire system. Actually this isn't all that bad. First the system is sparse and many fast solvers exist. Also in practice the cost of solving the system over a large time step is much more effective than taking many tiny steps with a cheap unstable solver.

Diffusing Densities

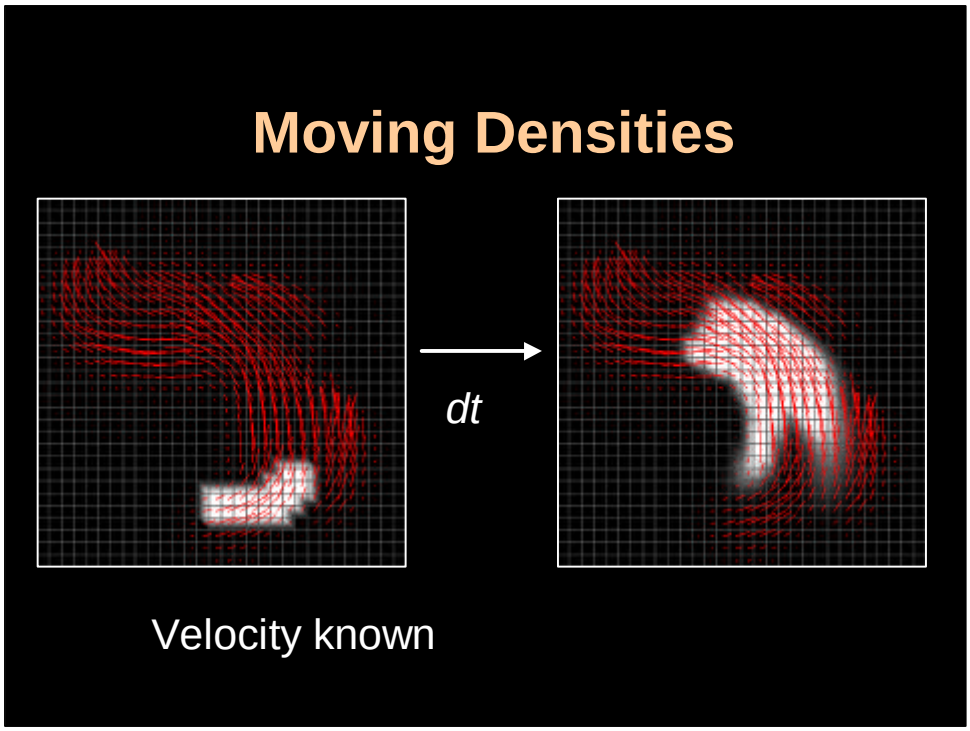
Linear solvers:

Name	Cost	Comments
Gaussian elimination	N^3	Use only for very small N (test code)
Jacobi/SOR relaxation	N^2	Easy to code but slow
FFT/cyclical reduction	N logN	Use when no internal boundaries
Conjugate gradient	$N^{1.5}$	Use when internal boundaries
Multi-grid	N	Slower than FFT in practice. Hard to code when internal boundaries present

So what kind of linear solvers are out there. The most naïve solver is to use Gaussian elimination. These solvers are only good for small or dense matrices. But they are available in many standard libraries and are useful to debug the solver code for small grids. The easiest solver to implement is simple relaxation such as Jacobi or Gauss-Seidel. The only problem is that they do not converge all that fast. I would start with this solver and later when the code is up and running switch to a more sophisticated technique such as the FFT based ones or the conjugate gradient. Both of these techniques converge much faster but are a little harder to implement. The FFT based techniques are really fast but only work when there are no internal boundaries, like an object in the fluid. These solvers are available in FISHPAK from netlib.org. It's in Fortran but you can convert it to c using the f2c tool also available at netlib.org. When objects are present in the flow I recommend using the conjugate gradient technique. It is fairly easy to code. In fact you can find some nice C++ templates (which I converted to C code) from the IML++ library at NIST. Finally, multigrid solvers are theoretically optimal. Although they are order N, the constant is quite high. And unless you consider huge grids both FISHPAK and the conjugate gradient will work much faster. Also multigrid methods are a total pain to code. Especially when objects are present or you consider grid sizes which are not powers of two.



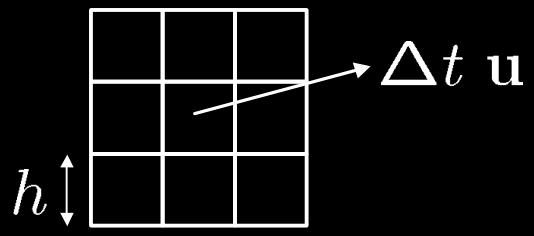
So this takes care of the diffusion step. Let's now turn to the last step in updating the density.



In this step we assume that the velocity is known. Over a time step we want to move the density along the velocity field.

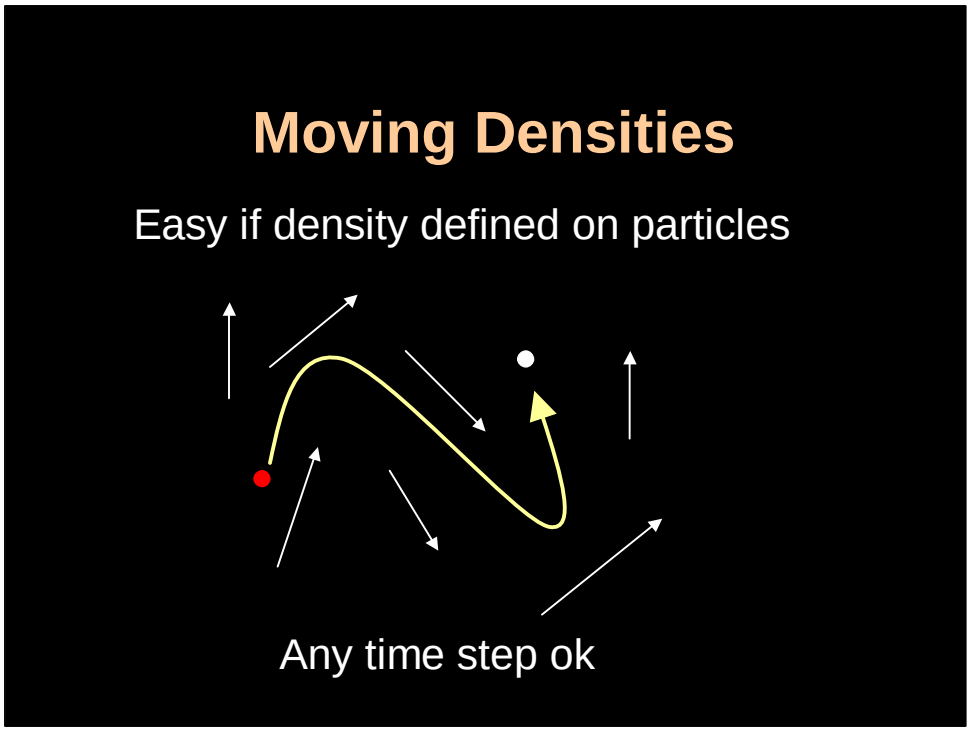
Moving Densities

Finite Differences: transfer only between neighbors

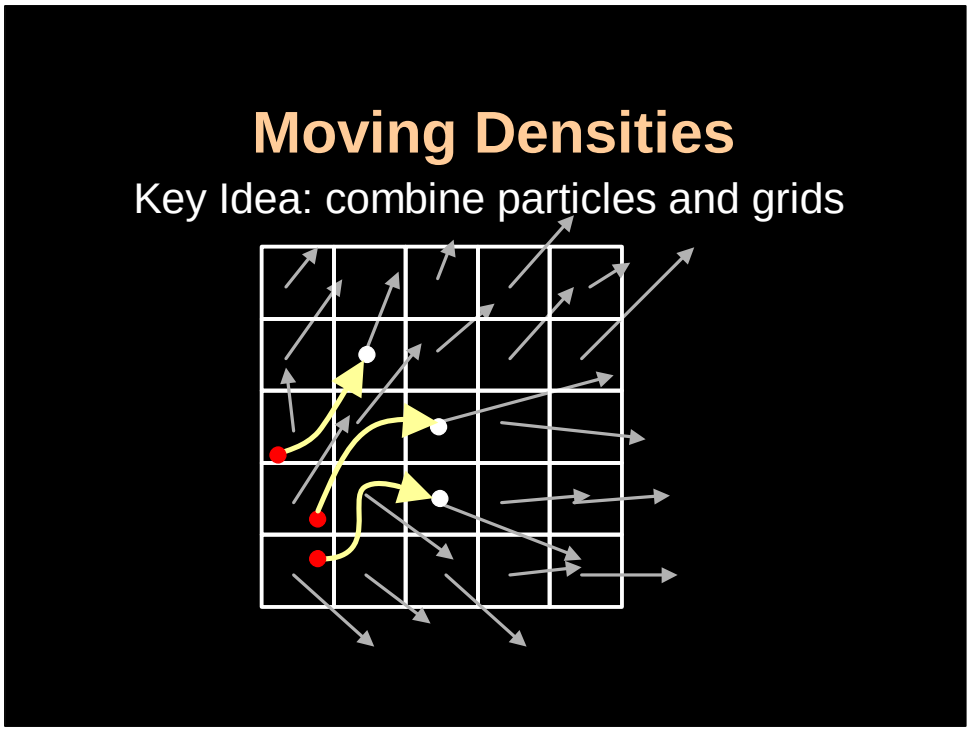


Unstable when $\Delta t\ |\mathbf{u}| > h$

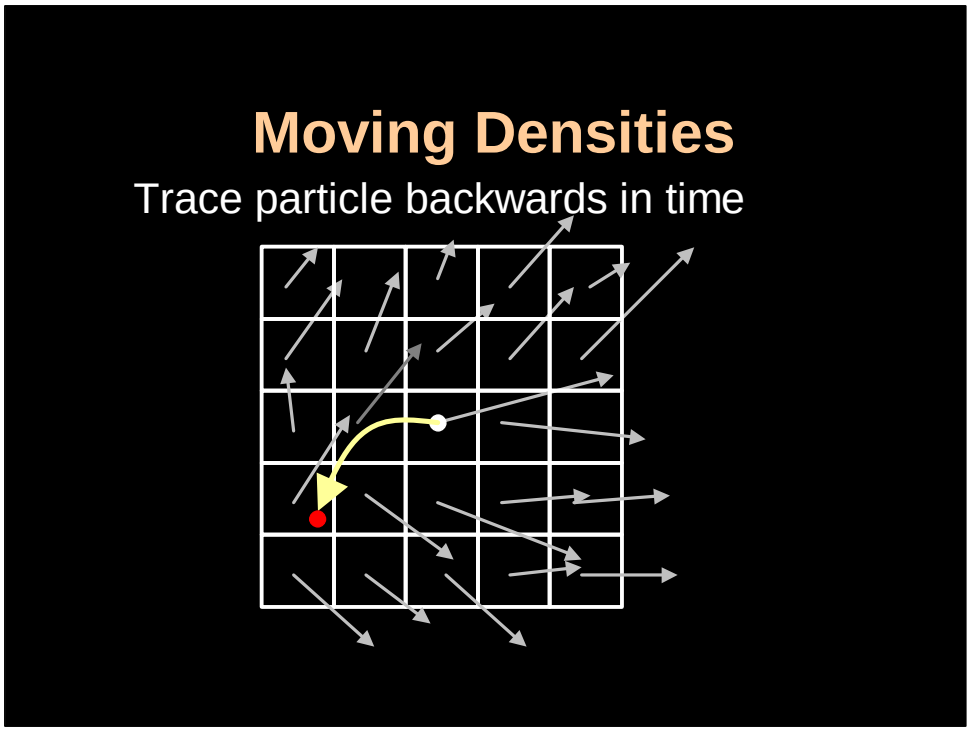
As for the diffusion step we can look at direct neighbors only and consider the fluxes. In this case the fluxes will be biased by the direction of the velocity field. The naïve implementation of this scheme again results in instabilities. The problem again is when the transfer is between neighbors that are more than one cell removed. This happened when either the velocity is too large, the time step too big or the grid spacing too small. We could use a stable implicit method, but the problem here is that the resulting linear system has a non-symmetrical matrix with varying constants. So the Conjugate Gradient solver cannot be used. Of course there are variant of the conjugate gradient such BiCGSTAB2 but they can become unstable when the system is ill-conditioned. However, it turns out there is a really simple technique that is stable and can solve for this step.



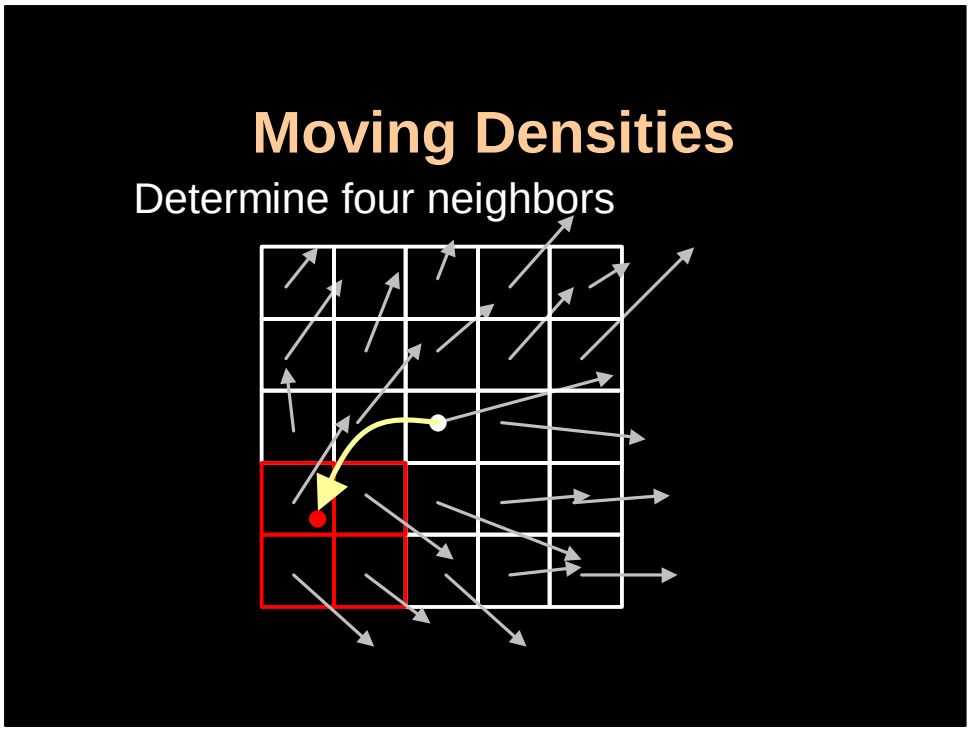
The basic idea is as follows. If the density was sampled on particles then this step would be trivial to compute: simply move the particles along the field using a particle tracer. Of course the problem is that we then have to use a particle solver for the other terms as well. And no good techniques are known to me for doing this. So we still want to keep the grids around.



Our idea is to find the positions of the particles that after one time step end up exactly at the grid centers.



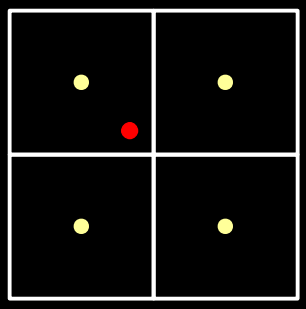
To find these particles we simply trace back each voxel center of the grid backwards through the field.



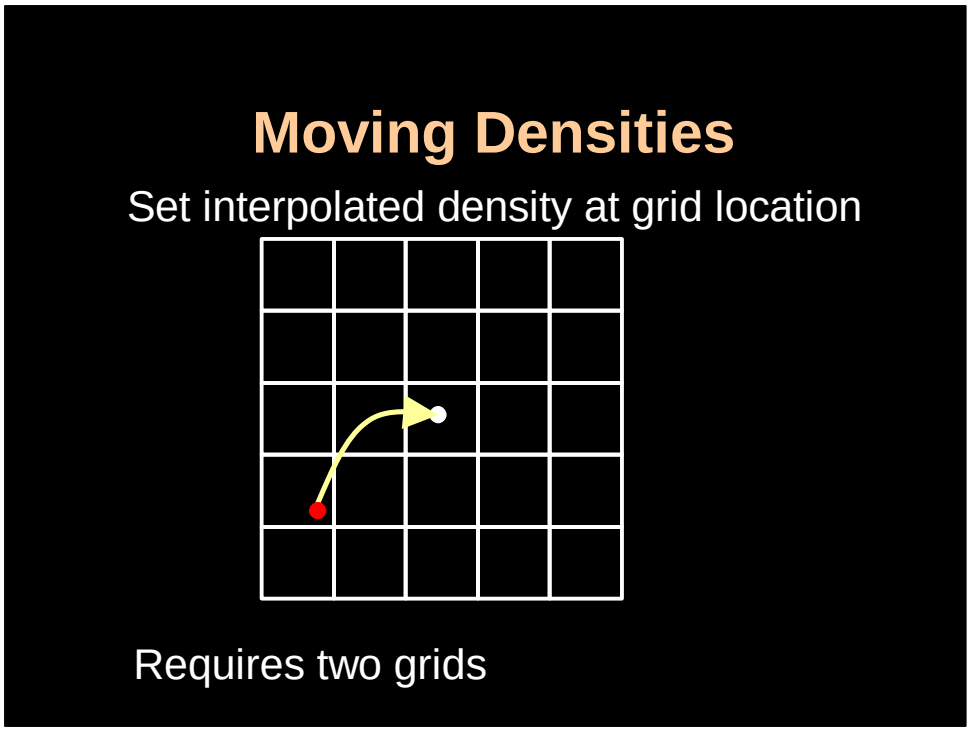
Doing this we will end up somewhere else in the grid. We first locate the four closest cells to the point.

Moving Densities

Interpolate the density at new location



And then we interpolate the density from these cells…



… and set the interpolated value as the new density of the departure cell. For this we require two grids. One that contains the density values of the previous time step and one that will contain the new interpolated values.

Moving Densities

This scheme is unconditionally stable:

$$\rho_{int} = (1 - s)\rho_0 + s\rho_1$$

$$\rho_0, \rho_1 \leq \rho_{max}$$

$$\rho_{int} \leq (1 - s + s)\rho_{max} \leq \rho_{max}$$

$\longrightarrow$ density is always bounded

The important property of this technique is that it is unconditionally stable: no matter how big the time step this technique will not blow up. This is why. Since we have a regular grid which is tensor product of one dimensional grids I only have to prove it for 1D data. Since the new data is a linear interpolation of previous data we have that the new maximum of the new densities is always bounded by the maximum density of the old values. So the density is always bounded no matter how big the time step and therefore will never blow up and become unstable.

Computing Velocities

Now let’s go back to the computation of the velocity.

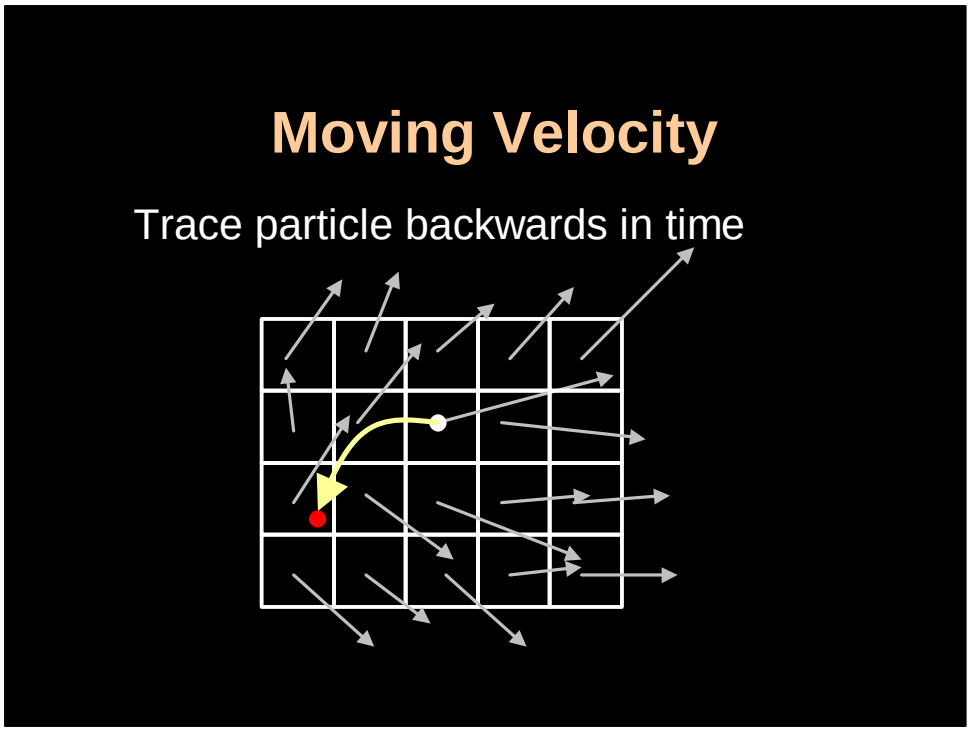
Computing Velocities

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

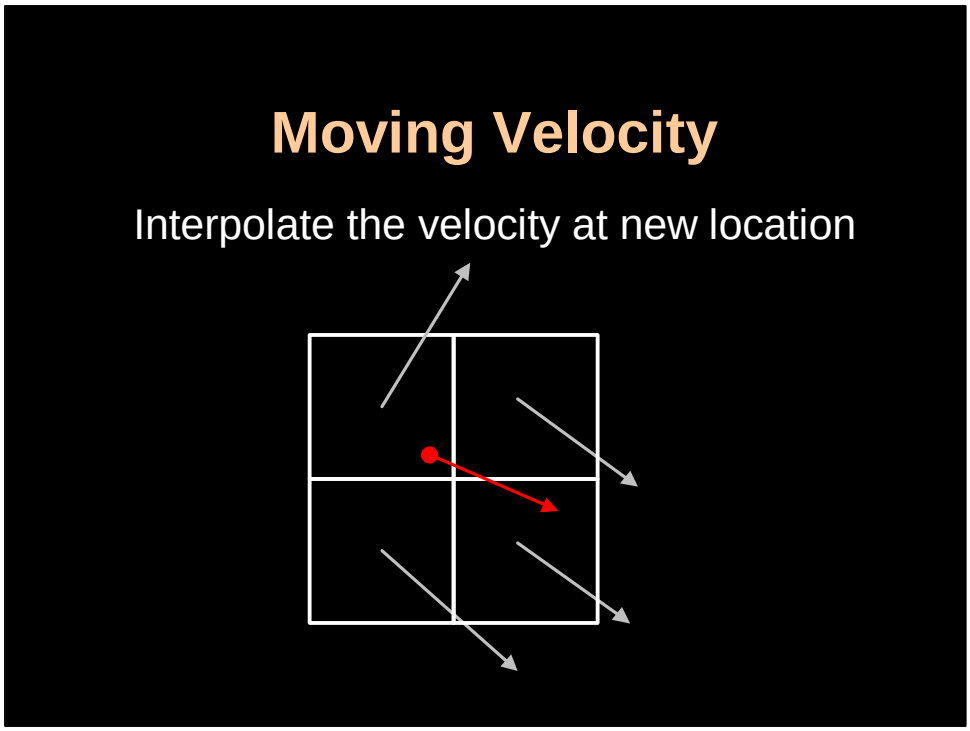
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

Velocity is moved by itself

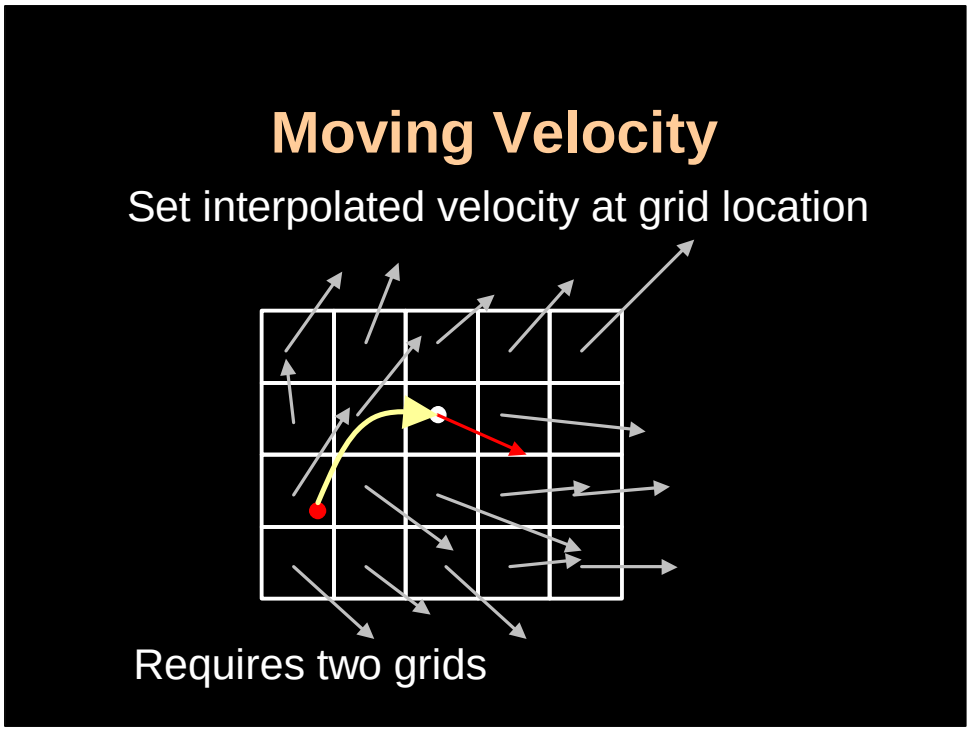
As I said earlier the equation I just showed you hot to solve for the density looks a lot like the equation for the velocity. The velocity also has a "diffusion" term which accounts for the effects of viscosity. The higher the viscosity constant nu, the more the velocity field will be smooth, resulting in viscous-like fluids. In this case we have to solve 2 diffusion equation in 2D and three equation in 3D, one for each component of the velocity field. The last term is the force term and can also be accounted for as for the sources in the density solver. Finally the first term is the most interesting one. It looks just like the corresponding term for the density, except that the velocity appears twice making it non-linear. In the case of the density this term states that the density should follow the velocity field. So we can interpret this term as saying that the "velocity should move along itself". This might sound weird at first but this is how we can interpret. And in fact we can blindly apply the same algorithm we used for the density. At first I never expected this to work and I was lucky that my first implementation didn't have bugs, if not I would have thought, Ah another crazy idea that doesn't work. Instead I was amazed that it worked so well.



Se here we go. Again we need two grids for the velocity. One that contains the old values and one that contains the new interpolated values. As before we trace each grid point backwards in term using the old velocities.



Doing this we end up somewhere else in the grid. And as for the density we interpolate a new velocity at that location from the neighboring grid cells.

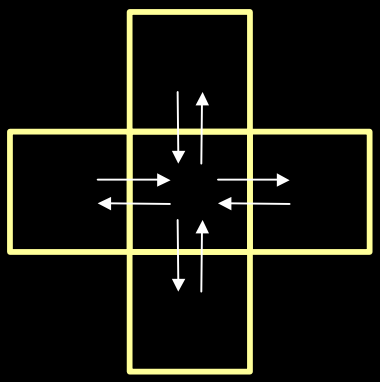


And then we set the new velocity to the interpolated one. Again this method is stable just like for the density solver. I learned from Ron Fedkiw that this technique was first invented in 1952 by Courant, Rees and Isaacson and has been rediscovered by many research in different fields. It is best known as a semi-Lagrangian technique. Oh well, at least I popularized the idea in computer graphics.

Conservation of Mass

There is still one step we have to enforce before we are done and that is that the fluid should conserve mass.

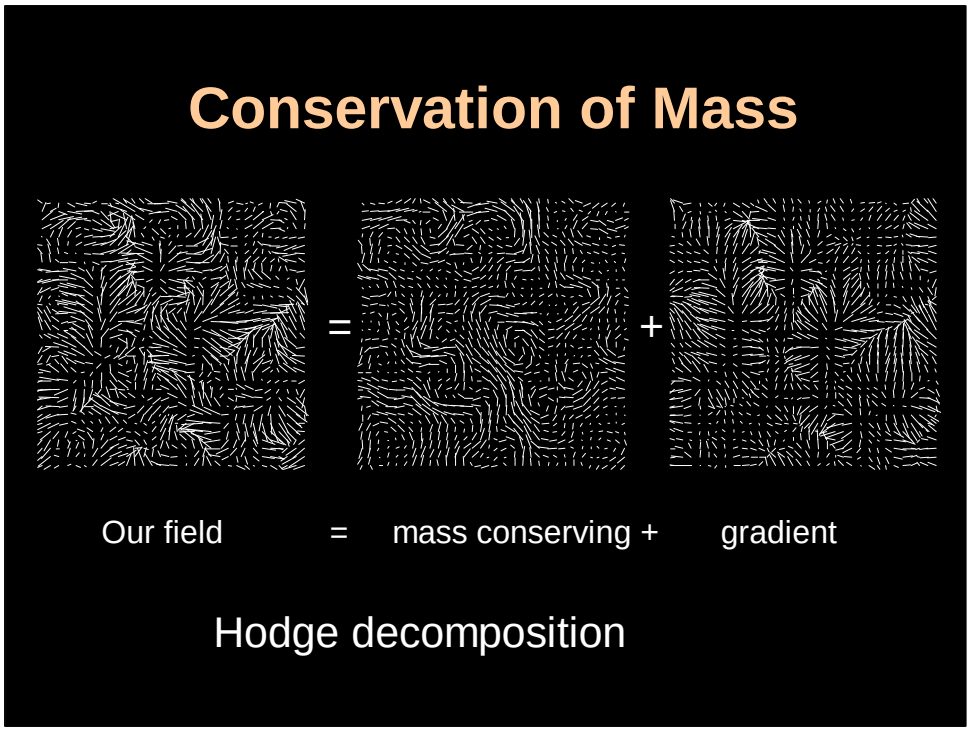
Conservation of Mass



Flow into cell = Flow out of the cell

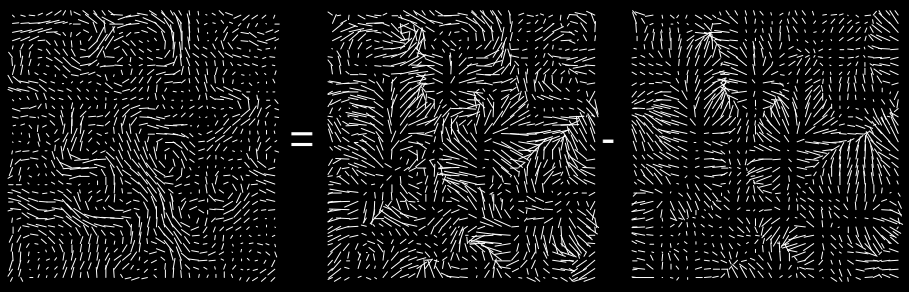
$U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1} = 0$ not

What this means is that we want to flow into a cell to be equal to flow out of it. This results in the constraint shown on the slide. In practice after the three previous steps this is never the case. So the idea is to correct the situation in a final step.



To to this we use a mathematical result know as the Hodge decomposition of a vector field. This result states that every vector field such as the one shown on the left is the sum of a mass conserving field and a gradient field. The mass conserving field is exactly the sort of vector field we want as it has nice vortices which will result in swirling looking flows. The gradient field on the hand is the worst possible case: at every point the flow either is all inward or outward. The gradient field can be visualized as being the slope function of some height field, it is defined entirely be a single scalar field.

Conservation of Mass



Mass conserving = our field - gradient

To get a mass conserving field from an arbitrary vector field we simply subtract the gradient part from it. This requires us to find the scalar function that defines the gradient field.

Conservation of Mass

Scalar field satisfies a Poisson Equation:

$$P_{i+1,j}+P_{i-1,j}+P_{i,j+1}+P_{i,j-1} - 4\,P_{i,j} = (U_{i+1,j} - U_{i-1,j} + V_{i,j+1} - V_{i,j-1})\,h$$

Linear system

It turns out that this gradient field can be computed by solving the following Poisson equation. Again we have a sparse symmetrical linear system that we can solve using any of the solvers that I mentioned when talking about the diffusion step. Typically conjugate gradient is a good idea with a good preconditioner, Jacobi did the job for me.

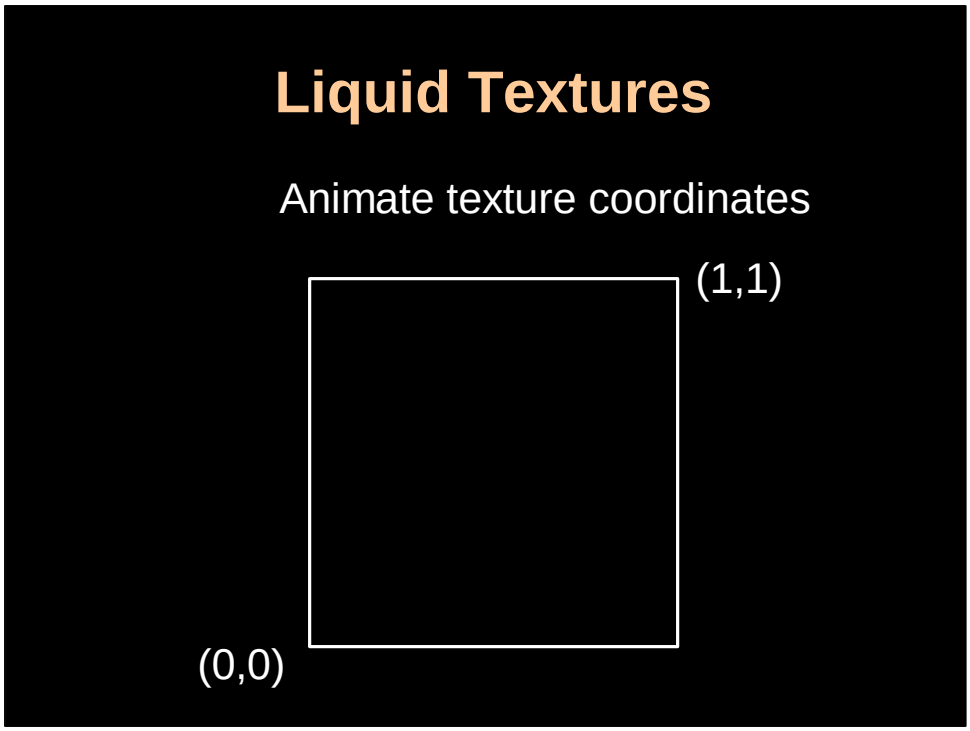
Summary

```
UpdateVelocity(U1,U0,F,visc,dt)
   AddForce(U1,U0,F,dt)
   Diffuse(U0,U1,visc,dt)
   Move(U1,U0,U0,dt)
   ConserveMass(U1,dt)
```

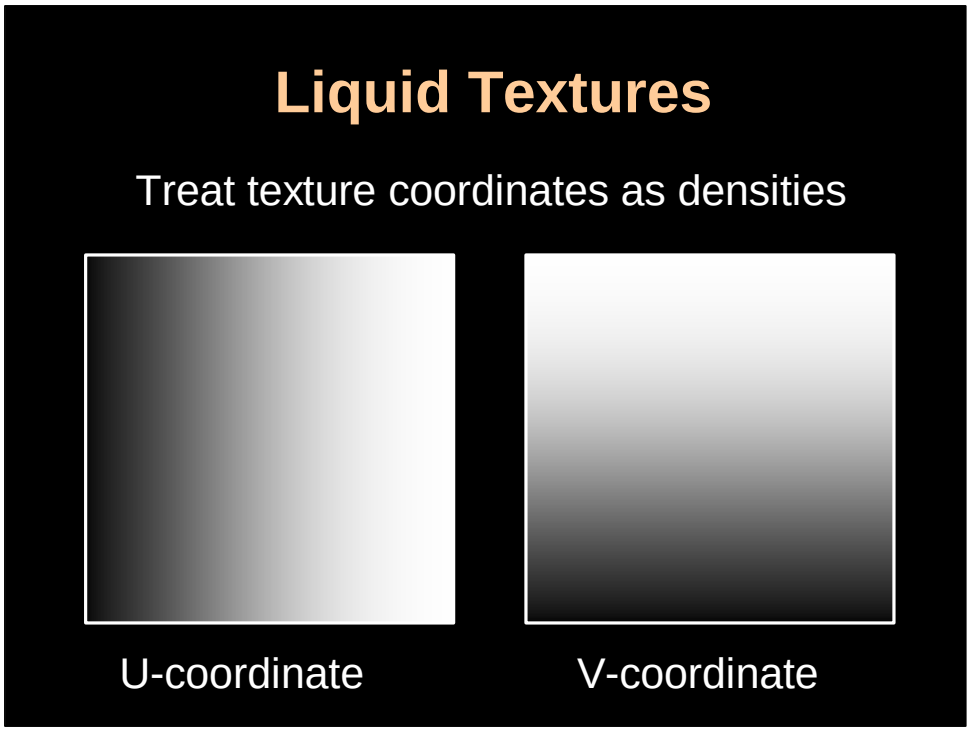
Very easy to code. Only need:
- Particle tracer + grid interpolator
- Linear solver (FISHPAK or CG)

So in summary here is all you need to write a fluid solver: a linear solver for the diffusion and the mass conservation step and a good particle tracer and grid interpolator for the self-advection step.

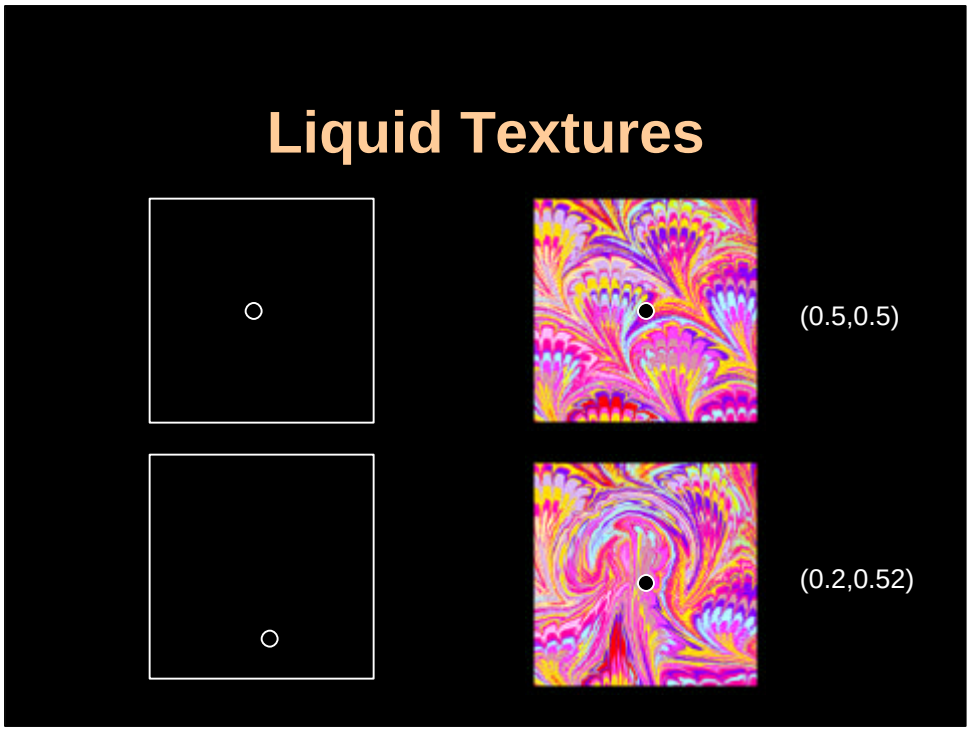
At this point I show a 2D demo that demonstrates each step of the algorithms just described.



A nice way to add detail to the flow is through texture maps. This is a popular way to add visual detail in computer graphics. We assume that our fluid density has intially the texture coordinates shown on the slide. (0,0) in the lower left corner and (1,1) in the upper right corner.

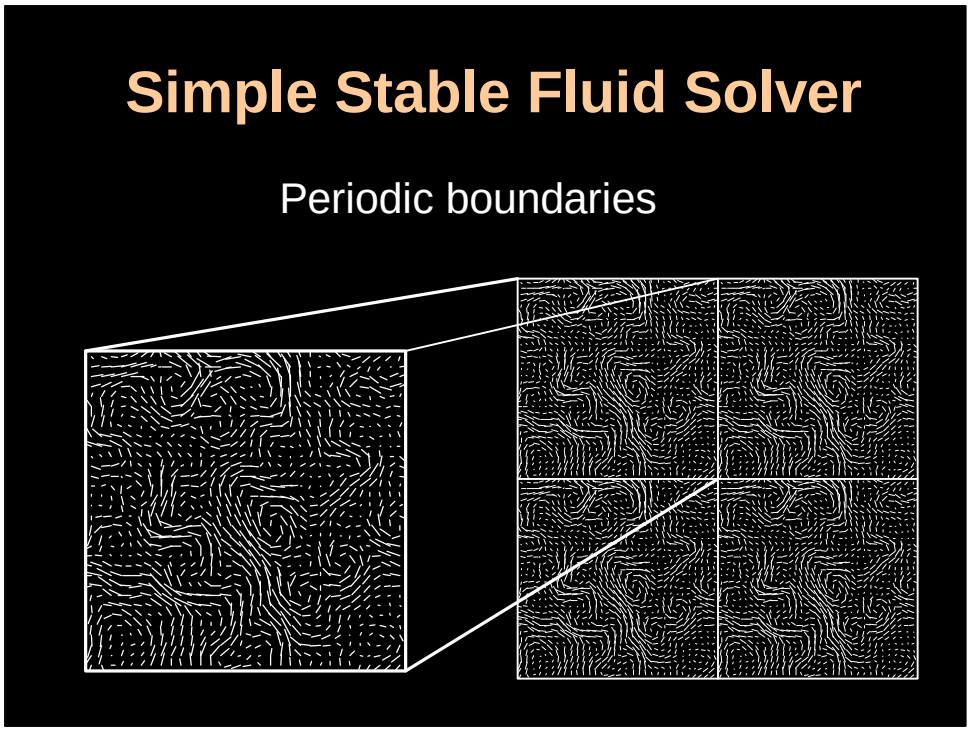


Now we can interpret the texture coordinates as density which initially are just equal to the ramps shown here. Then the idea is to feed these densities to the solver and let them evolve according to the velocity field. This will give the impression that the texture moves with the flow. So in essence we are simulating three densities: the density, the u coordinate and the v coordinate.

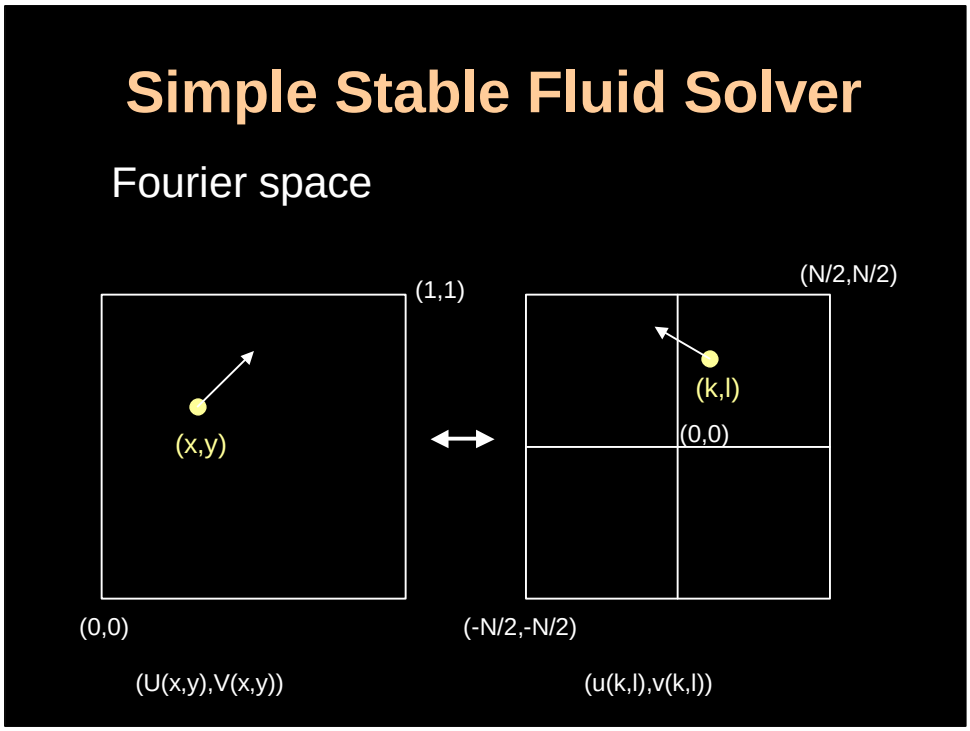


Here is an example. The point in the center of the texture initially corresponds to the center point in the fluid domain. However, when these texture coordinates are animated using the solver the coordinate in the center now point to a lower point in the texture map (when applying a force field upward). This will give the impression that the texture flows upward. Very simple trick that is pretty effective.

At this point I show some real time demos where I move these textures around. I also show a “wild paint” program where drops of different color fall down due to the effects of gravity.



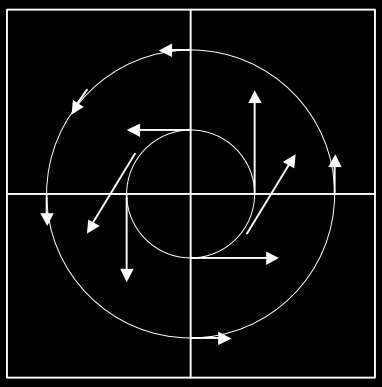
At the beginning of my talk I said that I could implement my solver in roughly 60 lines of code. You can do this in the case when the fluid is periodic. Which means that the flow is continuous across adjacent boundaries. So for example you can tile a single domain to the entire space. Like some funky wallpaper with arrows on them that also vary over time. These flows of course do not occur exactly in Nature, but they can be useful in computer graphics. For example it allows you to have field defined everywhere in space. So you can model some ambient turbulence that way. In fact if you throw some particles in the field, you will barely notice that the field is periodic. In fact I did something quite similar in my 1993 SIGGRAPH paper with Eugene Fiume.



Because the field is periodic we can use the Fourier transform. The Fourier transform of a vector field is also a vector field. Instead of assigning a vector to each point in space the Fourier transform assigns a vector for each wave number (k,l). The wave numbers have values between –N/2 and N/2, where N is the resolution of the grid. The reason that this is neat is that some operations are very simple to do in the Fourier space, while others are more easily done in the spatial domain.

Simple Stable Fluid Solver

diffusion = low pass filter: $\exp(-(k^2+l^2)??t)$
mass conservation = velocity perpendicular to the fourier directions



More specifically both the diffusion step (viscosity remember) and the mass conserving step are very easy to compute in Fourier domain. The effect of viscosity is simply a low pass filter which dampens the higher frequencies. So the arrows get tinier as you move away from the center of the Fourier space as shown on the slide. In the Fourier domain a mass conserving field has the nice property that the velocity is always perpendicular to the wave number. So all the vectors are perpendicular to the circles centered at the origin. This is shown on the slide. The mathematical reason that this is the case is that incompressibility equation which reads Nabla dot u, becomes wave number dot u in the Frequency domain. A dot product is zero if an d only if the vector are perpendicular. So what all this says is that the velocity field has a very simple structure in the Fourier domain. A fact that is not widely know I think. The self-advection term is best solved in the spatial domain using the simple semi-Lagrangian style technique, where you trace each point back and do the interpolation. So is the addition of external forces which are usually localized in space. Although to model turbulence it might be a better idea to do it in frequency domain using a Kolmogoroff spectrum.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                    s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                    s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

   for ( i=0 ; i<=n ; i+=2 )
   {
      x = 0.5*i;
      for ( j=0 ; j<n ; j++ )
      {
         y = j<=n/2 ? j : j-n;
         r = x*x+y*y;
         if ( r==0.0 ) continue;
         f = exp(-r*dt*visc);
         U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
         U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
         u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
         u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
         v0[i+  (n+2)*j] = f*(    -y*x/r *U[0] + (1-y*y/r)*V[0] );
         v0[i+1+(n+2)*j] = f*(    -y*x/r *U[1] + (1-y*y/r)*V[1] );
      }
   }

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

60 lines of (readable) C code

So here is the code. All it assumes is that you have a good FFT solver. The one I use is called the Fastest Fourier Transform in the West (FFTW) which you can get from MIT for free. I will know go over the different parts of the code. The input is simply the velocity of the previous time step (u,v), the forces (u0,v0), the viscosity visc and the time step dt.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                    s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                    s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

   for ( i=0 ; i<=n ; i+=2 )
   {
      x = 0.5*i;
      for ( j=0 ; j<n ; j++ )
      {
         y = j<=n/2 ? j : j-n;
         r = x*x+y*y;
         if ( r==0.0 ) continue;
         f = exp(-r*dt*visc);
         U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
         U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
         u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
         u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
         v0[i+  (n+2)*j] = f*(    -y*x/r *U[0] + (1-y*y/r)*V[0] );
         v0[i+1+(n+2)*j] = f*(    -y*x/r *U[1] + (1-y*y/r)*V[1] );
      }
   }

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

Add forces

First we simply add the force grid multiplied by the time step to the velocity field.

Simple Stable Fluid Solver

```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                        s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                        s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

   for ( i=0 ; i<=n ; i+=2 )
   {
      x = 0.5*i;
      for ( j=0 ; j<n ; j++ )
      {
         y = j<=n/2 ? j : j-n;
         r = x*x+y*y;
         if ( r==0.0 ) continue;
         f = exp(-r*dt*visc);
         U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
         U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
         u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
         u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
         v0[i+  (n+2)*j] = f*(    -y*x/r *U[0] + (1-y*y/r)*V[0] );
         v0[i+1+(n+2)*j] = f*(    -y*x/r *U[1] + (1-y*y/r)*V[1] );
      }
   }

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

Move velocity

Then we do the self-advection step. Again we trace each voxel center back in time. The interpolation is quite straightforward since we do not have to worry about boundaries. A particle that exits one boundary reenters the grid from the opposite boundary. This can easily be implemented using couple mod operations in C. See the code.

Simple Stable Fluid Solver

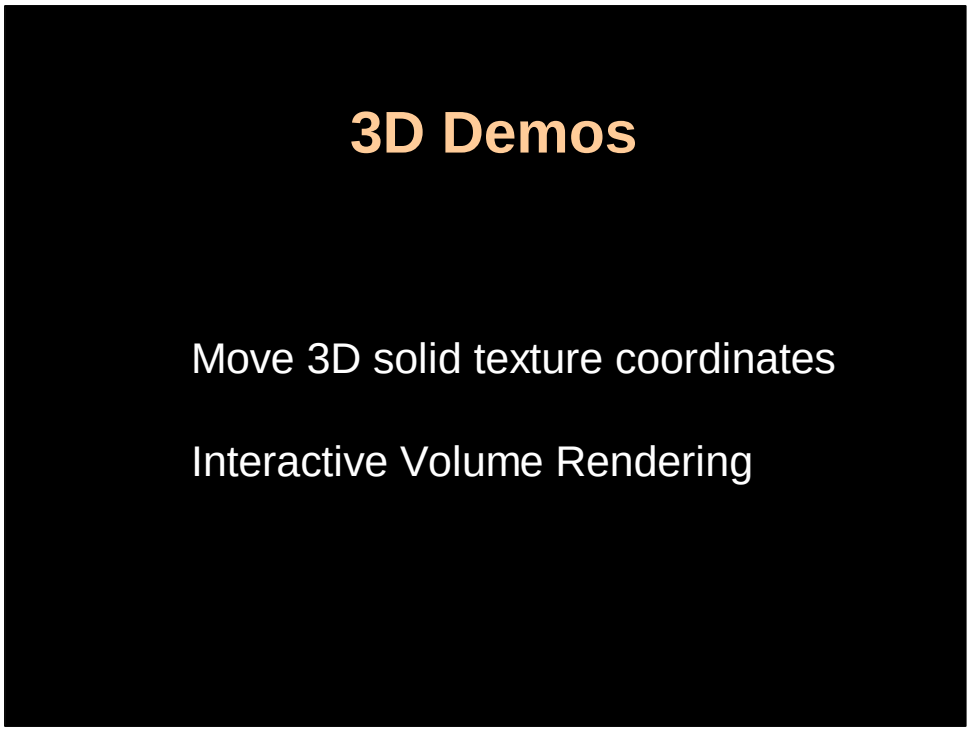
```
void stable_solve ( int n, float * u, float * v, float * u0, float * v0,
float visc, float dt )
{
   float x, y, x0, y0, f, r, U[2], V[2], s, t;
   int i, j, i0, j0, i1, j1;

   for ( i=0 ; i<n*n ; i++ )
   {
      u[i] += dt*u0[i]; u0[i] = u[i];
      v[i] += dt*v0[i]; v0[i] = v[i];
   }

   for ( x=0.5/n,i=0 ; i<n ; i++,x+=1.0/n )
   {
      for ( y=0.5/n,j=0 ; j<n ; j++,y+=1.0/n )
      {
         x0 = n*(x-dt*u0[i+n*j])-0.5; y0 = n*(y-dt*v0[i+n*j])-0.5;
         i0 = floor(x0); s = x0-i0; i0 = (n+(i0%n))%n; i1 = (i0+1)%n;
         j0 = floor(y0); t = y0-j0; j0 = (n+(j0%n))%n; j1 = (j0+1)%n;
         u[i+n*j] = (1-s)*((1-t)*u0[i0+n*j0]+t*u0[i0+n*j1])+
                        s *((1-t)*u0[i1+n*j0]+t*u0[i1+n*j1]);
         v[i+n*j] = (1-s)*((1-t)*v0[i0+n*j0]+t*v0[i0+n*j1])+
                        s *((1-t)*v0[i1+n*j0]+t*v0[i1+n*j1]);
      }
   }

   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u0[i+(n+2)*j] = u[i+n*j]; v0[i+(n+2)*j] = v[i+n*j]; }
```

```
   FFT(1,n,u0); FFT(1,n,v0);

   for ( i=0 ; i<=n ; i+=2 )
   {
      x = 0.5*i;
      for ( j=0 ; j<n ; j++ )
      {
         y = j<=n/2 ? j : j-n;
         r = x*x+y*y;
         if ( r==0.0 ) continue;
         f = exp(-r*dt*visc);
         U[0] = u0[i  +(n+2)*j]; V[0] = v0[i  +(n+2)*j];
         U[1] = u0[i+1+(n+2)*j]; V[1] = v0[i+1+(n+2)*j];
         u0[i  +(n+2)*j] = f*( (1-x*x/r)*U[0]     -x*y/r *V[0] );
         u0[i+1+(n+2)*j] = f*( (1-x*x/r)*U[1]     -x*y/r *V[1] );
         v0[i+  (n+2)*j] = f*(   -y*x/r *U[0] + (1-y*y/r)*V[0] );
         v0[i+1+(n+2)*j] = f*(   -y*x/r *U[1] + (1-y*y/r)*V[1] );
      }
   }

   FFT(-1,n,u0); FFT(-1,n,v0);

   f = 1.0/(n*n);
   for ( i=0 ; i<n ; i++ )
      for ( j=0 ; j<n ; j++ )
         { u[i+n*j] = f*u0[i+(n+2)*j]; v[i+n*j] = f*v0[i+(n+2)*j ]; }

   return;
}
```

Diffuse + project

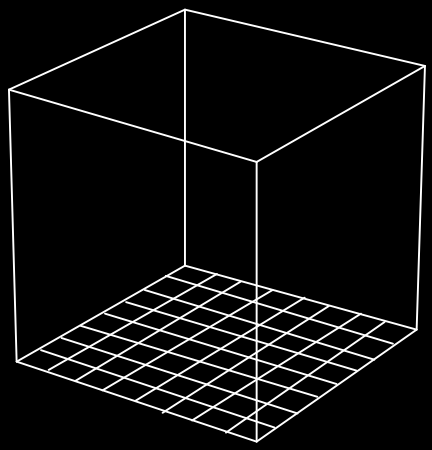
Next we transform our field to the Fourier domain where we do the diffusion and the mass conserving steps. The viscosity low pass filter depends on the wave number, the time step and the viscosity. The projection step is also very easy to implement. Notice that velocity values are complex numbers, however the storage is almost the same due to the symmetries of a Fourier transform of a real function. So in essence we only store the right half of the Fourier space. Another thing that is interesting is that the solver works in any dimension: simply add more arrays and for loops and the FFTW has version for transforms in any dimensions. It is not clear to me what the applications would be of 4 or 5 dimensional fluid solver. Maybe the evolution of some set of parameters is governed by a Navier-Stokes Equation. The vector field in this case models the flow of change of these parameters. So I have a solution here that is looking for a problem.

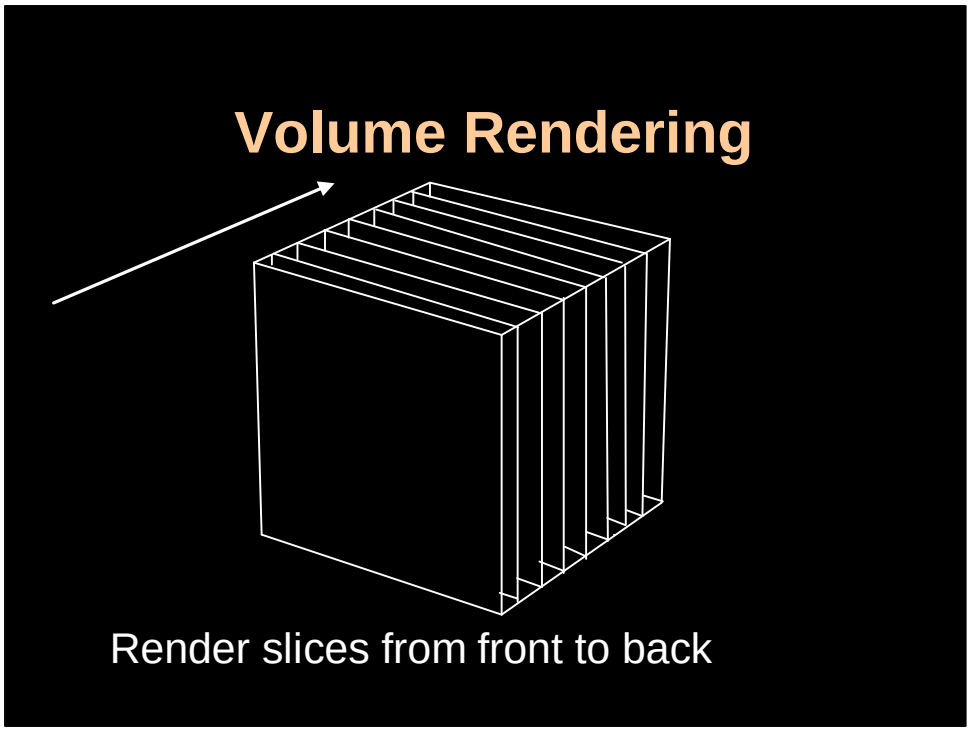
Here I show a demo of the simple solver where I interact in real time with a velocity field and switch back and force between the spatial and the Fourier domain.



As I said at the beginning of the talk, although everything was explained in a 2D setting the algorithm easily extends to 3D by simply adding an extra index and extra for loops. One issue is however how we display the density since it is now a volume. For this we need a fast volume renderer.

Volume Rendering





We do this in hardware by sampling the volume along slices and then rendering them from front to back as semi-transparent quads. The slices are chosen to be aligned with the main axis most aligned with the view. This makes the coding much easier. It results small popping artifacts in some cases. So don't use it in production but it allows to view the density in real-time. You can also self-shadow the volume by doing a shadow prepass from the light source and storing the accumulated transparencies. Essentially I use a 3D Bresenham with the origins at the center of the cells on the faces that are lit.

I now show various 3D demos which still run in real time. Showing the new hardware density rendering with self-shadowing. I also show densities which are textured using 3D texture maps. The Octane and the new nVidia cards have them in hardware, but there are ways to fake them with 2D texture maps, and in fact I have it working on my Dell 1GHz PIII laptop with a GeForce2 GO card in it. I also show some demos with boundaries where the smoke slides over them, etc. All this in real time of course.

Defeating Dissipation

Work with

Ronald Fedkiw & Henrik Wann Jensen

Stanford University

I will now talk about some recent work with Ron Fedkiw and Henrik Wenn Jensen both from Stanford University. As you could tell from the demos the motion is nice but tends to be dampened faster than actual flows. This is because to enforce stability additional damping is added to the flow. So the flows although they appear cool when stirred actually die out to fast. In this new work we address this problem and apply it specifically to smoke.

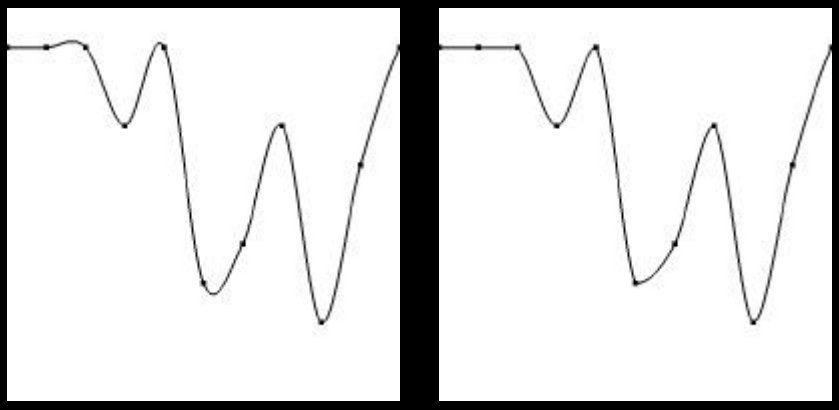
Higher Order Interpolation

Use Hermite interpolation instead of Linear interpolation.

The first technique is to improve the interpolation that is used during the self-advection step. We get higher accuracy with better interpolants such as the Hermite curve shown on the slide. The problem, however, is that these interpolants may under or overshoot the data. So we do not have the nice property anymore that the max of the new values is bounded by the max of the old values. And so potentially the simulation might blow up and we lose the nice property of stability.

Higher Order Interpolation

Force interpolant to be monotonic to avoid instabilities



To fix this problem we introduce some new splines which are guaranteed to not overshoot the data. The picture on left shows data interpolated with a standard Hermite spline. You can clearly see the overshoots and under shoots where the data isn't smooth, this is a basic behavior of any high order interpolant. The curve on the right shows our new interpolation scheme. Notice how there no over or undershoots and that the curve is discontinuous where it should be. Our solver with this interpolation scheme is stable for any time step.

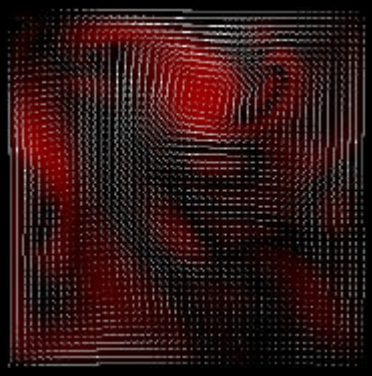
Higher Order Interpolation

linear

Hermite

Here are two animations where you can see the increase in precision using the new scheme. Of course the interpolation is more expensive, about 9 times more expensive than the linear interpolation. So the linear one is useful in pre-production phase, to get the motion basically right. While the more expensive solver can be useful at later stages to refine the motion.

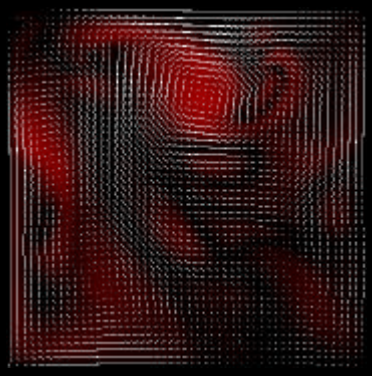
Vorticity Confinement



$$\omega = \nabla \times \mathbf{u}$$

So that's one way of reducing the numerical dissipation of the basic Stable Fluids algorithm. The next technique is very cool. The basic idea is to reinject the energy that is dissipated back into the flow through a clever force field. Instead of looking at the velocity we will consider the vorticity field which is defined as the curl of the velocity. Again this is a vector field. In 2D this vector always points out of the fluid plane and the vorticity can be treated as a scalar field. On the slide I show the velocity field I show the velocity field and the corresponding magnitude of the vorticity in red. Notice how the field is strongest where the rotation is largest. Vorticity is thus a measure of how much the flow rotates. Clearly for turbulent smoke we want the vorticity to be strong and localized. This will result in nice looking swirling smoke fields.

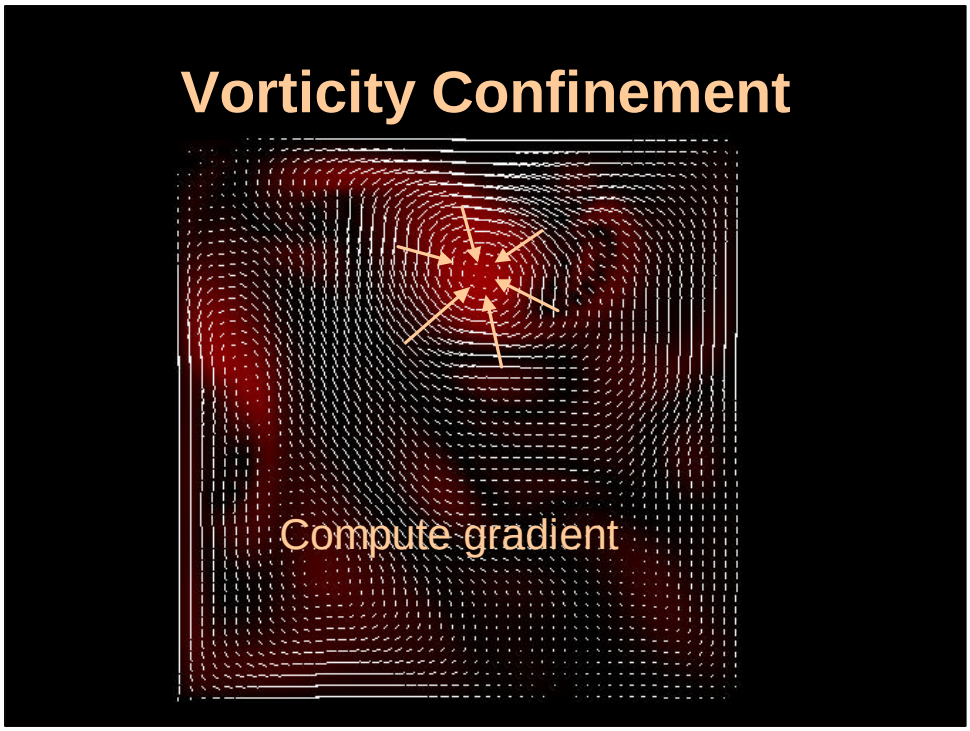
Vorticity Confinement



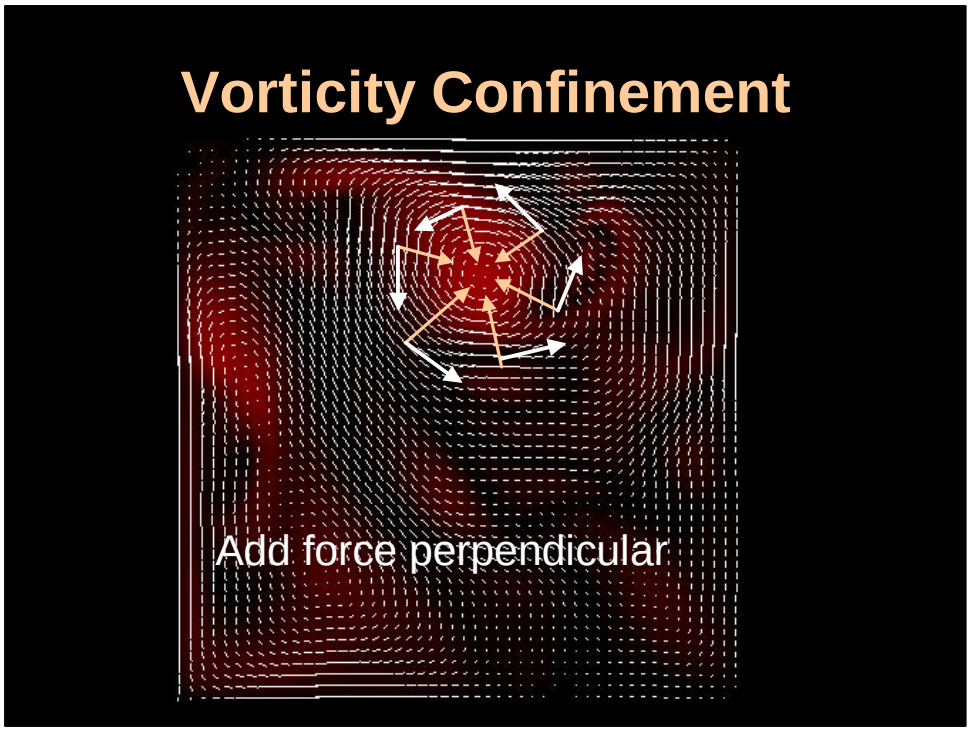
Basic idea :Increase vorticity

John Steinhoff 1987 (Flow Analysis)

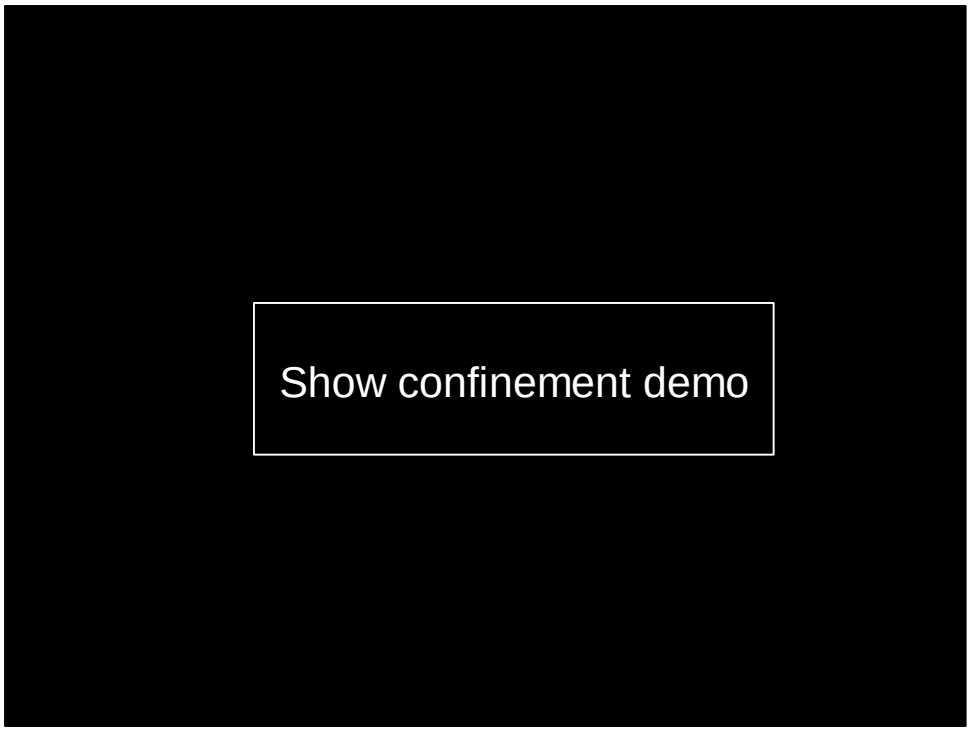
So the idea is to keep the vorticity alive using an external force. The basic idea is due to an amazing idea from John Steinhoff that he had some ten years ago. John is a physicist who has his own company called Flow Analysis, he has many other cool ideas that are implemented in his commercial software.



The first step in computing the confinement force, is to compute the gradient of the vorticity. So this field points inwards to the vortex center.



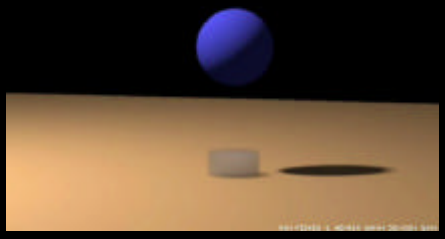
The next step is to add a force perpendicular to the gradient field. This will tend to keep the vortices alive and well localized.



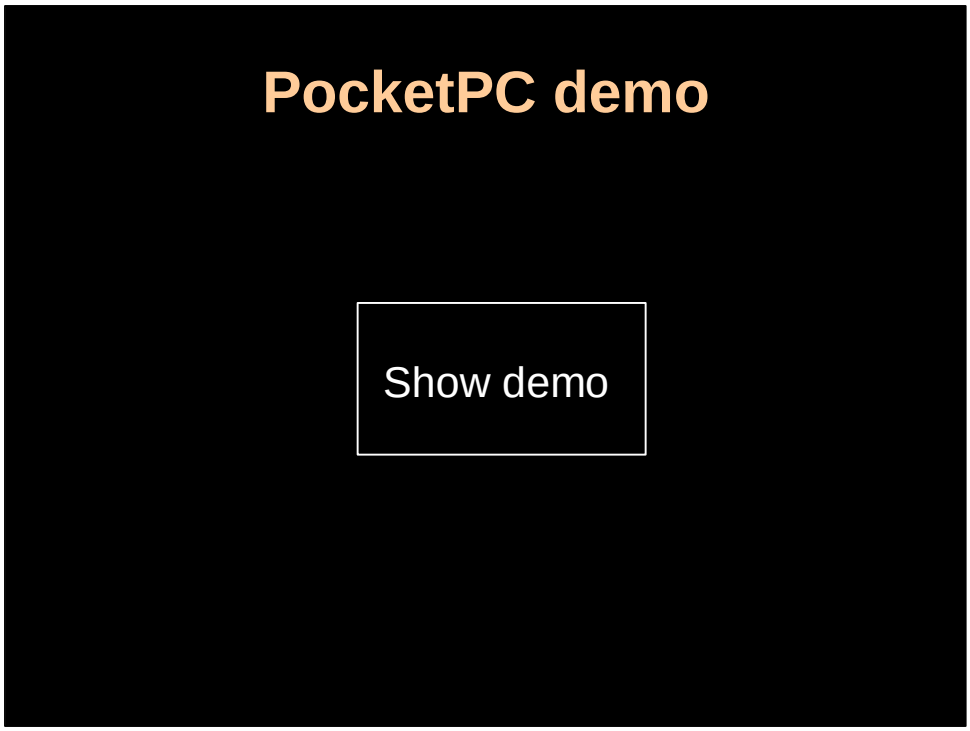
In this demo I do a side by side comparison of the solver with and without the confinement force. This is a real time demo and I display the vorticity field and also show how a density field has more detail with the confinement force added in. I also show some demos of a wind tunnel, in it you get the typical Karman vortex street pattern, this puts the Reynold's number of these flows at roughly 200. I also show 3D demos with the vorticity confinement, with object interacting with smoke, such as spheres and cloth.

This is a high quality animation rendered using Jensen’s photon map.

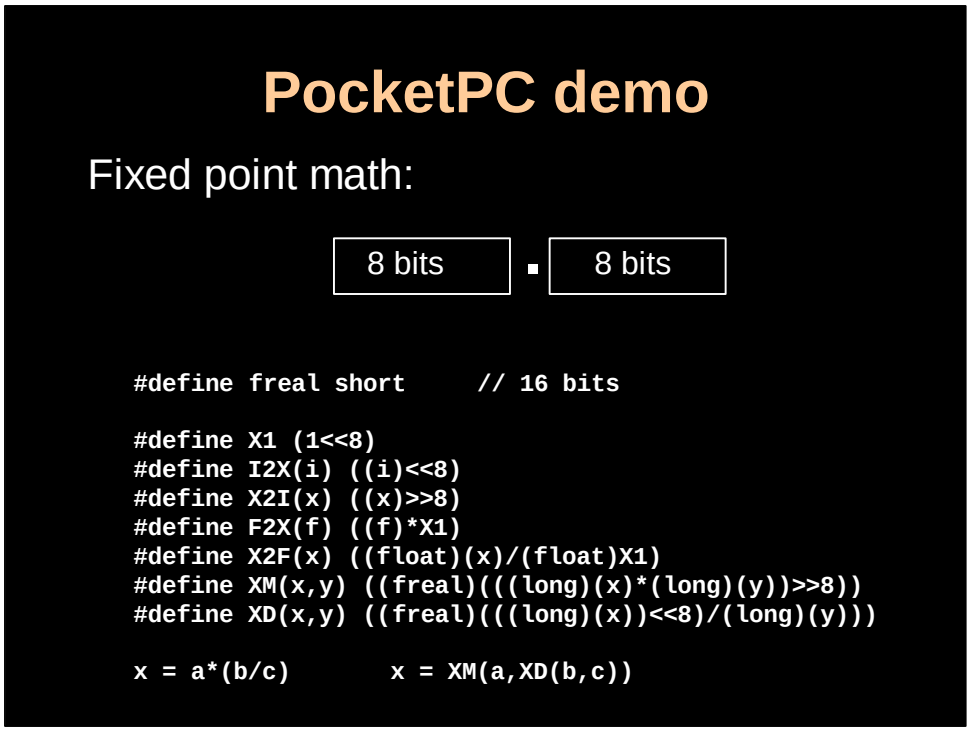
High Quality Renderings



Here is another one with a sphere



Finally the last demo is where I show a 2D solver that runs on my Pocket PC (Compaq iPaq H3600). I did all the rendering myself directly to screen space using the GAPI gamer interface for the Pocket PC. And I ported my entire solver to fixed point arithmetic since the CPU in the PocketPC (Intel StrongARM) does not have hardware support for floats.



In fixed point math all real are represented as 16 bit integers: 8 bit for the integer part and 8 for the fractional part. It is really easy to implement the equivalent operations for fixed point reals. All you have to do is define the following defines and use them instead of the usual operations. You have to be careful with certain algorithms that do not work well with fixed point due to underflows and overflows. For example the conjugate gradient fails miserably, so you are better off using a simple relaxation technique like Gauss-Seidel.

Future Work

- Handle free boundaries (water)
- Parallel implementation (in progress)
- Adaptive grids (in progress)
- “Smarter” texture maps

Here some of the stuff I am working on. I would like to have a real time simulation of water. This is harder as the boundary conditions change over time (at the water surface). I hope that I can reuse some of the ideas presented here and apply them to water. Also I would like to parallelize my code. Of course I do not have the hardware to test my ideas. That’s a problem. However, I was able to parallelize the code on my dual PIII at work. I simply feed the velcoity solver to one CPU and the density solver (+texture coord or temperature) to the second CPU, this gave some dramatic speedups. Also we are working on adaptive grids. The grid can be adapted to the region of interest. Imagine a thin trail of smoke, than it would be a waste to have a uniform grid for all spatial directions. Also I am looking into designing smarter texture maps to model things like pyroclastic flows. These flows are so highly detailed that they are beyond grid technology at this point, the same with clouds. So the idea is to add a smart texture map with billowing motion which is driven by a coarser simulation.

Thank you for your attention.

Any questions ?

Procedural Volumetric Modeling and Animation of Clouds and Other Gaseous Phenomena

David S. Ebert

ebertd@purdue.edu

1 An Introduction to Procedural Modeling and Animation

As mentioned in the course introduction, we are working in a very exciting time. The combination of increased CPU power with powerful, and now **programmable,** graphics processors (GPUs) available on affordable PCs has started an age where we can envision and realize interactive complex procedural models.

This chapter presents a framework for procedural modeling and animation of gaseous phenomena using volumetric procedural models: a general class of procedural techniques for modeling natural phenomena. Volumetric procedural models use three-dimensional volume density functions

(*vdf(x,y,z)*) that define the density of a continuous three-dimensional space. Volume density functions (vdf's) are the natural extension of solid texturing [Perlin1985] to describe the actual geometry of objects. Volume density functions are used extensively in computer graphics for modeling and animating gases, fire, fur, liquids, and other "soft" objects. Hypertextures [Perlin1989] and Inakage's flames [Inakage1991] are other examples of the use of volume density functions.

Many advanced geometric modeling techniques are inherently procedural. L-systems, fractals, particle systems, and implicit surfaces [Bloomenthal1997] (also called blobs and metaballs), are, to some extent, procedural and can be combined nicely into the framework of volumetric procedural models. This chapter will concentrate on the development and animation of several volume density functions for creating realistic images and animations of clouds. The inclusion of fractal techniques into vdf's is presented in most of the examples because they rely on a statistical simulation of turbulence that is fractal in nature. This chapter will also briefly discuss the inclusion of implicit function techniques into vdf's. A more complete discussion of the simulation of noise and turbulence functions, as well as a thorough presentation of procedural modeling and texturing techniques, can be found in *Texturing and Modeling: A Procedural Approach, 2nd edition,* [Ebert1998][1]. This text also contains a thorough discussion of the detailed development of volumetric procedural modeling, which forms a good basis to this chapter.

1.1 Procedural Techniques and Computer Graphics

Procedural techniques have been used throughout the history of computer graphics. Many early modeling and texturing techniques included procedural definitions of geometry and surface color. From these early beginnings, procedural techniques have exploded into an important, powerful modeling, texturing, and animation paradigm. During the mid- to late 1980s, procedural techniques for creating realistic textures, such as marble, wood, stone, and other natural material gained widespread use. These techniques were extended to procedural modeling, including models of

[1] As of March 1999, two of the co-authors of this book have won Academy Awards for their computer graphics contributions used in motion pictures: Darwyn Peachey and Ken Perlin.

water, smoke, steam, fire, planets, and even tribbles. The development of the RenderMan[2] shading language [Pixar1989] in 1989 greatly expanded the use of procedural techniques. Currently, most commercial rendering and animation systems even provide a procedural interface. Procedural techniques have become an exciting, vital aspect of creating realistic computer generated images and animations. As the field continues to evolve, the importance and significance of procedural techniques will continue to grow.

1.2 Definition and Power of Procedural Techniques

Procedural techniques are code segments or algorithms that specify some characteristic of a computer generated model or effect. For example, a procedural texture for a marble surface does not use a scanned-in image to define the color values. Instead, it uses algorithms and mathematical functions to determine the color.

One of the most important features of procedural techniques is ***abstraction***. In a procedural approach, rather than explicitly specifying and storing all the complex details of a scene or sequence, we *abstract* them into a function or an algorithm (i.e., a *procedure*) and evaluate that procedure when and where needed. We gain a storage savings, as the details are no longer explicitly specified but rather implicit in the procedure, and shift the time requirements for specification of details from the programmer to the computer. This allows us to create inherently multi-resolution models and textures that we can evaluate to the resolution needed.

We also gain the power of ***parametric control***, allowing us to assign to a parameter a meaningful concept (e.g., a number that makes mountains "rougher" or "smoother"). Parametric control also provides amplification of the modeler/animator's efforts: a few parameters yield large amounts of detail (Smith [Smith1984] referred to this as *database amplification*). This parametric control unburdens the user from the low-level control and specification of detail. We also gain the serendipity inherent in procedural techniques: we are often pleasantly surprised by the unexpected behaviors of procedures, particularly stochastic procedures. Procedural models also offer ***flexibility***. The designer of the procedures can capture the *essence* of the object, phenomenon, or motion

---

[2] RenderMan is a registered trademark of Pixar.

without being constrained by the complex laws of physics. Procedural techniques allow the inclusion of any amount of physical accuracy into the model that is desired. The designer may produce a wide range of effects, from accurately simulating natural laws to purely artistic effects.

1.3 Background

The techniques described in this chapter use procedurally defined volume density functions for modeling, texturing, and animating gases. There have been numerous previous approaches to modeling gases in computer graphics, including Kajiya's simple physical approximation [Kajiya1984], Gardner's solid textured ellipsoids [Gardner1985, Gardner1990], Max's height fields [Max1986], constant density media [Klassen1987, Nishita1987], and fractals [Voss1983]. I have developed several approaches for modeling and controlling the animation of gases [Ebert 1989, Ebert1990a, Ebert1990b, Ebert1991, Ebert1992]. Stam has used "fuzzy blobbies" as a three-dimensional model for animating gases [Stam1993] and has extended their use to modeling fire [Stam1995].

Volume rendering is essential for realistic images and animations of gases. Any procedure-based volume rendering system, such as Perlin's [Perlin1989], or my system [Ebert1990a, Ebert1991], can be used for rendering volumetric procedural models. A volume ray-tracer is also very easy to extend to allow the support of procedural volumetric modeling. My system is a hybrid rendering system that uses a fast scanline a-buffer rendering algorithm for the surface-defined objects in the scene, while volume modeled objects are volume rendered. This rendering system features a physically-based low-albedo illumination and atmospheric attenuation model for the gases. Volumetric shadows are also efficiently combined into the system through the use of three-dimensional shadow tables [Ebert1990a, Ebert1998]. Precomputing these procedures at a fixed resolution 3D grid defeats many of the advantages of the procedural model, but allows these to be loaded as 3D textures into the latest PC and workstation boards and enables interactive rendering and exploration of these procedural models. PC graphics boards are now available with 3D texture mapping hardware, which enables interactive or even real-time volume rendering with precomputed volumetric models. Additionally, 3D texture mapping is part of the DirectX 8.0 standard and the OpenGL 1.2 standard, so we should expect to find this feature on all high-end PC graphics boards shortly.

2 Volumetric Cloud Modeling with Implicit Functions

Modeling clouds is a very difficult task because of their complex, amorphous structure and because even an untrained eye can judge the realism of a cloud model. Their ubiquitous nature makes them an important modeling and animation task. This chapter describes a new volumetric procedural approach for cloud modeling and animation that allows easy, natural specification and animation of the clouds, flexibility to include as much physics or art as desired into the model, unburdens the user from detailed geometry specification, and produces realistic volumetric cloud models. This technique combines the flexibility of volumetric procedural modeling with the smooth blending and ease of control of primitive-based implicit functions (metaballs, blobs) to create a powerful new modeling technique. This technique also demonstrates the advantages of primitive-based implicit functions for modeling semi-transparent volumetric objects.

2.1 Background

Modeling clouds in computer graphics has been a challenge for over twenty years [Dungan1979] and major advances in cloud modeling still warrant presentation in the SIGGRAPH Papers Program (e.g., [Dobashi2000]). Many previous approaches have used semi-transparent surfaces to produce convincing images of clouds [Gardner1984, Gardner1985, Gardner1990, Voss1983]. Although these techniques can produce realistic images of clouds viewed from a distance, these cloud models are hollow and do not allow the user to seamlessly enter, travel through, and inspect the interior of the cloud model. To capture the three-dimensional structure of a cloud, volumetric density-based models must be used. Kajiya [Kajiya1984] produced the first volumetric cloud model in computer graphics, but the results are not photo-realistic. Stam [Stam1995], Foster [Foster1997], and Ebert [Ebert1994] have produced convincing volumetric models of smoke and steam, but have not done substantial work on modeling clouds. Neyret [Neyret1997] has recently produced some preliminary results of a convective cloud model based on general physical characteristics. This model may be promising for simulating convective clouds; however, it currently uses surfaces (large particles) to model the cloud structure. A general, flexible, easy-to-use, realistic volumetric cloud model is still needed in computer graphics.

In developing this new cloud modeling and animation system, I have chosen to build upon the recent work in advanced modeling techniques and volumetric procedural modeling. Many

advanced geometric modeling techniques, such as fractals [Peitgen1992], implicit surfaces [Blinn1982, Wyvill1986, Nishimura1985], grammar-based modeling [Smith1984, Prusinkiewicz1990], and volumetric procedural models/hypertextures [Perlin1985, Ebert1994] use procedural abstraction of detail to allow the designer to control and animate objects at a high level. Their inherent procedural nature provides flexibility, data amplification, abstraction of detail, and ease of parametric control. When modeling complex volumetric phenomena, such as clouds, this abstraction of detail and data amplification are necessary to make the modeling and animation tractable. It would be impractical for an animator to specify and control the detailed three-dimensional density of a cloud model. This system does not use a physics-based approach because it is computationally prohibitive and non-intuitive to use for many animators and modelers. Setting and animating correct physics parameters for dew point, particulate distributions, temperature and pressure gradients, etc. is a time-consuming, detailed task. This model was developed to allow the modeler and animator to work at a much higher level. I also didn't want to restrict the results by the laws of physics, but to allow for artistic expression.

Volumetric procedural models have all of the advantages of procedural techniques and are a natural choice for cloud modeling because they are the most flexible advanced modeling technique. Since a procedure is evaluated to determine the object's density, any advanced modeling technique, simple physics simulation, mathematical function or artistic algorithm can be included in the model.

Combining traditional volumetric procedural models with implicit functions creates a model that has the advantages of both techniques. Implicit functions have been used for many years as a modeling tool for creating solid objects and smoothly blended surfaces [Bloomenthal1997]. However, little work has been done to explore their potential for modeling volumetric density distributions of semi-transparent volumes. Nishita [Nishita1996] has used volume rendered implicits as a basic cloud model in his work on multiple scattering illumination models; however, this work has concentrated on illumination effects and not on realistic modeling of the cloud geometry. Stam has also used volumetric blobbies to create his models of smoke and clouds [Stam1991, Stam1993, Stam1995]. His work is related to the approach described in this chapter. My early work on using volume rendered implicit spheres to produce a fly-through of a volumetric cloud was described in [Ebert1997a]. This work has been developed further to use implicits to provide a natural way of specifying and animating the global structure of the cloud, while using

more traditional procedural techniques to model the detailed structure.

2.2 Volumetric Procedural Modeling With Implicit Functions

The volumetric cloud model uses a two-level: the cloud macrostructure and the cloud microstructure. Implicit functions and turbulent volume densities model these, respectively. The basic structure of the cloud model combines these two components to determine the final density of the cloud.

The cloud's microstructure is created by using procedural *turbulence* and *noise* functions, in a manner similar to my *basic_gas* function (see [Ebert1998]). This allows the procedural simulation of natural detail to the level needed. Simple mathematical functions are added to allow shaping of the density distributions and control over the sharpness of the density falloff.

Implicit functions were chosen to model the cloud macrostructure because of their ease of specification and smoothly blending density distributions. The user simply specifies the location, type, and weight of the implicit primitives to create the overall cloud shape. Any implicit primitive, including spheres, cylinders, ellipsoids, and skeletal implicits can be used to model the cloud macrostructure. Since these are volume rendered as a semi-transparent medium, the whole volumetric field function is being rendered, as compared to implicit surface rendering where only a small range of values of the field are used to create the objects. The implicit density functions are primitive-based density functions: they are defined by summed, weighted, parameterized, primitive implicit surfaces. A simple example of the implicit formulation of a sphere centered at the point ***center*** with radius ***r*** is the following:

$$F(x,y,z): \ (x - center.x)^2 + (y-center.y)^2 + (z-center.z)^2 - r^2 = 0.$$

The real power of implicit functions is the smooth blending of the density fields from separate primitive sources. I chose to use Wyvill's standard cubic function [Wyvill1986] as the density (blending) function for the implicit primitives:

$$F_{cub}(r) = -\frac{4}{9}\frac{r^6}{R^6} + \frac{17}{9}\frac{r^4}{R^4} - \frac{22}{9}\frac{r^2}{R^2} + 1.$$

In the above equation, ***r*** is the distance from the primitive. This density function is a cubic in the

distance squared and its value ranges from 1 when ***r=0*** (within the primitive) to 0 at ***r=R***. This density function has several advantages. First, its value drops off quickly to zero (at the distance ***R***), reducing the number of primitives that must be considered in creating the final surface. Second, it has zero derivatives at ***r=0*** and ***r=R*** and is symmetrical about the contour value ***0.5***, providing for smooth blends between primitives. The final implicit density value is then the weighted sum of the density field values of each primitive:

$$Density_{implicit}(p) = \sum_i (w_i F_{cub_i}(p - q))$$

where w_i is the weight of the ith primitive and ***q*** is the closest point on element ***i*** from ***p***.

To create non-solid implicit primitives, the location of the point is procedurally altered before the evaluation of the blending functions. This alteration can be the product of the procedure and the implicit function and/or a warping of the implicit space.

These techniques are combined into a simple cloud model as shown below:

```
volumetric_procedural_implicit_function(pnt, blend%, pixel_size)
   perturbed_point = procedurally alter pnt using noise and turbulence
   density1 = implicit_function(perturbed_point)
   density2 = turbulence(pnt, pixel_size)
   blend = blend% * density1 +(1 - blend%) * density2
   density = shape resulting density based on user controls for
             wispiness and denseness(e.g., use pow & exponential
             function)
   return(density)
```

The density from the implicit primitives is combined with a pure turbulence based density using a user specified ***blend%*** (60% to 80% gives good results). The blending of the two densities allows the creation of clouds that range from entirely determined by the implicit function density to entirely determined by the procedural turbulence function. When the clouds are completely determined by the implicit functions, they will tend to look more like cotton balls. The addition of the procedural alteration and turbulence is what gives them their naturalistic look.

2.3 Volumetric Cloud Modeling

The volumetric procedural implicit algorithm given above forms the basis of a flexible system for the modeling of volumetric objects. This chapter focuses on the use of these techniques for modeling and animating realistic clouds. The volume rendering of the clouds is not discussed in detail. For a description of the volume rendering system that was used to make my images of clouds in this book, please see [Ebert1990a, Ebert1998]. Any volume rendering system can be used with these volumetric cloud procedures; however, to get realistic effects, the system should accumulate densities using atmospheric attenuation, and a physics-based illumination algorithm should be used. For accurate images of cumulus clouds, a high-albedo illumination algorithm (e.g., [Max1994, Nishita1996]) is needed.

2.3.1 Cumulus Clouds

Cumulus clouds are very common in nature and can be easily simulated using spherical or elliptical implicit primitives. Figure 1 shows the type of result that can be achieved by using nine implicit spheres to model a cumulus cloud. The animator/modeler simply positions the implicit spheres to produce the general cloud structure. Procedural modification then alters the density distribution to create the detailed wisps. The algorithm used to create the clouds in Figure 1 and Figure 2 is the following:

```
cumulus(pnt,density,parms, pnt_w, vol)
     xyz_td  pnt;             /* location of point in cloud space */
     xyz_td  pnt_w;           /* location of point in world space */
     float   *density,*parms;
     vol_td  vol;
{
  float new_turbulence();     /* my turbulence function */
  float peachey_noise();      /* Darwyn Peachey's noise function */
  float metaball_evaluate();  /* function for evaluating the metaball primitives*/
  float mdens,                /* metaball density value */
        turb,                 /* turbulence amount */
        peach;                /* Peachey noise value */
  xyz_td path;                /* path for swirling the point */
  extern int frame_num;
  static int ncalcd=1;
  static float sin_theta_cloud, cos_theta_cloud, theta,
         path_x, path_y, path_z, scalar_x, scalar_y, scalar_z;

  /* calculate values that only depend on the frame number once per frame
   */
  if(ncalcd)
    {
      ncalcd=0;
      /* create gentle swirling in the cloud */
      theta =(frame_num%600)*01047196; /* swirling effect */
      cos_theta_cloud = cos(theta);
      sin_theta_cloud = sin(theta);
      path_x = sin_theta_cloud*.005*frame_num;
```

```
      path_y = .01215*(float)frame_num;
      path_z= sin_theta_cloud*.0035*frame_num;
      scalar_x = (.5+(float)frame_num*0.010);
      scalar_z = (float)frame_num*.0073;
    }

   /* Add some noise to the point's location
    */
   peach = peachey_noise(pnt); /* Use Darwyn Peachey's noise function */
   pnt.x -= path_x -peach*scalar_x;
   pnt.y = pnt.y - path_y +.5*peach;
   pnt.z += path_z - peach*scalar_z;

   /* Perturb the location of the point before evaluating the implicit primitives.
    */
   turb=fast_turbulence(pnt);
   turb_amount=parms[4]*turb;
   pnt_w.x += turb_amount;
   pnt_w.y -= turb_amount;
   pnt_w.z += turb_amount;

   mdens=(float)metaball_evaluate((double)pnt_w.x, (double)pnt_w.y,
                          (double)pnt_w.z, (vol.metaball));

  *density= parms[1]*(parms[3]*mdens + (1.0 - parms[3])*turb*mdens);
  *density= pow(*density,(double)parms[2]);
}
```

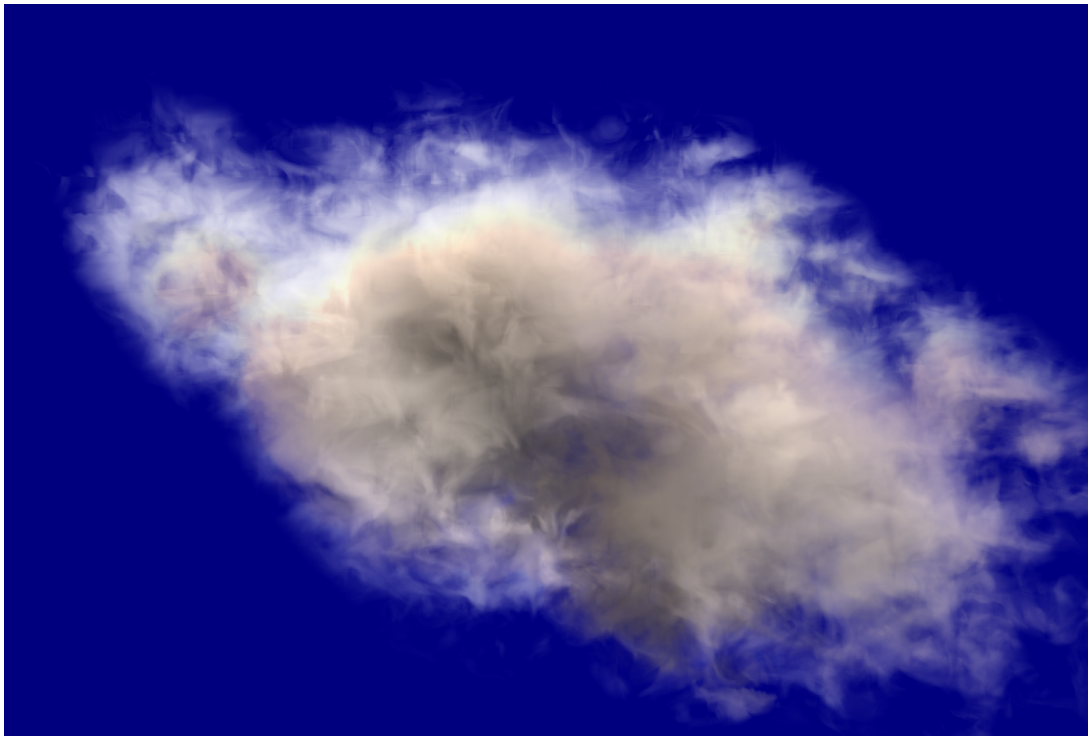


Figure 1: An example of a cumulus cloud. Copyright 2001 David S. Ebert

Parms[3] is the blending function value between implicit (metaball) density and the product of the turbulence density and the implicit density. This method of blending ensures that the entire cloud density is a product of the implicit field values, preventing cloud pieces from occurring outside the

defining primitives. Using a large ***parms[3]*** generates clouds that are mainly defined by their implicit primitives and are, therefore, ``smoother'' and less turbulent. ***Parms[1]*** is a density scaling factor, ***parms[2]*** is the exponent for the ***pow()*** function, and ***parms[4]*** controls the amount of turbulence to use in displacing the point before evaluation of the implicit primitives. For good images of cumulus clouds, useful values are the following: ***0.2 < parms[1] < 0.4***, ***parms[2] = 0.5***, ***parms[3]=0.4***, and ***parms[4] = 0.7*** .

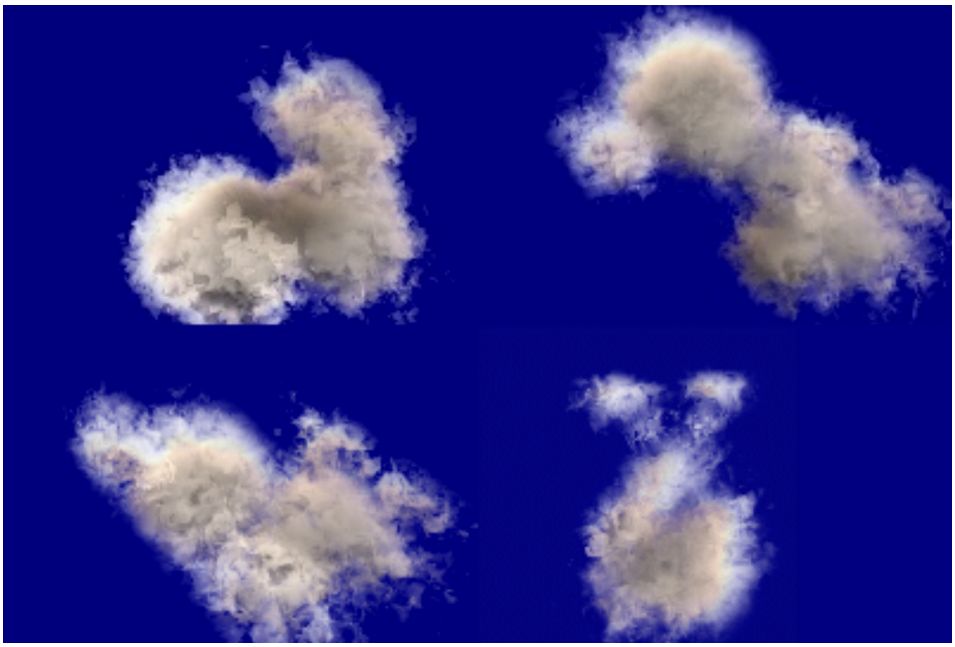


Figure 2: Example cloud creatures: Copyright 1998 David S. Ebert

2.3.2 Cirrus and Stratus Clouds

Cirrus clouds differ greatly from cumulus clouds in their density, thickness, and falloff. In general, cirrus clouds are thinner, less dense, and wispier. These effects can be created by altering the parameters to the *cumulus* cloud procedure and also by changing the implicit primitives. The density value parameter for a cirrus cloud is normally chosen as a smaller value and the exponent is chosen larger, producing larger areas of no clouds and a greater number of individual clouds. To create cirrus clouds, the user can simply specify the global shape (envelope) of the clouds with a few implicit primitives or specify implicit primitives to determine the location and shape of each cloud. In the former case, the shape of each cloud is mainly controlled by the volumetric

procedural function and turbulence simulation, as opposed to cumulus clouds where the implicit functions are the main shape control. It is also useful to modulate the densities along the direction of the jet stream to produce more natural wisps. This can be created by the user specifying a predominant direction of wind flow and using a turbulent version of this vector in controlling the densities as follows:

```
Cirrus(pnt,density,parms, pnt_w, vol, jet_stream)
     xyz_td  pnt;             /* location of point in cloud space */
     xyz_td  pnt_w;           /* location of point in world space */
     xyz_td  jet_stream;
     float   *density,*parms;
     vol_td  vol;
{
  float new_turbulence();     /* my turbulence function */
  float peachey_noise();      /* Darwyn Peachey's noise function */
  float metaball_evaluate();  /* function for evaluating the metaball primitives*/
  float mdens,                /* metaball density value */
        turb,                 /* turbulence amount */
        peach;                /* Peachey noise value */
  xyz_td path;                /* path for swirling the point */
  extern int frame_num;
  static int ncalcd=1;
  static float sin_theta_cloud, cos_theta_cloud, theta,
         path_x, path_y, path_z, scalar_x, scalar_y, scalar_z;

  /* calculate values that only depend on the frame number once per frame */
  if(ncalcd)
    { ncalcd=0;
      /* create gentle swirling in the cloud */
      theta =(frame_num%600)*01047196; /* swirling effect */
      cos_theta_cloud = cos(theta);
      sin_theta_cloud = sin(theta);
      path_x = sin_theta_cloud*.005*frame_num;
      path_y = .01215*(float)frame_num;
      path_z= sin_theta_cloud*.0035*frame_num;
      scalar_x = (.5+(float)frame_num*0.010);
      scalar_z = (float)frame_num*.0073;
    }

   /* Add some noise to the point's location */
   peach = peachey_noise(pnt); /* Use Darwyn Peachey's noise function */
   pnt.x -= path_x -peach*scalar_x;
   pnt.y = pnt.y - path_y +.5*peach;
   pnt.z += path_z - peach*scalar_z;

   /* Perturb the location of the point before evaluating the implicit
    * primitives.*/
   turb=fast_turbulence(pnt);
   turb_amount=parms[4]*turb;
   pnt_w.x += turb_amount;
   pnt_w.y -= turb_amount;
   pnt_w.z += turb_amount;

   /* make the jet stream turbulent */
   jet_stream.x + =.2*turb;
   jet_stream.y + =.3*turb;
   jet_stream.z + =.25*turb;

   /* warp point along the jet stream vector */
   pnt_w = warp(jet_stream, pnt_w);
   mdens=(float)metaball_evaluate((double)pnt_w.x, (double)pnt_w.y,
                        (double)pnt_w.z, (vol.metaball));
  *density= parms[1]*(parms[3]*mdens + (1.0 - parms[3])*turb*mdens);
  *density= pow(*density,(double)parms[2]);
}
```

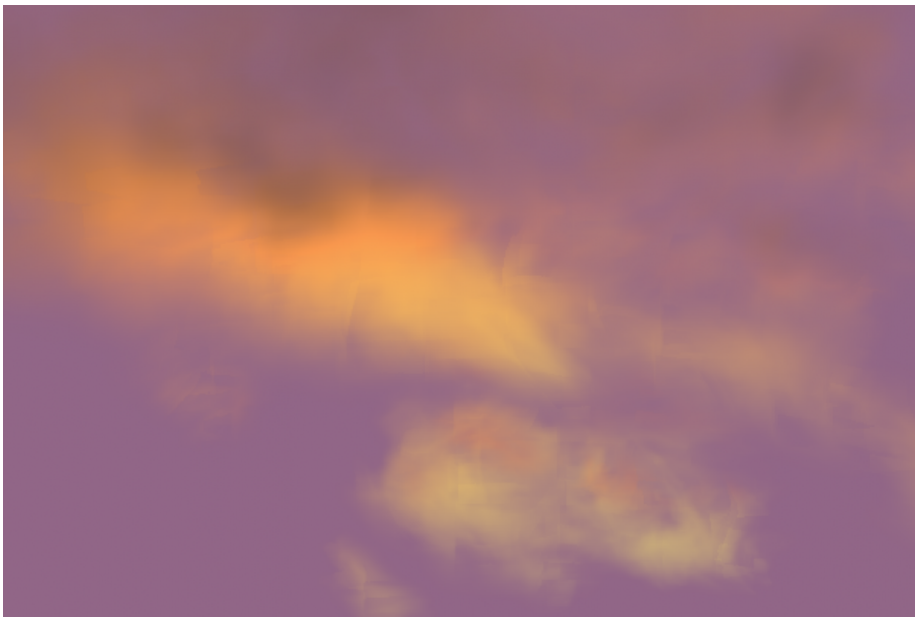


Figure 3: Cirrus Clouds. Copyright 1998 David S. Ebert

Several examples of cirrus cloud formations created using these techniques can be seen in Figure 3 and Figure 4**.** Figure 4 shows a higher cirrostratus layer created by a large elliptical primitive and a few individual lower cirrus clouds created with cylindrical primitives.

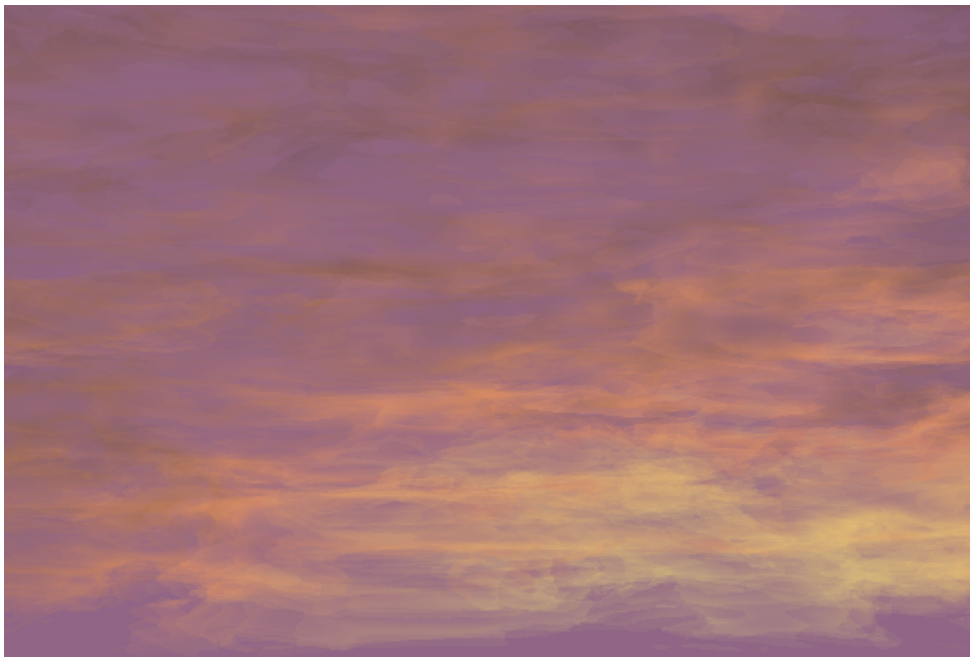


Figure 4: Another example of cirrostratus clouds. Copyright 1998 David S. Ebert

Stratus clouds can also be modeled by using a few implicits to create the global shape or extent of the stratus layer, while using volumetric procedural functions to define the detailed structure of all of the clouds within this layer. Stratus cloud layers are normally thicker and less wispy, as compared with cirrus clouds. This effect can be created by adjusting the size of the turbulent space (smaller/fewer wisps), using a smaller exponent value (creates more of a cloud layer effect), and increasing the density of the cloud. Using simple mathematical functions to shape the densities can create some of the more interesting stratus effects, such as a mackerel sky. The mackerel stratus cloud layer in Figure 5 was created by modulating the densities with turbulent sine waves in the x and y directions.

Figure 5: A mackerel stratus cloud layer. Copyright 1998 David S. Ebert

2.3.3 Cloud Creatures

The combination of implicit functions with volumetric procedural models provides an easy to use system for creating realistic clouds, artistic clouds, and cloud creatures. Some examples of cloud

creatures created using a simple graphical user interface (GUI) to position nine implicit spheres can be seen in Figure 2**.** They were designed in less than 15 minutes each, and a straw poll shows that viewers have seen many different objects in them (similar to real cloud shapes). Currently, the simple GUI only allows access to a small portion of the system. The rest of the controls are available through a text-based interface. More complex shapes, time-based deformations, and animations can be created by allowing the user access to more of the controls, implicit primitives, and parameters of the full cloud modeling system. These cloud creatures are easily designed and animated by controlling the implicit primitives and procedural parameters. The implicit primitives blend and deform smoothly, allowing the specification and animation of skeletal structures, and provide an intuitive interface to modeling amorphous volumetric creatures.

2.3.4 User Specification and Control

Since the system uses implicit primitives for the cloud macrostructure, the user creates the general cloud structure by specifying the location, type, and weight of each implicit primitive. For the image in Figure 1**,** nine implicit spheres were positioned to create the cumulus cloud. Figure 2 shows the wide range of cloud shapes and creatures that can be created by simply adjusting the location of each primitive and the overall density of the model through a simple GUI. The use of implicit primitives makes this a much more natural interface than with traditional procedural techniques. Each of the cloud models in this chapter was created in less than 30 minutes of design time.

The user of the system also specifies a density scaling factor, a power exponent for the density distribution (controls amount of wispiness), any warping procedures to apply to the cloud, and the name of the volumetric procedural function so that special effects can be programmed into the system.

2.4 Animating Volumetric Procedural Clouds

The volumetric cloud models described above produce nice still images of clouds and also clouds that gently evolve over time. The models can be animated using the procedural animation techniques described in [Ebert1991, Ebert1998] or by animating the implicit primitives. Procedural animation is the most flexible and powerful technique since any deformation, warp or physical

simulation can be added to the procedure. An animator ca use key frames or dynamics simulations to animate the implicit primitives. Several examples of applying these two animation techniques for various effects are described below.

2.4.1 Procedural Animation

Both the implicit primitives and the procedural cloud space can be animated algorithmically. One of most useful forms of implicit primitive animation is warping. A time varying warp function can be used to gradually warp the shape of the cloud over time to simulate the formation of clouds, their movement, and their deformation by wind and other forces. Cloud formations are usually altered based on the jet stream. To simulate this effect, all that is needed is to warp the primitives along a vector representing the jet stream. This can be done by warping the points before evaluating the implicit functions. The amount of warping can be controlled by the wind velocity, or gradually added in over time to simulate the initial cloud development. Implicits can be warped along the jet stream as follows:

```
perturb_pnt = procedurally alter pnt using noise and turbulence
height = relative height of perturb_pnt
vector = jet_stream + turbulence(pnt)
perturb_pnt = warp(perturb_pnt, vector, height)
density1 = implicit_function(perturbed_pnt)
...
```

To get more natural effects, it is useful to alter each point by a small amount of turbulence before warping it. Several frames from an animation of a cumulus cloud warping along the jet stream can be seen in Figure 6. To create this effect, ease-in and ease-out based on the frame number was used to animate the warp amount. The implicit primitives' locations do not move in this animation, but the warping function animates the space to move and distort the cloud along the jet stream vector. Other warping functions to simulate squash and stretch [Bloomenthal1997] and other effects can also be used. Instead of a single vector and velocity, a vector field is input into the program to define more complex weather patterns. The current system allows the specification of vector flow tables and tables of functional primitives (attractors, vortices) to control the motion and deformation of the clouds. This procedural warping technique was used successfully by Stam in animating gases [Stam1995].

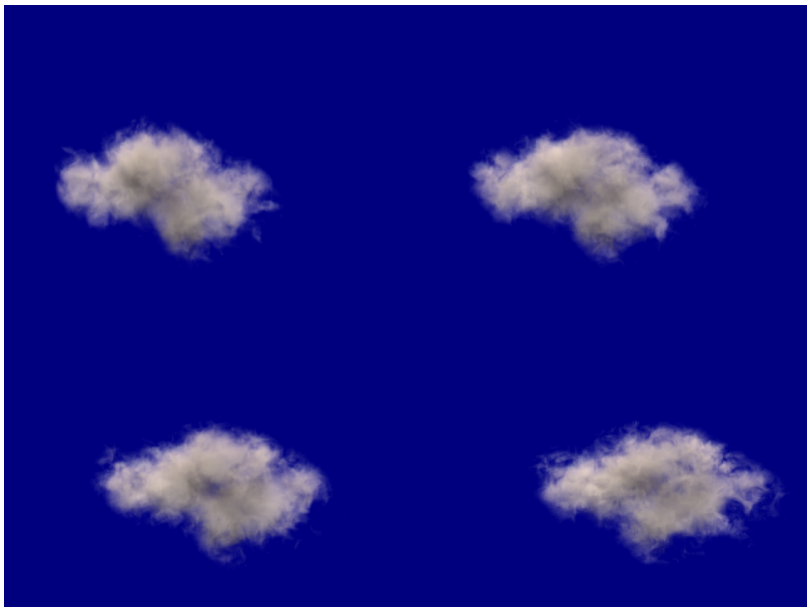


Figure 6: Example cloud warping. Copyright 1998 David S. Ebert

2.4.2 Implicit Primitive Animation

The implicit primitives can be animated in the same manner as implicit surfaces: each primitive's location, parameters (e.g., radii), and weight can be animated over time. This provides an easy to use high-level animation interface for cloud animation. This technique was used in the animation, "A Cloud is Born," [Ebert1997b] showing the birth of a cumulus cloud followed by a fly-through of it. Several stills from the formation sequence can be seen Figure 7. For this animation, the centers of the implicit spheres were moved over time to simulate three separate cloud elements merging and growing into a full cumulus cloud. The radii of the spheres were also increased over time. Finally, to create animation in the detailed cloud structure, each point was moved along a turbulent path over time before evaluation of the turbulence function, as illustrated in the *cumulus* procedure. A powerful animation tool for volumetric procedural implicit functions is the use of dynamics and physics-based simulations to control the movement of the implicits and the deformation of space. Since the implicits are modeling the macro-structure of the cloud while procedural techniques are modeling the microstructure, fewer primitives are needed to achieve complex cloud models. Dynamics simulations can be applied to the clouds by using particle system techniques, with each

particle representing a volumetric implicit primitive. The smooth blending and procedurally generated detail allow complex results with less than a few hundred primitives, a factor of 100 to 1000 less than needed with traditional particle systems. I have implemented a simple particle system for volumetric procedural implicit particles. The user specifies a few initial implicit primitives, dynamics information, such as speed, initial velocity, force function, and lifetime, and the system generates the location, number, size, and type of implicit for each frame. In our initial

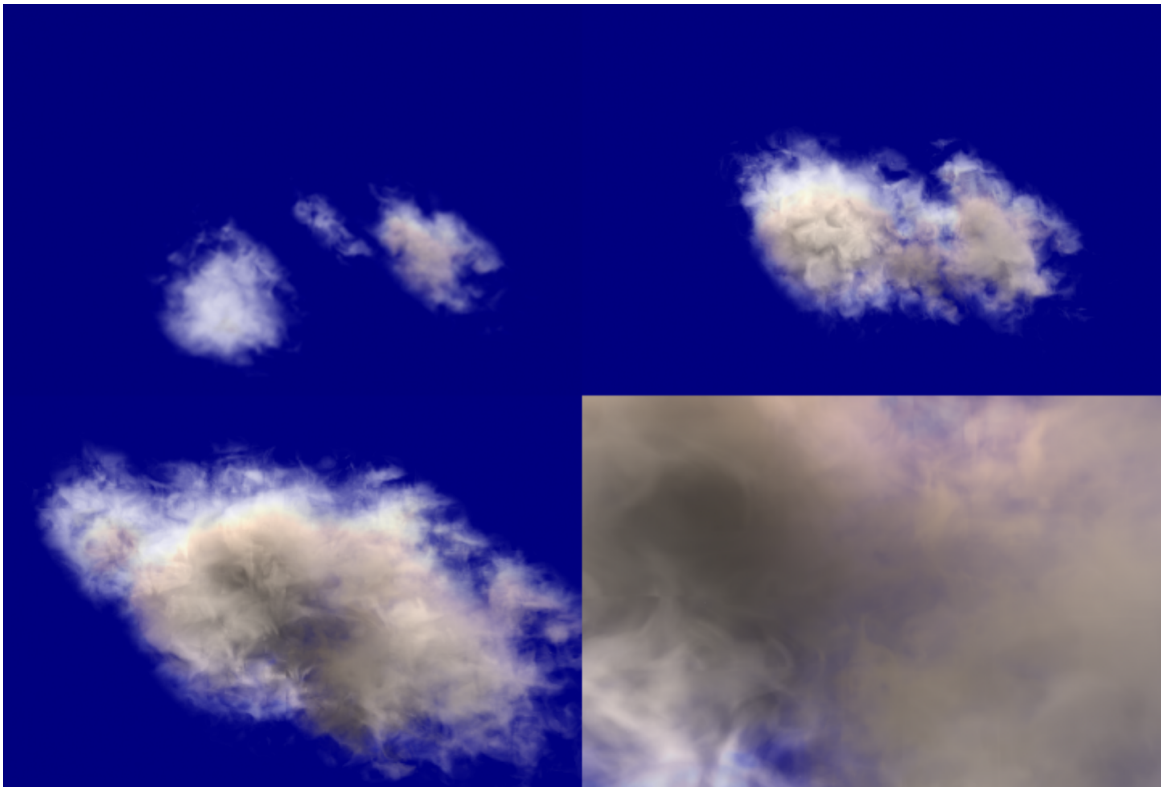


Figure 7: Several stills from "A Cloud is Born." Copyright 1997 David S. Ebert

tests, it took less than 1 minute to generate and animate the implicit particles for 200 frames. Unlike traditional particle systems, cloud implicit particles never die, they just become dormant.

Cumulus clouds created through this volumetric procedural implicit particle system can be seen in Figure 8. The stills in Figure 8 show a cloud created by an upward turbulent force. The number of children created from a particle was also controlled by the turbulence of the particle's location. For the animations in this figure, the initial number of implicit primitives was 12 and the final number was approximately 50.

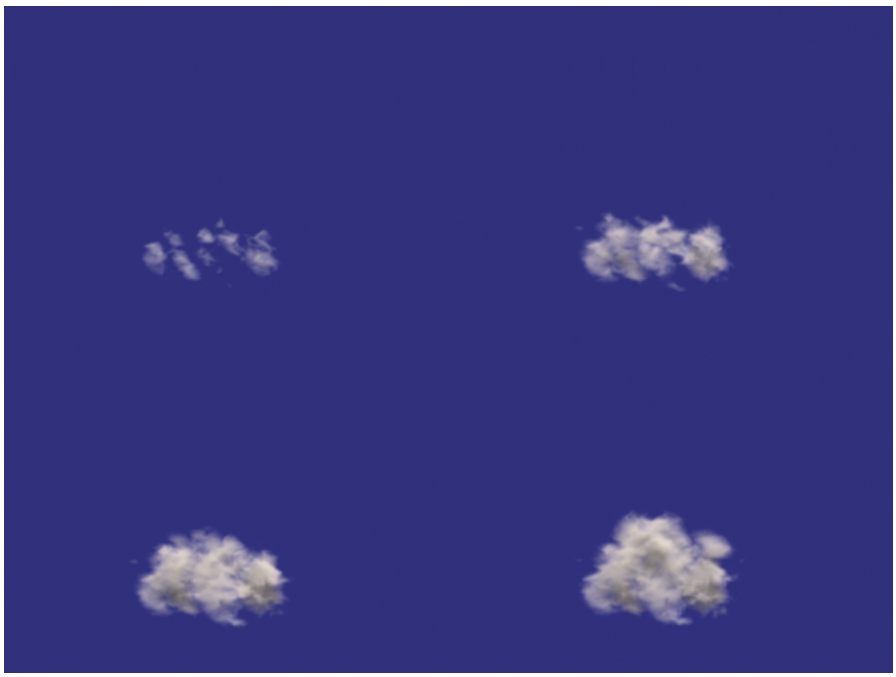


Figure 8: Formation of a cumulus cloud using a volumetric procedural implicit particle system. Copyright 1998 David S. Ebert

The animation and formation of cirrus and stratus clouds can also be controlled by the use of a volumetric procedural implicit particle system. For the formation of a large area of cirrus or cirrostratus clouds, the particle system can randomly seed space and then use turbulence to grow the clouds from the creation of new implicit primitives, as can be seen in Figure 9. The cirrostratus layer in this image contains 150 implicit primitives, which were generated from the user specifying 5 seed primitives.

To control the dynamics of the cloud particle system, any commercial particle animation program can also be used. A useful approach for cloud dynamics is to use *qualitative dynamics:* simple simulations of the observed properties and formation of clouds. The underlying physical forces that create a wide range of cloud formations are extremely complex to simulate, computationally expensive, and very restrictive. The incorporation of simple, parameterized rules that simulate observable cloud behavior will produce a powerful cloud animation system.

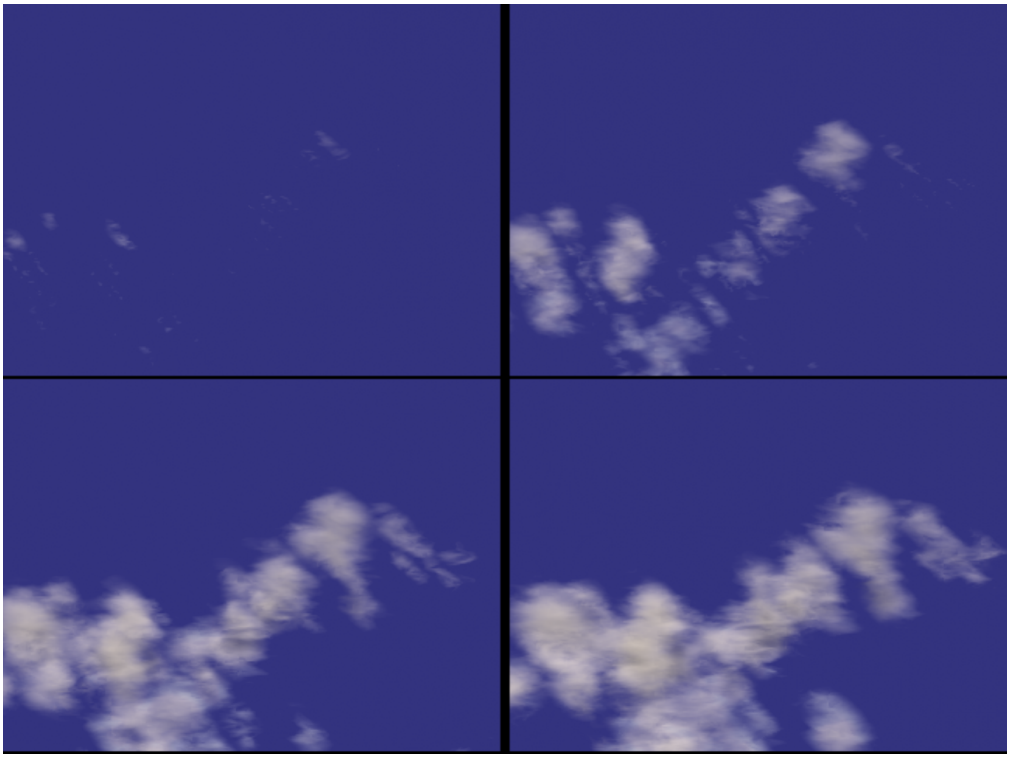


Figure 9: Cirrostratus cloud layer formation using a volumetric procedural implicit particle system. Copyright 1998 David S. Ebert

3 Interactivity and Clouds

3.1 Simple Interactive Cloud Models

There are several physical processes that govern the formation of clouds. Simple visual simulations of these techniques with particle systems can be used to get more convincing cloud formation animations. Neyret [Neyret97] suggested that the following physical processes are important to cloud formation simulations:

- Rayleigh Taylor instability: Rayleigh-Taylor instabilities result when a heavy fluid is supported by a less dense fluid against the force of gravity. Any perturbation along the

interface between the two fluids will grow.

- Bubbles: a small globule of gas in a thin liquid envelope.
- Rate variation of Temperature.
- Kelvin -Helmholtz instability: instability associated with airflows having marked vertical shear and weak thermal stratification. The common name for this instability is Kelvin-Helmholtz instability. These instabilities are often visualized as a row of horizontal eddies aligned within this layer of vertical shear.
- Vortices: A measure of the local rotation in a fluid flow. In weather analysis and forecasting, it usually refers to the vertical component of rotation.
- Bernard Cells: the hexagonal shaped convection eddies that can form in a solution that is being heated.

Neyret's model takes into account three phenomena at various scales: hot spot generation on the ground, simulation of bubbles rising bubble and reaching their dew point, bubble creation and evolution inside the cloud and their emergence as turrets on the borders. His proposed model incorporates:

- Bubble Generation - rising of hot air parcels due to buoyancy force caused by difference in density. Compute attraction force among parcels. Threshold of energy required for rising. Once it rises, fresh air takes its place and its probability to rise again decreases - emulates "Bernard Cell" behavior.
- Cloud evolution - the cloud is composed of static bubbles. The birth of a bubble inside the cloud is due to the local temperature gradient (Rayleigh Taylor instability).
- Direction of bubble depends on the local heat gradient.
- Small scale shape - it assumes a recursive structure for the small scale shape of the cloud. A bubble is considered as a sphere onto which are convected the main vortices, which were initially waves. The vortices are also assumed to be spherical and sub vortices are advected upon their parent vortex surface in a recursive fashion.

Results from an implementation of these techniques by Ruchigartha using MEL scripts in Maya

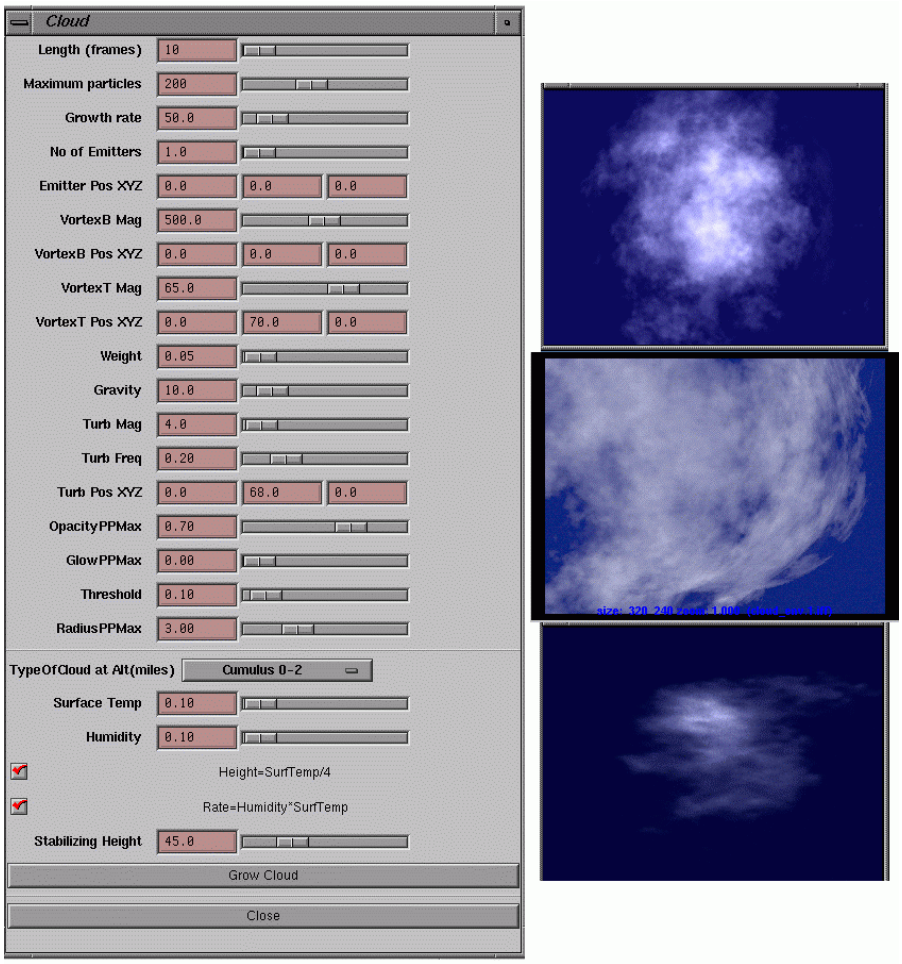


Figure 10: MEL GUI and sample images from Maya based cloud particle system generator.

can be found at http://www.cs.umbc.edu/~ebert/ruchi1 . An example of the output from her system and the GUI cloud control can be seen in Figure 10.

3.2 Rendering Clouds in Commercial Packages

The main component needed to effectively render clouds is volume rendering support in your renderer. Volumetric shadows, low- or high-albedo illumination and correct atmospheric attenuation are needed to get realistic looking clouds. An example of a volume rendering plug-in for Maya that can used to created volume rendered clouds can be found on the Maya Conductor

CD from 1999 and on the HighEnd3D web page http://www.highend3d.com/maya/plugins . The plug-in, volumeGas, by Marlin Rowley and Vlad Korolov, implements a simplified version of my volume renderer (which was used to produce the images in these notes).

3.3 Interactive Rendering and Interaction with Procedural Clouds on PC Hardware

With the advent of programmable hardware graphics pipelines, what are the important factors for generating true, interactive procedural models of natural phenomena? The following are several important factors that must be considered:

1. How much programmability is needed and available at the following levels:
   - Vertex
   - Fragment
2. Precision of the programmable operations:
   - Are these 8-bit quantities? 9-bit? 12-bit?
   - Are they fixed range (e.g, 0 to 1)?
   - What precision is needed to not produce artifacts?
   - Are they signed quantities (e.g., 8-bit plus sign bit)?
3. What mathematical operations are available?
   - Addition and Subtraction? (some new boards don't allow fragment subtraction)
   - Multiplication and Division?
   - More advanced operations (sqrt, sin, cos, etc.)?
4. Are dependent operations allowed (can the results of one operation change the next computation)?

All of the above factors greatly affect the type of procedural models that can be implemented at interactive rates. Current hardware finally allows some programmability of the GPU and we can

start to create interactive procedural models. However, the operations that are available at the fragment level are still limited and the limited precision is a serious limiting factor. The ability to compute a result at the fragment level and use this to change a texture coordinate used in the next texture look-up is a basic capability that is now available and enables basic procedural solid texturing, etc. Unfortunately, to create amazing, complex procedural models in real-time we need the ability to generate new geometry at the fragment level. We could then download to the GPU a small procedural model that creates geometry representing our procedural model in real-time. This is a feature that probably won't be available for several years. But, with clever programming, we can create some very interesting, advanced procedural effects with the programmability available in the latest graphics boards.

4 References

[Blinn1982] Blinn, J., "Light Reflection Functions for Simulation of Clouds and Dusty Surfaces," *Computer Graphics (SIGGRAPH '82 Proceedings),* 16(3)21:29, July 1982.

[Bloomenthal1997] Bloomenthal, J., Bajaj, C., Blinn, J., Cani-Gascuel, M.P., Rockwood, A., Wyvill, B., Wyvill, G., *Introduction to Implicit Surfaces,* Morgan Kaufman Publishers, 1997.

[Dungan1979] Dungan, W. Jr., "A Terrain and Cloud Computer Image Generation Model," *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19:41-44, July 1985.

[Dobashi2000] Yoshinori Dobashi and Kazufumi Kaneda and Hideo Yamashita and Tsuyoshi Okita and Tomoyuki Nishita. A Simple, Efficient Method for Realistic Animation of Clouds, *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 19-28 (July 2000). ACM Press / ACM SIGGRAPH / Addison Wesley Longman. Edited by Kurt Akeley. ISBN 1-58113-208-5.

[Ebert1989] Ebert, D., Boyer, K., Roble, D., "Once a Pawn a Foggy Knight…," [videotape], *SIGGRAPH Video Review,* 54. ACM SIGGRAPH, 1989.

[Ebert1990a] Ebert, D. and Parent, R., "Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques." *Computer Graphics (SIGGRAPH 90 Conference Proceedings)*, 24:357-366, August 1990.

[Ebert1990b] Ebert, D., Ebert, J., and Boyer, K., "Getting Into Art" [videotape], Department of Computer and Information Science, The Ohio State University, May1990.

[Ebert1991] Ebert, D., *Solid Spaces: A Unified Approach to Describing Object Attributes*. Ph.D. Thesis, The Ohio State University, 1991.

[Ebert1994] Ebert, D., Carlson, W., Parent, R., "Solid Spaces and Inverse Particle Systems for Controlling the Animation of Gases and Fluids," *The Visual Computer,* 10(4):179-190, 1994.

[Ebert1997a] Ebert, D., "Volumetric Modeling with Implicit Functions: A Cloud is Born," *SIGGRAPH 97 Visual Proceedings (Technical Sketch),* 147, ACM SIGGRAPH 1997.

[Ebert1997b] Ebert, D., Kukla, J., Bedwell, T., Wrights, S., "A Cloud is Born," *ACM SIGGRAPH Video Review (SIGGRAPH 97 Electronic Theatre Program),* ACM SIGGRAPH, August 1997.

[Ebert1998] Ebert, D., Musgrave, F., Peachey, D., Perlin, K., Worley, S., *Texturing and Modeling: A Procedural Approach, Second Edition,* AP Professional, July 1998.

[Foster1997] Foster, N., Metaxas, D., "Modeling the Motion of Hot Turbulent Gases," *SIGGRAPH 97 Conference Proceedings,* ACM SIGGRAPH 1997.

[Gardner1984] Gardner, G., "Simulation of Natural Scenes Using Textured Quadric Surfaces," *Computer Graphics (SIGGRAPH '84 Conference Proceedings),* 18, July 1984.

[Gardner1985] Gardner, G., "Visual Simulation of Clouds," *Computer Graphics (SIGGRAPH '85 Conference Proceedings),* 19(3), July 1985.

[Gardner1990] Gardner, G., "Forest Fire Simulation," *Computer Graphics (SIGGRAPH 90 Conference Proceedings),* 24, August 1990.

[Inakage1991] Inakage, M., "Modeling Laminar Flames," *SIGGRAPH 91 Course Notes 27,* ACM SIGGRAPH, July 1991.

[Kajiya1984] Kajiya, J., von Herzen, B., "Ray Tracing Volume Densities," *Computer Graphics (SIGGRAPH '84 Conference Proceedings),* 18, July 1984.

[Kajiya1989] Kajiya, J., Kay, T., "Rendering Fur with Three-dimensional Textures," *Computer Graphics (SIGGRAPH 89 Conference Proceedings),* 23, July 1989.

[Max1986] Max, N., "Light Diffusion Through Clouds and Haze," *Computer Vision, Graphics, and Image Processing,* 33(3), March 1986.

[Max1994] Max, N., "Efficient Light Propagation for Multiple Anisotropic Volume Scattering," *Fifth Eurographics Workshop on Rendering,* June 1994.

[Neyret1997] Neyret, F., "Qualitative Simulation of Convective Cloud Formation and Evolution," *Eight International Workshop on Computer Animation and Simulation,* Eurographics, September 1997.

[Nishimura1985] Nishimura, H., Hirai, A., Kawai, T., Kawata, T., Shirakawa, I., Omura, K., "Object Modeling by Distribution Function and a Method of Image Generation," *Journal of Papers Given at the Electronics Communication Conference '85,* J68-D(4), 1985.

[Nishita1987] Nishita, T., Miyawaki, Y., Nakamae, E., "A shading model for atmospheric scattering considering luminous intensity distribution of light sources*," Computer Graphics (SIGGRAPH '87Proceedings)*, 21, pages 303-310, July 1987.

[Nishita1996] Nishita, T., Nakamae, E., Dobashi, Y., "Display Of Clouds And Snow Taking Into Account Multiple Anisotropic Scattering And Sky Light," *SIGGRAPH 96 Conference Proceedings*, pages 379-386. ACM SIGGRAPH, August 1996.

[Peitgan1992] Peitgan, H., Saupe, D., eds., *Chaos and Fractals: New Frontiers of Science*, Springer-Verlag, New York, NY, 1992.

[Perlin1985] Perlin, K., "An Image Synthesizer," *Computer Graphics (SIGGRAPH '85 Conference Proceedings),* 19(3), July 1985.

[Perlin1989] Perlin, K., "Hypertextures", *Computer Graphics (SIGGRAPH 89 Conference Proceedings),* 23, July 1989.

[Pixar1989] Pixar, *The Renderman Interface: Version 3.1,* Pixar, San Rafael, Ca. 1989.

[Prusinkiewicz1990], Prusinkiewicz, P., Lindenmayer, A., *The Algorithmic Beauty of Plants,* Springer-Verlag, 1990.

[Smith1984] Smith, A., "Plants, Fractals, and Formal Languages," *Computer Graphics (SIGGRAPH '84 Conference Proceedings),* 18, July 1984.

[Stam1991] Stam, J., and Fiume, E., " A multiple-scale stochastic modeling primitive," *Proceedings Graphics Interface '91,* June 1991.

[Stam1993] Stam, J., and Fiume, E., " Turbulent wind fields for gaseous phenomena*," Computer Graphics (SIGGRAPH '93 Proceedings),* 27, pages 369-376, August 1993.

[Stam1995] Stam, J., and Fiume, E., " Depicting fire and other gaseous phenomena using diffusion processes," *SIGGRAPH 95 Conference Proceedings,* ACM SIGGRAPH 1995.

[Voss1983] Voss, R., "Fourier Synthesis of Gaussian fractals: 1/*f* noises, landscapes, and flakes," SIGGRAPH 83: Tutorial on State of the Art Image Synthesis, 10. ACM SIGGRAPH, 1983.

[Wyvill1986] Wyvill, G., McPheeters, C., Wyvill, B., "Data Structure for Soft Objects," *The Visual Computer,* 2:227-234, January 1986.

Procedural Volumetric Cloud Modeling, Animation, and Real-time Techniques

David S. Ebert
School of Electrical and Computer Engineering
Purdue University
ebertd@purdue.edu

Overview

- *Proceduralism*
- *Background*
- *Modeling Gases*

Overview

- *Cloud Modeling*
- *Examples Using Commercial Systems*
- *Hardware Issues and Real-Time Gases*
- *Conclusion*
- *Future Directions for Research*

Proceduralism: Advantages of Procedural Techniques

- *Flexibility*
- *Parametric Control*
- *Data Amplification*
- *Procedural Abstraction - High Level Control*
- *Complexity on Demand*
  - Inherently multi-resolution model
  - Computational savings
  - Ease of anti-aliasing

Background

- *Why Model Gases ?*
- *Important Visual Characteristics*
- *Rendering System Considerations*

Why Model Gases ?

- *Visual Realism*
- *Artistic Effects*

Important Visual Characteristics

- *Amorphous*
- *Swirling*
- *Attenuation of Light*
- *Shadowing*
- *Illumination*

Example: Fog

Rendering System Considerations: Issues / My System

Issues	My System
Volume Rendering Support	*Scanline A-buffer w/ Volume Tracing*
Illumination Issues • Participating media - scatters, reflects, absorbs light • Low-albedo models (single scattering) • High-albedo models (multiple scattering)	*Low-albedo Illumination Model*
Volume Shadowing	*3D Table-based Shadowing* • Fast, efficient • 10 to 15 times faster than ray-traced shadows
Modeling Capability	*Procedural Volume Density Functions*

Modeling Gases: Previous Approaches

Surface Approaches

- Hollow/flat objects
- Interaction problems
- Fast

Volume Approaches

- Greater realism, flexibility
- Slower

Volumetric Modeling Advantages

Accurate Shadowing

Realistic Illumination

Realistic Simulation of Natural Volumetric Phenomena (Clouds, Gases, Water, Fire)

Volumetric Procedural Modeling (VPM)

Basic VPM Primitives

- Any function of three-dimensions
- Stochastic:
  - *Noise, turbulence, fBm*
- Regular: implicit functions
  - *Smooth blending*
  - *Useful primitives (spheres, cylinders, ellipsoids, skeletons)*

Volumetric Procedural Gas Modeling

Turbulence-based Procedures

- Perlin's noise and turbulence functions

Shape Resulting Gas

- Simple mathematical functions

Defines Volume Density

Basic Gas Procedure

$$Density = (turbulence(pnt) * density_scaling)^{exponent}$$

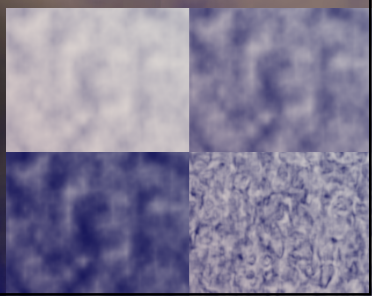
- Exponent typically 1.0 to 10.0

Gas Shaping Primitives

Power Function

Sine Function

Exponential Function



Steam Rising From a Teacup

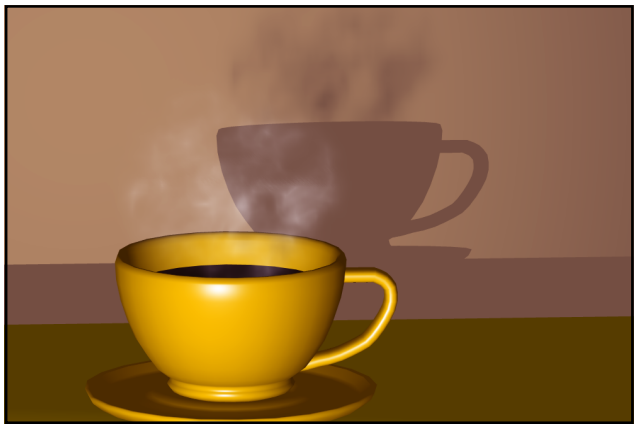
Volume of Gas Over the Teacup

Basic Gas Procedure Used for Density

Steam Rising ...

Shape Gas Spherically

Shape Gas Vertically



Volumetric Cloud Modeling: Volumetric Procedural Implicits

Previous Volumetric Procedural Implicit Modeling

- Perlin: hypertextures
- Stam: fire modeling, clouds
- Kisacikoglu: gas plasma - Sphere

Previous Cloud Modeling

- Surface-based (Gardner)
- Fractal-based (Voss)
- Volume-based (Kajiya, Stam)

Volumetric Procedural Implicit Modeling

Two Tiered Approach

- Cloud macrostructure
  - *Volumetrically rendered implicit primitives*
- Cloud microstructure
  - *Procedurally defined natural detail*
  - *Procedural volumetric density functions*

Cloud Macrostructure

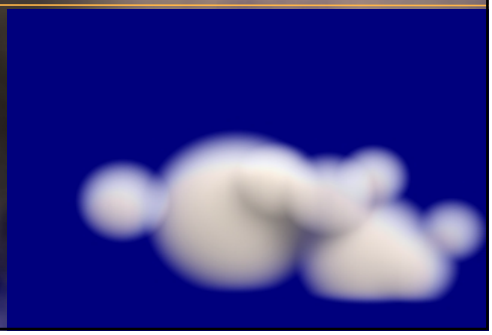
Primitive-Based Implicit Models

- Currently: spheres, cylinders, ellipsoids
- Wyvill's blending function

Ease of Specification, Animation, Global Deformation

- Easily controlled by particle system dynamics

Example Implicit Cloud



Cloud Microstructure

Volumetric Procedural Model

Built-in Multiresolution Model

Features:

- Main primitives: noise and turbulence
- Mathematical functions for shaping
- Natural controls

Simple Volumetric Procedural Model (VPM)

vpm(pnt)

- pnt = map pnt to procedural turbulence space
- turb = turbulence (pnt)
- density = pow(denseness*turb, wispiness)
- return(density)

Combined Model

Use Procedural Techniques to Perturb Sample Point

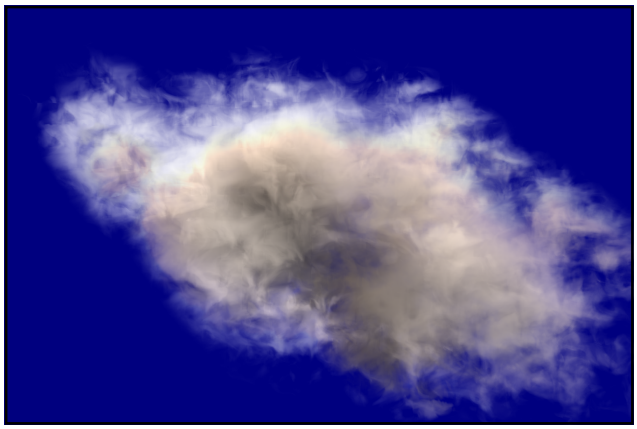
Calculate Implicit Density for Point

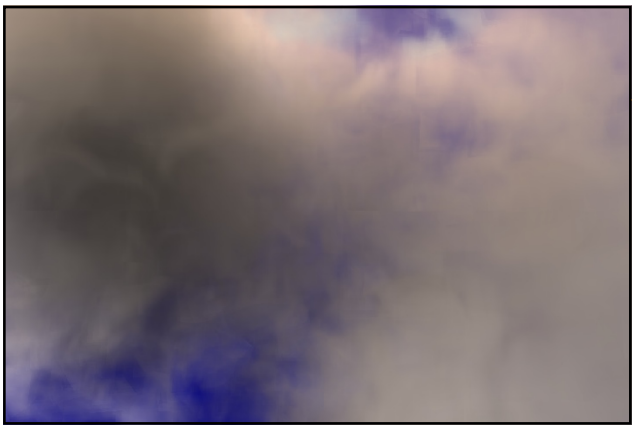
Calculate Procedural Density for Point

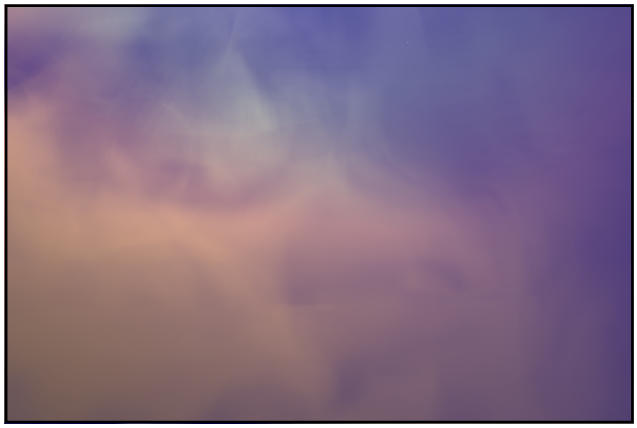
Blend These Densities

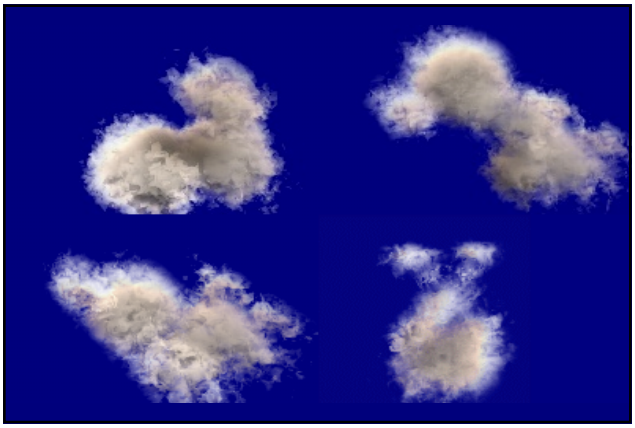
- blend = blend% * imp_density + (1-blend%)*proc_density*imp_density

Shape With Math Functions









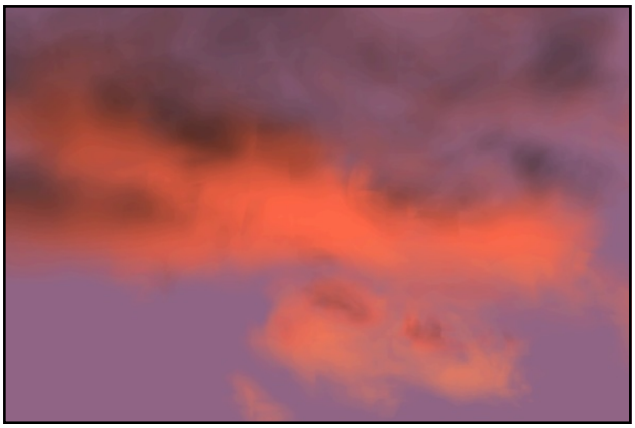
Stratus And Cirrus Cloud Effects

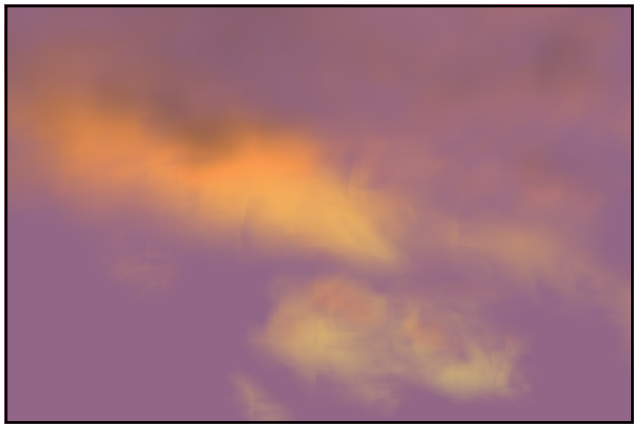
Stratus Clouds

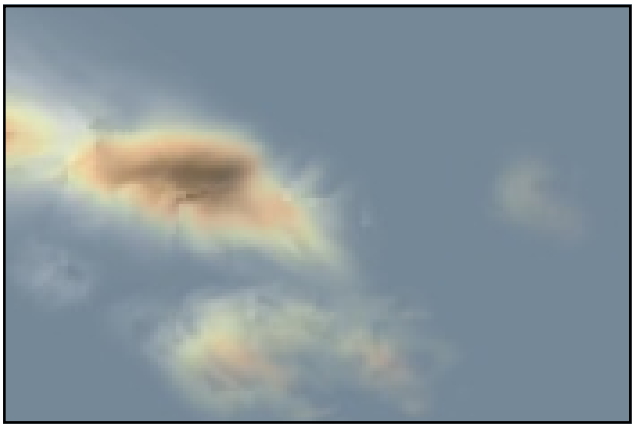
- Use a few implicits to specify extent of layer
- Use procedural techniques for details
- Denser and less wispy

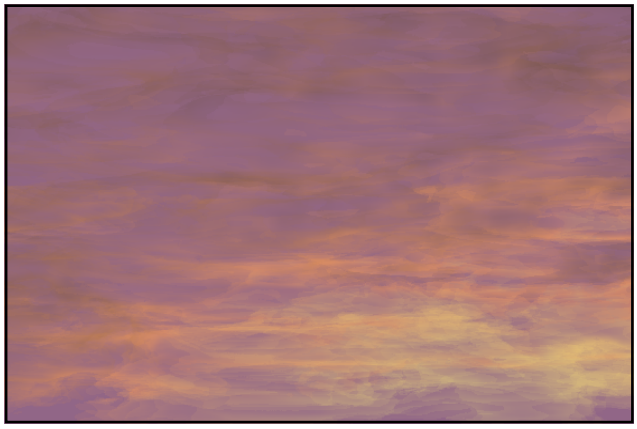
Cirrus Clouds

- Use implicits for each cloud or for global shape
- Thinner, less dense, wispier









Another Example (Henrik Wann Jensen)

Procedural Cloud Model Based on the Techniques Presented

- Generates a large number of points describing cloud density

Realistic Cloud and Environmental Illumination Using Photon Maps

Animation: Little Fluffy Clouds

- Cloud density is increased procedurally
- Sun rises, cloud layer forms, sun sets

Examples Using Commercial Systems: A/W Maya

Rendering:

- Volumetric cloud plug-in

Animation

- Cloud formation dynamics in MEL

Volumetric Cloud Plug-in
(Marlin Rowley, Vlad Korolev, David Ebert)

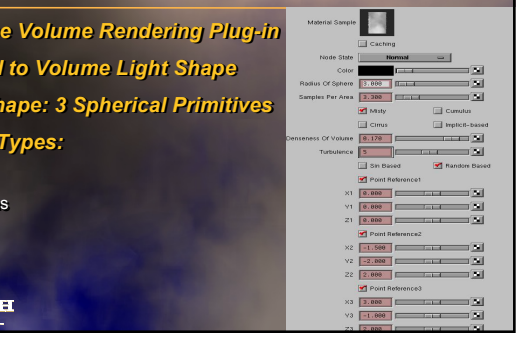
Prototype Volume Rendering Plug-in

Attached to Volume Light Shape

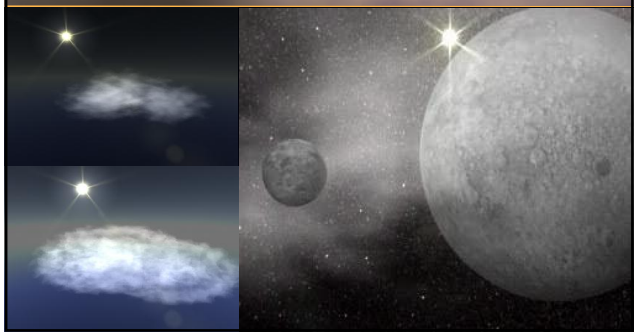
Cloud Shape: 3 Spherical Primitives

4 Cloud Types:

- Misty
- Cumulus
- Cirrus
- Implicit



Volumetric Cloud Plug-in: Examples



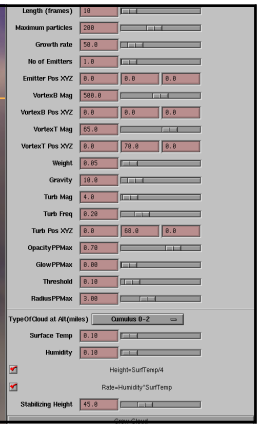
Plug-in Available

- High End 3D web site rendering (rendering section)

- www.highend3d

- v3 for NT released 5/31/2001

Cloud Dynamics in MEL (Ruchigartha)



Specialized Particle System

Dynamics Simulates

- Buoyant bubbles
- Temperature gradients - controls velocity
- Vortices
- Gravity
- Wind fields

Cloud Dynamics in MEL: Simulation

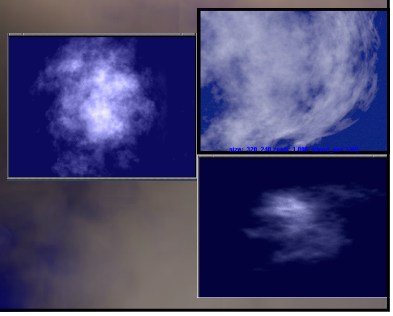
Particle Emitter

- Numerous settable attributes

Evaluate Forces on Particles

Create Children - Split Particles

Particle Death – Stabilize



Real-time Dense Gases: Issues

Volume Rendering vs. Approximations

Static vs. Dynamic Models

Semi-transparent Volume Accumulation

Illumination

Shadowing

Issues for Real-time Gases: Volume Approximations

Particle Systems – Only Practical for Thin Gases

- No inter-particle illumination, shadowing
- Often simple transparency model (or none) – depth sorted?
- Probabilistic shading and shadowing can be used

Imposters / Billboards – Good for Distant Clouds

- For close-ups and fly-throughs must integrate cloud slabs onto imposter
  - *Very time consuming – slows performance*
  - *Use pre-computed tables to improve performance*

Issues for Real-Time Gases: Volume Approximations (cont.)

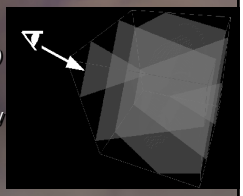
Textured Ellipsoids – Good for Distant Clouds

- Problem 1: need to handle view dependent illumination and shadowing
- Problem 2: fly-throughs
  - *must integrate cloud onto plane that slices through ellipsoid*
    - Need to update each frame
  - *Very time consuming – slows performance*
  - *Use pre-computed tables to improve performance*

Issues for Real-time Gases: Volume Rendering (Overview)

Hardware Approaches to Real-Time Volume Rendering

- Mitsubishi VolumePro board (>$5000)
- 3D texture mapping hardware
  - *Nvidia GeForce3, ATI Radeon (< $400)*
  - *SGI Octane, Onyx, … (>$10,000)*
- Limited resolution based on board memory
  - *256^3 (64Mb)?*



Interactive Software Solutions

- Splatting – Comes closest but is still seconds / frame

Issues for Volumetric Gases: Static Modeling

3D Textures for Gas Density

- Limited by resolution of 3D texture: 256^3 (64Mb)
  - *Not a very detailed cloud, want 1000^3 at least*
  - *What about shadow volume, illumination volumes, etc. => even more memory*
- Precision of densities / opacities: Is 8 bits enough?

Global Density Model + Volume Detail Texture (Noise Texture)

- Need dependent texture reads

Issues for Real-time Volumetric Gases: Dynamic Models

Dynamically Change 3D Texture Densities

- Need ability to update portions of 3D textures at 30 fps

Change 3D Texture Indices Algorithmically

- How quick can you change the texture coordinates on the slices?

Use a Changing Smaller Texture to Dynamically Offset the 3D Texture Lookup

Could Generate Geometry on the Fly (Micropolygons)

- Need capability to generate new triangles at the vertex or fragment processing level
  - *E.g. from a vertex program on a Nvidia chip*
  - *Can use dummy geometry – but no textures in v.p.*

Issues for Real-time Volumetric Gases: Opacity Accumulation

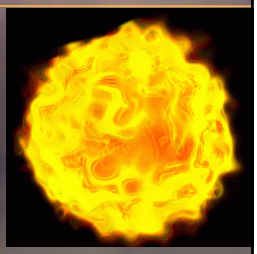
Need Exponential Accumulation of Gas Densities:

$$e^{-\tau \int_{t1}^{t2} density\ (p)\, dp}$$

- Most systems use simple linear blend

Can Pre-integrate Accumulated Opacity Within a Slab and Store That in the Texture (e.g., Engel 2001)

- Opacities at front and back plus step size become texture coordinates
- Requires dependent texture read



Courtesy of Klaus Engel, Pre-Integrated Volume Renderer V1.7, 15 fps, 2001

Issues for Real-time Volumetric Gases: Illumination

How to Simulate Bi-directional Reflection Function for Low-albedo Illumination

- 2D texture maps indexed by eye angle and light angle?
  - *Needs dependent texture read*

How to Simulate Multiple (High-albedo) Scattering?

- Could use pre-integrated tables
  - *Need to change for each move in observer position or light position*

Approximation of Isotropic Particle Scattering

- Only dependent on light direction

Issues for Real-time Volumetric Gases: Shadowing

How to Compute Real-time Shadows?

- 2D real-time shadow mapping
  - *Only would works for shadowing onto objects, not self-shadowing*
  - *Problem with transparent objects*
- Could create 3D shadow table using texture sliced renderer from direction of eye point
  - *Cuts frame rate approximately 25-50% depending on accuracy desired*
- Projected imposters to form shadow texture (Dobashi 2000)

What's Now Available for PC Graphics?

3D Textures - (e.g., ATI, 3dfx, Nvidia, X-box)

Programmable Vertex Shading (e.g., GeForce2, GeForce3)

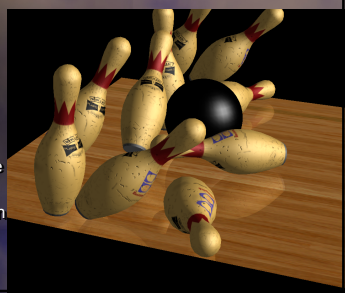
Dependent Texture Reads (e.g., ATI Radeon, GeForce3)

Programmable Pixel Shading (e.g., GeForce3)

What's Now Available for PC Graphics?

***Stanford Real-Time Programmable Shading Language** (Mark, Proudfoot, Hanrahan)*

- Great for real-time programmable shader development and volume shading design
- Re-targetable compiler to optimize passes through graphics pipeline
- Between OpenGL and Renderman



Hardware Issues With New Advances

How Much Flexibility in the New Programmability?

- Can you add, subtract, multiply, divide?
- Are conditionals allowed?
- How big is the temporary storage?
  - *Can you do noise tables?*
- Can you use 3D textures just like 2D textures in dependent reads?
- Any order of operations imposed by the hardware (implementation gotcha)?
- What operations are allowed in each part of the pipeline?

Hardware Issues With New Advances (cont.)

What Is the Range of the Values for Each Operation?

- 0 to 255, -255 to 255, fixed point, float

What Is the Precision?

- 8-bit, 9-bit, 12 bit, 16 bit?
  - *Affects complexity of operations that can be performed before quantization errors are visible*
- How does the precision vary at different stages of pipeline?
  - *E.g., Geforce 3 pixel shaders are floating point, but textures are 8-bit and combiners are 9-bit*

Conclusion

Procedural Modeling and Animation is :

Powerful

Flexible

Extensible

Conclusion

Important Aspects

- Flexible volume modeling system
- Accurate illumination and shadowing

Procedural Modeling

- Particle systems, L-systems, blobs can be included
- Flexible, turbulent volume modeling

Conclusion

Volumetric Procedural Implicit Cloud Modeling

- Ease of control and specification of implicits
- Smooth blending
- Natural appearance from turbulence simulation
- Procedural abstraction
- Parametric control

Conclusion

Real-time Gases Are On the Horizon

- Latest programmability and capabilities of PC hardware enables a vast array of techniques
- Procedural techniques are well suited for new hardware
  - *Eliminate the data transfer bottleneck*

Future Goal

- Download procedural cloud to GPU and generate geometry and render on the fly

SIGGRAPH 2002

Acknowledgements

Collaborators:

- RTSL: Bill Mark, Kekoa Proudfoot, Pat Hanrahan
- Klaus Engel, Rick Parent, Steve May
- Students: Marlin Rowley, Vlad Korolev, Ruchigartha

Funding: NSF, NASA, DoD, Electronic Arts

SIGGRAPH 2002

L-systems: from the Theory to Visual Models of Plants

Przemyslaw Prusinkiewicz[1], Jim Hanan[2], Mark Hammel[1] and Radomir Mech[1]

[1] Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4
[2] CSIRO — Cooperative Research Centre for Tropical Pest Management, Brisbane, Queensland, Australia

1 Introduction

In 1968, Aristid Lindenmayer introduced a formalism for simulating the development of multicellular organisms, subsequently named L-systems [18]. This formalism was closely related to abstract automata and formal languages, and attracted the immediate interest of theoretical computer scientists. The vigorous development of the mathematical theory of L-systems was followed by its applications to the modeling of plants. These applications gained momentum after 1984, when Smith introduced state-of-the art computer graphics techniques to visualize the structures and processes being modeled [52]. Smith also attracted attention to the phenomenon of *data-base amplification*, or the possibility of generating complex structures from compact data sets, which is inherent in L-systems and forms the cornerstone of L-system applications to image synthesis. Subsequent developments (presented here from our personal perspective, without covering the fast-growing array of contributions from many other researchers) included:

- introduction of turtle interpretation of L-systems [32, 53] and refinement of a programming language based on L-systems [11, 43], which facilitated specification of the models for simulation purposes and promoted the use of L-systems as a language for describing models in publications;
- recognition of the fractal character of structures generated by L-systems, which related them to the dynamically developing science of fractals [32, 43, 38];
- increased interest in the application of computer simulations to the understanding of living processes and structures, related to the emergence of the field of Artificial Life;
- extension of the range of phenomena that can be modeled using L-systems, including, most recently, incorporation of environmental factors into the models [30, 41];
- increased understanding of the modeling process, providing a methodology for constructing models according to biological observations and measurements [45, 48].

In this paper, we revisit basic mechanisms that control plant development: lineage (cellular descent), captured by the class of context free L-systems, and endogenous interaction (transfer of information between neighboring modules in the structure), captured by context-sensitive L-systems (*c.f.* [22]). Within this framework, we present several models that have been developed after the survey of L-systems in [43].

The original version of this apper appeared in M. T. Michalewicz (Ed.): *Plants to Ecosystems. Advances in Computational Life Sciences*, CSIRO, Collingwood, Australia 1997, pp. 1–27.

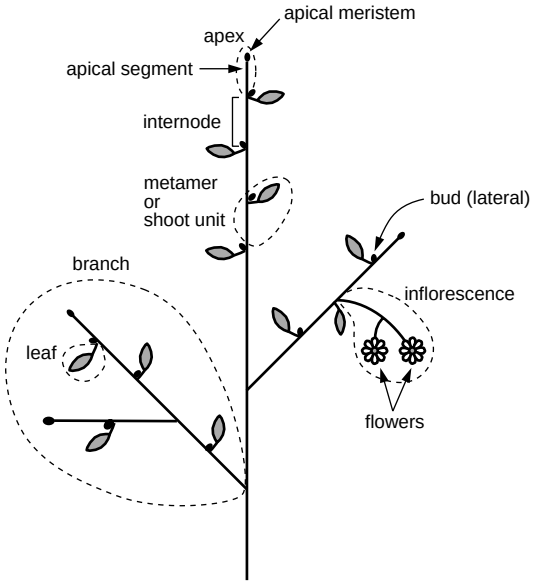


Figure 1: Selected modules and groups of modules (encircled with dashed lines) used to describe plants

2 The modular structure of plants

L-systems were originally introduced to model the development of simple *multicellular* organisms (for example, algae) in terms of division, growth, and death of individual cells [18, 19]. The range of L-system applications has subsequently been extended to higher plants and complex branching structures, in particular inflorescences [8, 9], described as configurations of modules in space. In the context of L-systems, the term *module* denotes any discrete constructional unit that is repeated as the plant develops, for example an internode, an apex, a flower, or a branch (Figure 1) [2, 12, 55]. The goal of modeling at the modular level is to describe the development of a plant as a whole, and in particular the emergence of plant shape, as the integration of the development of individual units.

3 Plant development as a rewriting process

The essence of development at the modular level can be conveniently captured by a parallel *rewriting system* that replaces individual *parent*, *mother*, or *ancestor* modules by configurations of *child*, *daughter*, or *descendant* modules. All modules belong to a finite *alphabet of module types*, thus the behavior of an arbitrarily large configuration of modules can be specified using a finite set of *rewriting rules* or *productions*. In the simplest case of *context-free*

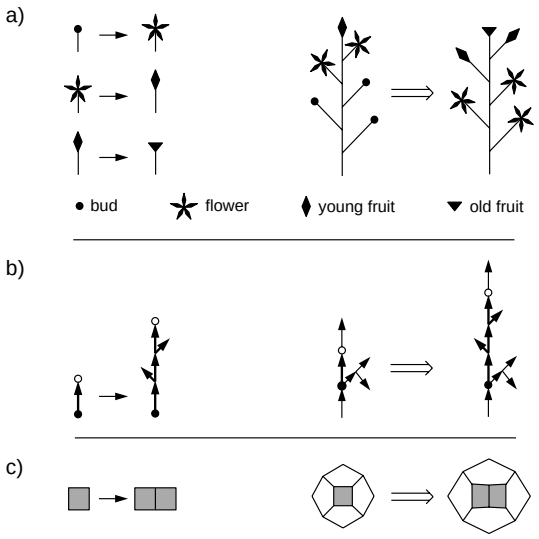


Figure 2: Examples of production specification and application: (a) development of a flower, (b) development of a branch, and (c) cell division.

rewriting, a production consists of a single module called the *predecessor* or the *left-hand side*, and a configuration of zero, one, or more modules called the *successor* or the *right-hand side*. A production p with the predecessor matching a given mother module can be applied by deleting this module from the rewritten structure and inserting the daughter modules specified by the production's successor.

Three examples of production application are shown in Figure 2. In case (a), modules located at the extremities of a branching structure are replaced without affecting the remainder of the structure. In case (b), productions that replace internodes divide the branching structure into a lower part (below the internode) and an upper part. The position of the upper part is adjusted to accommodate the insertion of the successor modules, but the shape and size of both the lower and upper part are not changed. Finally, in case (c), the rewritten structures are represented by graphs with cycles. The size and shape of the production successor does not exactly match the size and shape of the predecessor, and the geometry of the predecessor and the embedding structure had to be adjusted to accommodate the successor. The last case is most complex, since the application of a local rewriting rule may lead to a global change of the structure's geometry. Developmental models of cellular layers operating in this manner have been presented in [43, 4, 5, 7]. In this paper we focus on the rewriting of branching structures corresponding to cases (a) and (b).

Productions may be applied *sequentially*, to one module at a time, or they may be applied *in parallel*, with all modules being rewritten simultaneously in every *derivation step*. Parallel rewriting is more appropriate for the modeling of biological development, since development takes place simultaneously in all parts of an organism. A derivation step then corresponds to the progress of time over some interval. A sequence of structures obtained in consecutive derivation steps from a predefined *initial structure* or *axiom* is called a *developmental sequence*. It can be viewed as the result of a *discrete-time simulation* of development.

For example, Figure 3 illustrates the development of a stylized compound leaf including two module types, the *apices* (represented by

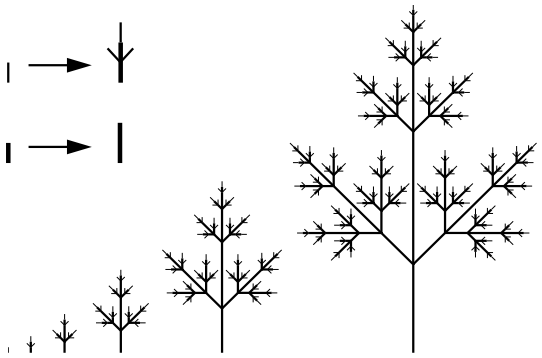


Figure 3: Developmental model of a compound leaf, modeled as a configuration of apices and internodes

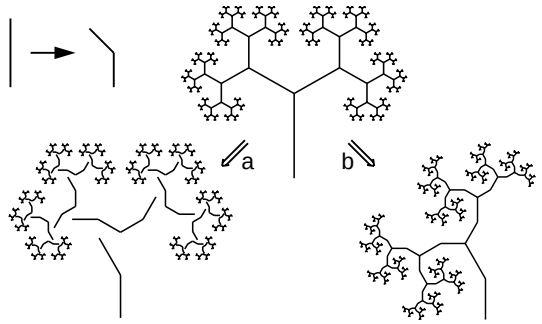


Figure 4: A comparison of the Koch construction (a) with a rewriting system preserving the branching topology of the modeled structures (b). The same production is applied in both cases, but the rules for incorporating the successor into the structure are different.

thin lines) and the *internodes* (thick lines). An apex yields a structure that consists of two internodes, two lateral apices, and a replica of the main apex. An internode elongates by a constant scaling factor. In spite of the simplicity of these rules, an intricate branching structure develops from a single apex over a number of derivation steps.

It is interesting to contrast simulation of development using rewriting rules with the well known Koch construction for generating fractals [29, page 39]. The essence of the Koch construction is the replacement of straight line segments by sets of lines. Their positions, orientations, and scales are determined by the position, orientation, and scale of the segment being replaced (Figure 4a). In contrast, in models of plants, the position and orientation of each module is determined by the chain of modules beginning at the base of the structure and extending to the module under consideration. For example, when the internodes bend, the subtended branches are rotated and displaced to maintain the connectivity of the structure (Figure 4b). Thus, development is simulated as a parallel application of productions, followed by a sequential connection of the child structures.

Rewriting processes maintaining the connectivity of branching structures can be defined directly in the geometric domain, but a more convenient approach is to express the generating rules and the resulting structures symbolically, using a string notation. A sequential geometric interpretation of these strings from the left (plant base) to right (branch extremities) automatically captures proper positioning of the higher branches on the lower ones. The rewriting

of branching structures in the string domain is the cornerstone of L-systems.

The basic notions of the theory of L-systems have been presented in many survey papers [22, 20, 21, 24, 25, 26] and books [43, 38, 13, 50, 51]. Consequently, we only describe parametric L-systems, which are a particularly convenient programming tool for expressing models of plant development. Our presentation closely follows the formalization introduced in [43, 39] (see also [11, 40]).

4 Parametric L-systems

Parametric L-systems operate on *parametric words*, which are strings of *modules* consisting of *letters* with associated *parameters*. The letters belong to an *alphabet* V, and the parameters belong to the set of *real numbers* $\Re$. A module with letter $A \in V$ and parameters $a_1, a_2, ..., a_n \in \Re$ is denoted by $A(a_1, a_2, ..., a_n)$. Every module belongs to the set $M = V \times \Re^*$, where $\Re^*$ is the set of all finite sequences of parameters. The set of all strings of modules and the set of all nonempty strings are denoted by $M^* = (V \times \Re^*)^*$ and $M^+ = (V \times \Re^*)^+$, respectively.

The real-valued *actual* parameters appearing in the words have a counterpart in the *formal* parameters, which may occur in the specification of L-system productions. If Σ is a set of formal parameters, then $C(\Sigma)$ denotes a *logical expression* with parameters from Σ, and $E(\Sigma)$ is an *arithmetic expression* with parameters from the same set. Both types of expressions consist of formal parameters and numeric constants, combined using the arithmetic operators $+$, $-$, $*$, $/$; the exponentiation operator $\wedge$, the relational operators $<$, $<=$, $>$, $>=$, $==$; the logical operators !, &&, || (not, and, or); and parentheses $()$. The expressions can also include calls to standard mathematical functions, such a natural logarithm, sine, floor, and functions returning random variables. The operation symbols and the rules for constructing syntactically correct expressions are the same as in the C programming language [17]. For clarity of presentation, however, we sometimes use Greek letters and symbols with subscripts in print. Relational and logical expressions evaluate to zero for false and one for true. A logical statement specified as the empty string is assumed to have value one. The sets of all correctly constructed logical and arithmetic expressions with parameters from Σ are noted $\mathcal{C}(\Sigma)$ and $\mathcal{E}(\Sigma)$.

A *parametric 0L-system* is defined as an ordered quadruple $G = \langle V, \Sigma, \omega, P \rangle$, where:

- V is the *alphabet* of the system,
- Σ is the *set of formal parameters*,
- $\omega \in (V \times \Re^*)^+$ is a nonempty parametric word called the *axiom*,
- $P \subset (V \times \Sigma^*) \times \mathcal{C}(\Sigma) \times (V \times \mathcal{E}(\Sigma)^*)^*$ is a finite *set of productions*.

The symbols : and $\rightarrow$ are used to separate the three components of a production: the *predecessor*, the *condition*, and the *successor*. Thus, a production has the format

$$pred : cond \rightarrow succ.$$

For example, a production with predecessor $A(t)$, condition $t > 5$ and successor $B(t+1)CD(t \wedge 0.5, t-2)$ is written as

$$A(t) : t > 5 \rightarrow B(t+1)CD(t \wedge 0.5, t-2). \qquad (1)$$

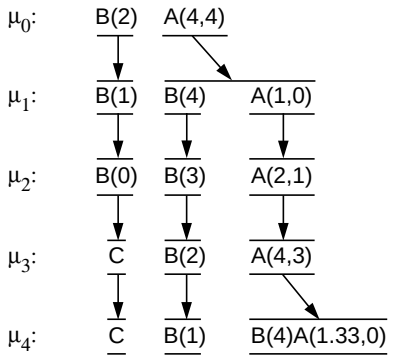


Figure 5: The initial sequence of strings generated by the parametric L-system specified in equation (2)

A production in a 0L-system *matches* a module in a parametric word if the following conditions are met:

- the letter in the module and the letter in the production predecessor are the same,
- the number of actual parameters in the module is equal to the number of formal parameters in the production predecessor, and
- the condition evaluates to *true* if the actual parameter values are substituted for the formal parameters in the production.

A matching production can be *applied* to the module, creating a string of modules specified by the production successor. The actual parameter values are substituted for the formal parameters according to their position. For example, production (1) above matches a module $A(9)$, since the letter A in the module is the same as in the production predecessor, there is one actual parameter in the module $A(9)$ and one formal parameter in the predecessor $A(t)$, and the logical expression $t > 5$ is true for t equal to 9. The result of the application of this production is a parametric word $B(10)CD(3, 7)$.

If a module a produces a parametric word χ as the result of a production application in an L-system G, we write $a \mapsto \chi$. Given a parametric word $\mu = a_1 a_2 ... a_m$, we say that the word $\nu = \chi_1 \chi_2 ... \chi_m$ is *directly derived* from (or *generated* by) μ and write $\mu \Longrightarrow \nu$ if and only if $a_i \mapsto \chi_i$ for all $i = 1, 2, ..., m$. A parametric word ν is generated by G in a *derivation of length* n if there exists a sequence of words $\mu_0, \mu_1, ..., \mu_n$ such that $\mu_0 = \omega$, $\mu_n = \nu$ and $\mu_0 \Longrightarrow \mu_1 \Longrightarrow ... \Longrightarrow \mu_n$.

An example of a parametric L-system is given below.

$$\begin{array}{lllll} \omega: & B(2)A(4,4) & & & \\ p_1: & A(x,y) & : y <= 3 & \rightarrow & A(x*2, x+y) \\ p_2: & A(x,y) & : y > 3 & \rightarrow & B(x)A(x/y, 0) \\ p_3: & B(x) & : x < 1 & \rightarrow & C \\ p_4: & B(x) & : x >= 1 & \rightarrow & B(x-1) \end{array} \qquad (2)$$

It is assumed that a module replaces itself if no matching production is found in the set P. The words obtained in the first few derivation steps are shown in Figure 5.

Productions in parametric 0L-systems are *context-free*, *i.e.*, applicable regardless of the context in which the predecessor appears. A *context-sensitive* extension is necessary to model information exchange between neighboring modules. In general, a context-sensitive production has the format

$$lc < pred > rc \quad : cond \rightarrow succ,$$

where symbols $<$ and $>$ separate the three components of the predecessor: a string of modules without brackets lc called the *left context*, a module $pred$ called the *strict predecessor*, and a well-nested bracketed string of modules rc called the *right context*. The remaining components of the production are the condition $cond$ and the successor $succ$, defined as for parametric 0L-systems.

A sample context-sensitive production is given below:

$$A(x) < B(y) > C(z) : x + y + z > 10 \rightarrow E((x+y)/2)F((y+z)/2). \quad (3)$$

The left context is separated from the strict predecessor by the symbol $<$. Similarly, the strict predecessor is separated from the right context by the symbol $>$. Production 3 can be applied to the module $B(5)$ that appears in a parametric word

$$\cdots A(4)B(5)C(6)\cdots \quad (4)$$

since the sequence of letters A, B, C in the production and in parametric word (4) are the same, the numbers of formal parameters and actual parameters coincide, and the condition $4 + 5 + 6 > 10$ is true. As a result of the production application, the module $B(5)$ will be replaced by a pair of modules $E(4.5)F(5.5)$. Naturally, the modules $A(4)$ and $C(6)$ will be replaced by other productions in the same derivation step.

Productions in 2L-systems use context on both sides of the strict predecessor. 1L-systems are a special case of 2L-systems in which context appears only on one side of the productions.

When no production explicitly listed as a member of the production set P matches a module in the rewritten string, we assume that an appropriate *identity production* belongs to P and replaces this module by itself. Under this assumption, a parametric L-system $G = \langle V, \Sigma, \omega, P \rangle$ is called *deterministic* if and only if for each module $A(t_1, t_2, \ldots, t_n) \in V \times \Re^*$ the production set includes exactly one matching production. Within this paper we only consider deterministic L-systems.

5 The turtle interpretation of L-systems

Strings generated by L-systems may be interpreted geometrically in many different ways. Below we outline the *turtle interpretation* of L-systems, introduced by Szilard and Quinton [53], and extended by Prusinkiewicz [32, 33] and Hanan [11, 10]. A tutorial exposition is included in [43], and subsequent results are presented in [11]. The summary below is based on [43, 39, 33, 15].

After a string has been generated by an L-system, it is scanned sequentially from left to right, and the consecutive symbols are interpreted as commands that maneuver a LOGO-style turtle [1, 31] in three dimensions. The turtle is represented by its *state*, which consists of turtle *position* and *orientation* in the Cartesian coordinate system, as well as various attribute values, such as current *color* and *line width*. The position is defined by a vector $\vec{P}$, and the orientation is defined by three vectors $\vec{H}$, $\vec{L}$, and $\vec{U}$, indicating the turtle's *heading* and the directions to the *left* and *up* (Figure 6a). These vectors have unit length, are perpendicular to each other, and satisfy the equation $\vec{H} \times \vec{L} = \vec{U}$. Rotations of the turtle are expressed by the equation:

$$\left[\begin{array}{ccc} \vec{H}' & \vec{L}' & \vec{U}' \end{array}\right] = \left[\begin{array}{ccc} \vec{H} & \vec{L} & \vec{U} \end{array}\right] \mathbf{R},$$

where $\mathbf{R}$ is a 3×3 rotation matrix [6]. Changes in the turtle's state are caused by interpretation of specific symbols, each of which may be followed by parameters. If one or more parameters are present,

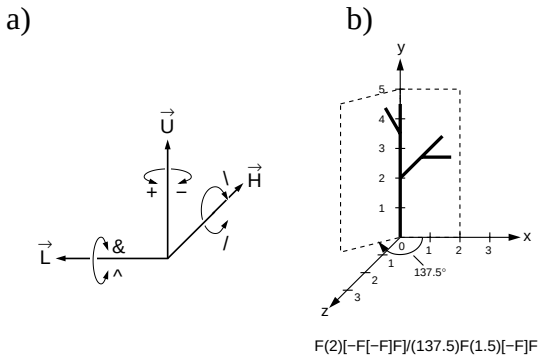


Figure 6: a) Controlling the turtle in three dimensions. b) Example of the turtle interpretation of a string.

the value of the first parameter affects the turtle's state. If the symbol is not followed by any parameter, default values specified outside the L-system are used. The following list specifies the basic set of symbols interpreted by the turtle.

Symbols that cause the turtle to move and draw

$F(s), G(s)$ Move forward a step of length s and draw a line segment from the original to the new position of the turtle.

$f(s), g(s)$ Move forward a step of length s without drawing a line.

$@O(r)$ Draw a sphere of radius r at the current position.

Symbols that control turtle orientation in space (Figure 6a)

$+(\theta)$ Turn left by angle θ around the $\vec{U}$ axis.

$-(\theta)$ Turn right by angle θ around the $\vec{U}$ axis.

$\&(\theta)$ Pitch down by angle θ around the $\vec{L}$ axis.

$\wedge(\theta)$ Pitch up by angle θ around the $\vec{L}$ axis.

$/(\theta)$ Roll left by angle θ around the $\vec{H}$ axis.

$\backslash(\theta)$ Roll right by angle θ around the $\vec{H}$ axis.

$|$ Turn $180°$ around the $\vec{U}$ axis. This is equivalent to $+(180)$ or $-(180)$.

Symbols for modeling structures with branches

$[$ Push the current state of the turtle (position, orientation and drawing attributes) onto a pushdown stack.

$]$ Pop a state from the stack and make it the current state of the turtle. No line is drawn, although in general the position and orientation of the turtle are changed.

Symbols for creating and incorporating surfaces

$\{$ Start saving the subsequent positions of the turtle as the vertices of a polygon to be filled.

$\}$ Fill the saved polygon.

$\sim X(s)$ Draw the surface identified by symbol X, scaled by s, at the turtle's current location and orientation. Such a surface is usually defined as a bicubic patch [33, 10].

Symbols that change the drawing attributes

$\#(w)$ Set line width to w, or increase the value of the current line width by the default width increment if no parameter is given.

$!(w)$ Set line width to w, or decrease the value of the current line width by the default width decrement if no parameter is given.

$;(n)$ Set the index of the color map to n, or increase the value of the current index by the default colour increment if no parameter is given.

$,(n)$ Set the index of the color map to n, or decrease the value of the current index by the default colour decrement if no parameter is given.

A sample string and its interpretation are shown in Figure 6b. The default length of lines represented by symbols F without a parameter is 1, and the default magnitude of the angles represented by symbols $+$ and $-$ is $45°$.

6 Examples of parametric D0L-system models

This section presents selected examples that illustrate the operation of deterministic 0L-systems (D0L-systems) with turtle interpretation and their application to the modeling of plants. Many other examples are included in [11, 43, 39].

6.1 Fractal generation

Fractal curves provide a convenient means for illustrating the basic principle of L-system operation [32, 43, 38, 44]. For example, the following L-system generates the well-known snowflake curve [29, 54].

$$\begin{array}{lll} \omega: & F(1)-(120)F(1)-(120)F(1) & \\ p_1: & F(s) \rightarrow F(s/3)+(60)F(s/3) & \\ & \quad -(120)F(s/3)+(60)F(s/3) & \end{array}$$

The axiom $F(1)-(120)F(1)-(120)F(1)$ draws an equilateral triangle, with edges of unit length. Production p_1 replaces each line segment with a polygonal shape, as shown at the top of Figure 7. Productions for symbols $+$ and $-$ are not listed, which means that the corresponding modules will be replaced by themselves during the derivation. The same effect could have been obtained by explicit inclusion of productions:

$$\begin{array}{ll} p_2: & +(a) \rightarrow +(a) \\ p_3: & -(a) \rightarrow -(a) \end{array}$$

The axiom and the figures obtained in the first three derivation steps are shown at the bottom of Figure 7.

6.2 Simulation of development

The next L-system generates the developmental sequence of the stylized compound leaf model presented in Figure 3.

$$\begin{array}{lllll} \omega: & !(1)F(1,1) & & & \\ p_1: & F(s) & \rightarrow & G(s)[-!(1)F(s)][+!(1)F(s)]G(s)!(1)F(s) \\ p_2: & G(s) & \rightarrow & G(2*s) \\ p_3: & !(w) & \rightarrow & !(3) \end{array}$$

The structure is built from two module types, apices F (represented by thin lines) and internodes G (thick lines). In both cases the parameter s determines the length of the line representing the module. An apex yields a structure that consists of two internodes, two lateral apices, and a replica of the main apex (production p_1). An internode elongates by a constant scaling factor (production p_2). Production p_3 is used to make the lines representing the internodes wider (3 units of width) than the lines representing the apices (1 unit). The branching angle associated with symbols $+$ and $-$ is set to $45°$ by a global variable outside the L-system.

6.3 Exploration of parameter space

Parametric L-systems provide a convenient mathematical framework for exploring the range of forms that can be captured by the same structural model with varying attributes (constants in the productions). Such *parameter space explorations* motivated some of the earliest computer simulations of biological structures: the models of sea shells devised by Raup and Michelson [46, 47] and the models of trees proposed by Honda [14] to study factors that determine overall tree shape. Parameter space exploration may reveal an unexpected richness of forms that can be produced by even the simplest models. For example, Figure 8 shows nine branching structures selected from a continuum generated by the following parametric D0L-system:

$$\begin{array}{lllll} \omega: & A(100, w_0) & & & \\ p_1: & A(s,w) & : s >= \mathit{min} & \rightarrow & !(w)F(s) \\ & & & & [+(\alpha_1)/(\varphi_1)A(s*r_1, w*q \wedge e)] \\ & & & & [+(\alpha_2)/(\varphi_2)A(s*r_2, w*(1-q) \wedge e)] \end{array}$$

The single non-identity production p_1 replaces apex A by an internode F and two new apices A. The angle values α_1, α_2, φ_1, and φ_2 determine the orientation of these apices with respect to the subtending internode. Parameters s and w specify internode length and

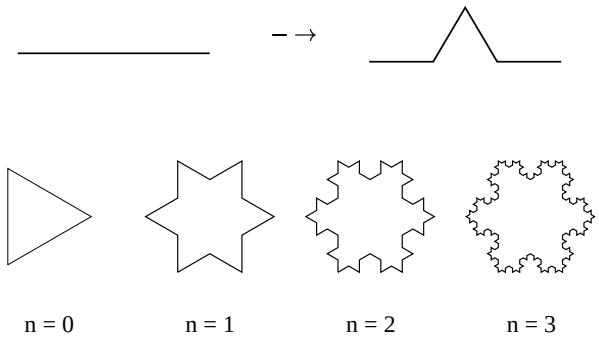


Figure 7: Visual interpretation of the production for the snowflake curve, and the curve after n = 0, 1, 2, and 3 derivation steps

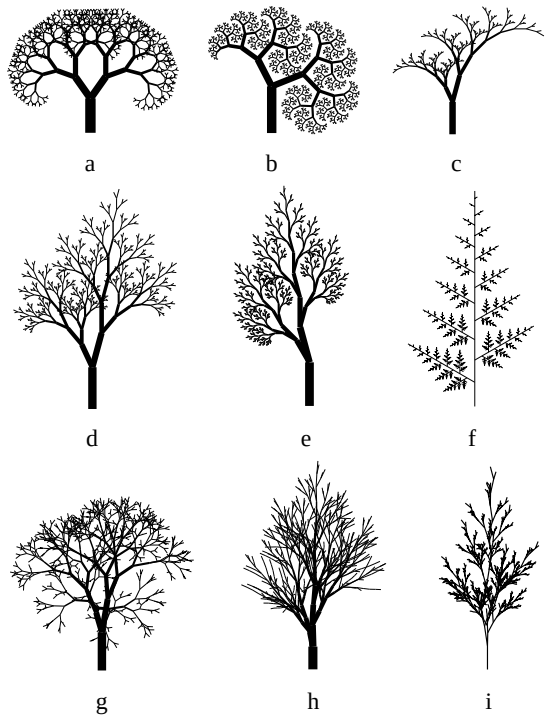


Figure 8: Sample structures generated by a parametric D0L-system with different values of constants

Table 1: The values of constants used to generate Figure 8

Figure	r_1	r_2	α_1	α_2	φ_1	φ_2	w_0	q	e	*min*	n
a	.75	.77	35	-35	0	0	30	.50	.40	0.0	10
b	.65	.71	27	-68	0	0	20	.53	.50	1.7	12
c	.50	.85	25	-15	180	0	20	.45	.50	0.5	9
d	.60	.85	25	-15	180	180	20	.45	.50	0.0	10
e	.58	.83	30	15	0	180	20	.40	.50	1.0	11
f	.92	.37	0	60	180	0	2	.50	.00	0.5	15
g	.80	.80	30	-30	137	137	30	.50	.50	0.0	10
h	.95	.75	5	-30	-90	90	40	.60	.45	25.0	12
i	.55	.95	-5	30	137	137	5	.40	.00	5.0	12

width. The constants r_1 and r_2 determine the gradual decrease in internode length that occurs while traversing the tree from its base towards the apices. The constants w_0, q, and e control the width of branches. The initial stem width is specified by w_0 in the second parameter of the axiom module A. For $e = 0.5$, the combined area of the descendant branches is equal to the area of the mother branch, as postulated by Leonardo da Vinci [29, page 156] (see also [28, pages 131–135]). The value q specifies the differences in width between descendant branches originating at the same vertex. Finally, the condition prevents formation of branches with length less then the threshold value *min*. The values of constants corresponding to each structure are collected in Table 1. The final column headed n indicates the number of derivation steps.

6.4 Modeling mesotonic and acrotonic structures

In spite of their apparent diversity, the structures generated by L-system of Section 6.3 share a common developmental pattern: in each derivation step, every apex gives rise to an internode terminated by a pair of new apices. This is a simple instance of *subapical branching*, a common developmental pattern in plants, in which new branches are initiated only near the apices of the existing axes. As a consequence of this pattern, the lower branches, being created first, have more time to develop than the branches further up, and a *basitonic* structure (more developed near the base than near the top) results (Figure 9a). In nature, however, one also finds *mesotonic* and *acrotonic* structures, in which the most developed branches are located near the middle or the top of the mother branch (Figures 9 b and c). As observed by Frijters and Lindenmayer [9], and formalized by Prusinkiewicz and Kari [42], arbitrarily large mesotonic and acrotonic structures cannot be generated by non-parametric deterministic 0L-systems with subapical branching. In contrast, *parametric* D0L-systems can generate such structures. For example, the following parametric D0L-system generates the mesotonic structure shown in Figure 9b.

$$\begin{array}{lll} \omega: & FA(0) & \\ p_1: & A(v) & \rightarrow [-FB(v)][+FB(v)]FA(v+1) \\ p_2: & B(v): v > 0 & \rightarrow FB(v-1) \end{array}$$

The axiom ω defines the initial structure as an internode F terminated by an apex A. In each derivation step, the apex A adds a new segment F to the main axis and initiates a pair of branches FB (production p_1). The value of parameter v assigned to the lateral apices B describes the maximum length to which each branch will grow (production p_2). This value is incremented acropetally (*i.e.*, in the ascending order of branches) by production p_1, yielding a sequence of branches of increasing length. This sequence is broken in the upper part of the structure, where the branches still grow. Consequently, the younger branches near the top are shorter than the older ones further down, and a mesotonic overall structure results.

A detailed discussion of the generation of mesotonic and acrotonic structures using a construct similar to parametric L-systems has been presented by Lück, Lück, and Bakkali [27].

6.5 The shedding of branches

The natural processes of plant development often involve shedding, or programmed removal of selected modules from the growing structure. In order to simulate shedding, Hanan [11] extended the formalism of L-systems with the *cut symbol* $\%$, which causes the removal of the remainder of the branch that follows it. For example, in the absence of other productions, the derivation step given below takes place:

$$a[b\%[cd]e[\%f]]g[h[\%i]j]k \Longrightarrow a[b]g[h[]j]k$$

A simple example of an L-system incorporating the cut symbol is given below:

$$\begin{array}{lllll} \omega: & A & & & \\ p_1: & A & & \rightarrow & F(1)[-X(3)B][+X(3)B]A \\ p_2: & B & & \rightarrow & F(1)B \\ p_3: & X(d) & : d > 0 & \rightarrow & X(d-1) \\ p_4: & X(d) & : d == 0 & \rightarrow & U\% \\ p_5: & U & & \rightarrow & F(0.3) \end{array}$$

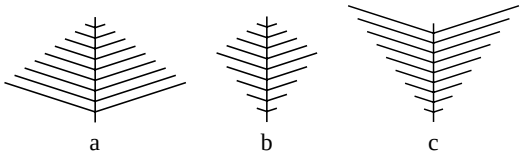


Figure 9: Schematic representation of a basitonic (a), mesotonic (b), and acrotonic (c) branching pattern. From [42].

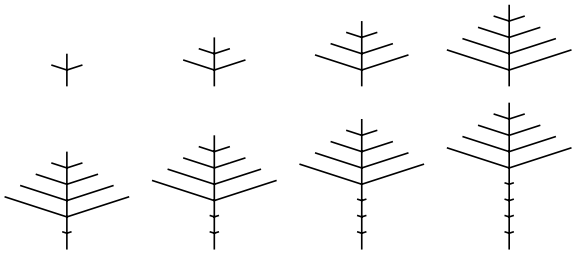


Figure 10: A developmental sequence generated by the L-system specified in Section 6.5. The images shown represent derivation steps 2 through 9.

According to production p_1, in each derivation step the apex of the main axis A produces an internode F of unit length and a pair of lateral apices B. Each apex B extends a branch by forming a succession of internodes F (production p_2). After three steps from branch initiation (controlled by production p_3), production p_4 inserts the cut symbol $\%$ and an auxiliary symbol U at the base of the branch. In the next step, the cut symbol removes the branch, while symbol U inserts a marker $F(0.3)$ indicating a "scar" left by the removed branch. The resulting developmental sequence is shown in Figure 10. The initial steps capture the growth of a basitonic structure. Beginning at derivation step 6, the oldest branches are shed, creating an impression of a tree crown of constant shape and size moving upwards. The crown is in a state of dynamic equilibrium: the addition of new branches and internodes at the apices is compensated by the loss of branches further down.

The state of dynamic equilibrium can be easily observed in the development of palms, where new leaves are created at the apex of the trunk while old leaves are shed at the base of the crown (Figure 11). Since both processes take place at the same rate, an adult palm carries an approximately constant number of leaves. This phenomenon has an interesting physiological explanation: palms are unable to gradually increase the diameter of their trunk over time, thus the flow of water and nutrients through the trunk can support only a crown of constant size.

7 Examples of context-sensitive L-system models

In this section we consider the propagation of control information through the structure of the developing plant (*endogenous information flow* [34]), which is captured by context-sensitive productions in the framework of L-systems. The conceptual elegance and expressive power of context-sensitive productions are among the most important assets of L-systems in modeling applications.

7.1 Development of a mesotonic structure

As outlined in Section 6.4, arbitrarily large mesotonic and acrotonic structures cannot be generated using deterministic 0L-systems without parameters [42]. The proposed mechanisms for modeling these structures can be divided into two categories: those using parameters to characterize the growth potential or vigor of individual apices, such as the L-system discussed in Section 6.4 and those postulating control of development by signals [8, 16]. The following L-system simulates the development of the mesotonic structure shown in Figure 12 using an acropetal (upward moving) signal.

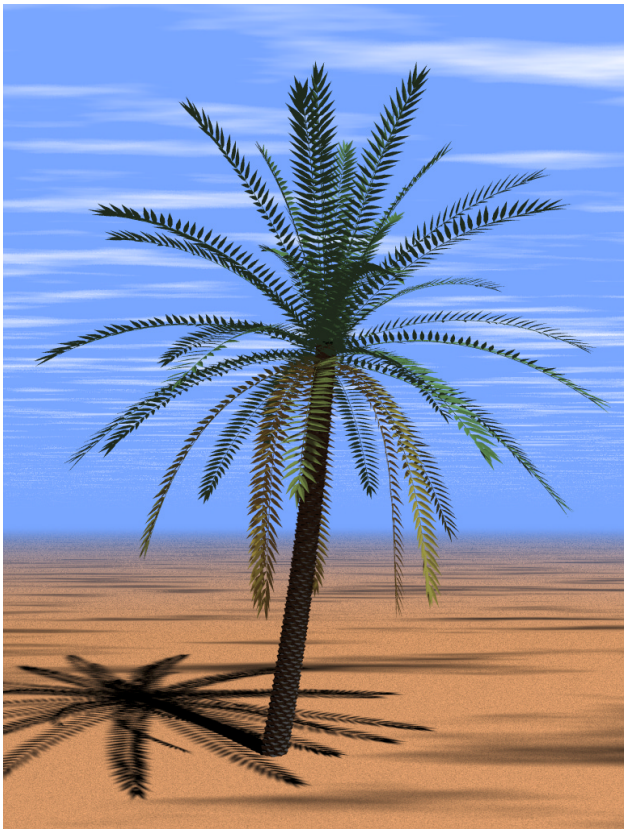


Figure 11: A model of the date palm (*Phoenix dactylifera*). This image was created using an L-system with the general structure specified in Section 6.5.

$$
\begin{array}{lll}
\#\text{define} & m & 3 \quad /*\,\text{plastochron} - \text{main axis}\,*/ \\
\#\text{define} & n & 4 \quad /*\,\text{plastochron} - \text{branc h}*/ \\
\#\text{define} & u & 4 \quad /*\,\text{propagation rate} - \text{main axis}\,*/ \\
\#\text{define} & v & 2 \quad /*\,\text{propagation rate} - \text{branc h}*/
\end{array}
$$

$$\text{ignore}: \quad +-/$$

$$
\begin{array}{ll}
\omega: & S(0)F(1,0)A(0) \\
p_1: & A(i) \quad : i < m-1 \rightarrow A(i+1) \\
p_2: & A(i) \quad : i == m-1 \rightarrow \\
 & \quad [+(60)F(1,1)B(0)]F(1,0)/(180)A(0) \\
p_3: & B(i) \quad : i < n-1 \rightarrow B(i+1) \\
p_4: & B(i) \quad : i == n-1 \rightarrow F(1,1)B(0) \\
p_5: & S(i) \quad : i < u+v \rightarrow S(i+1) \\
p_6: & S(i) \quad : i == u+v \rightarrow \varepsilon \\
p_7: & S(i) < F(l,o) : (o == 0)\&\&(i == u-1) \rightarrow \\
 & \quad \#F(l,o)!S(0) \\
p_8: & S(i) < F(l,o) : (o == 1)\&\&(i == v-1) \rightarrow \\
 & \quad \#F(l,o)!S(0) \\
p_9: & S(i) < B(j) \rightarrow \varepsilon
\end{array}
$$

The above L-system operates under the assumption that the context-sensitive production p_9 takes priority over p_3 or p_4. The ignore statement lists symbols that should not be taken for consideration for context-matching purposes. The axiom ω describes the initial structure as an internode F terminated by an apex A. A signal S

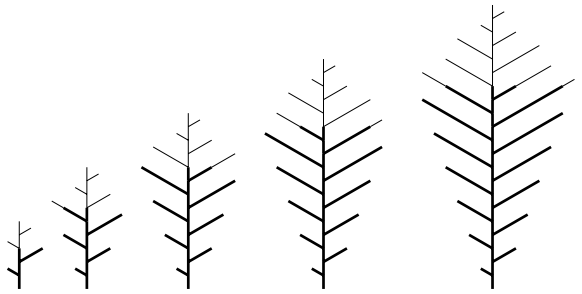


Figure 12: Development of a mesotonic branching structure controlled by an acropetal signal. Wide lines indicate the internodes reached by the signal. The stages shown correspond to derivation lengths 12, 24, 36, 48, and 60.

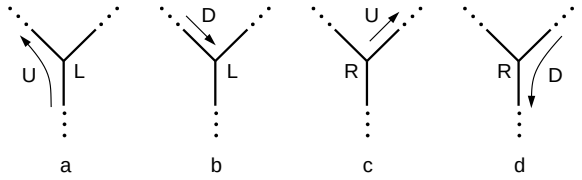


Figure 13: Insect's behavior at a branching point. An upward-moving insect U that approaches a branching point L is directed to the left daughter branch (a). A downward moving insect D that approaches a branching point marked L changes this marking to R, returns to state U, and enters the right branch (b and c). A downward moving insect D approaching a branching point R continues its downward motion (d).

is placed at the base of this structure. According to productions p_1 and p_2, the apex A periodically produces a lateral branch and adds an internode to the main axis. The period (called the *plastochron* of the main axis) is controlled by the constant m. Productions p_3 and p_4 describe the development of the lateral branches, where new segments F are added with plastochron n. Productions p_5 to p_8 describe the propagation of the signal through the structure. The signal propagation rate is u in the main axis, and v in the branches. Production p_9 removes the apex B when the signal reaches it, thus terminating the development of the corresponding lateral branch. Figure 12 shows that, for the values of plastochrons and signal propagation rates specified be the #define statements, the lower branches have less time to grow than the higher branches, and a mesotonic structure develops as a result.

A similar mechanism, based on the pursuit of apices by acropetal signals, has been proposed to model basipetal flowering sequences [43, 16, 23]. These sequences are characterized by the appearance of the first flower near the top of a plant, and a subsequent downward propagation of the flowering zone.

7.2 Attack of a plant by an insect

More complex information flow is considered in the next example. A hypothetical insect explores a growing branching structure and feeds on its apices. The insect always moves along the branches (*i.e.*, it does not jump or drop from one branch to another) and therefore can be treated as an endogenous signal. The insect's behavior at a branching point depends on its direction of motion and the state of the branching point, as explained in Figure 13. In a nutshell, the insect attempts to traverse the entire developing structure using the depth-first strategy. A context-sensitive L-system that integrates plant growth with the behavior of the insect is given below.

```
#define l_L   3     /* length of the left branch */
#define l_R   5     /* length of the right branch */
#define d     5     /* plastochron */
#define w     40    /* delay */
```

$$
\begin{array}{lllll}
\omega & : & W(w)FA(l_L, d) & & \\
p_1: & F & < A(n,m) & : m > 0 & \\
 & & & \rightarrow A(n, m-1) & \\
p_2: & F & < A(n,m) & : n > 0 \;\&\&\; m == 0 & \\
 & & & \rightarrow FA(n-1, d) & \\
p_3: & F & < A(n,m) & : n == 0 \;\&\&\; m == 0 & \\
 & & & \rightarrow L[+FA(l_L, d)][-FA(l_R, d)] & \\
p_4: & W(t) & : t > 0 & \rightarrow W(t-1) & \\
p_5: & W(t) & : t == 0 & \rightarrow U & \\
p_6: & U & < F & \rightarrow FU & \\
p_7: & U & & \rightarrow \varepsilon & \\
p_8: & UL & < + & \rightarrow +U & \\
p_9: & U & < A(n,m) & \rightarrow D & \\
p_{10}: & F & > D & \rightarrow DF & \\
p_{11}: & D & & \rightarrow \varepsilon & \\
p_{12}: & L & > [+D] & \rightarrow UR & \\
p_{13}: & UR & < - & \rightarrow -U & \\
p_{14}: & R & > [\,][-D] & \rightarrow D &
\end{array}
$$

Productions p_1 to p_3 describe the development of a simple branching structure. Starting with a single axis specified by axiom ω, the apex A appends a sequence of branch segments F to the current axis (productions p_1 and p_2), then initiates a pair of new lateral apices (production p_3) that recursively repeat the same pattern. Parameter m is used to count the derivation steps between the creation of consecutive segments F. Parameter n determines the remaining number of segments to be produced before the next branching occurs. The total number of segments in an axis is defined by constants l_L (for the main axis and the branches issued to the left) and l_R (for the branches issued to the right). A newly created branching point is marked by symbol L (production p_3).

After a delay of w steps introduced by production p_4, production p_5 places an insect in the state U at the base of the branching structure. This insect moves upwards, one branch segment per derivation step (productions p_6 and p_7), until it encounters the branching point marker L. The insect is then directed to the left daughter branch (production p_8). After crossing a number of segments and, possibly, further branching points, the insect eventually reaches an apex A. As specified by production p_9, this apex is then removed from the structure, thus stopping further growth of its axis, and the state of the insect is changed from U (moving upwards) to D (moving downwards). The downward movement is simulated by productions p_{10} and p_{11}. Returning to a branching point marked L, the insect changes this mark to R to indicate that the left branch has been already explored, reverts its own state to U, and enters the right branch (productions p_{12} and p_{13}). Coming back from that branch, the insect continues its downward movement (production p_{14}) until it reaches another branching point marked L and enters an unexplored right branch, or until it completes the traversal of the entire structure at its base.

A sequence of images obtained using a straightforward extension of the above L-system is shown in Figure 14. In this case, the insect feeds on the apices of a three-dimensional structure, and a branch that no longer carries any apices wilts.

Similar models can be constructed assuming different traversing

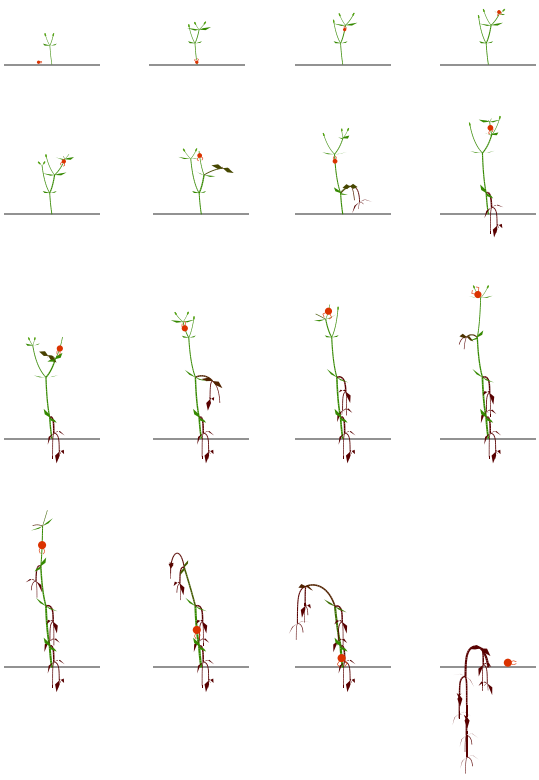


Figure 14: Simulation of the development of a plant attacked by an insect

and feeding strategies for one or many insects (which may interact with each other). Prospective applications of such models include simulation studies of insects used for weed control and of the impact of insects on crop plants [48, 49].

7.3 Development controlled by resource allocation

In the previous examples, discrete information was transferred between the modules of a developing structure. A signal (or insect) was either present or absent at any particular point, and affected the structure in an "all-or-nothing" manner, by removing the apices at the ends of branches. In nature, however, developmental processes are often controlled in a more modulated way, by the quantity of substances (*resources*) exchanged between the modules. For example, the growth of plants depends on the amount of water and minerals absorbed by the roots and carried acropetally (upwards), and by the amount of photosynthates produced by the leaves and transported basipetally. An early developmental model of branching structures making use of quantitative information flow was proposed by Borchert and Honda [3]. Below we restate the essence of this model using the formalism of L-systems, then we extend it to simulate interactions between the shoot and the roots in a growing plant.

Borchert and Honda postulated that the development of a branching structure is controlled by a flow or *flux* of substances, which propagate from the base of the structure towards the apices and supply them with materials needed for growth. When the flux reaching an apex exceeds a predefined threshold value, the apex bifurcates and initiates a lateral branch; otherwise it remains inactive. At branching points the flux is distributed according to the types of the supported internodes (straight or lateral) and the number of apices in the corresponding branches. These numbers are accumulated by messages that originate at the apices and propagate towards the base of the plant. Thus, development is controlled by a cycle of alternating acropetal and basipetal information flow.

An L-system that implements these mechanisms is given below.

#define α_1 10 /* branching angle - straight segment */
#define α_2 32 /* branching angle - lateral segment */
#define σ_0 17 /* initial flux */
#define η 0.89 /* controls input flux changes */
#define λ 0.7 /* flux distribution factor */
#define v_{th} 5.0 /* threshold flux for branching */

ignore: $+\ -/$

ω : $N(1)I(0,2,0,1)A$
p_1 : $N(k) < I(b,m,v,c) : b == 0\ \&\&\ m == 2$
$\rightarrow I(b,1,\sigma_0 * 2 \wedge (k-1) * (\eta \wedge k), c)$
p_2 : $N(k) > I(b,m,v,c) : b == 0\ \&\&\ m == 2 \rightarrow N(k+1)$
p_3 : $I(b_l,m_l,v_l,c_l) < I(b,m,v,c) : m_l == 1\ \&\&\ b == 1$
$\rightarrow I(b,m_l,v_l - v_l * (1-\lambda) * ((c_l - c)/c), c)$
p_4 : $I(b_l,m_l,v_l,c_l) < I(b,m,v,c) : m_l == 1\ \&\&\ b == 2$
$\rightarrow I(b,m_l,v_l * (1-\lambda) * (c/(c_l - c)), c)$
p_5 : $I(b,m,v,c) < A : m == 1\ \&\&\ v > v_{th}$
$\rightarrow /(180)[-(\alpha_2)I(2,2,v*(1-\lambda),1)A]$
$+(\alpha_1)I(1,2,v*\lambda,1)A$
p_6 : $I(b,m,v,c) > A : m == 1\ \&\&\ v <= v_{th} \rightarrow I(b,2,v,c)$
p_7 : $I(b,m,v,c) > [I(b_2,m_2,v_2,c_2) =]I(b_1,m_1,v_1,c_1) :$
$m == 0\ \&\&\ m_1 == 2\ \&\&\ m_2 == 2$
$\rightarrow I(b,2,v,c_1 + c_2)$
p_8 : $I(b,m,v,c) : m == 1 \rightarrow I(b,0,v,c)$
p_9 : $I(b_l,m_l,v_l,c_l) < I(b,m,v,c) : m_l == 2\ \&\&\ m == 2$
$\rightarrow I(b,0,v,c)$

This L-system operates on three types of modules: apices A, internodes I, and an auxiliary module N. The internodes are visualized as lines of unit length. Each internode has four parameters:

- **segment type** b, where 0 denotes base of the tree, 1 – a straight segment, and 2 – a lateral segment;
- **message type** m, where 0 denotes no message currently carried by the internode, 1 – an acropetal message (flux), and 2 – a basipetal message (apex count);
- **flux value** v, and
- **apex count** c.

All internodes are visualized as lines of unit length.

At the beginning of a developmental cycle, indicated by the presence of a basipetal message ($m = 2$) in the basal internode ($b = 0$), production p_1 calculates an input flux value. The expression used for this purpose, $v = \sigma_0 2^{(k-1)\eta^k}$, was introduced by Borchert and Honda to simulate a sigmoid increase of flux penetrating the base of a plant over time. The progress of time is captured by production p_2, which increments the current cycle number k in module N.

Productions p_3 and p_4 simulate acropetal flux propagation and distribute it between the straight segment and the lateral segment. If both the straight and lateral branch support the same number of

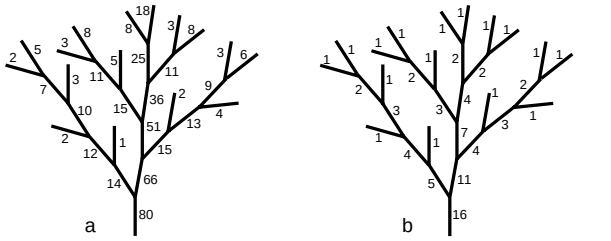


Figure 15: The structure generated by L-system of Section 7.3 at completion of the fifth developmental cycle. The numbers indicate the flow values v rounded to the nearest integer (a), and the numbers of apices c in the branches supported by each internode (b).

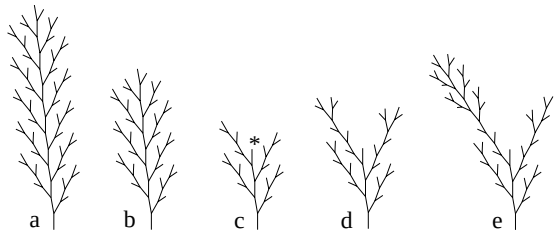


Figure 16: Development of a branching structure simulated using an L-system implementation of the model by Borchert and Honda. (a) Development not affected by pruning; (b, c) the structure immediately before and after pruning; (d, e) the subsequent development of the pruned structure. Based on [3].

apices, the straight segment will obtain a predefined fraction λ of the flux v_l reaching the branching point; the lateral segment will obtain the remainder, $(1-\lambda)v_l$. If a lateral branch supports c apices and its sister straight branch supports c_s apices, the flux reaching the lateral branch is further multiplied by the ratio c/c_s. The number c_s is not directly available to the lateral branch, but it can be calculated as the difference between the number of apices supported by this branch and its mother, $c_s = c_l - c$. In total, the flux directed towards the lateral branch is equal to $v_l(1-\lambda)(c/(c_l-c)$ (production p_3). The remaining flux reaches the straight segment. The parameter c denotes, in this case, the number of apices supported by the straight segment, and the resulting expression is $v_l - v_l(1-\lambda)((c_l-c)/c)$ (production p_4).

Productions p_5 and p_6 control the addition of new segments to the structure. According to production p_5, if the internode preceding an apex A reaches a sufficient flux $v > v_{th}$, the apex will create two new internodes I terminated by apices A. The new segments are assigned an initial message type $m = 2$, which triggers the basipetal signal propagation needed to update the count of apices supported by each segment. Alternatively, if the flux reaching an apex is not sufficient for bifurcation ($v \leq v_{th}$), the supporting internode itself starts the propagation of the basipetal signal (production p_6).

Production p_7 adds the number of apices supported by the daughter branches (c_1 and c_2), and propagates the result to the mother internode. Both input numbers must be available ($m_1 = 2$ and $m_2 = 2$) before basipetal message propagation takes place.

The remaining productions reset the message value m to zero, after the flux values have been transferred acropetally (p_8) or the apex count has been passed basipetally (p_9).

The initial state of the model is determined by the axiom ω. The value of the parameter to module N sets the current cycle number to 1. The initial structure consists of a single internode I terminated by an apex A. The message type indicates the presence of a basipetal message ($m = 2$) which triggers the application of productions p_1 and p_2, initiating the first full developmental cycle. The state of the structure after 35 derivation steps (completion of the fifth developmental cycle) is shown in Figure 15.

A remarkable feature of Borchert and Honda's model is its ability to simulate the response of a plant to its environment. Specifically, after a branch has been pruned, the model redirects the fluxes to the remaining branches and accelerates their growth to compensate for the loss. A sequence of structures that illustrates this phenomenon is shown in Figure 16. In accordance with [3], the L-system used in this case extends the L-system discussed above with parameters and productions needed to capture the effect of aging. Consequently, a branch that was unable to grow for a given number of developmental cycles dies: it loses the ability to develop further and stops taking any fluxes.

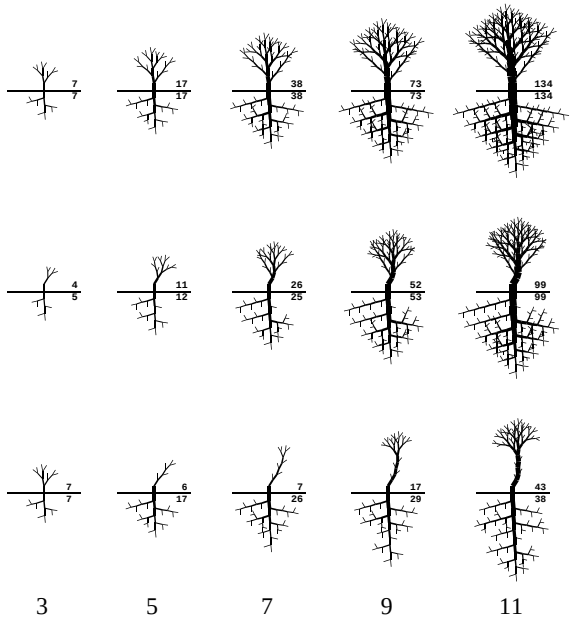


Figure 17: Application of the Borchert and Honda's model to the simulation of a complete plant, showing development unaffected by pruning (top row), affected by pruning during the third cycle of development (middle row), and affected by pruning during the fifth cycle of development (bottom row). The numbers of live apices in the shoot and root are indicated above and below the ground level. The numbers at the base of the figure indicate the number of completed developmental cycles.

Similar behavior is shown in Figure 17. In this case, two structures representing the shoot and the root of a plant are generated simultaneously. The flux penetrating the root at the beginning of a developmental cycle is assumed to be proportional to the number of apices in the shoot; reciprocally, the flux penetrating the shoot is proportional to the number of apices in the root. These assumptions form a crude approximation of plant physiology, whereby the photosynthates produced by the shoot fuel the development of the root, and water and mineral compounds gathered by the root are required for the development of the shoot. The model also assumes an increase of internode width over time and a gradual rotation of a lateral segment to the straight segment position, after the straight segment has

been lost. The developmental sequence shown in the top row of Figure 17 is unaffected by pruning. The shoot and the root develop in concert. The next two rows illustrate development affected by a loss of branches. The removal of a shoot branch slows down the development of the root; on the other hand, the large size of the root, compared to the remaining shoot, fuels a fast re-growth of the shoot. Eventually, the plant is able to redress the balance between the size of the shoot and the root. This is a non-obvious consequence of the model, which illustrates the usefulness of L-systems in predicting the global behavior of plants, given the specification of their components.

8 Conclusions

L-system models integrate local processes, taking place at the level of individual modules, into developmental patterns and structures of entire plants. Consequently, they address the central problem of morphogenesis: the description and understanding of mechanisms through which living organisms acquire their form. This aspect of modeling motivated the original biological applications of L-systems investigated by Lindenmayer and his collaborators, and continues to play a key role in current biological research using L-systems. The emergence of global forms and developmental patterns is also important in the application of L-systems to computer graphics, because it makes it possible to create realistic representations of growing plants using relatively easy to specify, compact sets of data.

In principle, the mathematical formulation of L-systems should also make it possible to address biologically relevant questions in the form of a deductive theory of plant development. The results of this theory could be potentially more general than simulations, which are inherently limited to case studies. Unfortunately, construction of such a theory still seems quite remote. One reason is the lack of a precise mathematical description of plant form. This is not of crucial importance in simulations, where the results are evaluated visually, but impedes the formulation of theorems and proofs. Another difficulty is the discrepancy between studies on the theory of L-systems and the needs of biological modeling. Most theoretical results are pertinent to non-parametric 0L-systems that operate on non-branching strings without geometric interpretation (for examples, see [50]). In contrast, L-system models of biological phenomena often involve parameters, interactions between modules, and geometric features of the modeled structures. We hope that further development of the theory of L-systems will bridge this gap.

9 Acknowledgements

An overview of L-systems was the subject of several invited lectures and tutorials presented recently by P. Prusinkiewicz. Consequently, this paper includes sections of previous surveys [36, 37], and coincides with [35]. The idea of using L-systems to simulate the interaction between plants and insects was proposed by Peter Room. The reported research has been sponsored by grants and graduate scholarships from the Natural Sciences and Engineering Council of Canada.

References

[1] H. Abelson and A. A. diSessa. *Turtle geometry*. M.I.T. Press, Cambridge, 1982.

[2] A. Bell. *Plant form: An illustrated guide to flowering plants.* Oxford University Press, Oxford, 1991.

[3] R. Borchert and H. Honda. Control of development in the bifurcating branch system of *Tabebuia rosea*: A computer simulation. *Botanical Gazette*, 145(2):184–195, 1984.

[4] M. J. M. de Boer. *Analysis and computer generation of division patterns in cell layers using developmental algorithms*. PhD thesis, University of Utrecht, 1989.

[5] M. J. M. de Boer, F. D. Fracchia, and P. Prusinkiewicz. A model for cellular development in morphogenetic fields. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 351–370. Springer-Verlag, Berlin, 1992.

[6] J. D. Foley and A. Van Dam. *Fundamentals of interactive computer graphics*. Addison-Wesley, Reading, 1982.

[7] F. D. Fracchia, P. Prusinkiewicz, and M. J. M. de Boer. Animation of the development of multicellular structures. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Animation '90*, pages 3–18, Tokyo, 1990. Springer-Verlag.

[8] D. Frijters and A. Lindenmayer. A model for the growth and flowering of *Aster novae-angliae* on the basis of table (1,0)L-systems. In G. Rozenberg and A. Salomaa, editors, *L Systems*, Lecture Notes in Computer Science 15, pages 24–52. Springer-Verlag, Berlin, 1974.

[9] D. Frijters and A. Lindenmayer. Developmental descriptions of branching patterns with paracladial relationships. In A. Lindenmayer and G. Rozenberg, editors, *Automata, languages, development*, pages 57–73. North-Holland, Amsterdam, 1976.

[10] J. S. Hanan. *PLANTWORKS: A software system for realistic plant modelling*. Master's thesis, University of Regina, November 1988.

[11] J. S. Hanan. *Parametric L-systems and their application to the modelling and visualization of plants*. PhD thesis, University of Regina, June 1992.

[12] J. L. Harper and A. D. Bell. The population dynamics of growth forms in organisms with modular construction. In R. M. Anderson, B. D. Turner, and L. R. Taylor, editors, *Population dynamics*, pages 29–52. Blackwell, Oxford, 1979.

[13] G. T. Herman and G. Rozenberg. *Developmental systems and languages.* North-Holland, Amsterdam, 1975.

[14] H. Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 31:331–338, 1971.

[15] M. James, J. Hanan, and P. Prusinkiewicz. CPFG version 2.0 user's manual. Manuscript, Department of Computer Science, University of Calgary, 1993, 50 pages.

[16] J. M. Janssen and A. Lindenmayer. Models for the control of branch positions and flowering sequences of capitula in *Mycelis muralis* (L.) Dumont (Compositae). *New Phytologist*, 105:191–220, 1987.

[17] B. W. Kernighan and D. M. Ritchie. *The C programming language. Second edition.* Prentice Hall, Englewood Cliffs, 1988.

[18] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[19] A. Lindenmayer. Developmental systems without cellular interaction, their languages and grammars. *Journal of Theoretical Biology*, 30:455–484, 1971.

[20] A. Lindenmayer. Developmental algorithms for multicellular organisms: A survey of L-systems. *Journal of Theoretical Biology*, 54:3–22, 1975.

[21] A. Lindenmayer. Algorithms for plant morphogenesis. In R. Sattler, editor, *Theoretical plant morphology*, pages 37–81. Leiden University Press, The Hague, 1978.

[22] A. Lindenmayer. Developmental algorithms: Lineage versus interactive control mechanisms. In S. Subtelny and P. B. Green, editors, *Developmental order: Its origin and regulation*, pages 219–245. Alan R. Liss, New York, 1982.

[23] A. Lindenmayer. Positional and temporal control mechanisms in inflorescence development. In P. W. Barlow and D. J. Carr, editors, *Positional controls in plant development*. University Press, Cambridge, 1984.

[24] A. Lindenmayer. Models for multicellular development: Characterization, inference and complexity of L-systems. In A. Kelemenová and J. Kelemen, editors, *Trends, techniques and problems in theoretical computer science*, Lecture Notes in Computer Science 281, pages 138–168. Springer-Verlag, Berlin, 1987.

[25] A. Lindenmayer and H. Jürgensen. Grammars of development: Discrete-state models for growth, differentiation and gene expression in modular organisms. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 3–21. Springer-Verlag, Berlin, 1992.

[26] A. Lindenmayer and P. Prusinkiewicz. Developmental models of multicellular organisms: A computer graphics perspective. In C. G. Langton, editor, *Artificial Life*, pages 221–249. Addison-Wesley, Redwood City, 1988.

[27] J. Lück, H. B. Lück, and M. Bakkali. A comprehensive model for acrotonic, mesotonic, and basitonic branching in plants. *Acta Biotheoretica*, 38:257–288, 1990.

[28] N. MacDonald. *Trees and networks in biological models.* J. Wiley & Sons, New York, 1983.

[29] B. B. Mandelbrot. *The fractal geometry of nature*. W. H. Freeman, San Francisco, 1982.

[30] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. Proceedings of SIGGRAPH '96 (New Orleans, Louisiana, August 4–9, 1996). ACM SIGGRAPH, New York, 1996, pp. 397–410.

[31] S. Papert. *Mindstorms: Children, computers and powerful ideas.* Basic Books, New York, 1980.

[32] P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86 — Vision Interface '86*, pages 247–253, 1986.

[33] P. Prusinkiewicz. Applications of L-systems to computer imagery. In H. Ehrig, M. Nagl, A. Rosenfeld, and G. Rozenberg, editors, *Graph grammars and their application to computer science; Third International Workshop*, pages 534–548. Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 291.

[34] P. Prusinkiewicz. Visual models of morphogenesis. *Artificial Life*, 1:61–74, 1994.

[35] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. L-systems: from the theory to visual models of plants. *Machine Graphics and Vision*, 5(1/2):365–392, 1996.

[36] P. Prusinkiewicz, M. Hammel, J. Hanan, and R. Měch. Visual models of plant development. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, Vol. III*: Beyond words*, pages 535–597. Springer, Berlin, 1997.

[37] P. Prusinkiewicz, M. Hammel, R. Měch, and J. Hanan. The artificial life of plants. In D. Terzopoulos, editor, *SIGGRAPH 1995 Course Notes on Artificial Life*, pages 1–1 – 1–38. ACM SIGGRAPH, 1995.

[38] P. Prusinkiewicz and J. Hanan. *Lindenmayer systems, fractals, and plants*, volume 79 of *Lecture Notes in Biomathematics*. Springer-Verlag, Berlin, 1989 (second printing 1992).

[39] P. Prusinkiewicz and J. Hanan. Visualization of botanical structures and processes using parametric L-systems. In D. Thalmann, editor, *Scientific visualization and graphics simulation*, pages 183–201. J. Wiley & Sons, Chichester, 1990.

[40] P. Prusinkiewicz and J. Hanan. L-systems: From formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 193–211. Springer-Verlag, Berlin, 1992.

[41] P. Prusinkiewicz, M. James, and R. Měch. Synthetic topiary. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994). ACM SIGGRAPH, New York, 1994, pp. 351–358.

[42] P. Prusinkiewicz and L. Kari. Subapical bracketed L-systems. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph grammars and their application to computer science; Fifth International Workshop*, Lecture Notes in Computer Science 1073, pages 550–564. Springer-Verlag, Berlin, 1996.

[43] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[44] P. Prusinkiewicz, A. Lindenmayer, and F. D. Fracchia. Synthesis of space-filling curves on the square grid. In H.-O. Peitgen, J. M. Henriques, and L. F. Penedo, editors, *Fractals in the fundamental and applied sciences*, pages 341–366. North-Holland, Amsterdam, 1991.

[45] P. Prusinkiewicz, W. Remphrey, C. Davidson, and M. Hammel. Modeling the architecture of expanding *Fraxinus pennsylvanica* shoots using L-systems. *Canadian Journal of Botany*, 72:701–714, 1994.

[46] D. M. Raup. Geometric analysis of shell coiling: general problems. *Journal of Paleontology*, 40:1178–1190, 1966.

[47] D. M. Raup and A. Michelson. Theoretical morphology of the coiled shell. *Science*, 147:1294–1295, 1965.

[48] P. M. Room, J. S. Hanan, and P. Prusinkiewicz. Virtual plants: new perspectives for ecologists, pathologists, and agricultural scientists. *Trends in Plant Science*, 1(1):33–38, 1996.

[49] P. M. Room, L. Maillette, and J. Hanan. Module and metamer dynamics and virtual plants. *Advances in Ecological Research*, 25:105–157, 1994.

[50] G. Rozenberg and A. Salomaa. *The mathematical theory of L systems.* Academic Press, New York, 1980.

[51] A. Salomaa. *Formal languages.* Academic Press, New York, 1973.

[52] A. R. Smith. Plants, fractals, and formal languages. Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 22–27, 1984). In *Computer Graphics*, 18, 3 (July 1984), pages 1–10, ACM SIGGRAPH, New York, 1984.

[53] A. L. Szilard and R. E. Quinton. An interpretation for DOL systems by computer graphics. *The Science Terrapin*, 4:8–13, 1979.

[54] H. von Koch. Une méthode géométrique élémentaire pour l'étude de certaines questions de la théorie des courbes planes. *Acta Mathematica*, 30:145–174, 1905.

[55] D. M. Waller and D. A. Steingraeber. Branching and modular growth: Theoretical models and empirical patterns. In J. B. C. Jackson and L. W. Buss, editors, *Population biology and evolution of clonal organisms*, pages 225–257. Yale University Press, New Haven, 1985.

Visual Models of Plants Interacting with Their Environment

Radomír Měch and Przemyslaw Prusinkiewicz[1]
University of Calgary

ABSTRACT

Interaction with the environment is a key factor affecting the development of plants and plant ecosystems. In this paper we introduce a modeling framework that makes it possible to simulate and visualize a wide range of interactions at the level of plant architecture. This framework extends the formalism of Lindenmayer systems with constructs needed to model bi-directional information exchange between plants and their environment. We illustrate the proposed framework with models and simulations that capture the development of tree branches limited by collisions, the colonizing growth of clonal plants competing for space in favorable areas, the interaction between roots competing for water in the soil, and the competition within and between trees for access to light. Computer animation and visualization techniques make it possible to better understand the modeled processes and lead to realistic images of plants within their environmental context.

CR categories: F.4.2 **[Mathematical Logic and Formal Languages]**: Grammars and Other Rewriting Systems: *Parallel rewriting systems*, I.3.7 **[Computer Graphics]**: Three-Dimensional Graphics and Realism, I.6.3 **[Simulation and Modeling]**: Applications, J.3 **[Life and Medical Sciences]**: Biology.

Keywords: scientific visualization, realistic image synthesis, software design, L-system, modeling, simulation, ecosystem, plant development, clonal plant, root, tree.

1 INTRODUCTION

Computer modeling and visualization of plant development can be traced back to 1962, when Ulam applied cellular automata to simulate the development of branching patterns, thought of as an abstract representation of plants [53]. Subsequently, Cohen presented a more realistic model operating in continuous space [13], Lindenmayer proposed the formalism of L-systems as a general framework for plant modeling [38, 39], and Honda introduced the first computer model of tree structures [32]. From these origins, plant modeling emerged as a vibrant area of interdisciplinary research, attracting the efforts of biologists, applied plant scientists, mathematicians, and computer scientists. Computer graphics, in particular, contributed a wide range of models and methods for synthesizing images of plants. See [18, 48, 54] for recent reviews of the main results.

One aspect of plant structure and behavior neglected by most models is the interaction between plants and their environment (including other plants). Indeed, the incorporation of interactions has been identified as one of the main outstanding problems in the domain of plant modeling [48] (see also [15, 18, 50]). Its solution is needed to construct predictive models suitable for applications ranging from computer-assisted landscape and garden design to the determination of crop and lumber yields in agriculture and forestry.

Using the information flow between a plant and its environment as the classification key, we can distinguish three forms of interaction and the associated models of plant-environment systems devised to date:

1. The plant is affected by global properties of the environment, such as day length controlling the initiation of flowering [23] and daily minimum and maximum temperatures modulating the growth rate [28].

2. The plant is affected by local properties of the environment, such as the presence of obstacles controlling the spread of grass [2] and directing the growth of tree roots [26], geometry of support for climbing plants [2, 25], soil resistance and temperature in various soil layers [16], and predefined geometry of surfaces to which plant branches are pruned [45].

3. The plant interacts with the environment in an information feedback loop, where the environment affects the plant and the plant reciprocally affects the environment. This type of interaction is related to *sighted* [4] or *exogenous* [42] mechanisms controlling plant development, in which parts of a plant influence the development of other parts of the same or a different plant through the space in which they grow. Specific models capture:

   - competition for *space* (including collision detection and access to light) between segments of essentially two-dimensional schematic branching structures [4, 13, 21, 22, 33, 34, 36];
   - competition between root tips for *nutrients* and *water* transported in soil [12, 37] (this mechanism is related to competition between growing branches of corals and sponges for nutrients diffusing in water [34]);

[1]Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4 (mech|pwp@cpsc.ucalgary.ca)

Published in the Proceedings of SIGGRAPH '96 (New Orleans, LA, August 4–9, 1996). In *Computer Graphics* Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, New York, pp. 397–410.

– competition for *light* between three-dimensional shoots of herbaceous plants [25] and branches of trees [9, 10, 11, 15, 33, 35, 52].

Models of exogenous phenomena require a comprehensive representation of both the developing plant and the environment. Consequently, they are the most difficult to formulate, implement, and document. Programs addressed to the biological audience are often limited to narrow groups of plants (for example, poplars [9] or trees in the pine family [21]), and present the results in a rudimentary graphical form. On the other hand, models addressed to the computer graphics audience use more advanced techniques for realistic image synthesis, but put little emphasis on the faithful reproduction of physiological mechanisms characteristic to specific plants.

In this paper we propose a general *framework* (defined as a modeling methodology supported by appropriate software) for modeling, simulating, and visualizing the development of plants that bi-directionally interact with their environment. The usefulness of modeling frameworks for simulation studies of models with complex (emergent) behavior is manifested by previous work in theoretical biology, artificial life, and computer graphics. Examples include cellular automata [51], systems for simulating behavior of cellular structures in discrete [1] and continuous [20] spaces, and L-system-based frameworks for modeling plants [36, 46]. Frameworks may have the form of a general-purpose simulation program that accepts models described in a suitable mini-language as input, *e.g.* [36, 46], or a set of library programs [27]. Compared to special-purpose programs, they offer the following benefits:

- At the conceptual level, they facilitate the design, specification, documentation, and comparison of models.
- At the level of model implementation, they make it possible to develop software that can be reused in various models. Specifically, graphical capabilities needed to visualize the models become a part of the modeling framework, and do not have to be reimplemented.
- Finally, flexible conceptual and software frameworks facilitate interactive experimentation with the models [46, Appendix A].

Our framework is intended both for purpose of image synthesis and as a research and visualization tool for model studies in plant morphogenesis and ecology. These goals are addressed at the levels of the simulation system and the modeling language design. The underlying paradigm of plant-environment interaction is described in Section 2. The resulting design of the simulation software is outlined in Section 3. The language for specifying plant models is presented in Section 4. It extends the concept of environmentally-sensitive L-systems [45] with constructs for bi-directional communication with the environment. The following sections illustrate the proposed framework with concrete models of plants interacting with their environment. The examples include: the development of planar branching systems controlled by the crowding of apices (Section 5), the development of clonal plants controlled by both the crowding of ramets and the quality of terrain (Section 6), the development of roots controlled by the concentration of water transported in the soil (Section 7), and the development of tree crowns affected by the local distribution of light (Section 8) The paper concludes with an evaluation of the results and a list of open problems (Section 9).

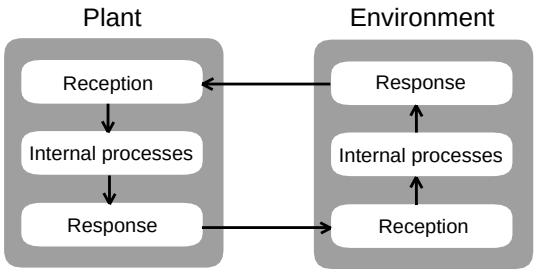


Figure 1: Conceptual model of plant and environment treated as communicating concurrent processes

2 CONCEPTUAL MODEL

As described by Hart [30], every environmentally controlled phenomenon can be considered as a chain of causally linked events. After a stimulus is perceived by the plant, information in some form is transported through the plant body (unless the site of stimulus perception coincides with the site of response), and the plant reacts. This reaction reciprocally affects the environment, causing its modification that in turn affects the plant. For example, roots growing in the soil can absorb or extract water (depending on the water concentration in their vicinity). This initiates a flow of water in the soil towards the depleted areas, which in turn affects further growth of the roots [12, 24].

According to this description, the interaction of a plant with the environment can be conceptualized as two concurrent processes that communicate with each other, thus forming a feedback loop of information flow (Figure 1). The plant process performs the following functions:

- reception of information about the environment in the form of scalar or vector values representing the stimuli perceived by specific organs;
- transport and processing of information inside the plant;
- generation of the response in the form of growth changes (*e.g.* development of new branches) and direct output of information to the environment (*e.g.* uptake and excretion of substances by a root tip).

Similarly, the environmental process includes mechanisms for the:

- perception of the plant's actions;
- simulation of internal processes in the environment (*e.g.* the diffusion of substances or propagation of light);
- presentation of the modified environment in a form perceivable by the plant.

The design of a simulation system based on this conceptual model is presented next.

3 SYSTEM DESIGN

The goal is to create a framework, in which a wide range of plant structures and environments can be easily created, modified, and

used for experimentation. This requirement led us to the following design decisions:

- The plant and the environment should be modeled by separate programs and run as two communicating processes. This design is:
  - compatible with the assumed conceptual model of plant-environment interaction (Figure 1);
  - consistent with the principles of structured design (modules with clearly specified functions jointly contribute to the solution of a problem by communicating through a well defined interface; information local to each module is hidden from other modules);
  - appropriate for interactive experimentation with the models; in particular, changes in the plant program can be implemented without affecting the environmental program, and *vice versa*;
  - extensible to distributed computing environments, where different components of a large ecosystem may be simulated using separate computers.
- The user should have control over the type and amount of information exchanged between the processes representing the plant and the environment, so that all the needed but no superfluous information is transferred.
- Plant models should be specified in a language based on L-systems, equipped with constructs for bi-directional communication between the plant and the environment. This decision has the following rationale:
  - A succinct description of the models in an interpreted language facilitates experimentation involving modifications to the models;
  - L-systems capture two fundamental mechanisms that control development, namely flow of information from a mother module to its offspring (cellular descent) and flow of information between coexisting modules (endogenous interaction) [38]. The latter mechanism plays an essential role in transmitting information from the site of stimulus perception to the site of the response. Moreover, L-systems have been extended to allow for input of information from the environment (see Section 4);
  - Modeling of plants using L-systems has reached a relatively advanced state, manifested by models ranging from algae to herbaceous plants and trees [43, 46].
- Given the variety of processes that may take place in the environment, they should be modeled using special-purpose programs.
- Generic aspects of modeling, not specific to particular models, should be supported by the modeling system. This includes:
  - an L-system-based plant modeling program, which interprets L-systems supplied as its input and visualizes the results, and
  - the support for communication and synchronization of processes simulating the modeled plant and the environment.

A system architecture stemming from this design is shown in Figure 2. We will describe it from the perspective of extensions to the formalism of L-systems.

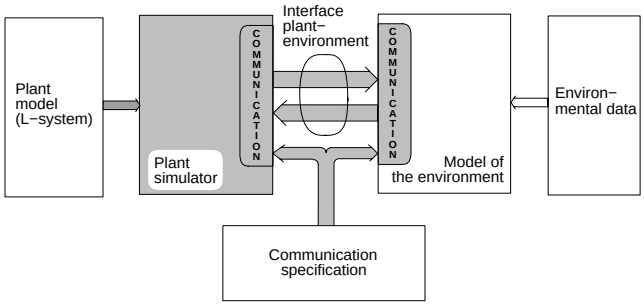


Figure 2: Organization of the software for modeling plants interacting with their environment. Shaded rectangles indicate components of the modeling framework, clear rectangles indicate programs and data that must be created by a user specifying a new model of a plant or environment. Shaded arrows indicate information exchanged in a standardized format.

4 OPEN L-SYSTEMS

Historically, L-systems were conceived as closed cybernetic systems, incapable of simulating any form of communication between the modeled plant and its environment. In the first step towards the inclusion of environmental factors, Rozenberg defined *table L-systems*, which allow for a change in the set of developmental rules (the production set of the L-system) in response to a change in the environment [31, 49]. Table L-systems were applied, for example, to capture the switch from the production of leaves to the production of flowers by the apex of a plant due to a change in day length [23]. *Parametric L-systems* [29, 46], introduced later, made it possible to implement a variant of this technique, with the environment affecting the model in a quantitative rather than qualitative manner. In a case study illustrating this possibility, weather data containing daily minimum and maximum temperatures were used to control the rate of growth in a bean model [28]. *Environmentally-sensitive L-systems* [45] represented the next step in the inclusion of environmental factors, in which local rather than global properties of the environment affected the model. The new concept was the introduction of query symbols, returning current position or orientation of the turtle in the underlying coordinate system. These parameters could be passed as arguments to user-defined functions, returning local properties of the environment at the queried location. Environmentally-sensitive L-systems were illustrated by models of topiary scenes. The environmental functions defined geometric shapes, to which trees were pruned.

Open L-systems, introduced in this paper, augment the functionality of environmentally-sensitive L-systems using a reserved symbol for bilateral communication with the environment. In short, parameters associated with an occurrence of the communication symbol can be set by the environment and transferred to the plant model, or set by the plant model and transferred to the environment. The environment is no longer represented by a simple function, but becomes an active process that may react to the information from the plant. Thus, plants are modeled as open cybernetic systems, sending information to and receiving information from the environment.

In order to describe open L-systems in more detail, we need to recall the rudiments of L-systems with turtle interpretation. Our presentation is reproduced from [45].

An L-system is a parallel rewriting system operating on branching structures represented as *bracketed strings* of symbols with associated numerical parameters, called *modules*. Matching pairs of square brackets enclose branches. Simulation begins with an initial string called the *axiom*, and proceeds in a sequence of discrete *derivation steps*. In each step, *rewriting rules* or *productions* replace all modules in the predecessor string by successor modules. The applicability of a production depends on a predecessor's context (in context-sensitive L-systems), values of parameters (in productions guarded by conditions), and on random factors (in stochastic L-systems). Typically, a production has the format:

$$id : lc < pred > rc : cond \rightarrow succ : prob$$

where id is the production identifier (label), lc, $pred$, and rc are the left context, the strict predecessor, and the right context, $cond$ is the condition, $succ$ is the successor, and $prob$ is the probability of production application. The strict predecessor and the successor are the only mandatory fields. For example, the L-system given below consists of axiom ω and three productions p_1, p_2, and p_3.

ω: A(1)B(3)A(5)
p_1: A(x) $\rightarrow$A(x+1) : 0.4
p_2: A(x) $\rightarrow$B(x–1) : 0.6
p_3: A(x) < B(y) > A(z) : y < 4 $\rightarrow$B(x+z)[A(y)]

The stochastic productions p_1 and p_2 replace module $A(x)$ by either $A(x+1)$ or $B(x-1)$, with probabilities equal to 0.4 and 0.6, respectively. The context-sensitive production p_3 replaces a module $B(y)$ with left context $A(x)$ and right context $A(z)$ by module $B(x+z)$ supporting branch $A(y)$. The application of this production is guarded by condition $y < 4$. Consequently, the first derivation step may have the form:

$$\text{A(1)B(3)A(5)} \Rightarrow \text{A(2)B(6)[A(3)]B(4)}$$

It was assumed that, as a result of random choice, production p_1 was applied to the module $A(1)$, and production p_2 to the module $A(5)$. Production p_3 was applied to the module $B(3)$, because it occurred with the required left and right context, and the condition $3 < 4$ was true.

In the L-systems presented as examples we also use several additional constructs (*cf.* [29, 44]):

- Productions may include statements assigning values to local variables. These statements are enclosed in curly braces and separated by semicolons.
- The L-systems may also include arrays. References to array elements follow the syntax of C; for example, $MaxLen[order]$.
- The list of productions is ordered. In the deterministic case, the first matching production applies. In the stochastic case, the set of all matching productions is established, and one of them is chosen according to the specified probabilities.

For details of the L-system syntax see [29, 43, 46].

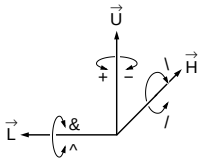


Figure 3: Controlling the turtle in three dimensions

In contrast to the parallel application of productions in each derivation step, the interpretation of the resulting strings proceeds sequentially, with reserved modules acting as commands to a LOGO-style turtle [46]. At any point of the string, the *turtle state* is characterized by a position vector $\vec{P}$ and three mutually perpendicular orientation vectors $\vec{H}$, $\vec{U}$, and $\vec{L}$, indicating the turtle's heading, the up direction, and the direction to the left (Figure 3). Module F causes the turtle to draw a line in the current direction. Modules $+$, $-$, &, $\wedge$, / and \ rotate the turtle around one of the vectors $\vec{H}, \vec{U}$, or $\vec{L}$, as shown in Figure 3. The length of the line and the magnitude of the rotation angle can be given globally or specified as parameters of individual modules. During the interpretation of branches, the opening square bracket pushes the current position and orientation of the turtle on a stack, and the closing bracket restores the turtle to the position and orientation popped from the stack. A special interpretation is reserved for the module %, which cuts a branch by erasing all symbols in the string from the point of its occurrence to the end of the branch [29]. The meaning of many symbols depends on the context in which they occur; for example, $+$ and $-$ denote arithmetic operators as well as modules that rotate the turtle.

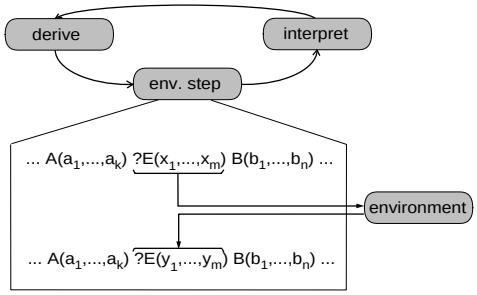


Figure 4: Information flow during the simulation of a plant interacting with the environment, implemented using an open L-system

The turtle interpretation of L-systems described above was designed to visualize models in a postprocessing step, with no effect on the L-system operation. Position and orientation of the turtle are important, however, while considering environmental phenomena, such as collisions with obstacles and exposure to light. The environmentally-sensitive extension of L-systems makes these attributes accessible during the rewriting process [45]. The generated string is interpreted after each derivation step, and turtle attributes found during the interpretation are returned as parameters to reserved *query modules*. Syntactically, the query modules have the form $?X(x, y, z)$, where $X = P, H, U,$ or L. Depending on the actual symbol X, the values of parameters x, y, and z represent a position or an orientation vector.

Open L-systems are a generalization of this concept. *Communication modules* of the form $?E(x_1, \ldots, x_m)$ are used both to send and receive environmental information represented by the values of parameters $x_1, \ldots, x_m$ (Figure 4). To this end, the string resulting from a derivation step is scanned from left to right to determine the state of the turtle associated with each symbol. This phase is similar to the graphical interpretation of the string, except that the results need not be visualized. Upon encountering a communication symbol, the plant process creates and sends a message to the environment including all or a part of the following information:

- the address (position in the string) of the communication module (mandatory field needed to identify this module when a reply comes from the environment),
- values of parameters x_i,
- the state of the turtle (coordinates of the position and orientation

vector, as well as some other attributes, such as current line width),

- the type and parameters of the module following the communication module in the string (not used in the examples discussed in this paper).

The exact message format is defined in a *communication specification file*, shared between the programs modeling the plant and the environment (Figure 2). Consequently, it is possible to include only the information needed in a particular model in the messages sent to the environment. Transfer of the last message corresponding to the current scan of the string is signaled by a reserved end-of-transmission message, which may be used by the environmental process to start its operation.

The messages output by the plant modeling program are transferred to the process that simulates the environment using an interprocess communication mechanism provided by the underlying operating system (a pair of UNIX pipes or shared memory with access synchronized using semaphores, for example). The environment processes that information and returns the results to the plant model using messages in the following format:

- the address of the target communication module,
- values of parameters y_i carrying the output from the environment.

The plant process uses the received information to set parameter values in the communication modules (Figure 4). The use of addresses makes it possible to send replies only to selected communication modules. Details of the mapping of messages received by the plant process to the parameters of the communication modules are defined in the communication specification file.

After all replies generated by the environment have been received (a fact indicated by an end-of-transmission message sent by the environment), the resulting string may be interpreted and visualized, and the next derivation step may be performed, initiating another cycle of the simulation.

Note that, by preceding every symbol in the string with a communication module it is possible to pass complete information about the model to the environment. Usually, however, only partial information about the state of a plant is needed as input to the environment. Proper placement of communication modules in the model, combined with careful selection of the information to be exchanged, provide a means for keeping the amount of transferred information at a manageable level.

We will illustrate the operation of open L-systems within the context of complete models of plant-environment interactions, using examples motivated by actual biological problems.

5 A MODEL OF BRANCH TIERS

Background. Apical meristems, located at the endpoints of branches, are engines of plant development. The apices grow, contributing to the elongation of branch segments, and from time to time divide, spawning the development of new branches. If all apices divided periodically, the number of apices and branch segments would increase exponentially. Observations of real branching structures show, however, that the increase in the number of segments is less than exponential [8]. Honda and his collaborators modeled several hypothetical mechanisms that may control the extent of branching in order to prevent overcrowding [7, 33] (see also [4]). One of the models [33], supported by measurements and earlier simulations of the tropical tree *Terminalia catappa* [19], assumes an exogenous interaction mechanism. *Terminalia* branches form horizontal tiers, and the model is limited to a single tier, treated as a two-dimensional structure. In this case, the competition for light effectively amounts to collision detection between the apices and leaf clusters. We reproduce this model as the simplest example illustrating the methodology proposed in this paper.

Communication specification. The plant communicates with the environment using communication modules of the form $?E(x)$. Messages sent to the environment include the turtle position and the value of parameter x, interpreted as the vigor of the corresponding apex. On this basis, the environmental process determines the fate of each apex. A parameter value of $x = 0$ returned to the plant indicates that the development of the corresponding branch will be terminated. A value of $x = 1$ allows for further branching.

The model of the environment. The environmental process considers each apex or non-terminal node of the developing tier as the center of a circular leaf cluster, and maintains a list of all clusters present. New clusters are added in response to messages received from the plant. All clusters have the same radius ρ, specified in the environmental data file (*cf.* Figure 2). In order to determine the fate of the apices, the environment compares apex positions with leaf cluster positions, and authorizes an apex to grow if it does not fall into an existing leaf cluster, or if it falls into a cluster surrounding an apex with a smaller vigor value.

The plant model. The plant model is expressed as an open L-system. The values of constants are taken from [33].

```
#define r1 0.94      /* contraction ratio and vigor 1 */
#define r2 0.87      /* contraction ratio and vigor 2 */
#define α1 24.4      /* branching angle 1 */
#define α2 36.9      /* branching angle 2 */
#define φ 138.5      /* divergence angle */
ω: –(90)[F(1)?E(1)A(1)]+(φ)[F(1)/?E(1)A(1)]
   +(φ)[F(1)?E(1)A(1)]+(φ)[F(1)/?E(1)A(1)]
   +(φ)[F(1)?E(1)A(1)]

p1: ?E(x) < A(v) : x == 1 →
        [+(α2)F(v*r2)?E(r2)A(v*r2)] –(α1)F(v*r1)/?E(r1)A(v*r1)
p2: ?E(x) → ε
```

The axiom ω specifies the initial structure as a whorl of five branch segments F. The divergence angle φ between consecutive segments is equal to $138.5°$. Each segment is terminated by a communication symbol $?E$ followed by an apex A. In addition, two branches include module $/$, which changes the directions at which subsequent branches will be issued (left *vs.* right) by rotating the apex $180°$ around the segment axis.

Production p_1 describes the operation of the apices. If the value of parameter x returned by a communication module $?E$ is not 1, the associated apex will remain inactive (do nothing). Otherwise the apex will produce a pair of new branch segments at angles α_1 and α_2 with respect to the mother segment. Constants r_1 and r_2 determine the lengths of the daughter segments as fractions of the length of their mother segment. The values r_1 and r_2 are also passed to the process simulating the environment using communication modules $?E$. Communication modules created in the previous derivation step are no longer needed and are removed by production p_2.

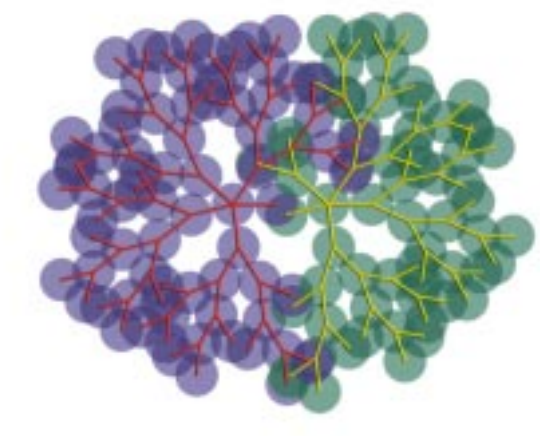


Figure 5: Competition for space between two tiers of branches simulated using the Honda model

Simulation. Figure 5 illustrates the competition for space between two tiers developing next to each other. The extent of branching in each tier is limited by collisions between its apices and its own or the neighbor's leaf clusters. The limited growth of each structure in the direction of its neighbor illustrates the phenomenon of *morphological plasticity*, or adaptation of the form of plants to their environment [17].

6 A MODEL OF FORAGING IN CLONAL PLANTS

Background. Foraging (propagation) patterns in clonal plants provide another excellent example of response to crowding. A clonal plant spreads by means of horizontal stem segments (*spacers*), which form a branching structure that grows along the ground and connects individual plants (*ramets*) [3]. Each ramet consists of a leaf supported by an upright stem and one or more buds, which may give rise to further spacers and ramets. Their gradual death, after a certain amount of time, causes gradual separation of the whole structure (the *clone*) into independent parts.

Following the surface of the soil, clonal plants can be captured using models operating in two dimensions [5], and in that respect resemble *Terminalia* tiers. We propose a model of a hypothetical plant that responds to favorable environmental conditions (high local intensity of light) by more extensive branching and reduced size of leaves (allowing for more dense packing of ramets). It has been inspired by a computer model of clover outlined by Bell [4], the analysis of responses of clonal plants to the environment presented by Dong [17], and the computer models and descriptions of vegetative multiplication of plants involving the death of intervening connections by Room [47].

Communication specification. The plant sends messages to the environment that include turtle position and two parameters associated with the communications symbol, $?E(type, x)$. The first parameter is equal to 0, 1, or 2, and determines the type of exchanged information as follows:

- The message $?E(0, x)$ represents a request for the light intensity (irradiance [14]) at the position of the communication module. The environment responds by setting x to the intensity of incoming light, ranging from 0 (no light) to 1 (full light).
- The message $?E(1, x)$ notifies the environment about the creation of a ramet with a leaf of radius x at the position of the communication module. No output is generated by the environment in response to this message.
- The message $?E(2, x)$ notifies the environment about the death of a ramet with a leaf of radius x at the position of the communication module. Again, no output is generated by the environment.

The model of the environment. The purpose of the environment process is to determine light intensity at the locations requested by the plant. The ground is divided into patches (specified as a raster image using a paint program), with different light intensities assigned to each patch. In the absence of shading, these intensities are returned by the environmental process in response to messages of type 0. To consider shading, the environment keeps track of the set of ramets, adding new ramets in response to a messages of type 1, and deleting dead ramets in response to messages of type 2. If a sampling point falls in an area occupied by a ramet, the returned light intensity is equal to 0 (leaves are assumed to be opaque, and located above the sampling points).

The plant model. The essential features of the plant model are specified by the following open L-system.

```
#define α 45              /* branching angle */
#define MinLight 0.1      /* light intensity threshold */
#define MaxAge 20         /* lifetime of ramets and spacers */
#define Len 2.0           /* length of spacers */
#define Prob_B(x)         (0.12+x*0.42)
#define Prob_R(x)         (0.03+x*0.54)
#define Radius(x)         (sqrt(15–x*5)/π)
ω:   A(1)?E(0,0)

p1:  A(dir) > ?E(0,x) : x >= MinLight
                  → R(x)B(x,dir)F(Len,0)A(–dir)?E(0,0)
p2:  A(dir) > ?E(0,x) : x < MinLight  → ε

p3:  B(x,dir) → [+(α*dir)F(Len,0)A(–dir)?E(0,0)] : Prob_B(x)
p4:  B(x,dir) → ε : 1–Prob_B(x)

p5:  R(x) → [@o(Radius(x),0)?E(1,Radius(x))] : Prob_R(x)
p6:  R(x) → ε : 1–Prob_R(x)

p7:  @o(radius,age): age < MaxAge  → @o(radius,age+1)
p8:  @o(radius,age): age == MaxAge  → ?E(2,radius)

p9:  F(len,age): age < MaxAge  → F(len,age+1)
p10: F(len,age): age == MaxAge  → f(len)

p11: ?E(type,x) → ε
```

The initial structure specified by the axiom ω consists of an apex A followed by the communication module $?E$. If the intensity of light x reaching an apex is insufficient (below the threshold $MinLight$), the apex dies (production p_2). Otherwise, the apex creates a ramet initial R (*i.e.*, a module that will yield a ramet), a branch initial B, a spacer F, and a new apex A terminated by communication module $?E$ (production p_1). The parameter dir, valued either 1 or -1, controls the direction of branching. Parameters of the spacer module specify its length and age.

A branch initial B may create a lateral branch with its own apex A and communication module $?E$ (production p_3), or it may die and disappear from the system (production p_4). The probability of survival is an increasing linear function $Prob_B$ of the light intensity x that has reached the mother apex A in the previous derivation step. A similar stochastic mechanism describes the production of a ramet by the ramet initial R (productions p_5 and p_6), with the probability of ramet formation controlled by an increasing linear function $Prob_R$. The ramet is represented as a circle $@o$; its radius is a decreasing function $Radius$ of the light intensity x. As in the case of spacers, the second parameter of a ramet indicates its age, initially set to 0. The environment is notified about the creation of the ramet using a communication module $?E$.

The subsequent productions describe the aging of spacers (p_7) and ramets (p_9). Upon reaching the maximum age $MaxAge$, a ramet is removed from the system and a message notifying the environment about this fact is sent by the plant (p_8). The death of the spacers is simulated by replacing spacer modules F with invisible line segments f of the same length. This replacement maintains the relative position of the remaining elements of the structure. Finally, production p_{11} removes communication modules after they have performed their tasks.

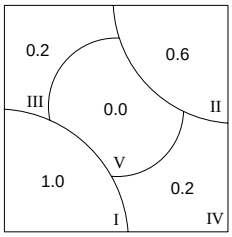


Figure 6: Division of the ground into patches

Simulations. Division of the ground into patches used in the simulations is shown in Figure 6. Arabic numerals indicate the intensity of incoming light, and Roman numerals identify each patch. The development of a clonal plant assuming this division is illustrated in Figure 7. As an extension of the basic model discussed above, the length of the spacers and the magnitude of the branching angle have been varied using random functions with a normal distribution. Ramets have been represented as trifoliate leaves.

The development begins with a single ramet located in relatively good (light intensity 0.6) patch II at the top right corner of the growth area (Figure 7, step 9 of the simulation). The plant propagates through the unfavorable patch III without producing many branches and leaves (step 26), and reaches the best patch I at the bottom left corner (step 39). After quickly spreading over this patch (step 51), the plant searches for further favorable areas (step 62). The first attempt to reach patch II fails (step 82). The plant tries again, and this time succeeds (steps 101 and 116). Light conditions in patch II are not sufficient, however, to sustain the continuous presence of the plant (step 134). The colony disappears (step 153) until the patch is reached again by a new wave of propagation (steps 161 and 182).

The sustained occupation of patch I and the repetitive invasion of patch II represent an emerging behavior of the model, difficult to predict without running simulations. Variants of this model, including other branching architectures, responses to the environment, and layouts of patches in the environment, would make it possible to analyze different foraging strategies of clonal plants. A further extension could replace the empirical assumptions regarding plant responses with a more detailed simulation of plant physiology (for example, including production of photosynthates and their transport and partition between ramets). Such physiological models could provide insight into the extent to which the foraging patterns optimize plants' access to resources [17].

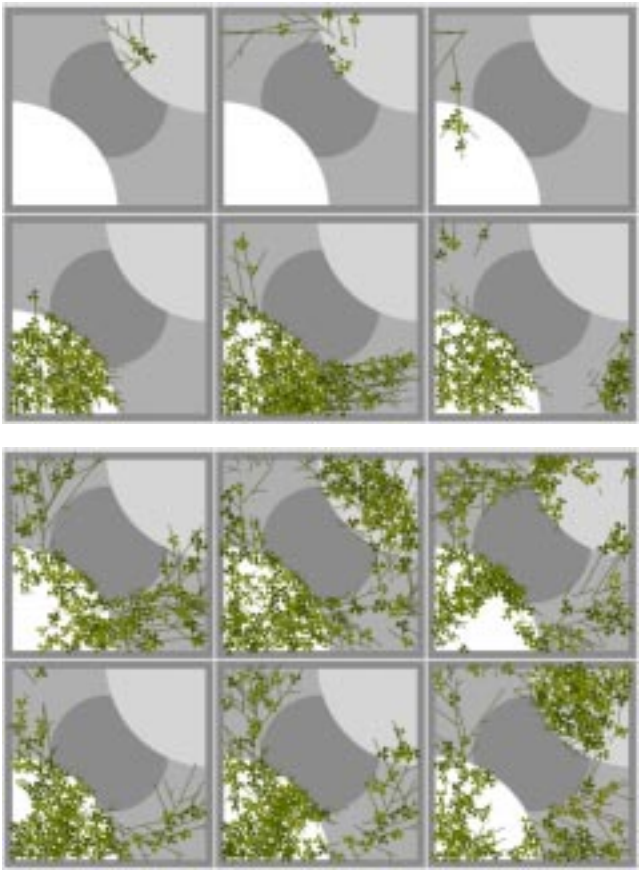


Figure 7: Development of a hypothetical clonal plant simulated using an extension of L-system 3. The individual images represent structures generated in 9, 26, 39, 51, 62, and 82 derivation steps (top), followed by structures generated in 101, 116, 134, 153, 161, and 182 steps (bottom).

7 A MODEL OF ROOT DEVELOPMENT

Background. The development of roots provides many examples of complex interactions with the environment, which involve mechanical properties, chemical reactions, and transport mechanisms in the soil. In particular, the main root and the rootlets absorb water from the soil, locally changing its concentration (volume of water per unit volume of soil) and causing water motion from water-rich to depleted regions [24]. The tips of the roots, in turn, follow the gradient of water concentration [12], thus adapting to the environment modified by their own activities.

Below we present a simplified implementation of the model of root development originally proposed by Clausnitzer and Hopmans [12]. We assume a more rudimentary mechanism of water transport, namely diffusion in a uniform medium, as suggested by Liddell and Hansen [37]. The underlying model of root architecture is similar to that proposed by Diggle [16]. For simplicity, we focus on model operation in two-dimensions.

Communication specification. The plant interacts with the environment using communication modules $?E(c, \theta)$ located at the apices of the root system. A message sent to the environment includes the turtle position $\vec{P}$, the heading vector $\vec{H}$, the value of

parameter c representing the requested (optimal) water uptake, and the value of parameter θ representing the tendency of the apex to follow the gradient of water concentration. A message returned to the plant specifies the amount of water actually received by the apex as the value of parameter c, and the angle biasing direction of further growth as the value of θ.

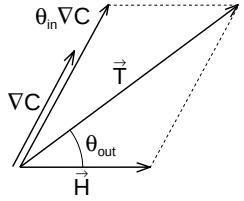


Figure 8: Definition of the biasing angle θ_{out}

The model of the environment. The environment maintains a field C of water concentrations, represented as an array of the amounts of water in square sampling areas. Water is transported by diffusion, simulated numerically using finite differencing [41]. The environment responds to a request for water from an apex located in an area (i, j) by granting the lesser of the values requested and available at that location. The amount of water in the sampled area is then decreased by the amount received by the apex. The environment also calculates a linear combination $\vec{T}$ of the turtle heading vector $\vec{H}$ and the gradient of water concentration ∇C (estimated numerically from the water concentrations in the sampled area and its neighbors), and returns an angle θ between the vectors $\vec{T}$ and $\vec{H}$ (Figure 8). This angle is used by the plant model to bias turtle heading in the direction of high water concentration.

The root model. The open L-system representing the root model makes use of arrays that specify parameters for each branching order (main axis, its daughter axes, *etc.*). The parameter values are loosely based on those reported by Clausnitzer and Hopmans [12].

```
#define N 3                          /* max. branching order + 1 */
Define: { array
Req[N] = {0.1, 0.4, 0.05},           /* requested nutrient intake */
MinReq[N] = {0.01, 0.06, 0.01},      /* minimum nutrient intake */
ElRate[N] = {0.55, 0.25, 0.55},      /* maximum elongation rate */
MaxLen[N] = {200, 5, 0.8},           /* maximum branch length */
Sens[N] = {10, 0, 0},                /* sensitivity to gradient */
Dev[N] = {30, 75, 75},               /* deviation in heading */
Del[N–1] = {30, 60},                 /* delay in branch growth */
BrAngle[N–1] = {90, 90},             /* branching angle */
BrSpace[N–1] = {1, 0.5}              /* distance between branches */
}
ω: A(0,0,0)?E(Req[0],Sens[0])

p1: A(n,s,b) > ?E(c,θ) : (s > MaxLen[n]) || (c < MinReq[n]) → ε
p2: A(n,s,b) > ?E(c,θ) :
        (n >= N–1) || (b < BrSpace[n]) {h=c/Req[n]*ElRate[n];}
        → +(nran(θ,Dev[n]))F(h) A(n,s+h,b+h)?E(Req[n],Sens[n])
p3: A(n,s,b) > ?E(c,θ) :
        (n < N–1) && (b >= BrSpace[n]) {h=c/Req[n]*ElRate[n];}
        → +(nran(θ,Dev[n]))B(n,0)F(h)
          /(180)A(n,s+h,h)?E(Req[n],Sens[n])
p4: B(n,t) : t < Del[n]  → B(n,t+1)
p5: B(n,t) : t >= Del[n]
        → [+(BrAngle[n])A(n+1,0,0)?E(Req[n+1],Sens[n+1])]
p6: ?E(c,θ)  → ε
```

The development starts with an apex A followed by a communication module $?E$. The parameters of the apex represent the branch order (0 for the main axis, 1 for its daughter axes, *etc.*), current axis length, and distance (along the axis) to the nearest branching point.

Figure 9: A two-dimensional model of a root interacting with water in soil. Background colors represent concentrations of water diffusing in soil (blue: high, black: low). The initial and boundary values have been set using a paint program.

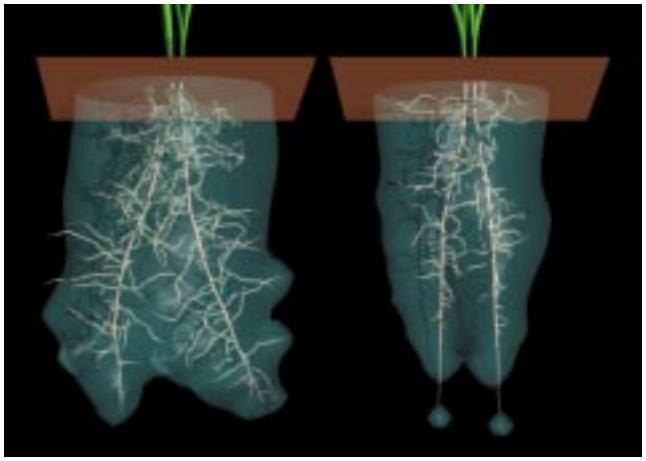


Figure 10: A three-dimensional extension of the root model. Water concentration is visualized by semi-transparent iso-surfaces [55] surrounding the roots. As a result of competition for water, the roots grow away from each other. The divergence between their main axes depends on the spread of the rootlets, which grow faster on the left then on the right.

Productions p_1 to p_3 describe possible fates of the apex as described below.

If the length s of a branch axis exceeds a predefined maximum value $MaxLen[n]$ characteristic to the branch order n, or the amount of water c received by the apex is below the required minimum $MinReq[n]$, the apex dies, terminating the growth of the axis (production p_1).

If the branch order n is equal to the maximum value assumed in the model ($N - 1$), or the distance b to the closest branching point on the axis is less than the threshold value $BrSpace[n]$, the apex adds a new segment F to the axis (production p_2). The length h of F is the product of the nominal growth increment $ElRate[n]$ and the ratio of the amount of water received by the apex c to the amount requested $Req[n]$. The new segment is rotated with respect to its predecessor by an angle $nran(\theta, Dev[n])$, where $nran$ is a random function with a normal distribution. The mean value θ, returned by the environment, biases the direction of growth towards regions of

higher water concentration. The standard deviation $Dev[n]$ characterizes the tendency of the root apex to change direction due to various factors not included explicitly in the model.

If the branch order n is less than the maximum value assumed in the model ($N-1$), and the distance b to the closest branching point on the axis is equal to or exceeds the threshold value $BrSpace[n]$, the apex creates a new branch initial B (production p_3). Other aspects of apex behavior are the same as those described by production p_2.

After the delay of $Del[n]$ steps (production p_4), the branch initial B is transformed into an apex A followed by the communication module $?E$ (production p_5), giving rise to a new root branch. Production p_6 removes communication modules that are no longer needed.

Simulations. A sample two-dimensional structure obtained using the described model is shown in Figure 9. The apex of the main axis follows the gradient of water concentration, with small deviations due to random factors. The apices of higher-order axes are not sensitive to the gradient and change direction at random, with a larger standard deviation. The absorption of water by the root and the rootlets decreases water concentration in their neighborhood; an effect that is not fully compensated by water diffusion from the water-rich areas. Low water concentration stops the development of some rootlets before they have reached their potential full length.

Figure 10 presents a three-dimensional extension of the previous model. As a result of competition for water, the main axes of the roots diverge from each other (left). If their rootlets grow more slowly, the area of influence of each root system is smaller and the main axes are closer to each other (right). This behavior is an emergent property of interactions between the root modules, mediated by the environment.

8 MODELS OF TREES CONTROLLED BY LIGHT

Background. Light is one of the most important factors affecting the development of plants. In the essentially two-dimensional structures discussed in Section 5, competition for light could be considered in a manner similar to collision detection between leaves and apices. In contrast, competition for light in three-dimensional structures must be viewed as long-range interaction. Specifically, shadows cast by one branch may affect other branches at significant distances.

The first simulations of plant development that take the local light environment into account are due to Greene [25]. He considered the entire sky hemisphere as a source of illumination and computed the amount of light reaching specific points of the structure by casting rays towards a number of points on the hemisphere. Another approach was implemented by Kanamaru *et al.* [35], who computed the amount of light reaching a given sampling point by considering it a center of projection, from which all leaf clusters in a tree were projected on a surrounding hemisphere. The degree to which the hemisphere was covered by the projected clusters indicated the amount of light received by the sampling point. In both cases, the models of plants responded to the amount and the direction of light by simulating heliotropism, which biased the direction of growth towards the vector of the highest intensity of incoming light. Subsequently, Chiba *et al.* extended the models by Kanamaru *et al.* using more involved tree models that included a mechanism simulating the flow of hypothetical endogenous information within the tree [10, 11]. A biologically better justified model, formulated in terms of production and use of photosynthates by a tree, was proposed by Takenaka [52]. The amount of light reaching leaf clusters was calculated by sampling a sky hemisphere, as in the work by Greene. Below we reproduce the main features of the Takenaka's model using the formalism of open L-systems. Depending on the underlying tree architecture, it can be applied to synthesize images of deciduous and coniferous trees. We focus on a deciduous tree, which requires a slightly smaller number of productions.

Communication specification. The plant interacts with the environment using communication modules $?E(r)$. A message sent by the plant includes turtle position $\vec{P}$, which represents the center of a spherical leaf cluster, and the value of parameter r, which represents the cluster's radius. The environment responds by setting r to the flux [14] of light from the sky hemisphere, reaching the cluster.

The model of the environment. Once all messages describing the current distribution of leaves on a tree have been received, the environmental process computes the extent of the tree in the x, y, and z directions, encompasses the tree in a tight grid ($32 \times 32 \times 32$ voxels in our simulations), and allocates leaf clusters to voxels to speed up further computations. The environmental process then estimates the light flux Φ from the sky hemisphere reaching each cluster (shadows cast by the branches are ignored). To this end, the hemisphere is represented by a set of directional light sources S (9 in the simulations). The flux densities (radiosities) B of the sources approximate the non-uniform distribution of light from the sky (*cf.* [52]). For each leaf cluster L_i and each light source S, the environment determines the set of leaf clusters L_j that may shade L_i. This is achieved by casting a ray from the center of L_i in the direction of S and testing for intersections with other clusters (the grid accelerates this process). In order not to miss any clusters that may partially occlude L_i, the radius of each cluster L_j is increased by the maximum value of cluster radius r_{max}.

To calculate the flux reaching cluster L_i, this cluster and all clusters L_j that may shade it according to the described tests are projected on a plane P perpendicular to the direction of light from the source S. The impact of a cluster L_j on the flux Φ reaching cluster L_i is then computed according to the formula:

$$\Phi = (A_i - A_{ij})B + A_{ij}\tau B$$

where A_i is the area of the projection of L_i on P, A_{ij} is the area of the intersection between projections of L_i and L_j, and τ is the light transmittance through leaf cluster L_j (equal to 0.25 in the simulations). If several clusters L_j shade L_i, their influences are multiplied. The total flux reaching cluster L_i is calculated as the sum of the fluxes received from each light source S.

The plant model. In addition to the communication module $?E$, the plant model includes the following types of modules:

- Apex $A(vig, del)$. Parameter vig represents vigor, which determines the length of branch segments (internodes) and the diameter of leaf clusters produced by the apex. Parameter del is used to introduce a delay, needed for propagating products of photosynthesis through the tree structure between consecutive stages of development (years).
- Leaf $L(vig, p, age, del)$. Parameters denote the leaf radius vig, the amount of photosynthates produced in unit time according to the leaf's exposure to light p, the number of years for which a leaf has appeared at a given location age, and the delay del, which plays the same role as in the apices.
- Internode $F(vig)$. Consistent with the turtle interpretation, the parameter vig indicates the internode length.

- Branch width symbol $!(w, p, n)$, also used to carry the endogenous information flow. The parameters determine: the width of the following internode w, the amount of photosynthates reaching the symbol's location p, and the number of terminal branch segments above this location n.

The corresponding L-system is given below.

```
#define φ  137.5      /* divergence angle */
#define α0  5         /* direction change - no branching */
#define α1  20        /* branching angle - main axis */
#define α2  32        /* branching angle - lateral axis */
#define W  0.02       /* initial branch width */
#define VD  0.95      /* apex vigor decrement */
#define Del  30       /* delay */
#define LS  5         /* how long a leaf stays */
#define LP  8         /* full photosynthate production */
#define LM  2         /* leaf maintenance */
#define PB  0.8       /* photosynthates needed for branching */
#define PG  0.4       /* photosynthates needed for growth */
#define BM  0.32      /* branch maintenance coefficient */
#define BE  1.5       /* branch maintenance exponent */
#define Nmin  25      /* threshold for shedding */
Consider: ?E[]!L      /* for context matching */
ω:  !(W,1,1)F(2)L(1,LP,0,0)A(1,0)[!(0,0,0)]!(W,0,1)

p1: A(vig,del) : del<Del  → A(vig,del+1)
p2: L(vig,p,age,del) : (age<LS)&&(del<Del–1)  → L(vig,p,age,del+1)
p3: L(vig,p,age,del) : (age<LS)&&(del==Del–1)
                → L(vig,p,age,del+1)?E(vig*0.5)
p4: L(vig,p,age,del) > ?E(r) : (age<LS) && (r*LP>=LM)
                && (del == Del) → L(vig,LP*r–LM,age+1,0)
p5: L(vig,p,age,del) > ?E(r) : ((age == LS)||(r*LP<=LM))
                && (del == Del) → L(0,0,LS,0)
p6: ?E(r) < A(vig,del) : r*LP–LM>PB {vig=vig*VD;}
                → /(φ)[+(α2)!(W,–PB,1)F(vig)L(vig,LP,0,0)A(vig,0)
                      [!(0,0,0)]!(W,0,1)]
                  –(α1)!(W,0,1)F(vig)L(vig,LP,0,0)/A(vig,0)
p7: ?E(r) < A(vig,del) : r*LP–LM > PG {vig=vig*VD;}
                → /(φ)–(α0)[!(0,0,0)]
                  !(W,–PG,1)F(vig)L(vig,LP,0,0)A(vig,0)
p8: ?E(r) < A(vig,del) : r*LP–LM <= PG  → A(vig,0)
p9: ?E(r)  → ε
p10: !(w0,p0,n0) > L(vig,pL,age,del) [!(w1,p1,n1)]!(w2,p2,n2) :
                {w=(w1^ 2+w2^ 2)^ 0.5; p=p1+p2+pL–BM*(w/W)^ BE}
                (p>0) || (n1+n2 >=Nmin)  → !(w,p,n1+n2)
p11: !(w0,p0,n0) > L(vig,pL,age,del) [!(w1,p1,n1)]!(w2,p2,n2)
                → !(w0,0,0)L%
```

The simulation starts with a structure consisting of a branch segment F, supporting a leaf L and an apex A (axiom ω). The first branch width symbol ! defines the segment width. Two additional symbols ! following the apex create "virtual branches," needed to provide proper context for productions p_{10} and p_{11}. The tree grows in stages, with the delay of $Del + 1$ derivation steps between consecutive stages introduced by production p_1 for the apices and p_2 for the leaves. Immediately before each new growth stage, communication symbols are introduced to inform the environment about the location and size of the leaf clusters (p_3). If the flux r returned by the environment results in the production of photosynthates $r * LP$ exceeding the amount LM needed to maintain a cluster, it remains in the structure (p_4). Otherwise it becomes a liability to the tree and dies (p_5). Another condition to production p_5 prevents a leaf from occupying the same location for more than LS years.

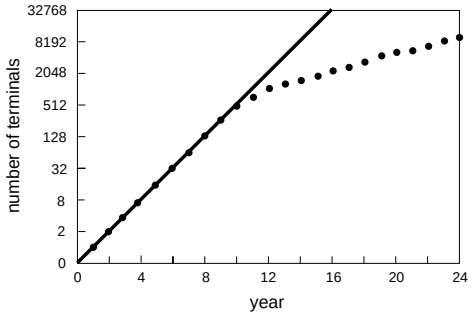


Figure 11: The number of terminal branch segments resulting from unrestricted bifurcation of apices (continuous line), compared to the number of segments generated in a simulation (isolated points)

The flux r also determines the fate of the apex, captured by productions p_6 to p_8. If the amount of photosynthates $r * LP - LM$ transported from the nearby leaf exceeds a threshold value PB, the apex produces two new branches (p_6). The second parameter in the first branch symbol ! is set to $-PB$, to subtract the amount of photosynthates used for branching from the amount that will be transported further down. The length of branch segments vig is reduced with respect to the mother segment by a predefined factor VD, reflecting a gradual decrease in the vigor of apices with age. The branch width modules ! following the first apex A are introduced to provide context required by productions p_{10} and p_{11}, as in the axiom.

If the amount of photosynthates $r * LP - LM$ transported from the leaf is insufficient to produce new branches, but above the threshold PG, the apex adds a new segment F to the current branch axis without creating a lateral branch (p_7). Again, a virtual branch containing the branch width symbol ! is being added to provide context for productions p_{10} and p_{11}.

If the amount of photosynthates is below PG, the apex remains dormant (p_8). Communication modules no longer needed are removed from the structure (p_9).

Production p_{10} captures the endogenous information flow from leaves and terminal branch segments to the base of the tree. First, it determines the radius w of the mother branch segment as a function of the radii w_1 and w_2 of the supported branches:

$$w = \sqrt{w_1^2 + w_2^2}.$$

Thus, a cross section of the mother segment has an area equal to the sum of cross sections of the supported segments, as postulated in the literature [40, 46]. Next, production p_{10} calculates the flow p of photosynthates into the mother segment. It is defined as the sum of the flows p_L, p_1 and p_2 received from the associated leaf L and from both daughter branches, decreased by the amount $BM * (w/W)^{BE}$ representing the cost of maintaining the mother segment. Finally, production p_10 calculates the number of terminal branch segments n supported by the mother segment as the sum of the numbers of terminal segments supported by the daughter branches, n_1 and n_2.

Production p_{10} takes effect if the flow p is positive (the branch is not a liability to the tree), or if the number n of supported terminals is above a threshold N_{min}. If these conditions are not satisfied,

Figure 12: A tree model with branches competing for access to light, shown without the leaves

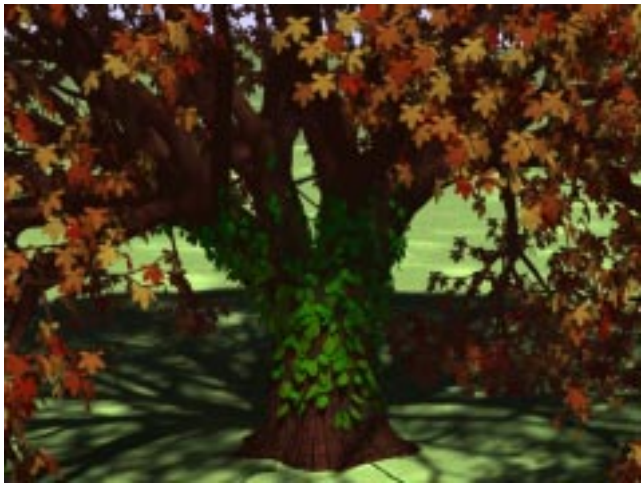


Figure 13: A climbing plant growing on the tree from the previous figure

production p_{11} removes (sheds) the branch from the tree using the cut symbol %.

Simulations. The competition for light between tree branches is manifested by two phenomena: reduced branching or dormancy of apices in unfavorable local light conditions, and shedding of

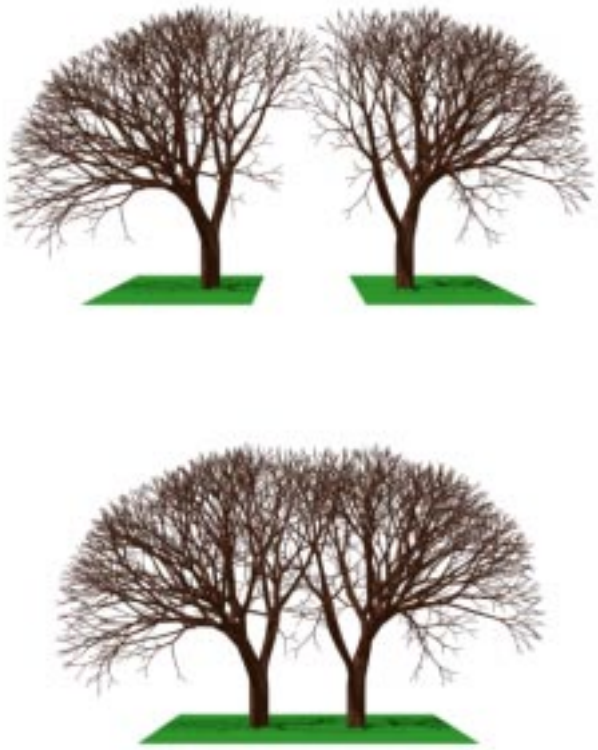


Figure 14: A model of deciduous trees competing for light. The trees are shown in the position of growth (top) and moved apart (bottom) to reveal the adaptation of crown geometry to the presence of the neighbor tree.

branches which do not receive enough light to contribute to the whole tree. Both phenomena limit the extent of branching, thus controlling the density of the crown. This property of the model is supported by the simulation results shown in Figure 11. If the growth was unlimited (production p_6 was always chosen over p_7 and p_8), the number of terminal branch segments would double every year. Due to the competition for light, however, the number of terminal segments observed in an actual simulation increases more slowly. For related statistics using a different tree architecture see [52].

A tree image synthesized using an extension of the presented model is shown in Figure 12. The key additional feature is a gradual reduction of the branching angle of a young branch whose sister branch has been shed. As the result, the remaining branch assumes the role of the leading shoot, following the general growth direction of its supporting segment. Branch segments are represented as texture-mapped generalized cylinders, smoothly connected at the branching points (*cf.* [6]). The bark texture was created using a paint program.

As an illustration of the flexibility of the modeling framework presented in this paper, Figure 13 shows the effect of seeding a hypothetical climbing plant near the same tree. The plant follows the surface of the tree trunk and branches, and avoids excessively dense colonization of any particular area. Thus, the model integrates sev-

Figure 15: A model of coniferous trees competing for light. The trees are shown in the position of growth (top) and moved apart (bottom).

eral environmentally-controlled phenomena: the competition of tree branches for light, the following of surfaces by a climbing plant, and the prevention of crowding as discussed in Section 6. Leaves were modeled using cubic patches (*cf.* [46]).

In the simulations shown in Figure 14 two trees described by the same set of rules (younger specimens of the tree from Figure 12) compete for light from the sky hemisphere. Moving the trees apart after they have grown reveals the adaptation of their crowns to the presence of the neighbor tree. This simulation illustrates both the necessity and the possibility of incorporating the adaptive behavior into tree models used for landscape design purposes.

The same phenomenon applies to coniferous trees, as illustrated in Figure 15. The tree model is similar to the original model by Takenaka [52] and can be viewed as consisting of approximately horizontal tiers (as discussed in Section 5) produced in sequence by the apex of the tree stem. The lower tiers are created first and therefore potentially can spread more widely then the younger tiers higher up (the *phase effect* [46]). This pattern of development is affected by the presence of the neighboring tree: the competition for light prevents the crowns from growing into each other.

The trees in Figure 15 retain branches that do not receive enough light. In contrast, the trees in the stand presented in Figure 16 shed branches that do not contribute photosynthates to the entire tree,

Figure 16: Relationship between tree form and its position in a stand.

using the same mechanism as described for the deciduous trees. The resulting simulation reveals essential differences between the shape of the tree crown in the middle of a stand, at the edge, or at the corner. In particular, the tree in the middle retains only the upper part of its crown. In lumber industry, the loss of lower branches is usually a desirable phenomenon, as it reduces knots in the wood and the amount of cleaning that trees require before transport. Simulations may assist in choosing an optimal distance for planting trees, where self-pruning is maximized, yet there is sufficient space between trees too allow for unimpeded growth of trunks in height and diameter.

9 CONCLUSIONS

In this paper, we introduced a framework for the modeling and visualization of plants interacting with their environment. The essential elements of this framework are:

- a system design, in which the plant and the environment are treated as two separate processes, communicating using a standard interface, and
- the language of open L-systems, used to specify plant models that can exchange information with the environment.

We demonstrated the operation of this framework by implementing models that capture collisions between branches, the propagation of clonal plants, the development of roots in soil, and the development of tree crowns competing for light. We found that the proposed framework makes it possible to easily create and modify models spanning a wide range of plant structures and environmental processes. Simulations of the presented phenomena were fast enough to allow interactive experimentation with the models (Table 1).

There are many research topics that may be addressed using the simulation and visualization capabilities of the proposed framework. They include, for instance:

- Fundamental analysis of the role of different forms of information flow in plant morphogenesis (in particular, the relationship between endogenous and exogenous flow). This is a continuation

Fig.	Number of		Derivation		Time[a]	
	branch segments	leaf clusters	steps	yrs	sim.	render.
5	138	140	5	5	1 s	1 s
7	786	229	182	NA	50 s	2 s
9	4194	34[b]	186	NA	67 s	3 s
10	37228	448[b]	301	NA	15 min	70 s
12	22462	19195	744	24	22 min	13 s[c]
15	13502	3448	194	15	4 min	8 s[d]

[a] Simulation and rendering using OpenGL on a 200MHz/64MB Indigo^2 Extreme
[b] active apices
[c] without generalized cylinders and texture mapping
[d] branching structure without needles

Table 1: Numbers of primitives and simulation/rendering times for generating and visualizing selected models

of the research pioneered by Bell [4] and Honda *et al.* [7, 33].

- Development of a comprehensive plant model describing the cycling of nutrients from the soil through the roots and branches to the leaves, then back to the soil in the form of substances released by fallen leaves.
- Development of models of specific plants for research, crop and forest management, and for landscape design purposes. The models may include environmental phenomena not discussed in this paper, such as the global distribution of radiative energy in the tree crowns, which affects the amount of light reaching the leaves and the local temperature of plant organs.

The presented framework itself is also open to further research. To begin, the precise functional specification of the environment, implied by the design of the modeling framework, is suitable for a formal analysis of algorithms that capture various environmental processes. This analysis may highlight tradeoffs between time, memory, and communication complexity, and lead to programs matching the needs of the model to available system resources in an optimal manner.

A deeper understanding of the spectrum of processes taking place in the environment may lead to the design of a mini-language for environment specification. Analogous to the language of L-systems for plant specification, this mini-language would simplify the modeling of various environments, relieving the modeler from the burden of low-level programming in a general-purpose language. Fleischer and Barr's work on the specification of environments supporting collisions and reaction-diffusion processes [20] is an inspiring step in this direction.

Complexity issues are not limited to the environment, but also arise in plant models. They become particularly relevant as the scope of modeling increases from individual plants to groups of plants and, eventually, entire plant communities. This raises the problem of selecting the proper level of abstraction for designing plant models, including careful selection of physiological processes incorporated into the model and the spatial resolution of the resulting structures.

The complexity of the modeling task can be also addressed at the level of system design, by assigning various components of the model (individual plants and aspects of the environment) to different components of a distributed computing system. The communication structure should then be redesigned to accommodate information transfers between numerous processes within the system.

In summary, we believe that the proposed modeling methodology and its extensions will prove useful in many applications of plant modeling, from research in plant development and ecology to landscape design and realistic image synthesis.

Acknowledgements

We would like to thank Johannes Battjes, Campbell Davidson, Art Diggle, Heinjo During, Michael Guzy, Naoyoshi Kanamaru, Bruno Moulia, Zbigniew Prusinkiewicz, Bill Remphrey, David Reid, and Peter Room for discussions and pointers to the literature relevant to this paper. We would also like to thank Bruno Andrieu, Mark Hammel, Jim Hanan, Lynn Mercer, Chris Prusinkiewicz, Peter Room, and the anonymous referees for helpful comments on the manuscript. Most images were rendered using the ray tracer `rayshade` by Craig Kolb. This research was sponsored by grants from the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] AGRAWAL, P. The cell programming language. *Artificial Life 2*, 1 (1995), 37–77.

[2] ARVO, J., AND KIRK, D. Modeling plants with environment-sensitive automata. In *Proceedings of Ausgraph'88* (1988), pp. 27 – 33.

[3] BELL, A. *Plant form: An illustrated guide to flowering plants.* Oxford University Press, Oxford, 1991.

[4] BELL, A. D. The simulation of branching patterns in modular organisms. *Philos. Trans. Royal Society London, Ser. B 313* (1986), 143–169.

[5] BELL, A. D., ROBERTS, D., AND SMITH, A. Branching patterns: the simulation of plant architecture. *Journal of Theoretical Biology 81* (1979), 351–375.

[6] BLOOMENTHAL, J. Modeling the Mighty Maple. Proceedings of SIGGRAPH '85 (San Francisco, California, July 22-26, 1985), in *Computer Graphics*, 19, 3 (July 1985), pages 305–311, ACM SIGGRAPH, New York, 1985.

[7] BORCHERT, R., AND HONDA, H. Control of development in the bifurcating branch system of *Tabebuia rosea*: A computer simulation. *Botanical Gazette 145*, 2 (1984), 184–195.

[8] BORCHERT, R., AND SLADE, N. Bifurcation ratios and the adaptive geometry of trees. *Botanical Gazette 142*, 3 (1981), 394–401.

[9] CHEN, S. G., CEULEMANS, R., AND IMPENS, I. A fractal based *Populus* canopy structure model for the calculation of light interception. *Forest Ecology and Management* (1993).

[10] CHIBA, N., OHKAWA, S., MURAOKA, K., AND MIURA, M. Visual simulation of botanical trees based on virtual heliotropism and dormancy break. *The Journal of Visualization and Computer Animation 5* (1994), 3–15.

[11] CHIBA, N., OHSHIDA, K., MURAOKA, K., MIURA, M., AND SAITO, N. A growth model having the abilities of growth-regulations for simulating visual nature of botanical trees. *Computers and Graphics 18*, 4 (1994), 469–479.

[12] CLAUSNITZER, V., AND HOPMANS, J. Simultaneous modeling of transient three-dimensional root growth and soil water flow. *Plant and Soil 164* (1994), 299–314.

[13] COHEN, D. Computer simulation of biological pattern generation processes. *Nature 216* (October 1967), 246–248.

[14] COHEN, M., AND WALLACE, J. *Radiosity and realistic image synthesis*. Academic Press Professional, Boston, 1993. With a chapter by P. Hanrahan and a foreword by D. Greenberg.

[15] DE REFFYE, P., HOULLIER, F., BLAISE, F., BARTHELEMY, D., DAUZAT, J., AND AUCLAIR, D. A model simulating above- and below-ground tree architecture with agroforestry applications. *Agroforestry Systems 30* (1995), 175–197.

[16] DIGGLE, A. J. ROOTMAP - a model in three-dimensional coordinates of the structure and growth of fi brous root systems. *Plant and Soil 105* (1988), 169–178.

[17] DONG, M. *Foraging through morphological response in clonal herbs*. PhD thesis, Univeristy of Utrecht, October 1994.

[18] FISHER, J. B. How predictive are computer simulations of tree architecture. *International Journal of Plant Sciences 153 (Suppl.)* (1992), 137–146.

[19] FISHER, J. B., AND HONDA, H. Computer simulation of branching pattern and geometry in *Terminalia* (Combretaceae), a tropical tree. *Botanical Gazette 138*, 4 (1977), 377–384.

[20] FLEISCHER, K. W., AND BARR, A. H. A simulation testbed for the study of multicellular development: The multiple mechanisms of morphogenesis. In *Artificial Life III*, C. G. Langton, Ed. Addison-Wesley, Redwood City, 1994, pp. 389–416.

[21] FORD, E. D., AVERY, A., AND FORD, R. Simulation of branch growth in the *Pinaceae*: Interactions of morphology, phenology, foliage productivity, and the requirement for structural support, on the export of carbon. *Journal of Theoretical Biology 146* (1990), 15–36.

[22] FORD, H. Investigating the ecological and evolutionary signifi cance of plant growth form using stochastic simulation. *Annals of Botany 59* (1987), 487–494.

[23] FRIJTERS, D., AND LINDENMAYER, A. A model for the growth and fbwering of *Aster novae-angliae* on the basis of table (1,0)L-systems. In *L Systems*, G. Rozenberg and A. Salomaa, Eds., Lecture Notes in Computer Science 15. Springer-Verlag, Berlin, 1974, pp. 24–52.

[24] GARDNER, W. R. Dynamic aspects of water availability to plants. *Soil Science 89*, 2 (1960), 63–73.

[25] GREENE, N. Voxel space automata: Modeling with stochastic growth processes in voxel space. Proceedings of SIGGRAPH '89 (Boston, Mass., July 31–August 4, 1989), in *Computer Graphics* 23, 4 (August 1989), pages 175–184, ACM SIGGRAPH, New York, 1989.

[26] GREENE, N. Detailing tree skeletons with voxel automata. SIGGRAPH '91 Course Notes on Photorealistic Volume Modeling and Rendering Techniques, 1991.

[27] GUZY, M. R. A morphological-mechanistic plant model formalized in an object-oriented parametric L-system. Manuscript, USDA-ARS Salinity Laboratory, Riverside, 1995.

[28] HANAN, J. Virtual plants —Integrating architectural and physiological plant models. In *Proceedings of ModSim 95* (Perth, 1995), P. Binning, H. Bridgman, and B. Williams, Eds., vol. 1, The Modelling and Simulation Society of Australia, pp. 44–50.

[29] HANAN, J. S. *Parametric L-systems and their application to the modelling and visualization of plants*. PhD thesis, University of Regina, June 1992.

[30] HART, J. W. *Plant tropisms and other growth movements*. Unwin Hyman, London, 1990.

[31] HERMAN, G. T., AND ROZENBERG, G. *Developmental systems and languages*. North-Holland, Amsterdam, 1975.

[32] HONDA, H. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology 31* (1971), 331–338.

[33] HONDA, H., TOMLINSON, P. B., AND FISHER, J. B. Computer simulation of branch interaction and regulation by unequal fbw rates in botanical trees. *American Journal of Botany 68* (1981), 569–585.

[34] KAANDORP, J. *Fractal modelling: Growth and form in biology*. Springer-Verlag, Berlin, 1994.

[35] KANAMARU, N., CHIBA, N., TAKAHASHI, K., AND SAITO, N. CG simulation of natural shapes of botanical trees based on heliotropism. *The Transactions of the Institute of Electronics, Information, and Communication Engineers J75-D-II*, 1 (1992), 76–85. In Japanese.

[36] KURTH, W. *Growth grammar interpreter GROGRA 2.4: A software tool for the 3-dimensional interpretation of stochastic, sensitive growth grammars in the context of plant modeling. Introduction and reference manual*. Forschungszentrum Waldökosysteme der Universität Göttingen, Göttingen, 1994.

[37] LIDDELL, C. M., AND HANSEN, D. Visualizing complex biological interactions in the soil ecosystem. *The Journal of Visualization and Computer Animation 4* (1993), 3–12.

[38] LINDENMAYER, A. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology 18* (1968), 280–315.

[39] LINDENMAYER, A. Developmental systems without cellular interaction, their languages and grammars. *Journal of Theoretical Biology 30* (1971), 455–484.

[40] MACDONALD, N. *Trees and networks in biological models*. J. Wiley & Sons, New York, 1983.

[41] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. *Numerical recipes in C: The art of scientific computing. Second edition*. Cambridge University Press, Cambridge, 1992.

[42] PRUSINKIEWICZ, P. Visual models of morphogenesis. *Artificial Life 1*, 1/2 (1994), 61–74.

[43] PRUSINKIEWICZ, P., HAMMEL, M., HANAN, J., AND MĚCH, R. Visual models of plant development. In *Handbook of formal languages*, G. Rozenberg and A. Salomaa, Eds. Springer-Verlag, Berlin, 1996. To appear.

[44] PRUSINKIEWICZ, P., AND HANAN, J. L-systems: From formalism to programming languages. In *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, G. Rozenberg and A. Salomaa, Eds. Springer-Verlag, Berlin, 1992, pp. 193–211.

[45] PRUSINKIEWICZ, P., JAMES, M., AND MĚCH, R. Synthetic topiary. Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994), pages 351–358, ACM SIGGRAPH, New York, 1994.

[46] PRUSINKIEWICZ, P., AND LINDENMAYER, A. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[47] ROOM, P. M. 'Falling apart' as a lifestyle: the rhizome architecture and population growth of *Salvinia molesta*. *Journal of Ecology 71* (1983), 349–365.

[48] ROOM, P. M., MAILLETTE, L., AND HANAN, J. Module and metamer dynamics and virtual plants. *Advances in Ecological Research 25* (1994), 105–157.

[49] ROZENBERG, G. T0L systems and languages. *Information and Control 23* (1973), 357–381.

[50] SACHS, T., AND NOVOPLANSKY, A. Tree from: Architectural models do not suffi ce. *Israel Journal of Plant Sciences 43* (1995), 203–212.

[51] SIPPER, M. Studying artifi cial life using a simple, general cellular model. *Artificial Life 2*, 1 (1995), 1–35.

[52] TAKENAKA, A. A simulation model of tree architecture development based on growth response to local light environment. *Journal of Plant Research 107* (1994), 321–330.

[53] ULAM, S. On some mathematical properties connected with patterns of growth of fi gures. In *Proceedings of Symposia on Applied Mathematics* (1962), vol. 14, American Mathematical Society, pp. 215–224.

[54] WEBER, J., AND PENN, J. Creation and rendering of realistic trees. Proceedings of SIGGRAPH '95 (Los Angeles, California, August 6–11, 1995), pages 119–128, ACM SIGGRAPH, New York, 1995.

[55] WYVILL, G., MCPHEETERS, C., AND WYVILL, B. Data structure for soft objects. *The Visual Computer 2*, 4 (February 1986), 227–234.

Realistic modeling and rendering of plant ecosystems

Oliver Deussen[1] Pat Hanrahan[2] Bernd Lintermann[3] Radomír Měch[4]
Matt Pharr[2] Przemyslaw Prusinkiewicz[4]

[1] Otto-von-Guericke University of Magdeburg
[2] Stanford University
[3] The ZKM Center for Art and Media Karlsruhe
[4] The University of Calgary*

Abstract

Modeling and rendering of natural scenes with thousands of plants poses a number of problems. The terrain must be modeled and plants must be distributed throughout it in a realistic manner, reflecting the interactions of plants with each other and with their environment. Geometric models of individual plants, consistent with their positions within the ecosystem, must be synthesized to populate the scene. The scene, which may consist of billions of primitives, must be rendered efficiently while incorporating the subtleties of lighting in a natural environment.

We have developed a system built around a pipeline of tools that address these tasks. The terrain is designed using an interactive graphical editor. Plant distribution is determined by hand (as one would do when designing a garden), by ecosystem simulation, or by a combination of both techniques. Given parametrized procedural models of individual plants, the geometric complexity of the scene is reduced by *approximate instancing*, in which similar plants, groups of plants, or plant organs are replaced by instances of representative objects before the scene is rendered. The paper includes examples of visually rich scenes synthesized using the system.

CR categories: I.3.7 **[Computer Graphics]**: Three-Dimensional Graphics and Realism, I.6.3 **[Simulation and Modeling]**: Applications, J.3 **[Life and Medical Sciences]**: Biology.

Keywords: realistic image synthesis, modeling of natural phenomena, ecosystem simulation, self-thinning, plant model, vector quantization, approximate instancing.

1 INTRODUCTION

Synthesis of realistic images of terrains covered with vegetation is a challenging and important problem in computer graphics. The challenge stems from the visual complexity and diversity of the modeled scenes. They include natural ecosystems such as forests or grasslands, human-made environments, for instance parks and gardens, and intermediate environments, such as lands recolonized by vegetation after forest fires or logging. Models of these ecosystems have a wide range of existing and potential applications, including computer-assisted landscape and garden design, prediction and visualization of the effects of logging on the landscape, visualization of models of ecosystems for research and educational purposes, and synthesis of scenes for computer animations, drive and flight simulators, games, and computer art.

Beautiful images of forests and meadows were created as early as 1985 by Reeves and Blau [50] and featured in the computer animation *The Adventures of André and Wally B.* [34]. Reeves and Blau organized scene modeling as a sequence of steps: specification of a terrain map that provides the elevation of points in the scene, interactive or procedural placement of vegetation in this terrain, modeling of individual plants (grass and trees), and rendering of the models. This general scheme was recently followed by Chiba *et al.* [8] in their work on forest rendering, and provides the basis for commercial programs devoted to the synthesis of landscapes [2, 49].

The complexity of nature makes it necessary to carefully allot computing resources — CPU time, memory, and disk space — when recreating natural scenes with computer graphics. The size of the database representing a scene during the rendering is a particularly critical item, since the amount of geometric data needed to represent a detailed outdoor scene is more than can be represented on modern computers. Consequently, a survey of previous approaches to the synthesis of natural scenes reflects the quest for a good tradeoff between the realism of the images and the amount of resources needed to generate them.

The scenes synthesized by Reeves and Blau were obtained using (structured) particle systems, with the order of one million particles per tree [50]. To handle large numbers of primitive elements contributing to the scene, the particle models of individual trees were generated procedurally and rendered sequentially, each model discarded as soon as a tree has been rendered. Consequently, the size of memory needed to generate the scene was proportional to the number of particles in a single tree, rather than the total number of particles in the scene. This approach required approximate shading calculations, since the detailed information about the neighborhood trees was not available. Approximate shading also reduced the time needed to render the scenes.

Another approach to controlling the size of scene representation is the reduction of visually unimportant detail. General methods for achieving this reduction have been the subject of intense research (for a recent example and further references see [24]), but the results do not easily apply to highly branching plant structures. Consequently, Weber and Penn [63] introduced a heuristic multiresolution representation specific to trees, which allows for reducing

* Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4 ($pwp@cpsc.ucalgary.ca$)

Published in the Proceedings of SIGGRAPH '98 (Orlando, Florida, July 19–24, 1998). In *Computer Graphics* Proceedings, Annual Conference Series, 1998, ACM SIGGRAPH, New York, pp. 275–286.

the number of geometric primitives in the models that occupy only a small portion on the screen. A multiresolution representation of botanical scenes was also explored by Marshall *et al.* [35], who integrated polygonal representations of larger objects with tetrahedral approximations of the less relevant parts of a scene.

A different strategy for creating visually complex natural scenes was proposed by Gardner [17]. In this case, the terrain and the trees were modeled using a relatively small number of geometric primitives (quadric surfaces). Their perceived complexity resulted from procedural textures controlling the color and the transparency of tree crowns. In a related approach, trees and other plants were represented as texture-mapped flat polygons (for example, see [49]). This approach produced visible artifacts when the position of the viewpoint was changed. A more accurate image-based representation was introduced by Max [37], who developed an algorithm for interpolating between precomputed views of trees. A multiresolution extension of this method, taking advantage of the hierarchical structure of the modeled trees, was presented in [36]. Shade *et al.* described a hybrid system for walkthroughs that uses a combination of geometry and textured polygons [53].

Kajiya and Kay [26] introduced volumetric textures as an alternative paradigm for overcoming the limitations of texture-mapped polygons. A method for generating terrains with volumetric textures representing grass and trees was proposed by Neyret [40, 41]. Chiba *et al.* [8] removed the deformations of plants caused by curvatures of the underlying terrain by allowing texels to intersect.

The use of volumetric textures limits the memory or disk space needed to represent a scene, because the same texel can be re-applied to multiple areas. The same idea underlies the oldest approach to harnessing visually complex scenes, object instancing [59]. According to the paradigm of instancing, an object that appears several times in a scene (possibly resized or deformed by an affine transformation) is defined only once, and its different occurrences (instances) are specified by affine transformations of the prototype. Since the space needed to represent the transformations is small, the space needed to represent an entire scene depends primarily on the number and complexity of *different* objects, rather than the number of their instances. Plants are particularly appealing objects of instancing, because repetitive occurrences can be found not only at the level of plant species, but also at the level of plant organs and branching structures. This leads to compact hierarchical data structures conducive to fast ray tracing, as discussed by Kay and Kajiya [27], and Snyder and Barr [56]. Hart and DeFanti [20, 21] further extended the paradigm of instancing from hierarchical to recursive (self-similar) structures. All the above papers contain examples of botanical scenes generated using instancing.

The complexity of natural scenes makes them not only difficult to render, but also to specify. Interactive modeling techniques, available in commercial packages such as Alias/Wavefront Studio 8 [1], focus on the direct manipulation of a relatively small number of surfaces. In contrast, a landscape with plants may include many millions of individual surfaces — representing stems, leaves, flowers, and fruits — arranged into complex branching structures, and further organized in an ecosystem. In order to model and render such scenes, we employ the techniques summarized below.

Multilevel modeling and rendering pipeline. Following the approach initiated by Reeves and Blau [50], we decompose the process of image synthesis into stages: modeling of the terrain, specification of plant distribution, modeling of the individual plants, and rendering of the entire scene. Each of these stages operates at a different level of abstraction, and provides a relatively high-level input for the next stage. Thus, the modeler is not concerned with plant distribution when specifying the terrain, and plant distribution is determined (interactively or algorithmically) without considering details of the individual plants. This is reminiscent of the simulations of flocks of birds [51], models of flower heads with phyllotactic patterns [16], and models of organic structures based on morphogenetic processes [14], where simulations were performed using geometrically simpler objects than those used for visualization. Blumberg and Galyean extended this paradigm to *multi-level* direction of autonomous animated agents [5]. In an analogous way, we apply it to multi-level modeling.

Open system architecture. By clearly specifying the formats of inputs and outputs for each stage of the pipeline, we provide a framework for incorporating independently developed modules into our system. This open architecture makes it possible to augment the complexity of the modeled scenes by increasing the range of the available modules, and facilitates experimentation with various approaches and algorithms.

Procedural models. As observed by Smith [55], procedural models are often characterized by a significant *data-base amplification*, which means that they can generate complex geometric structures from small input data sets. We benefit from this phenomenon by employing procedural models in all stages of the modeling pipeline.

Approximate instancing. We use object instancing as the primary paradigm for reducing the size of the geometric representation of the rendered scenes. To increase the degree of instancing, we cluster scene components (plants and their parts) in their parameter spaces, and approximate all objects within a given cluster with instances of a single representative object. This idea was initially investigated by Brownbill [7]; we extend it further by applying vector quantization (*c.f.* [18]) to find the representative objects in multidimensional parameter spaces.

Efficient rendering. We use memory- and time-efficient rendering techniques: decomposition of the scenes into subscenes that are later composited [12], ray tracing with instancing and a support for rendering many polygons [56], and memory-coherent ray tracing [43] with instancing.

By employing these techniques, we have generated scenes with up to 100,000 detailed plant models. This number could be increased even further, since none of the scenes required more than 150MB to store. However, with 100,000 plants, each plant is visible on average only in 10 pixels of a 1K $\times$ 1K image. Consequently, we seem to have reached the limits of useful scene complexity, because the level of visible detail is curbed by the size and resolution of the output device.

2 SYSTEM ARCHITECTURE

The considerations presented in the previous section led us to the modular design of our modeling and rendering system `EcoSys`, shown schematically in Figure 1.

The modeling process begins with the specification of a terrain. For this purpose, we developed an interactive editor `TerEdit`, which integrates a number of terrain modeling techniques (Section 3). Its output, a *terrain data* file, includes the altitudes of a grid of points superimposed on the terrain, normal vectors indicating the local slope of the terrain, and optional information describing environmental conditions, such as soil humidity.

The next task is to determine plant distribution on a given terrain. We developed two techniques for this purpose: visual editing of plant densities and simulation of plant interactions within an ecosystem

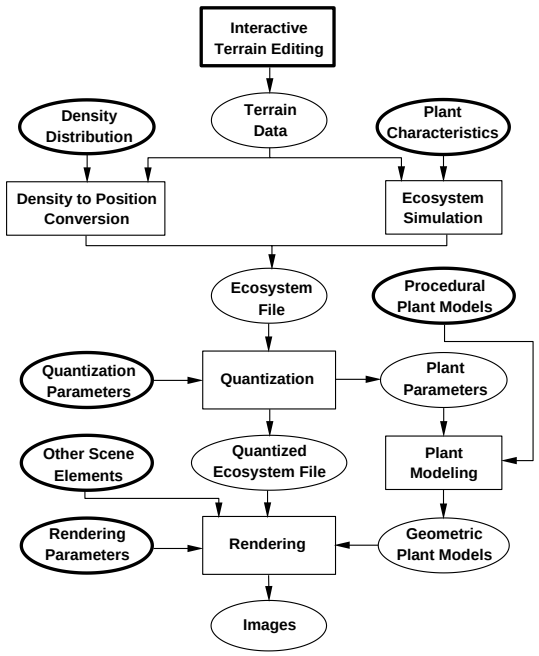


Figure 1: Architecture of the scene synthesis system. Bold frames indicate interactive programs and input files specified by the user.

(Section 4). The editing approach is particularly well suited to model environments designed by people, for example orchards, gardens, or parks. The user specifies the distribution of plant densities using a paint program. To convert this information into positions of individual plants, we developed the program `densedis` based on a half-toning algorithm: each dot becomes a plant. We can also specify positions of individual plants explicitly; this is particularly important in the synthesis of scenes that include detailed views of individual plants in the foreground.

To define plant distribution in natural environments, such as forests or meadows, we apply an ecosystem simulation model. Its input consists of terrain data, ecological characteristics of plant species (for example, annual or perennial growth and reproduction cycle, preference for wet or dry soil, and shade tolerance) and, optionally, the initial distribution of plants. The growth of plants is simulated accounting for competition for space, sunlight, resources in the soil, aging and death, seed distribution patterns, *etc*. We perform these simulations using the L-system-based plant modeling program `cpfg` [47], extended with capabilities for simulating interactions between plants and their environments [39]. To allow for simulations involving thousands of plants, we use their simplified geometrical representations, which are subsequently replaced by detailed plant models for visualization purposes.

Specification of a plant distribution may involve a combination of interactive and simulation techniques. For example, a model of an orchard may consist of trees with explicitly specified positions and weeds with positions determined by a simulation. Conversely, the designer of a scene may wish to change its aspects after an ecological simulation for aesthetic reasons. To allow for these operations, both `densedis` and `cpfg` can take a given plant distribution as input for further processing.

Plant distribution, whether determined interactively or by ecosystem simulation, is represented in an *ecosystem* file. It contains the information about the type, position and orientation of each plant (which is needed to assemble the final scene), and parameters of individual plants (which are needed to synthesize their geometric models).

Since we wish to render scenes that may include thousands of plants, each possibly with many thousands of polygons, the creation and storage of a separate geometric plant model for each plant listed in the ecosystem file is not practical. Consequently, we developed a program `quantv` that clusters plants in their parameter space and determines a representative plant for each cluster (Section 6). The algorithm performs quantization adaptively, thus the result depends on the characteristics of plants in the ecosystem. The quantization process produces two outputs: a *plant parameter* file, needed to create geometric models of the representative plants, and a *quantized ecosystem* file, which specifies positions and orientations of the instances of representative plants throughout the scene.

We employ two modeling programs to create the representative plants: the interactive plant modeler `xfrog` [10, 32, 33] and the L-system-based simulator `cpfg` [39, 47]. These programs input parametrized *procedural plant models* and generate specific *geometric plant models* according to the values in the plant parameter file (Section 5). To reduce the amount of geometric data, we extended the concept of instancing and quantization to components of plants. Thus, if a particular plant or group of plants has several parts (such as branches, leaves, or flowers) that are similar in their respective parameter spaces, we replace all occurrences of these parts with instances of a representative part.

Finally, the ecosystem is rendered. The input for rendering consists of the quantized ecosystem file and the representative plant models. Additional information may include geometry of the terrain and human-made objects, such as houses or fences. In spite of the quantization and instancing, the resulting scene descriptions may still be large. We experimented with three renderers to handle this complexity (Section 7). One renderer, called `fshade`, decomposes the scene into sub-scenes that are rendered individually and composited to form final images. Unfortunately, separating the scene into sub-scenes makes it impossible to properly capture global illumination effects. To alleviate this problem, we use the ray-tracer `rayshade` [29], which offers support for instancing and time-efficient rendering of scenes with many polygons, as long as the scene description fits in memory. When the scene description exceeds the available memory, we employ the memory-efficient ray-tracer `toro` [43], extended with a support for instancing.

In the following sections we describe the components of the `EcoSys` modeling pipeline in more detail. In Section 8, we present examples that illustrate the operation of the system as a whole.

3 TERRAIN SPECIFICATION

We begin the modeling of a scene with the specification of a terrain. The goal of this step is to determine elevation data, local orientations of the terrain, and additional characteristics, such as the water content in the soil, which may affect the type and vigor of plants at different locations.

Terrain data may have several sources. Representations of real terrains are available, for example, from the U.S. Geological Survey [30]. Several techniques have also been developed for creating synthetic terrains. They include: hand-painted height maps [65], methods for generating fractal terrains (reviewed in [38]), and models based on the simulation of soil erosion [28, 38].

In order to provide detailed control over the modeled terrain while taking advantage of the data amplification of fractal methods [55],

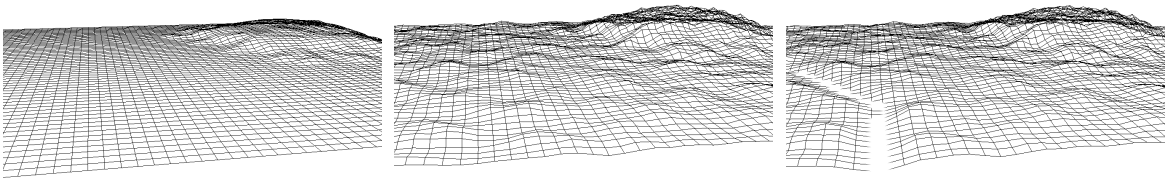


Figure 2: Three stages in creating a terrain: after loading a height map painted by hand (left), with hills added using noise synthesis (middle), and with a stream cut using the stream mask (right).

we designed and implemented an interactive terrain editing system `TerEdit`, which combines various techniques in a procedural manner. Terrain editing consists of *operations*, which modify the terrain geometry and have the spatial scope limited by *masks*. A similar paradigm is used in Adobe Photoshop [9], where *selections* can be used to choose an arbitrary subset of an image to edit.

We assume that masks have values between zero and one, allowing for smooth blending of the effects of operations. Both masks and operations can depend on the horizontal coordinates and the altitude of the points computed so far. Thus, it is possible to have masks that select terrain above some altitude or operations that are functions of the current altitude. The user's editing actions create a pipeline of operations with associated masks; to compute the terrain altitude at a point, the stages of this pipeline are evaluated in order. Undo and redo operations are easily supported by removing and re-adding operations from the pipeline and re-evaluating the terrain.

Examples of editing operations include translation, scaling, nonlinear scaling, and algorithmic synthesis of the terrain. The synthesis algorithm is based on noise synthesis [38], which generates realistic terrains by adding multiple scales of Perlin's noise function [42]. The user can adjust a small number of parameters that control the overall roughness of the terrain, the rate of change in roughness across the surface of the terrain, and the frequency of the noise functions used. Noise synthesis allows terrain to be easily evaluated at a single point, without considering the neighboring points; this makes it possible to have operations that act locally. Another advantage of noise synthesis is efficiency of evaluation; updating the wireframe terrain view (based on 256×256 samples of the region of interest) after applying an operation typically takes under a second. On a multiprocessor machine, where terrain evaluation is multi-threaded, the update time is not noticeable.

The editor provides a variety of masks, including ones that select rectangular regions of terrain from a top view, masks that select regions based on their altitude, and masks defined by image files. One of the most useful masks is designed for cutting streams through terrain. The user draws a set of connected line segments over the terrain, and the influence of the mask is based on the minimum distance from a sample point to any of these segments. A spline is applied to smoothly increase the influence of the mask close to the segments. When used with a scaling operation, the terrain inside and near the stream is scaled towards the water level, and nearby terrain is ramped down, while the rest of the terrain is unchanged.

The specification of a terrain using `TerEdit` is illustrated in Figure 2. In the first snapshot, the hill in the far corner was defined by loading in a height map that had been painted by hand. Next, small hills were added to the entire terrain using noise synthesis. The last image shows the final terrain, after the stream mask was used to cut the path of a stream. A total of five operators were applied to make this terrain, and the total time to model it was approximately fifteen minutes.

Once the elevations have been created, additional parameters of the terrain can be computed as input for ecosystem simulations or a direct source of parameters for plant models. Although the user can interactively paint parameters on the terrain, simulation provides a more sound basis for the modeling of natural ecosystems. Consequently, `TerEdit` incorporates a simulator of rain water flow and distribution in the soil, related to both the erosion algorithm by Musgrave *et al.* [38] and the particle system simulation of water on building facades by Dorsey *et al.* [11]. Water is dropped onto the terrain from above; some is absorbed immediately while the rest flows downhill and is absorbed by the soil that it passes through. A sample terrain with the water distribution generated using this approach is shown in Figure 3.

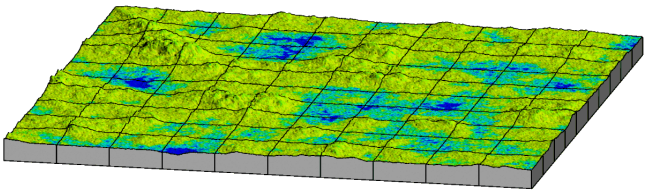


Figure 3: A sample terrain with the water concentration ranging from high (blue) to low (yellow)

4 SPECIFICATION OF PLANT POPULATIONS

The task of populating a terrain with plants can be addressed using methods that offer different tradeoffs between the degree of user control, time needed to specify plant distribution, and biological validity of the results. The underlying techniques can be divided into *space-occupancy* or *individual-based* techniques. This classification is related to paradigms of spatially-explicit modeling in ecology [3, 19], and parallels the distinction between space-based and structure-based models of morphogenesis [44].

The space-occupancy techniques describe the distribution of the *densities* of given plant species over a terrain. In the image synthesis context, this distribution can be be obtained using two approaches:

Explicit specification. The distribution of plant densities is measured in the field (by counting plants that occupy sample plots) or created interactively, for example using a paint program.

Procedural generation. The distributions of plant densities is obtained by simulating interactions between plant populations using an ecological model. The models described in the literature are commonly expressed in terms of *cellular automata* [19] or *reaction-diffusion* processes [23].

The individual-based techniques provide the location and attributes of *individual plants*. Again, we distinguish two approaches:

Explicit specification. Plant positions and attributes represent field data, for example obtained by surveying a real forest [25], or specified interactively by the user.

Procedural generation. Plant positions and attributes are obtained using a *point pattern generation model*, which creates a distribution of points with desired statistical properties [66], or using an *individual-based population model* [13, 58], which is applied to simulate interactions between plants within an ecosystem.

Below we describe two methods for specifying plant distribution that we have developed and implemented as components of `EcoSys`. The first method combines interactive editing of plant densities with a point pattern generation of the distribution of individual plants. The second method employs individual-based ecological simulations.

4.1 Interactive specification of plant populations

To specify a plant population in a terrain, the user creates a set of gray-level images with a standard paint program. These images define the spatial distributions of plant densities and of plant characteristics such as the age and vigor.

Given an image that specifies the distribution of densities of a plant species, positions of individual plants are generated using a halftoning algorithm. We have used the Floyd-Steinberg algorithm [15] for this purpose. Each black pixel describes the position of a plant in the raster representing the terrain. We also have implemented a relaxation method that iteratively moves each plant position towards the center of mass of its Voronoi polygon [6]. This reduces the variance of distances between neighboring plants, which sometimes produces visually pleasing effects.

Once the position of a plant has been determined, its parameters are obtained by referring to the values of appropriate raster images at the same point. These values may control the plant model directly or provide arguments to user-specified mathematical expressions, which in turn control the models. This provides the user with an additional means for flexibly manipulating the plant population.

Operations on raster images make it possible to capture some interactions between plants. For example, if the radius of a tree crown is known, the image representing the projection of the crown on the ground may be combined with user-specified raster images to decrease the density or vigor of plants growing underneath.

4.2 Simulation of ecosystems

Individual-based models of plant ecosystems operate at various levels of abstraction, depending on the accuracy of the representation of individual plants [58]. Since our goal is to simulate complex scenes with thousands of plants, we follow the approach of Firbank and Watkinson [13], and represent plants coarsely as circles positioned in a continuous landscape. Each circle defines the *ecological neighborhood* of the plant in its center, that is the area within which the plant interacts with it neighbors. Biologically motivated rules govern the outcomes of interactions between the intersecting circles. Global behavior of the ecosystem model is an emergent property of a system of many circles.

We implemented the individual-based ecosystem models using the framework of *open L-systems* [39]. Since L-systems operate on branching structures, we represent each plant as a circle located at the end of an invisible positioning line. All lines are connected into a branching structure that spreads over the terrain.

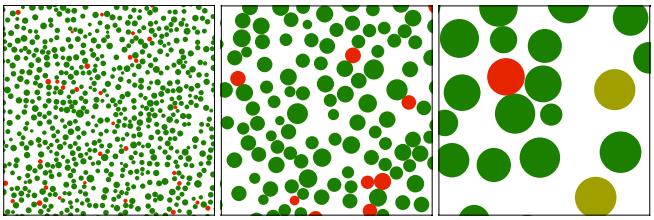


Figure 4: Steps 99, 134, and 164 of a sample simulation of the self-thinning process. Colors represents states of the plants: not dominated (green), dominated (red), and old (yellow). The simulation began with 62,500 plants, placed at random in a square field. Due to the large number of plants, only a part of the field is shown.

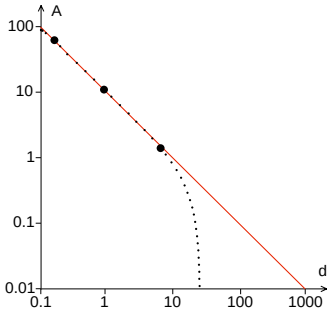


Figure 5: The average area of plants as a function of their density. Small dots represent the results of every tenth simulation step. Large dots correspond to the states of simulation shown in Figure 4.

For example, let us consider a model of plant distribution due to a fundamental phenomenon in plant population dynamics, *self-thinning*. This phenomenon is described by Ricklefs as follows [52, page 339]: "If one plots the logarithm of average plant weight as a function of the logarithm of density, data points fall on a line with a slope of approximately $-\frac{3}{2}$ [called the self-thinning curve]. [...] When seeds are planted at a moderate density, so that the beginning combination of density and average dry weight lies below the self-thinning curve, plants grow without appreciable mortality until the population reaches its self-thinning curve. After that point, the intense crowding associated with further increase in average plant size causes the death of smaller individuals."

Our model of self-thinning is a simplified version of that by Firbank and Watkinson [13]. The simulation starts with an initial set of circles, distributed at random in a square field, and assigned random initial radii from a given interval. If the circles representing two plants intersect, the smaller plant dies and its corresponding circle is removed from the scene. Plants that have reached a limit size are considered old and die as well.

Figure 4 shows three snapshots of the simulation. The corresponding plot shows the average area of the circles as a function of their density (Figure 5). The slope of the self-thinning curve is equal to -1; assuming that mass is proportional to volume, which in turn is proportional to area raised to the power of $-\frac{3}{2}$, the self-thinning curve in the density-mass coordinates would have the slope of $-\frac{3}{2}$. Thus, in spite of its simplicity, our model captures the essential characteristic of plant distribution before and after it has reached the self-thinning curve.

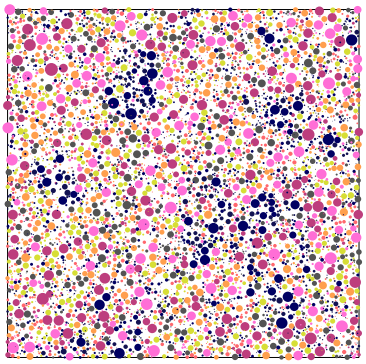


Figure 6: Simulated distribution of eight plant species in a terrain from Figure 3. Colors indicate plant species. Plants with a preference for wet areas are shown in blue.

A slightly more complicated model describes plant distribution in a population of different plant species. Each species is defined by a set of values that determine: (i) the number of new plants added to the field per simulation step, (ii) the maximum size of the plants, (iii) their average growth rate, (iv) the probability of surviving the domination by a plant with a competitive advantage, and (v) a preference for wet or dry areas. An individual plant is characterized by: (i) the species to which it belongs, (ii) its size, and (iii) its vigor. The vigor is a number in the range from 0 to 1, assigned to each plant as a randomized function of water concentration at the plant's location and the plant's preference for wet or dry areas. The competitive ability of a plant is determined as a product of its vigor and its relative size (the ratio between the actual and maximum size). When the circles representing two plants intersect, their competitive abilities are compared. The plant with a smaller competitive ability is dominated by the other plant and may die with the defined probability.

Figure 6 presents the result of a simulation involving a mix of eight plant species growing in a terrain shown in Figure 3. Plants with a preference for wet areas are represented by blue circles. Plants with a preference for dry areas have been assigned other colors. Through the competition between the species, a segregation of plants between the wet and dry areas has emerged.

Similar models can be developed to capture other phenomena that govern the development of plant populations.

5 MODELING OF PLANTS

Interactive editing of plant populations and the simulation of ecosystems determine positions and high-level characteristics of all plants in the modeled scene. On this basis, geometric models of individual plants must now be found.

Recent advances in plant measuring techniques have made it possible to construct a geometric model of a specific plant according to detailed measurements of its structure [54]. Nevertheless, for the purpose of visualizing plants differing by age, vigor, and possibly other parameters, it is preferable to treat geometric models as a product of the underlying procedural models. Construction of such models for computer graphics and biological purposes has been a field of active study, recently reviewed in [45]. Consequently, below we discuss only the issue of model parametrization, that is the incorporation of high-level parameters returned by the population model into the plant models. We consider two different approaches, which reflect different predictive values of *mechanistic* and *descriptive* models [60].

Mechanistic models operate by simulating the processes that control the development of plants over time. They inherently capture how the resulting structure changes with age [46, 47]. If a mechanistic model incorporates environmental inputs, the dependence of the resulting structure on the corresponding environmental parameters is an emergent feature of the model [39]. The model predicts the effect of various combinations of environmental parameters on the structure, and no explicit parametrization is needed. L-systems [47] and their extensions [39] provide a convenient tool for expressing mechanistic models. Within `EcoSys`, mechanistic models have been generated using `cpfg`.

Descriptive models capture plant architecture without simulating the underlying developmental processes. Consequently, they do not have an inherent predictive value. Nevertheless, if a family of geometric models is constructed to capture the key "postures" of a plant at different ages and with different high-level characteristics, we can obtain the in-between geometries by interpolation. This is equivalent to fitting functions that map the set of high-level parameters to the set of lower-level variables present in the model, and can be accomplished by regression [57] (see [48] for an application example). In the special case of plant postures characterized by a single parameter, the interpolation between key postures is analogous to key-framing [62], and can be accomplished using similar techniques. We applied interpolation to parametrize models created using both `xfrog` and `cpfg`.

6 APPROXIMATE INSTANCING

Geometric plant models are often large. A detailed polygonal representation of a herbaceous plant may require over 10MB to store; a scene with one thousand plants (a relatively small number in ecosystem simulations) would require 10GB. One approach for reducing such space requirements is to simplify geometric representations of objects that have a small size on the screen. We incorporated a version of this technique into our system by parameterizing the procedural plant models so that they can produce geometries with different polygonizations of surfaces. However, this technique alone was not sufficient to reduce the amount of data to manageable levels.

Instancing was used successfully in the past for compactly representing complex outdoor scenes (*e.g.* [56]). According to the paradigm of instancing [59], geometric objects that are identical up to affine transformations become instances of one object. To achieve a further reduction in the size of geometric descriptions, we extended the paradigm of instancing to objects that resemble each other, but are not exactly the same. Thus, sets of similar plants are represented by instances of a single representative plant. Furthermore, the hierarchical structure of plant scenes, which may be decomposed into groups of plants, individual plants, branches of different order, plant organs such as leaves and flowers, *etc.*, lends itself to instancing at different levels of the hierarchy. We create hierarchies of instances by quantizing model components in their respective parameter spaces, and reusing them.

Automatic generation of instance hierarchies for plant models expressed using a limited class of L-systems was considered by Hart [20, 21]. His approach dealt only with exact instancing. Brownbill [7] considered special cases of approximate instancing of plants, and analyzed tradeoffs between the size of the geometric models and their perceived distortion (departure from the original

geometry caused by the reduction of diversity between the components). He achieved reductions of the database size ranging from 5:1 to 50:1 with a negligible visual impact on the generated images (a tree and a field of grass). This result is reminiscent of the observation by Smith [55] that the set of random numbers used in stochastic algorithms for generating fractal mountains and particle-system trees can be reduced to a few representative values without significantly affecting the perceived complexity of the resulting images.

We generalize Brownbill's approach by relating it to clustering. Assuming that the characteristics of each plant are described by a vector of real numbers, we apply a clustering algorithm to the set of these vectors in order to find representative vectors. Thus, we reduce the problem of finding representative plants and instancing them to the problem of finding a set of representative points in the parameter space and mapping each point to its representative. We assume that plants with a similar appearance are described by close points in their parameter space; if this is not the case (for example, because the size of a plant is a nonlinear function of its age), we transform the parameter space to become perceptually more linear. We cluster and map plant parts in the same manner as the entire plants.

This clustering and remapping can be stated also in terms of vector quantization [18]: we store a code book of plants and plant parts and, for each input plant, we store a mapping to an object in the code book rather than the plant geometry itself. In computer graphics, vector quantization has been widely used for color image quantization [22]; more recent applications include reduction of memory needs for texture mapping [4] and representing light fields [31].

We use a multi-dimensional clustering algorithm developed by Wan *et al.* [61], which subdivides the hyperbox containing data points by choosing splitting planes to minimize the variance of the resulting clusters of data. We extended this algorithm to include an "importance weight" with each input vector. The weights make it possible to further optimize the plant quantization process, for example by allocating more representative vectors to the plants that occupy a large area of the screen.

7 RENDERING

Rendering natural scenes raises two important questions: (i) dealing with scene complexity, and (ii) simulating illumination, materials and atmospheric effects. Within `EcoSys`, we addressed these questions using two different strategies.

The first strategy is to to split the scene into sub-scenes of manageable complexity, render each of them independently using ray-casting, and composite the resulting RGBαZ images into the final image [12]. The separation of the scene into sub-scenes is a byproduct of the modeling process: both `densedis` and `cpfg` can output the distribution of a single plant species to form a sub-scene. The ray-casting algorithm is implemented in `fshade`, which creates the scene geometry procedurally by invoking the `xfrog` plant modeler at run time. This reduces file I/O and saves disk space compared to storing all of the geometric information for the scene on disk and reading it in while rendering. For example, the poplar tree shown in Figure 16 is 16 KB as a procedural model (plant template), but 6.7 MB in a standard text geometry format.

A number of operations can be applied to the RGBαZ images before they are combined. Image processing operations, such as saturation and brightness adjustments, are often useful. Atmospheric effects can be introduced in a post process, by modifying colors according to the pixel depth. Shadows are computed using shadow maps [64].

The scene separation makes it possible to render the scene quickly and re-render its individual sub-scenes as needed to improve the image. However, complex lighting effects cannot be easily included, since the renderer doesn't have access to the entire scene description at once.

The second rendering strategy is ray tracing. It lacks the capability to easily re-render parts of scenes that have been changed, but makes it possible to include more complex illumination effects. In both ray-tracers that we have used, `rayshade` [29] and `toro` [43], procedural geometry descriptions are expanded into triangle meshes, complemented with a hierarchy of grids and bounding boxes needed to speed up rendering [56]. `Rayshade` requires the entire scene description (object prototypes with their bounding boxes and a hierarchy of instancing transformations) to be kept in memory, otherwise page swapping significantly decreases the efficiency of rendering. In the case of `toro`, meshes are stored on disk; these are read in parts to a memory cache as needed for rendering computations and removed from memory when a prescribed limit amount of memory has been used. Consequently, the decrease in performance when the memory size has been reached is much slower [43]. We have made the straightforward extension of memory-coherent ray-tracing algorithms to manage instanced geometry: along with non-instanced geometry, the instances in the scene are also managed by the geometry cache.

Because rays can be traced that access the entire scene, more complex lighting effects can be included. For example, we have found that attenuating shadow rays as they pass through translucent leaves of tree crowns greatly improves their realism and visual richness.

8 EXAMPLES

We evaluated our system by applying it to create a number of scenes. In the examples presented below, we used two combinations of the modules: (i) ecosystem simulation and plant modeling using `cpfg` followed by rendering using `rayshade` or `toro`, and (ii) interactive specification of plant distribution using `densedis` in conjunction with plant generation using `xfrog` and rendering using `fshade`.

Figure 7 presents visualizations of two stages of the self-thinning process, based on distributions shown in Figure 4. The plants represent hypothetical varieties of *Lychnis coronaria* [47] with red, blue, and white flowers. Plant size values returned by the ecosystem simulation were quantized to seven representative values for each plant variety. The quantized values were mapped to the age of the modeled plants. The scene obtained after 99 simulation steps had 16,354 plants. The `rayshade` file representing this scene without instancing would be 3.5 GB (estimated); with instancing it was 6.7 MB, resulting in the compression ratio of approximately 500:1. For the scene after 164 steps, the corresponding values were: 441 plants, 125 MB, 5.8 MB, compression 21:1.

The mountain meadow (Figure 8 top) was generated by simulating an ecosystem of eight species of herbaceous plants, as discussed in Section 5. The distribution of plants is qualitatively similar to that shown schematically in Figure 6, but it includes a larger number of smaller plants. The individual plants were modeled with a high level of detail, which made it possible to zoom in on this scene and view individual plants. The complete scene has approximately 102,522 plants, comprising approximately $2 \cdot 10^9$ primitives (polygons and cylinders). The `rayshade` file representing this scene without instancing would be 200 GB (estimated), with the instancing it was 151 MB, resulting in a compression ratio of approximately 1,300:1.

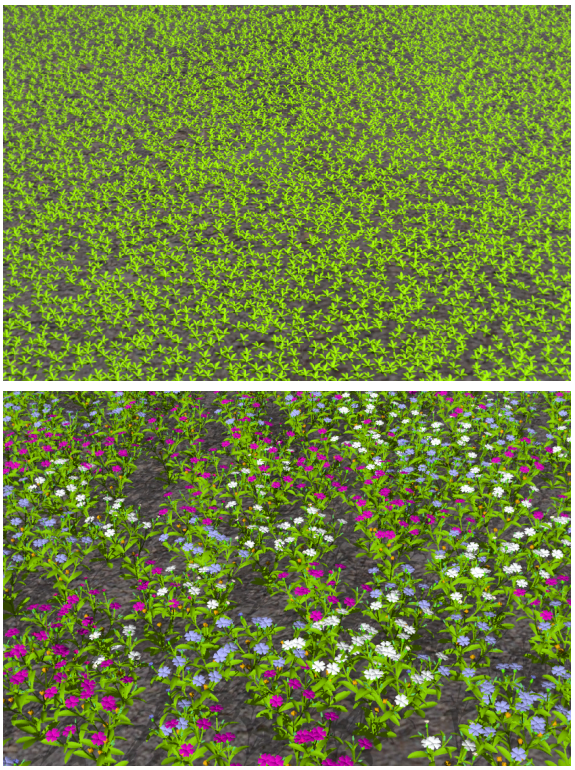


Figure 7: A *Lychnis coronaria* field after 99 and 164 simulation steps

The time needed to model this scene on a 150 MHz R5000 Silicon Graphics Indy with 96 MB RAM was divided as follows: simulation of the ecosystem (25 steps): 35 min, quantization (two-dimensional parameter space, each of the 8 plant species quantized to 7 levels): 5 min, generation of the 56 representative plants using `cpfg`: 9 min. The rendering time using `rayshade` on a 180 MHz R10000 Silicon Graphics Origin 200 with 256 MB RAM (1024×756 pixels, 4 samples per pixel) was approximately 8 hours. (It was longer using `toro`, but in that case the rendering time did not depend critically on the amount of RAM.)

In the next example, the paradigm of parameterizing, quantizing, and instancing was applied to entire groups of plants: tufts of grass with daisies. The number of daisies was controlled by a parameter (Figure 9). The resulting lawn is shown in Figure 10. For this image, ten different sets of grass tufts were generated, each instanced twenty times on average. The total reduction in geometry due to quantization and instancing (including instancing of grass blades and daisies within the tufts) was by a factor of 130:1. In Figure 11, a model parameter was used to control the size of the heaps of leaves. The heap around the stick and the stick itself were modeled manually.

Interactive creation of complex scenes requires the proper use of techniques to achieve an aesthetically pleasing result. To illustrate the process that we followed, we retrace the steps that resulted in the stream scene shown in Figure 15.

We began with the definition of a hilly terrain crossed by a little stream (Figure 2). To cover it with plants, we first created procedural models of plant species fitting this environment (Figure 12). Next, we extracted images representing the terrain altitudes and the stream position (Figures 13a and 13b) from the original terrain data. This

Figure 8: Zooming in on a mountain meadow

provided visual cues needed while painting plant distributions, for example, to prevent plants from being placed in the stream.

After that, we interactively chose a viewpoint, approximately at human height. With the resulting perspective view of the terrain as a reference, we painted a gray scale image for each plant species to define its distribution. We placed vegetation only in the areas visible from the selected viewpoint to speed up the rendering later on. For example, Figure 13c shows the image that defines the density distribution of stinging nettles. Since the stinging nettles grow on wet ground, we specified high density values along the stream. The density image served as input to `densedis`, which determined positions of individual plants. The dot diagram produced by `densedis` (Figure 13d) provided visual feedback that was used to refine the density image step by step until the desired distribution was found.

Figure 9: Grass tufts with varying daisy concentrations

Figure 10: Lawn with daisies

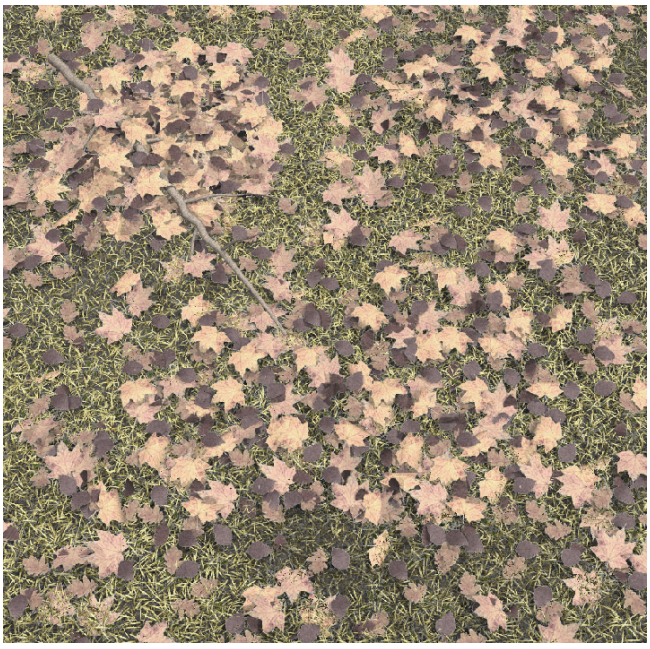


Figure 11: Leaves on grass

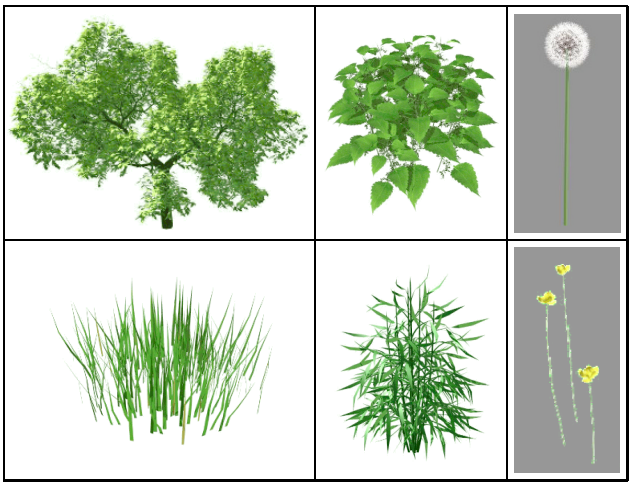


Figure 12: Sample plant models used in the stream scene. Top row: apple, stinging nettle, dandelion; bottom row: grass tuft, reed, yellow flower.

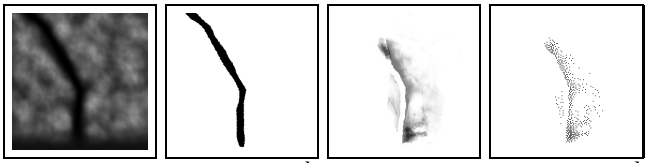


Figure 13: Creating distribution of stinging nettle: the heightmap of the covered area (a), the river image (b), the plant density distribution painted by the user (c), and the resulting plant positions (d).

Once the position of plants was established, we employed additional parameters to control the appearance of the plants. The vigor of stinging nettle plants, which affects the length of their stems and the number of leaves, was controlled using the density image for the nettles. To control the vigor of grass we used the height map: as a result, grass tufts have a slightly less saturated color on top of the hill than in the lower areas. Each tuft was oriented along a weighted sum of the terrain's surface normal vector and the up vector.

At this point, the scene was previewed and further changes in the density image were made until a satisfying result was obtained.

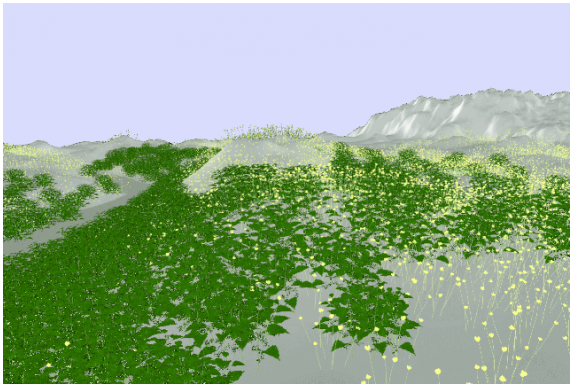


Figure 14: OpenGL preview of the stream scene including stinging nettle and yellow flowers

Figure 15: Stream scene

Figure 14 shows the preview of the distribution of stinging nettles and yellow flowers. To prevent intersections between these plants, the painted density image for the yellow flowers was multiplied by the inverted density image for the nettles.

The apple trees were placed by painting black dots on a white image. The final scene (Figure 15) was rendered using `fshade`. Images representing each species were rendered separately, and the resulting sub-scenes were composited as described in Section 7. The clouds were then added using a real photograph as a texture map. To increase the impression of depth in the scene, color saturation and contrast were decreased with increasing depth in a postprocessing step, and colors were shifted towards blue. Table 1 provides statistics about the instancing and geometric compression for this scene. The creation of this image took two days plus one day for defining the plant models. The actual compute time needed to synthesize this scene on a 195 MHz R10000 8-processor Silicon Graphics Onyx with 768MB RAM (1024 $\times$ 756 pixels, 9 samples per pixel) was 75 min.

Figures 16 and 17 present further examples of scenes with interactively created plant distributions. To simulate the effect of shadowing on the distribution of the yellow flowers in Figure 16, we rendered a top view of the spheres that approximate the shape of the apple trees, and multiplied the resulting image (further modified interactively) with the initial density image for the yellow flowers. We followed a similar strategy in creating Figure 17: the most important trees were positioned first, then rendered from above to provide visual cues for the further placements. Table 2 contains statistics about the geometry quantization in Figure 17.

plant	obj.	inst.	plant	obj.	inst.
apple	1	4	grass tuft	15	2577
reed	140	140	stinging nettle	10	430
dandelion	10	55	yellow flower	10	2751

Table 1: Number of prototype objects and their instances in the stream scene (Figure 15). Number of polygons without instancing: 16,547,728, with instancing: 992,216. Compression rate: 16.7:1.

plant	obj.	inst.	plant	obj.	inst.
weeping willow	16	16	reed	15	35
birch	43	43	poppy	20	128
distant tree	20	119	cornflower	72	20
St. John's wort	20	226	dandelion	20	75
grass tuft	15	824			

Table 2: Number of prototype objects and their instances in the Dutch scene (Figure 17). Number of polygons without instancing: 40,553,029, with instancing: 6,737,036. Compression rate: 6.0:1

Figure 16: Forest scene

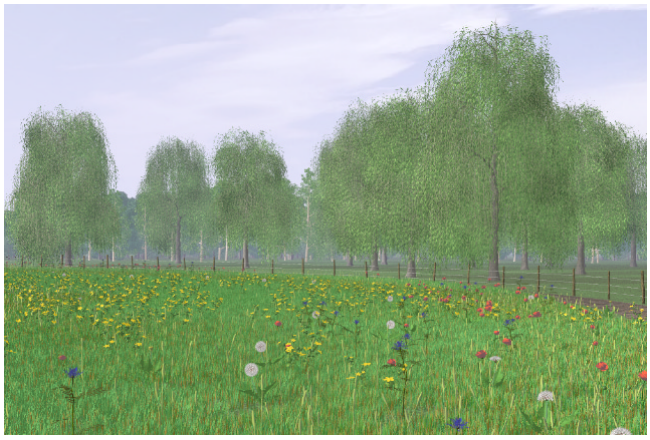


Figure 17: Dutch landscape

9 CONCLUSIONS

We presented the design and experimental implementation of a system for modeling and rendering scenes with many plants. The central issue of managing the complexity of these scenes was addressed with a combination of techniques: the use of different levels of abstraction at different stages of the modeling and rendering pipeline, procedural modeling, approximate instancing, and the employment of space- and time-efficient rendering methods. We tested our system by generating a number of visually complex scenes. Consequently, we are confident that the presented approach is operational and can be found useful in many practical applications.

Our work is but an early step in the development of techniques for creating and visualizing complex scenes with plants, and the presented concepts require further research. A fundamental problem is the evaluation of the impact of quantization and approximate instancing on the generated scenes. The difficulty in studying this problem stems from: (i) the difficulty in generating non-instanced reference images for visual comparison purposes (the scenes are too large), (ii) the lack of a formally defined error metric needed to evaluate the artifacts of approximate instancing in an objective manner, and (iii) the difficulty in generalizing results that were obtained by the analysis of specific scenes. A (partial) solution to this problem would set the stage for the design and analysis of methods that may be more suitable for quantizing plants than the general-purpose variance-based algorithm used in our implementation.

Other research problems exposed by our experience with `EcoSys` include: (i) improvement of the terrain model through its coupling with the plant population model (in nature vegetation affects terrain, for example by preventing erosion); (ii) design of algorithms for converting plant densities to positions, taking into account statistical properties of plant distributions found in natural ecosystems [66]); (iii) incorporation of morphogenetic plasticity (dependence of the plant shape on its neighbors [58]) into the multi-level modeling framework; this requires transfer of information about plant shapes between the population model and the procedural plant models; (iv) extension of the modeling method presented in this paper to animated scenes (with growing plants and plants moving in the wind); (v) design of methods for conveniently previewing scenes with billions of geometric primitives (for example, to select close views of details); and (vi) application of more faithful local and global illumination models to the rendering of plant scenes (in particular, consideration of the distribution of diffuse light in the canopy).

Acknowledgements

We would like to acknowledge Craig Kolb for his implementation of the variance-based quantization algorithm, which we adapted to the needs of our system, and Christain Jacob for his experimental implementations and discussions pertinent to the individual-based ecosystem modeling. We also thank: Stefania Bertazzon, Jim Hanan, Richard Levy, and Peter Room for discussions and pointers to the relevant literature, the referees for helpful comments on the manuscript, Chris Prusinkiewicz for editorial help, and Darcy Grant for system support in Calgary. This research was sponsored in part by the National Science Foundation grant CCR-9508579-001 to Pat Hanrahan, and by the Natural Sciences and Engineering Research Council of Canada grant OGP0130084 to Przemyslaw Prusinkiewicz.

REFERENCES

[1] Alias/Wavefront; a division of Silicon Graphics Ltd. Studio V8. SGI program, 1996.

[2] AnimaTek, Inc. AnimatTek's World Builder. PC program, 1996.

[3] R. A. Armstrong. A comparison of index-based and pixel-based neighborhood simulations of forest growth. *Ecology*, 74(6):1707–1712, 1993.

[4] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed textures. In *SIGGRAPH 96 Conference Proceedings*, pages 373–378, August 1996.

[5] B. M. Blumberg and T. A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *SIGGRAPH 95 Conference Proceedings*, pages 47–54, August 1995.

[6] B.N. Boots. *Spatial tesselations: concepts and applications of Voronoi diagrams*. John Wiley, 1992.

[7] A. Brownbill. Reducing the storage required to render L-system based models. Master's thesis, University of Calgary, October 1996.

[8] N. Chiba, K. Muraoka, A. Doi, and J. Hosokawa. Rendering of forest scenery using 3D textures. *The Journal of Visualization and Computer Animation*, 8:191–199, 1997.

[9] Adobe Corporation. Adobe Photoshop.

[10] O. Deussen and B. Lintermann. A modelling method and user interface for creating plants. In *Proceedings of Graphics Interface 97*, pages 189–197, May 1997.

[11] J. Dorsey, H. Køhling Pedersen, and P. Hanrahan. Flow and changes in appearance. In *SIGGRAPH 96 Conference Proceedings*, pages 411–420, August 1996.

[12] T. Duff. Compositing 3-D rendered images. *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19(3):41–44, 1985.

[13] F. G. Firbank and A. R. Watkinson. A model of interference within plant monocultures. *Journal of Theoretical Biology*, 116:291–311, 1985.

[14] K. W. Fleischer, D. H. Laidlaw, B. L. Currin, and A. H. Barr. Cellular texture generation. In *SIGGRAPH 95 Conference Proceedings*, pages 239–248, August 1995.

[15] R. W. Floyd and L. Steinberg. An adaptive algorithm for spatial gray scale. In *SID 75, Int. Symp. Dig. Tech. Papers*, pages 36–37, 1975.

[16] D. R. Fowler, P. Prusinkiewicz, and J. Battjes. A collision-based model of spiral phyllotaxis. *Computer Graphics (SIGGRAPH 92 Proceedings)*, 26(2):361–368, 1992.

[17] G. Y. Gardner. Simulation of natural scenes using textured quadric surfaces. *Computer Graphics (SIGGRAPH 84 Proceedings)*, 18(3):11–20, 1984.

[18] A. Gersho and R. M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, 1991.

[19] D. G. Green. Modelling plants in landscapes. In M. T. Michalewicz, editor, *Plants to ecosystems. Advances in computational life sciences I*, pages 85–96. CSIRO Publishing, Melbourne, 1997.

[20] J. C. Hart and T. A. DeFanti. Efficient anti-aliased rendering of 3D linear fractals. *Computer Graphics (SIGGRAPH 91 Proceedings)*, 25:91–100, 1991.

[21] J.C. Hart. The object instancing paradigm for linear fractal modeling. In *Proceedings of Graphics Interface 92*, pages 224–231, 1992.

[22] P. Heckbert. Color image quantization for frame buffer display. *Computer Graphics (SIGGRAPH 82 Proceedings)*, 16:297–307, 1982.

[23] S. I. Higgins and D. M. Richardson. A review of models of alien plant spread. *Ecological Modelling*, 87:249–265, 1996.

[24] H. Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 97 Conference Proceedings*, pages 189–198, August 1997.

[25] D. H. House, G. S. Schmidt, S. A. Arvin, and M. Kitagawa-DeLeon. Visualizing a real forest. *IEEE Computer Graphics and Applications*, 18(1):12–15, 1998.

[26] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *Computer Graphics (SIGGRAPH 89 Proceedings)*, 23(3):271–289, 1989.

[27] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. *Computer Graphics (SIGGRAPH 86 Proceedings)*, 20(4):269–278, 1986.

[28] A. D. Kelley, M. C. Malin, and G. M. Nielson. Terrain simulation using a model of stream erosion. *Computer Graphics (SIGGRAPH 88 Proceedings)*, 22(4):263–268, 1988.

[29] C. Kolb. Rayshade. http://graphics.stanford.edu/∼cek/rayshade.

[30] M. P. Kumler. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica*, 31(2), 1994.

[31] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH 96 Conference Proceedings*, pages 31–42, August 1996.

[32] B. Lintermann and O. Deussen. Interactive structural and geometrical modeling of plants. To appear in the *IEEE Computer Graphics and Applications*.

[33] B. Lintermann and O. Deussen. Interactive modelling and animation of natural branching structures. In R. Boulic and G. Hégron, editors, *Computer Animation and Simulation 96*. Springer, 1996.

[34] Lucasfilm Ltd. The Adventures of André and Wally B. Film, 1984.

[35] D. Marshall, D. S. Fussel, and A. T. Campbell. Multiresolution rendering of complex botanical scenes. In *Proceedings of Graphics Interface 97*, pages 97–104, May 1997.

[36] N. Max. Hierarchical rendering of trees from precomputed multi-layer Z-buffers. In X. Pueyo and P. Schröder, editors, *Rendering Techniques 96*, pages 165–174 and 288. Springer Wien, 1996.

[37] N. Max and K. Ohsaki. Rendering trees from precomputed Z-buffer views. In P. M. Hanrahan and W. Purgathofer, editors, *Rendering Techniques 95*, pages 74–81 and 359–360. Springer Wien, 1995.

[38] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. *Computer Graphics (SIGGRAPH 89 Proceedings)*, 23(3):41–50, 1989.

[39] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH 96 Conference Proceedings*, pages 397–410, August 1996.

[40] F. Neyret. A general and multiscale model for volumetric textures. In *Proceedings of Graphics Interface 95*, pages 83–91, 1995.

[41] F. Neyret. Synthesizing verdant landscapes using volumetric textures. In X. Pueyo and P. Schröder, editors, *Rendering Techniques 96*, pages 215–224 and 291, Wien, 1996. Springer-Verlag.

[42] K. Perlin. An image synthesizer. *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19(3):287–296, 1985.

[43] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH 97 Conference Proceedings*, pages 101–108, August 1997.

[44] P. Prusinkiewicz. Visual models of morphogenesis. *Artificial Life*, 1(1/2):61–74, 1994.

[45] P. Prusinkiewicz. Modeling spatial structure and development of plants: a review. *Scientia Horticulturae*, 74(1/2), 1998.

[46] P. Prusinkiewicz, M. Hammel, and E. Mjolsness. Animation of plant development. In *SIGGRAPH 93 Conference Proceedings*, pages 351–360, August 1993.

[47] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[48] P. Prusinkiewicz, W. Remphrey, C. Davidson, and M. Hammel. Modeling the architecture of expanding *Fraxinus pennsylvanica* shoots using L-systems. *Canadian Journal of Botany*, 72:701–714, 1994.

[49] Questar Productions, LLC. World Construction Set Version 2. PC program, 1997.

[50] W. T. Reeves and R. Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. *Computer Graphics (SIGGRAPH 85 Proceedings)*, 19(3):313–322, 1985.

[51] C. W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21(4):25–34, 1987.

[52] R. E. Ricklefs. *Ecology. Third Edition.* W. H. Freeman, New York, 1990.

[53] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH 96 Conference Proceedings*, pages 75–82, August 1996.

[54] H. Sinoquet and R. Rivet. Measurement and visualization of the architecture of an adult tree based on a three-dimensional digitising device. *Trees*, 11:265–270, 1997.

[55] A. R. Smith. Plants, fractals, and formal languages. *Computer Graphics (SIGGRAPH 84 Proceedings)*, 18(3):1–10, 1984.

[56] J. M. Snyder and A. H. Barr. Ray tracing complex models containing surface tessellations. *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21(4):119–128, 1987.

[57] R. R. Sokal and F. J. Rohlf. *Biometry. Third Edition.* W. H. Freeman, New York, 1995.

[58] K. A. Sorrensen-Cothern, E. D. Ford, and D. G. Sprugel. A model of competition incorporating plasticity through modular foliage and crown development. *Ecological Monographs*, 63(3):277–304, 1993.

[59] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. Proceedings of the Spring Joint Computer Conference, 1963.

[60] J. H. M. Thornley and I. R. Johnson. *Plant and crop modeling: A mathematical approach to plant and crop physiology*. Oxford University Press, New York, 1990.

[61] S. J. Wan, S. K. M. Wong, and P. Prusinkiewicz. An algorithm for multidimensional data clustering. *ACM Trans. Math. Software*, 14(2):135–162, 1988.

[62] A. Watt and M. Watt. *Advanced animation and rendering techniques: Theory and practice*. Addison-Wesley, Reading, 1992.

[63] J. Weber and J. Penn. Creation and rendering of realistic trees. In *SIGGRAPH 95 Conference Proceedings*, pages 119–128, August 1995.

[64] L. Williams. Casting curved shadows on curved surfaces. *Computer Graphics (SIGGRAPH 78 Proceedings)*, 12(3):270–274, 1978.

[65] L. Williams. Shading in two dimensions. In *Proceedings of Graphics Interface 91*, pages 143–151, June 1991.

[66] H. Wu, K. W. Malafant, L. K. Pendridge, P. J. Sharpe, and J. Walker. Simulation of two-dimensional point patterns: application of a latice framework approach. *Ecological Modelling*, 38:299–308, 1997.

Generating Spatial Distributions for Multilevel Models of Plant Communities

Brendan Lane Przemyslaw Prusinkiewicz

Department of Computer Science
University of Calgary
laneb | pwp @ cpsc.ucalgary.ca

Abstract

The simulation and visualization of large groups of plants has many applications. The extreme visual complexity of the resulting scenes can be captured using multilevel models. For example, in two-level models, plant distributions may be determined using coarse plant representations, and realistic visualizations may be obtained by substituting detailed plant models for the coarse ones. In this paper, we focus on the coarse aspect of modeling, the specification of plant distribution. We consider two classes of models: local-to-global models, rooted in the individual-based ecosystem simulations, and inverse, global-to-local models, in which positions of individual plants are inferred from a given distribution of plant densities. We extend previous results obtained using both classes of models with additional phenomena, including clustering and succession of plants. We also introduce the formalism of multiset L-systems to formalize the individual-based simulation models.

Key words: realistic image synthesis, multilevel modeling, plant ecosystem, spatial distribution, clustering, succession, multiset L-system

1 Introduction

The simulation and visualization of plant ecosystems has many theoretical and practical applications. They include fundamental research in ecology, visual impact analysis of forestry practices, and synthesis of complex scenery for computer animations, among others. The inherent complexity of the scenes resulting from the ecosystem simulations can be managed using the multilevel approach to modeling [3]. Rather than model the entire ecosystem at the detailed level of plant organs, such as leaves, flowers, apices, and internodes [13], the multilevel approach employs a hierarchy of models. For example, in the simplest, two-level case, a high-level model determines the distribution of the plants, and lower-level models determine the plants' shapes. The models are coupled so that information created at a higher level can affect the outcome of the model at the lower level.

In this paper, we focus on the generation of the spatial distribution of plants. Specifically, we extend the methods reported in [3] with the ecologically and visually important phenomena of clustering and succession of plants. We also introduce the formalism of *multiset L-systems* to formalize some of these models.

Previous work on multilevel modeling of plant ecosystems is summarized in Section 2. Following the approach introduced there, we distinguish the *local-to-global* approach, in which the distribution of plant densities is determined by a simulation of interactions between the individual plants, and the *global-to-local* approach, in which positions of individual plants are inferred from given large-scale density distributions. In Section 3.1, we introduce multiset L-systems as an extension of the L-system modeling framework. This extension allows us to use L-systems, long an individual plant modeling paradigm, to express local-to-global algorithms for generating plant distributions as well. Sample applications of multiset L-systems are given in Sections 3.2 to 3.4. The concept and examples of the global-to-local modeling of plant distribution are presented in Section 4, which extends preliminary results reported in [7]. Conclusions are presented in Section 5.

2 Previous work

Multilevel modeling of plant communities for image synthesis purposes was introduced in [3], although related techniques had been used earlier, *e.g.* [1]. The main concept was to consider the generation of a plant ecosystem as a hierarchy of tasks: specification of the terrain, generation of plant distribution using coarse plant models, synthesis of detailed plant models as needed to populate the scene, and the rendering of the final scene using instances of these detailed models.

Two different methods were used in [3] to create plant distributions. The first one was an individual-based ecosystem simulation, based on a model of Firbank and Watkinson [4]. Following that model, simulated plants were placed in the field at random, then iteratively

Published in the Proceedings of Graphics Interface 2002

'grown', and 'killed' when dominated by larger plants. The resulting distribution fit the *self-thinning curve* of plant ecology [8], a relationship between the average mass and average density in a monoculture of plants of the same age. The individual-based approach was also used in [3] to produce a hierarchy (distribution) of plant sizes (*c.f.* [8]), similar to that observed in nature.

The second method was intended to allow more user control in defining the local density of plants. The input was a greyscale image representing a map of the density of plants throughout the field. The Floyd-Steinberg error diffusion algorithm [5] was used to create the positions of individual plants conforming to these densities. The points produced by this algorithm were slightly jittered to make the distribution appear more random.

These two methods exemplify two different approaches to ecological modeling. The individual-based simulation is representative of the local-to-global approach. It is characterized by the emergence of global features from the local interactions of individual plants. In contrast, the error diffusion method is an example of the inverse, global-to-local approach, in which local characteristics are derived from global properties of the distribution. This distinction is similar to the distinction between local-to-global and global-to-local methods used to model individual plants [14].

The methods described in [3] tend to create uniform plant distributions. In reality, however, plants often are clustered. *Clustering*, also known as *clumping* or *underdispersion* [2], is a common phenomenon, caused by environmental factors (plants of the same type tend to cluster in the areas favorable to their growth), propagation (seeds fall close to their parent plants, or plants propagate by runners), as well as other mechanisms. It has a significant impact on the appearance of plant distributions, which is why we are seeking to model it.

The effect of clustering can be quantified using several statistical measures. We use the *Hopkins index* [6], which is defined as the average distance from a randomly chosen point to its nearest plant within a given region, divided by the average distance from a randomly chosen plant to its nearest plant:

$$H = \frac{\langle min_i(\|x - p_i\|)\rangle_x}{\langle min_i(\|p_j - p_i\|)\rangle_j}.$$

Distributions that are completely uncorrelated ('random') have an H value of 1. Distributions that are more dispersed than random ('regular') have an H value less than 1, and distributions that are clustered have an H value greater than 1. For example, Figure 1 compares an overdispersed distribution with a Hopkins index of 0.4 and a clustered distribution with a Hopkins index of 2.4.

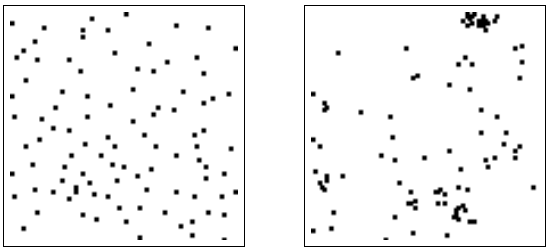


Figure 1: The effect of clustering on plant distribution. Left: an overdispersed distribution with H = 0.4. Right: a clustered distribution with H = 2.4.

3 Local-to-global modeling of plant distribution

3.1 Multiset L-systems

We model the individual plants using L-systems [13] and a related technique based on Chomsky grammars [14]. In addition, we extend the L-system formalism to generate plant distributions using the local-to-global approach. To this end, we introduce the notion of *multiset L-systems*.

An L-system model generates plants represented as strings of symbols [9] with optional parameters [13]. These strings define both the topology and the geometry of the resulting structures. An L-system specification consists of three components: the *alphabet*, which is the set of symbols that represent distinct components of the plant; the *axiom*, which represents the initial state of the modeled structure; and a list of *productions*, which define the development of the plant's components over steps of time. The alphabet may be defined implicitly, as the set of symbols that appear in the productions. The development of a plant is simulated in a sequence of *derivation steps*. In any step, each symbol is rewritten using the first applicable production on the list (or rewritten into itself if no production applies), yielding a new string. An extension of L-systems called *pseudo-L-systems* [11] makes it possible to rewrite two or more symbols using a single production. Another extension, called *open L-systems* [10], makes is possible to capture the interactions between the modeled plants and their environment.

Multiset L-systems unify and extend to branching structures two previously defined notions of the L-system theory: developmental systems with finite axiom sets [16] and L-systems with fragmentation [17]. In multiset L-systems, the set of productions operates on a multiset of strings that represent many plants, rather than a single string that represents an individual plant. New strings can be dynamically added to or removed from this multiset, representing organisms that are added to or removed from the population.

Formally, a context-free non-parametric multiset L-system is a four-tuple $G = \langle V, \%, \Omega, P\rangle$ where V is the

alphabet (a finite set of symbols), $\% \notin V$ is a reserved *fragmentation symbol*, $\Omega \subset V^\star$ is a finite set of words over V called the *axiom*, and $P \subset V \times (V \cup \{\%\})^\star$ is a finite set of *productions*. The alphabet V may contain, in particular, a pair of brackets, [and], which are used to delimit branches in the *bracketed string* notation of tree structures [13].

A derivation step in a multiset L-system consists of two sub-steps. First, all words x_i in the predecessor multiset are replaced by the intermediate successor words y_i using productions in P. The individual derivations $x_i \rightarrow y_i$ are performed as in an ordinary L-system. Second, the words y_i that contain one or more fragmentation symbols % are subdivided. In this process, symbol % acts as the marker of positions at which branches y_{ik} are cut off the tree y_i. The remaining part of the tree y_i and the cut off branches y_{ik} become the members of the successor multiset.

For example, let us consider the multiset L-system specified below.

Alphabet: {A, B, I, [,] }
Axiom: { A, B }
Productions: 1. A → I[B]A
2. B → B%A

Starting with the axiom, the first two derivation steps yield the multisets listed in the Table 1.

step	intermediate multiset	final multiset
0	{A, B}	{ A, B }
1	{ I[B]A, B%A }	{I[B]A, B, A }
2	{I[B%A]I[B]A, B%A, I[B]A }	{ I[B]I[B]A, A, B, A, I[B]A }

Table 1: Operation of a sample multiset L-system

Extensions of L-systems, such as pseudo-L-systems and open L-systems, also apply to the multiset L-systems. In particular, in the simulations of ecosystems we rely extensively on the *communication symbols* ?E, introduced in [10] as a part of the open L-system formalism. The communication symbol is a vehicle for information exchange between plant models and their environment. It can be associated with one or more parameters, which are set by the environmental program interfaced with the L-system-based simulator.

The plant models used in the ecosystem simulations are extremely simplified, in order to accommodate a large number of plants. We have used the L-system-based plant modeling software `L-studio/cpfg` [12], extended with multiset capabilities, to both generate plant distributions and model the individual plants.

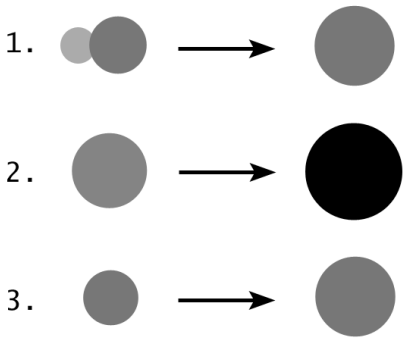


Figure 2: Diagrammatic representation of a model of self-thinning. Dark grey circles represent growing plants, the light grey circle represents a dominated plant, and the black circle represents a mature plant that no longer grows.

3.2 Self-thinning

As the first illustration of the concepts described above, let us consider a multiset L-system implementation of the individual-based self-thinning model outlined in [3]. Self-thinning takes place among a group of plants of the same species and age. As the plants grow and compete with each other for resources, smaller and weaker plants become *dominated* by larger, stronger plants, and eventually die. The essence of this process can be captured using the set of rules shown in Figure 2. The corresponding L-systems is given below:

Axiom: { T($\vec{x}_1$,r_1)?E(1) ,
T($\vec{x}_2$,r_2)?E(1) ,
... ,
T($\vec{x}_n$,r_n)?E(1) }

1. T($\vec{x}$,r)?E(c) : c == 0 → ϵ
2. T($\vec{x}$,r) : $r \geq R$ → T($\vec{x}$,R)
3. T($\vec{x}$,r)?E(c) → T($\vec{x}$,r + grow($r, \Delta t$))

Each plant is described by module T($\vec{x}$,r) followed by the communication module ?E(c). Vector $\vec{x}$ and number r represent position and size (shoot radius) of the plant. Parameter c is used for communication with the environmental process, which sets c to 1 if the plant is not dominated and to 0 if it is dominated. The environmental process considers each plant as a circle of a radius r, and determines which circles are intersecting. The smaller of any pair of intersecting circles is considered dominated.

The axiom introduces n plants with random positions and sizes (the initial distribution of plants could also be generated algorithmically). The first production, guarded

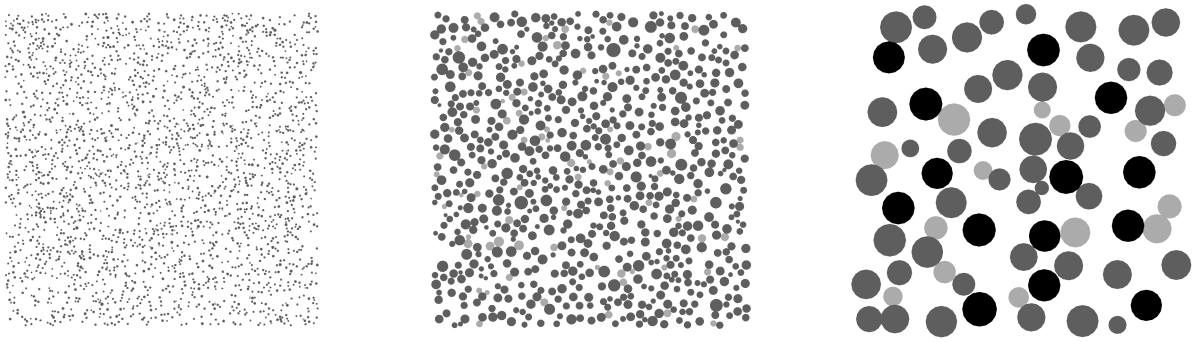


Figure 3: Three stages of simulation of the self-thinning process. Dark grey circles are growing plants, light grey circles are dominated plants, and black circles are mature plants, as in Figure 2.

by the condition c == 0, removes any dominated plant and its associated communication module from the population. Production 2 stops the growth of a plant that has reached its the maximum size R. Finally, production 3 increases the size of a plant that is neither dominated nor mature. The user-defined function grow($r, \Delta t$) captures growth of a plant of radius r over time interval Δt.

Figure 3 shows three stages of a self-thinning process simulated using this L-system. As the plant community develops over time, dominated plants gradually disappear and thin out the distribution. Sample visualizations obtained by substituting realistic plant models for the individual circles are shown in [3].

3.3 Plant succession

An extension to the previous L-system transforms it into a model of interaction between two plant species:

Axiom: { X }

1. X → T($\vec{x}_1$,r_1,1)?E(1) %
   ...
   T($\vec{x}_n$,r_n,1)?E(1) %
   T($\vec{x}_{n+1}$,r_{n+1},2)?E(1) %
   ...
   T($\vec{x}_{n+m}$,r_{n+m},2)?E(1) % X

2. T($\vec{x}$,r,sp) > ?E(c) : c == 0 &&
   random(1) < shaded[sp] → T($\vec{x}$,r,sp)
3. T($\vec{x}$,r,sp) ?E(c) : c == 0 → ϵ

4. T($\vec{x}$,r,sp) : $r \geq R$ && random(1) < oldage[sp]
   → T($\vec{x}$,R,sp)
5. T($\vec{x}$,r,sp) : $r \geq R \rightarrow \epsilon$

6. T($\vec{x}$,r,sp) → T($\vec{x}$,r + grow(r,sp, Δt),sp)

In this model a plant is represented by the module T($\vec{x}$,r,sp). Parameters $\vec{x}$ and r denote the plant's position and radius, as in the previous model. Parameter sp is the plant's species identifier, either 1 or 2. Production 1 adds n new plants of species 1 and m new plants of species 2 to the population. The production predecessor X reappears in the successor multiset, thus new plants are added in every simulation step. Productions 2 and 3 remove a dominated plant with probability $1 - \text{shaded}[sp]$[1]. The value shaded[sp], called the *shade tolerance* of the plant, is a measure of how likely it is to survive in shadow.

Productions 4 and 5 model the senescence of plants. Once a plant has reached the radius R, it survives with the probability oldage[sp]; a plant that does not survive dies and is removed from the community. Production 6 uses the growth function grow(r,sp,Δt) to simulate the growth of plants that are neither dominated nor old, according to their size and species.

With the right parameterization, this model captures the phenomenon of *succession* [8]. If species 1 has a higher growth rate but lower shade tolerance and old-age survivorship than species 2 (grow(r,1Δt) > grow(r,2,Δt), shaded[1] < shaded[2], oldage[1] < oldage[2]), then an initially empty field will be populated in stages. First, the field will be dominated by species 1. As the largest members of species 1 die, smaller members of species 2, which have survived due to their greater shade tolerance and now have a size advantage over young seedlings of species 1, will fill in the gaps. Eventually, the field will be dominated by members of species 2. A straightforward extension of this model to three plant species is illustrated in Figure 4.

[1] Recall that the production list is ordered, thus the string rewriting mechanism will first attempt to apply production 2, and only use production 3 if the condition of production 2 is not satisfied.

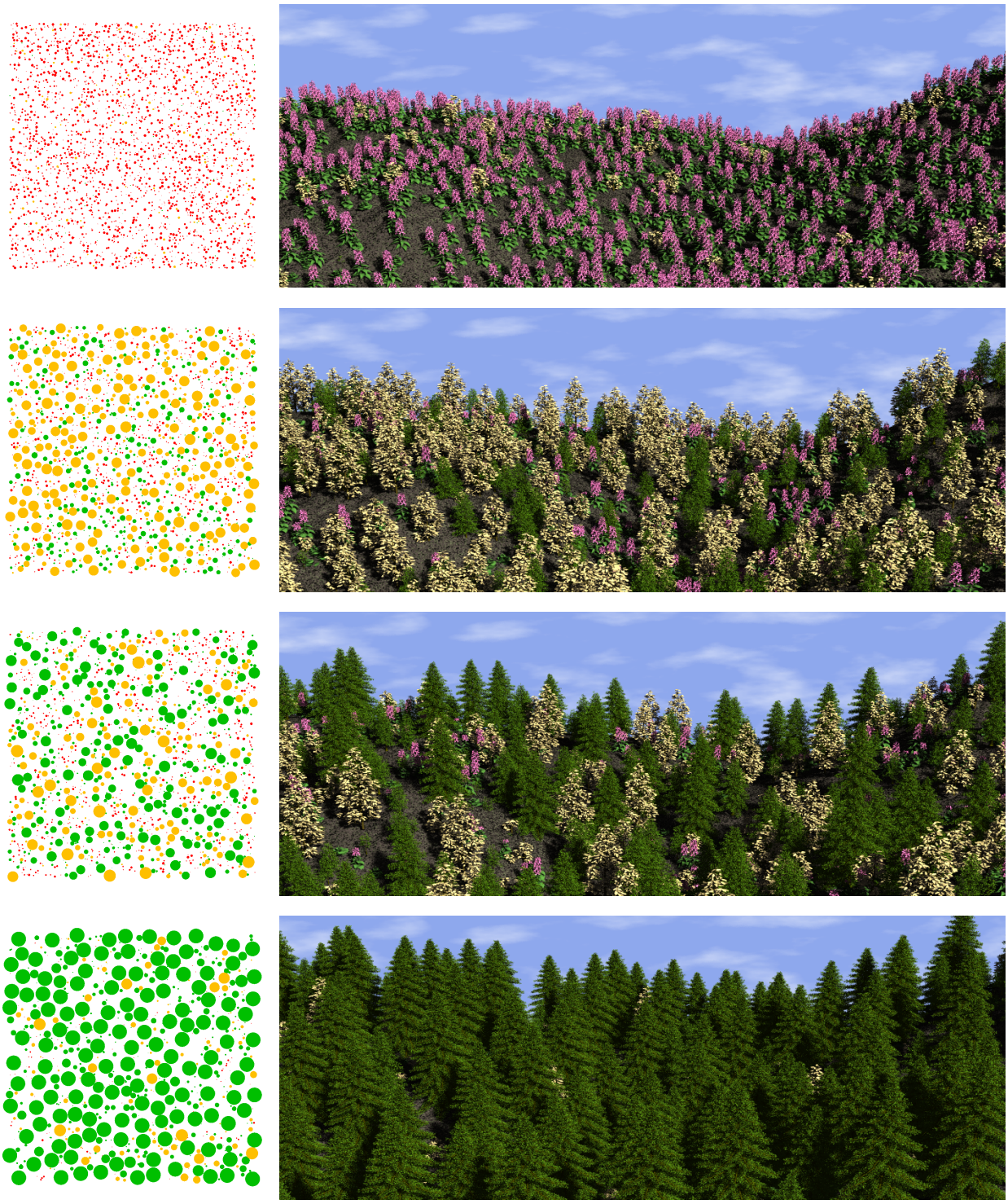


Figure 4: Four stages of ecosystem simulation using the plant succession model. Left: results of coarse-level simulation, using pink circles to indicate position of herbaceous plants (fireweed), orange circles to indicate positions of the early-succession deciduous trees, and green circles to indicate positions of the late-succession coniferous trees. Right: synthetic images obtained by placing tree models at the locations generated by the coarse-level simulation.

3.4 Plant propagation

The evaluation of the Hopkins index for the distributions shown in Figure 3 yields values of H equal to 0.8, 0.4, and 0.4, respectively. Similarly, the evaluation of the Hopkins index for the distributions in Figure 4 results in H values of 0.6, 0.7, 0.7, and 0.6. This shows that the competition for space leads to overdispersed plant distributions.

We can see why this is the case. If any two plants so much as touch each other, one of them will become dominated. In our self-thinning model, the dominated plant immediately dies; in the succession model, it dies with some probability per derivation step. In either case, the competition for space drives the plants apart, and there is no opposite mechanism encouraging plants to cluster.

One clustering mechanism observed in nature is local propagation. We can capture it, for instance, by 'sowing' new plants near the parent plants of the same species, instead of making them appear at random throughout the field. The resulting alteration of the succession model is given below.

Axiom: { T($\vec{x}_1$,r_1,1)?E(1) ,
... ,
T($\vec{x}_n$,r_n,1)?E(1) ,
T($\vec{x}_{n+1}$,r_{n+1},2)?E(1) ,
... ,
T($\vec{x}_{n+m}$,r_{n+m},2)?E(1) }

1. T($\vec{x}$,r,sp) > ?E(c) : c == 0 &&
   random(1) < shaded[sp] → T($\vec{x}$,r,sp)
2. T($\vec{x}$,r,sp) ?E(c) : c == 0 → ϵ

3. T($\vec{x}$,r,sp) : $r \geq R$ && random(1) > oldage[sp]
   → T($\vec{x}$,R,sp)
4. T($\vec{x}$,r,sp) : $r \geq R$ → ϵ

5. T($\vec{x}$,r,sp) ?E(c) → T($\vec{x}$,r + grow(sp,r,Δt),sp) ?E(c)
   % T($\vec{x} + \Delta\vec{x}$,r_0,sp) ?E(c)

The axiom defines the initial state of the model by placing n plants of species 1 and m plants of species 2 at random in the field. The subsequent productions are the same as in the succession model, except for production 5. According to it, a plant that is not dominated creates a new plant at position $\vec{x} + \Delta\vec{x}$, where $\Delta\vec{x}$ is a small random vector. Since the new plant is in close proximity to its parent, this propagation mechanism encourages clustering in the distribution.

Figure 5 illustrates the operation of this model. At the beginning, plants are randomly distributed. As the ecosystem develops, the two species become spatially segregated, creating large clusters of plants of each species. For example, the Hopkins indices of species 1 at the three stages shown are equal to 1.1, 4.2, and 11, respectively.

4 Global-to-local modeling of plant communities

4.1 The deformation-kernel method

The effect one plant in the self-thinning model (Section 3.2) has on the probability of finding another plant nearby is shown diagramatically in Figure 6. Within the reference plant radius r_t, the probability of finding another plant is very small; outside that radius, the probability is not affected by the reference plant.

The function K shown in Figure 6 is an example of a *deformation kernel*. If we suppose there is a field of values that characterizes the probability of placing a new plant at various locations, the deformation kernel captures the impact of an existing plant on this field. Various interactions between plants can be described using deformation kernels of different shapes, as suggested in Figure 7.

A simple plant placement algorithm can now be developed using this deformation kernel idea. We maintain a *joint probability density function* [15] $f(x, y)$, which characterizes the probability $f(x, y)dxdy$ of placing a new plant in the area $dxdy$ centered at point (x, y). The plants are placed one at a time; as each is placed, its deformation kernel modifies the probability function f that will be used to determine the position of the next plant. In this manner, a distribution of plants will eventually be formed.

Formally, the joint density function f defines a probability field, where the probability of a new plant growing in the rectangle $[0, x_s] \times [0, y_s]$, with $0 \leq x_s \leq x_{max}$ and $0 \leq y_s \leq y_{max}$, is given by the cumulative probability distribution function

$$\begin{aligned} F(x_s, y_s) &= P\{x_t \leq x_s, y_t \leq y_s\} \\ &= \int_0^{x_s}\int_0^{y_s} f(x, y)\, dx\, dy. \end{aligned}$$

Obviously, the probability that the plant will be found in the whole field is one, thus the density function must satisfy the normalizing equation

$$\int_0^{x_{max}}\int_0^{y_{max}} f(x, y)\, dx\, dy = 1. \qquad (1)$$

We find the position (x_t, y_t) of the plant to be added by calculating first its y, then its x coordinate. To this end, given the two-dimensional density function $f(x, y)$, we create the *marginal distribution function* $F_Y(y_s)$. That

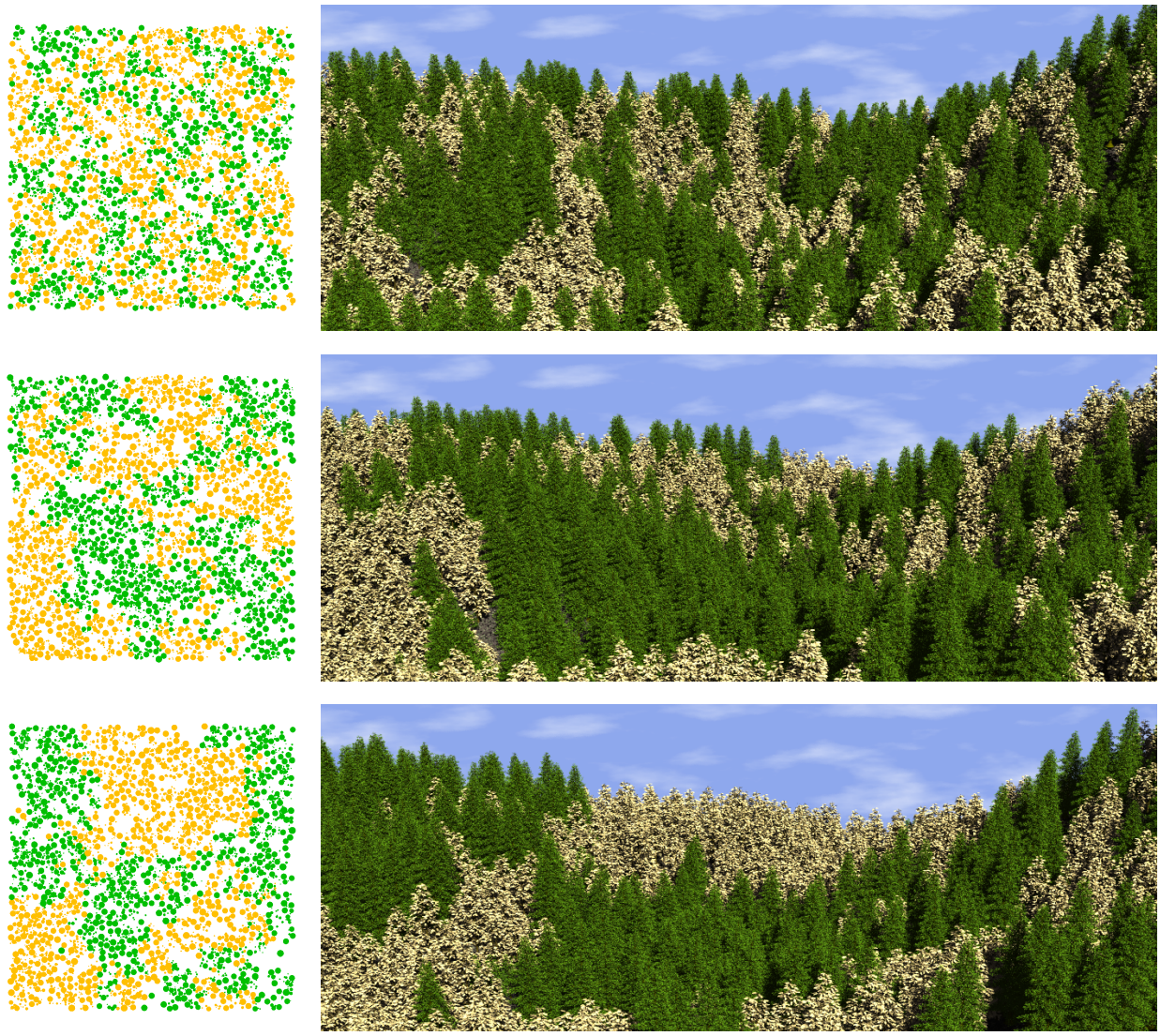


Figure 5: Three stages of ecosystem development simulated using the plant propagation model. Left: results of coarse-level simulation, using orange circles to indicate positions of poplar trees and green circles to indicate positions of spruce trees. Right: synthetic images obtained by placing tree models at the locations generated by the coarse-level simulation.

distribution function describes the probability that $y_t \leq y_s$ independently of the choice of x_t [2] :

$$F_Y(y_s) = P\{x_t \leq x_{max}, y_t \leq y_s\} = F(x_{max}, y_s).$$

We choose the y_t coordinate for the plant using the *inverse transformation method* [15]. To this end, we generate a random number u from the uniform distribution on $[0, 1]$. We then perform a binary search on $F_Y(y)$ to find the value y_t such that $F_Y(y_t) = u$. As F_Y is monotone and continuous, y_t exists and is unique. This is our plant's y coordinate.

Once we have chosen y_t, we calculate the *conditional distribution* $F_{X|Y}(x_s \mid y_t)$, which describes the probability that $x_t \leq x_s$ for a given a y value y_t:

$$F_{X|Y}(x_s \mid y_t) = P\{x_t \leq x_s \mid y_t\} = \frac{\int_0^{x_s} f(x, y_t)\, dx}{\int_0^{x_{max}} f(x, y_t)\, dx}.$$

We then apply the inverse transformation method to find coordinate x_t, given $F_{X|Y}(x_s \mid y_t)$.

[2] There is a corresponding marginal distribution function $F_X(x_s)$, which describes the probability that $x_t \leq x_s$, independent of what is chosen for y_t.

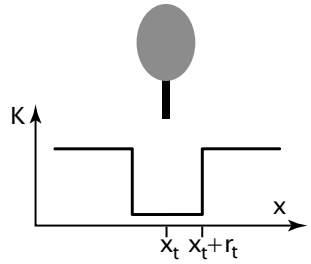


Figure 6: The effect of a reference plant on the probability of finding neighboring plants

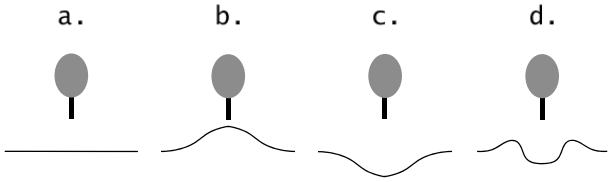


Figure 7: Examples of deformation kernels: a) kernel that has no effect on the neighboring plants, b) kernel that has a promotional effect, c) kernel that has an inhibitory effect, d) kernel that has an inhibitory short-range effect and promotional longer-range effect.

Having placed a plant of size r_t at position (x_t, y_t), we now deform the probability density function $f(x, y)$ in order to simulate the effects of that plant on the placement of nearby plants. To this end, we first multiply the probability density function $f(x, y)$ by the plant's deformation kernel $K(x, y)$,

$$f_{temp}(x, y) = f(x, y)K(x, y),$$

then renormalize the function $f_{temp}(x, y)$ to satisfy Equation 1. The deformation kernel typically is a function of the form

$$K(x, y) = \kappa\left(\frac{\sqrt{(x - x_t)^2 + (y - y_t)^2}}{r_t}\right),$$

where the function $\kappa(r)$ measures the effect a unit-sized plant has on the formation of plants at a distance r from it.

An obvious way to implement the above concepts is to represent values of the probability density function $f(x, y)$ using an n by n array of samples f_{ij}. To calculate position of a new plant, we create a vector **R** of partial sums of the rows, where

$$R_k = \sum_{i=0}^{k} \sum_{j=0}^{n-1} f_{ij}, \quad k = 0, 1, \ldots, n - 1.$$

We determine the coordinate y_t of the newly placed plant using the inverse transformation method. To this end, we pick a random number u from the uniform distribution on the interval $[0, R_{n-1}]$, then perform a binary search to locate R_i such that $R_i \leq u < R_{i+1}$. We then linearly interpolate between (i, R_i) and $(i+1, R_{i+1})$ to find (y_t, u).

Now a vector **C** of values representing the conditional distribution $F_{X|Y}(x_s \mid y_t)$ is computed by interpolating rows i and $i + 1$ of the array f_{ij}.

$$C_k = (y_0 - i) \sum_{j=0}^{k} f_{(i+1)j} + ((i + 1) - y_0) \sum_{j=0}^{k} f_{ij},$$

Given the values C_k, $k = 0, 1, \ldots, n - 1$, we choose a value from the uniform random distribution on the interval $[0, C_{n-1}]$, and use the inverse transformation method to find x_t.

The kernel is applied by simply calculating the distance d of every sampling point (i, j) from (x_t, y_t), then multiplying the value f_{ij} of the distribution function f at that point by $\kappa(\frac{d}{r_t})$.

If there are m plants to be placed and the function f is represented using n^2 values f_{ij}, the above algorithm will take $O(mn^2)$ time to run, since the sums R_k must be recalculated each time a new plant is placed. We improve on this result by updating the array **R** incrementally. When a kernel is applied to the distribution, the differences between f_{ij} and $\kappa(\frac{d}{r_t})f_{ij}$ are summed for each (i, j) within the range of the plant, and the differences are applied to the array **R**. Assuming that the kernel is only applied to a small fraction of the cells in the grid, this operation can be performed in $O(n)$ time per plant.

The operation of the kernel method is illustrated in Figure 8. The deformation kernel is that of Figure 7d. The initial distribution is uniform; $f(x, y) = c$. In the middle, a single plant has been added to the field; the density function has been altered in the plant's neighborhood. On the bottom, four more plants have been added; the density function has been modified near each of them.

Figure 9 shows point patterns generated using this algorithm with different deformation kernels. The Hopkins indices of these patterns are 0.4, 1.0, 1.2, and 2.4, confirming the visual observation that the kernel method is capable of creating a range of distributions, from overdispersed to random and clustered.

In Figure 10 points have been replaced by simple models of daisies, created using an L-system. The overdispersed pattern at the top of Figure 10 looks less realistic than the clustered pattern at the bottom, which justifies the introduction of clustering into the model.

The distributions shown in Figure 10 have been generated after initializing the probability density function

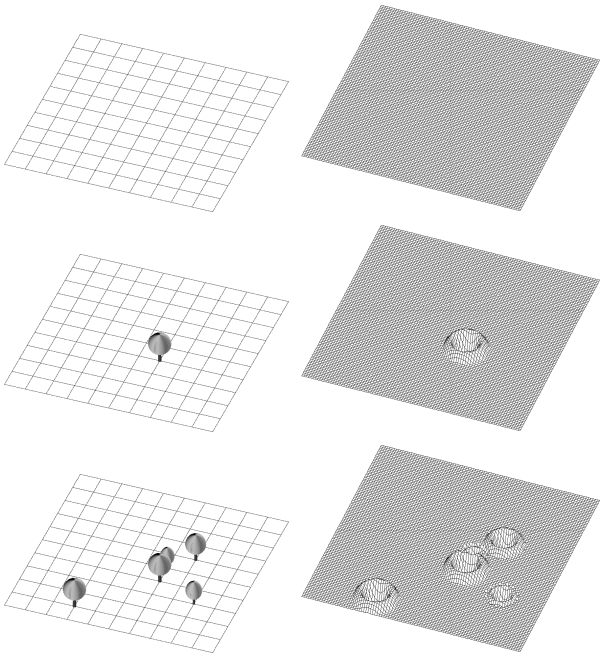


Figure 8: An example of the deformation kernel algorithm, using the kernel of Figure 7d. Left: the plant distribution. Right: the joint probability density function f. From top to bottom: the initial state, the state after placing the first plant, and the state after placing four more plants.

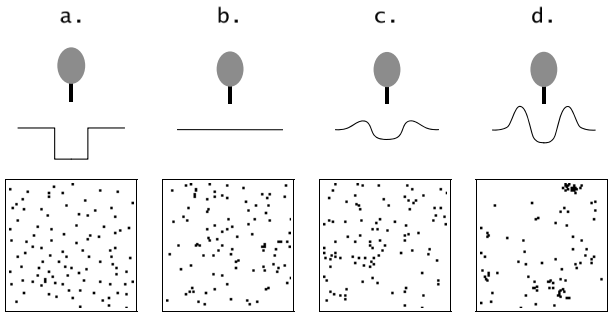


Figure 9: Some kernels and the point patterns they generate. The kernels are drawn at a larger scale than the point patterns.

f to a constant value. If, instead, we initialize f to a user-defined field (with a paint program, for example), we can generate spatial distributions of plants that conform to this field, as shown in Figure 11.

This result improves on the methodology described in [3], which allowed for the application of a user-specified density map, but did not make it possible to control the degree of plant clustering.

Figure 10: Point patterns corresponding to Figures 9a and 9c, rendered as fields of daisies.

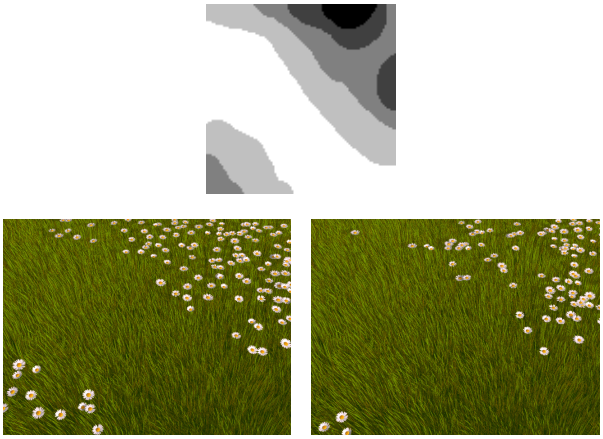


Figure 11: Plant distributions created from the user-defined density map, shown at the top. The Hopkins index values are $H \approx 1.2$ (left) and $H \approx 1.9$ (right).

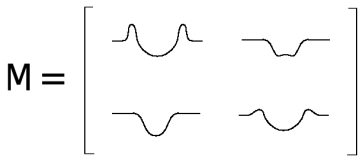


Figure 12: The concept of the kernel matrix M for two species

4.2 Extensions

The kernel-based method can be extended to include information about the plants' sizes, as well as to model the interaction of several species.

Size information is taken into account by placing plants in order of size, largest first. Larger, older plants then affect the positioning of smaller, younger plants, as it is to be expected. The actual sizes may be drawn from a distribution that gives few large plants, more plants of medium size, and still more small plants.

Plants of species 1 may have a different effect on other plants of species 1 than they do on plants of species 2. Different kernels are thus required to capture these effects. In fact, for two species, the total of four kernels is required: one for the effects species 1 has on itself, one for the effects species 1 has on species 2, one for the effects species 2 has on species 1, and one for the effects species 2 has on itself. This leads, in the general case, to a *kernel matrix* M (Figure 12), which defines the effects that each species has on itself and on each other species.

In an extension of the kernel placement method to n species, we keep track of n different probability density functions f_i. As plants are placed, each probability density function is deformed by the relevant kernel; thus if a plant of species j is placed, each function f_i is deformed by kernel $M_{i,j}$, where $i = 1, 2, \ldots, n$. This process is illustrated in Figure 13.

A sample application of the above method is presented in Figure 14. The modeling of trees incorporated in this scenes was discussed in [14].

5 Conclusions

We formalized and extended the methods for defining plant distributions that had originally been proposed in [3]. To this end, we introduced multiset L-systems, an extension of the L-system formalism, to model groups of plants, rather than single plants alone. We then applied these L-systems to simulate the essence of self-thinning, succession, and clustering of plants in an ecosystem.

We also improved the interactive techniques for generating plant distributions, making is possible to specify not only global densities, but also levels of clustering. The resulting kernel deformation method can produce plant distributions with a wide range of clustering values, from overdispersed to random to highly clustered. We have illustrated the use of this method using several models of plant communities.

Fig.	Number of			Time taken[a]		
	plants	different plants	primitives (millions)	distribution generation[b]	plants	rendering
4a	2688	20	34	9 sec	30 sec	8 min
b	1593	20	18	72 sec	27 sec	8 min
c	1409	20	30	2 min	30 sec	9 min
d	834	20	78	3 min	40 sec	11 min
5a	5584	24	141	2 min	30 sec	15 min
b	5526	24	147	22 min	30 sec	14 min
c	3427	24	100	44 min	30 sec	12 min
10	100	8	1.5	0.05 sec	2 sec	2 min
14	1599	36	147	0.5 sec	65 sec	11 min

[a] Times recorded on a 733 MHz Pentium III processor.
[b] For Figures 4 and 5, the times given show how long the simulation took to reach the given frame.

Table 2: Statistics pertinent to Figures 4, 5, 10, and 14

Both methods make it possible to generate plant ecosystems at the speeds required for their practical applications to realistic image synthesis. Statistics pertinent to the scenes included in this paper are given in Table 2.

We believe that the proposed methods can be applied to model diverse plant ecosystems and have various practical applications, such as scene dressing for computer animation purposes, and visual impact analysis of tree cutting and regrowth on the landscape. The realism of the resulting scenes can be further improved using more accurate models of the underlying biological processes, and more sophisticated rendering methods.

Acknowledgments

We thank Martin Fuhrer for the model of fireweed used in Figure 4, Lars Mündermann for the model of grass used in Figures 10 and 11, and the referees for their insightful comments. We acknowledge Radoslaw Karwowski and Radomir Měch for the development of the `L-studio/cpfg` plant modeling software, and Craig Kolb for the development of the `rayshade` ray tracer, which we used to render the scenes. The partial support of this research by grants from the Natural Sciences and Engineering Research Council of Canada is also gratefully acknowledged.

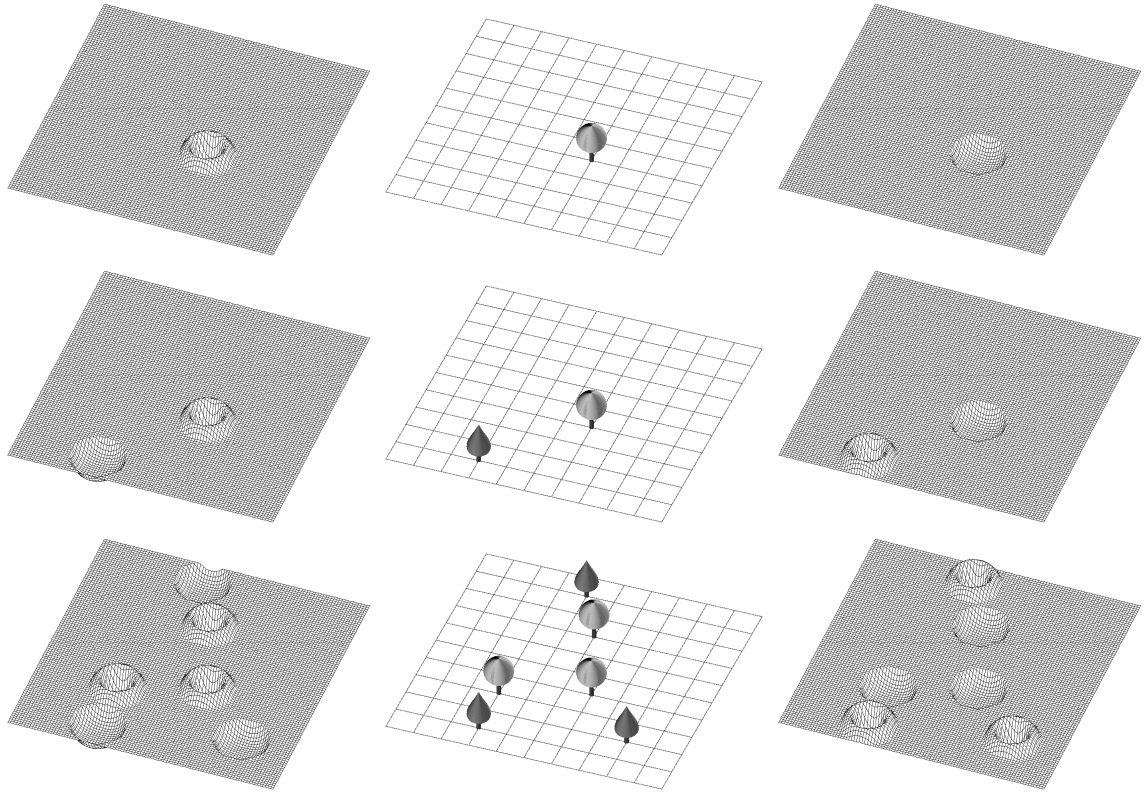


Figure 13: Generation of a two-species distribution using the deformation kernel algorithm with the kernel matrix shown in Figure 12. On the left, the probability density function of species 1; on the right, that of species 2. From top to bottom: the state of the system after placing the first plant of species 1, after placing the first plant of species 2, and after placing several plants of each species.

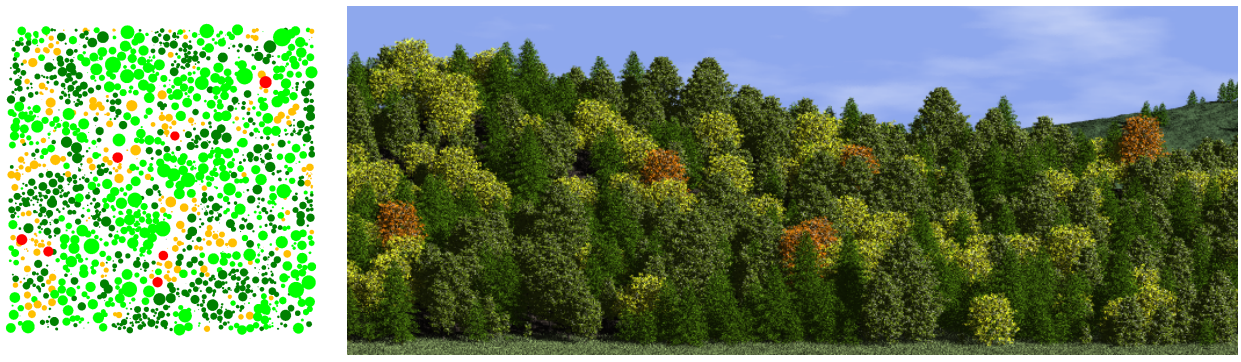


Figure 14: A forest model consisting of four species of trees. Left: distribution of plants. Right: realistic visualization of the model.

References

[1] N. Chiba, K. Muraoka, A. Doi, and J. Hosokawa. Rendering of forest scenery using 3D textures. *Journal of Visualization and Computer Animation*, 8(4):191–199, 1997.

[2] M. R. T. Dale. *Spatial Pattern Analysis in Plant Ecology*. Cambridge Studies in Ecology. Cambridge University Press, Cambridge, UK, 1999.

[3] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Proceeding of SIGGRAPH 98 (Orlando, Florida, July 19-24, 1998)*, pages 275–286, 1998.

[4] F. G. Firbank and A. R. Watkinson. A model of interference within plant monocultures. *Journal of Theoretical Biology*, 116:291–311, 1985.

[5] R. W. Floyd and L. Steinberg. An adaptive algorithm for spatial greyscale. *Proceedings of the Society for Information Display*, 17:75–77, 1975.

[6] B. Hopkins. A new method for determining the type of distribution of plant individuals. *Annals of Botany*, XVIII:213–226, 1954.

[7] B. Lane and P. Prusinkiewicz. Randomized generation of nonuniform plant distributions. *Proceedings of Western Computer Graphics Symposium 2000*, pages 61–66, 2000.

[8] L. Legendre and P. Legendre. *Numerical Ecology*. Elsevier, Amsterdam, 1983.

[9] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[10] R. Měch and P. Prusinkiewicz. Visual models of plants interacting with their environment. *Proceedings of SIGGRAPH 1996*, pages 397–410, August 1996.

[11] P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86 — Vision Interface '86*, pages 247–253, 1986.

[12] P. Prusinkiewicz, R. Karwowski, R. Měch, and J. Hanan. L-studio/cpfg: A software system for modeling plants. In M. Nagl, A. Schürr, and M. Münch, editors, *Applications of graph transformation with industrial relevance*, Lecture Notes in Computer Science 1779, pages 457–464. Springer-Verlag, Berlin, 2000.

[13] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[14] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. *Proceeding of SIGGRAPH 2001 (Los Angeles, California, August 11-17, 2001)*, pages 289–300, 2001.

[15] S. M. Ross. *Introduction to Probability Models*. Academic Press, 1997.

[16] G. Rozenberg and K. P. Lee. Developmental systems with finite axiom sets. Part I. Systems without interactions. *International Journal of Computer Mathematics*, 4:43–68, 1974.

[17] G. Rozenberg, K. Ruohonen, and A. Salomaa. Developmental systems with fragmentation. *International Journal of Computer Mathematics*, 5:177–191, 1976.

The use of positional information in the modeling of plants

Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, Brendan Lane

Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4
pwp|lars|radekk|laneb@cpsc.ucalgary.ca

Abstract

We integrate into plant models three elements of plant representation identified as important by artists: posture (manifested in curved stems and elongated leaves), gradual variation of features, and the progression of the drawing process from overall silhouette to local details. The resulting algorithms increase the visual realism of plant models by offering an intuitive control over plant form and supporting an interactive modeling process. The algorithms are united by the concept of expressing local attributes of plant architecture as functions of their location along the stems.

CR categories: F.4.2 **[Mathematical Logic and Formal Languages]**: Grammars and Other Rewriting Systems, I.3.5 **[Computer Graphics]**: Computational Geometry and Object Modeling, J.3 **[Life and Medical Sciences]**: Biology.

Keywords: realistic image synthesis, interactive procedural modeling, plant, positional information, phyllotaxis, Chomsky grammar, L-system, differential turtle geometry, generalized cylinder.

1 Introduction

Forward simulation of development is a well established paradigm for modeling plants. It underlies, for example, the AMAP simulation software [9] and modeling methods based on L-systems [28]. In both cases, a plant is modeled using a set of rules that describe the emergence and growth of individual plant components. The simulation program traces their fate over time, and integrates them into the structure of the whole plant.

Over the years, the simulation paradigm has been extended to include a wide range of interactions between plants and their environments [15, 21]. The resulting models have gained acceptance as a research tool in biology and have led to increasingly convincing visualizations. In image synthesis applications, however, the simulation-based approach has several drawbacks:

- Visual realism of the models is linked to the biological and physical accuracy of simulations. This requires the modeler to have a good understanding of the underlying processes, makes comprehensive models complicated, and results in long simulation times.

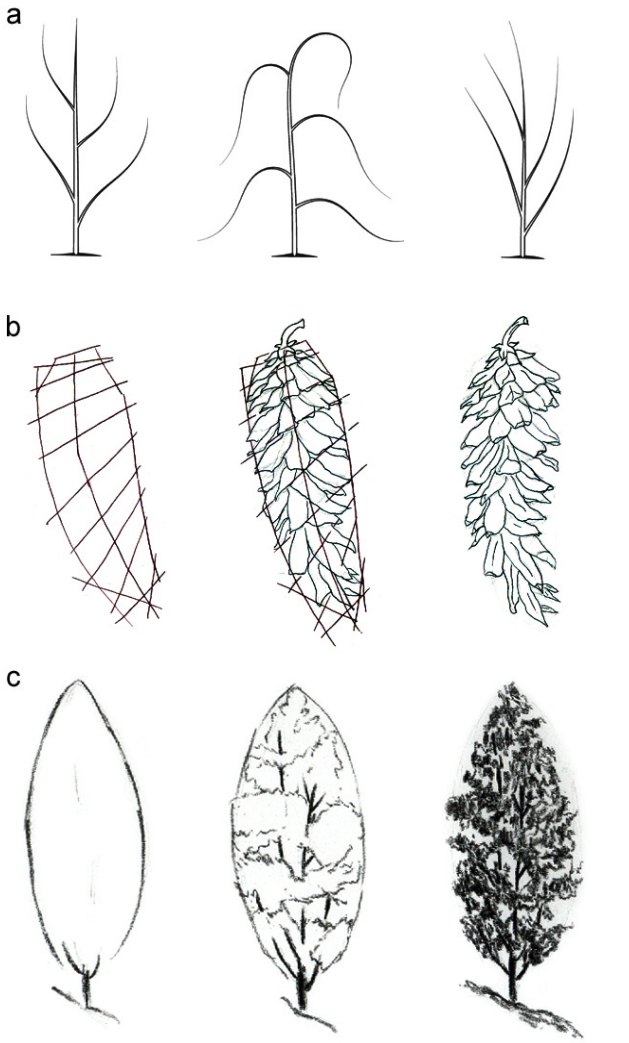


Figure 1: Selected elements of the artistic representation of plants: (a) posture, (b) regular arrangement and gradual variation of organs along an axis, and (c) progression from silhouette to detail in the drawing process. Figure (a) is based on [37, page 41], (b) is redrawn from [38, page 68], and (c) is redrawn from [24, page 13].

- Global characteristics of plant appearance, such as the curvature of plant axes, the density of organ distribution, and the overall silhouette of the plant, are emergent properties of the models and therefore are difficult to control.

Methods for creating visually realistic representations of plants

ACM SIGGRAPH 2001 (Los Angeles, California, August12-17, 2001), pp. 289 - 300.

have long been understood by artists (Figure 1). Important elements include plant *posture*, defined by the angles of insertion and curvature of organs, and the *arrangement* and gradual *variation* of organs on their supporting stems. The drawing process progresses in a *global-to-local* fashion, from silhouette to detail. Inspired by the quality of botanical illustrations, we have developed a plant modeling method that supports similar elements and processes. The proposed method *inverts* the local-to-global operation of simulation-based models by progressing from global plant characteristics specified by the user to algorithmically generated details. The algorithms are united by their use of *positional information*, which we define as the position of plant components along the *axes* of their supporting stems or branches. User-defined functions map this information to *morphogenetic gradients* [2], which describe the distributions of features along the axes.

The notions of positional information and morphogenetic gradients unify and generalize several plant-modeling concepts that have already appeared in botanical and computer graphics literature. Following their review (Section 2), we outline our modeling software environment, focusing on the language that we use to formally describe the algorithms and models (Section 3). We then develop the mathematical foundations of plant modeling based on positional information: the modeling and framing of individual axes (Section 4), and their partitioning into internodes (Section 5). In Section 6, we present the resulting modeling method from the modeler's perspective, and illustrate its applications using plants and plant structures organized along a single axis. In Section 7, we address the important special case of organ arrangement in closely packed spiral phyllotactic patterns. Finally, in Section 8, we extend the proposed modeling method to plants with higher-order branches, including trees. We conclude the paper with a discussion of the results, applications, and problems open for further research (Section 9). Proofs of selected mathematical results pertinent to the use of positional information in the modeling of plants are presented in the Appendices.

2 Previous work

Applications of positional information have their origins in early descriptive plant models created by biologists: the poplar model by Burk *et al.* [7] and the larch sapling model by Remphrey and Powell [30]. In both cases, the length of lateral branches was expressed as a function of their position on the main stem. The models were visualized as two-dimensional line drawings.

In computer graphics, a related concept was first applied to model trees by Reeves and Blau [29], who expressed the length of first-order branches as the distance from the branching point to a user-specified surface defining the silhouette of the tree. Higher-order branches were generated algorithmically, with "many parameters inherited from the parent."

A more elaborated model was introduced by Weber and Penn [36]. They characterized a tree using several positional functions, and pointed to an advantage of this technique: "Since our parameters can address the character of an entire stem and not just its segment-to-segment nature, we allow users to make changes on a level they can more easily understand and visualize."

Lintermann and Deussen incorporated positional information into their interactive plant modeling program `xfrog` [18, 19]. The position of a sample point along an axis may affect the length of an internode, the length of a branch, the magnitude of a branching angle, and other attributes. Functions that map positions to attribute values

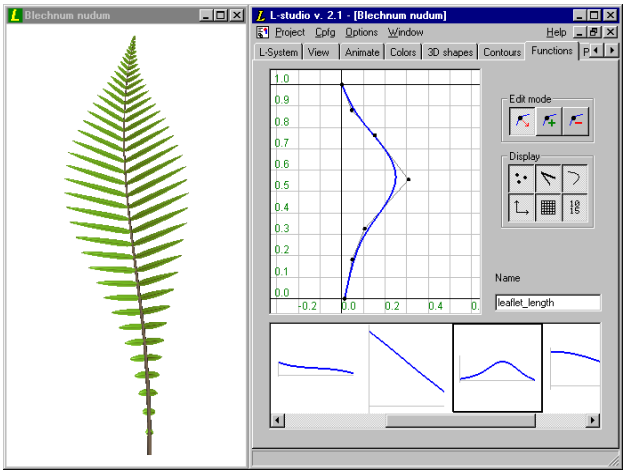


Figure 2: A snapshot of the L-studio/cpfg screen. The model can be manipulated using textual and graphical editors displayed on the right side of the screen. In this example, the outline of the fishbone water fern leaf (*Blechnum nudum*) is being defined using a graphical function editor. A *gallery* under the editor's window provides access to various functions used in this model. The second row of tabs near the top of the screen makes it possible to select other editors, such as the textual editor of the L-system that has been used to specify the algorithmic structure of this model.

can be specified graphically, by editing function plots, or textually, by editing algebraic expressions. The authors did not describe in detail the algorithms underlying their software, but experience with `xfrog` was an inspiration for our work. From the user interface perspective, the editing of function plots is an extension of the interactive manipulation of plant parameters using sliders [23, 28].

3 The modeling environment

We have adapted the L-system-based modeling software `L-studio/cpfg` [27] to the needs of modeling using positional information. A screenshot of the system in operation is shown in Figure 2.

An L-studio model consists of two basic components: a description of a generative algorithm in the `cpfg` modeling language [25], and a set of graphically defined entities. These entities can be defined and manipulated using the L-studio *function*, *curve*, *surface* and *material* editors [27], or imported from external sources.

The fundamental constructs of the `cpfg` language are *rewriting rules*, or *productions*. The program supports both parallel application of productions, characteristic of L-systems [28], and sequential application of productions, characteristic of Chomsky grammars [8]. In the context of plant modeling, these formalisms compare as follows.

L-system productions capture the *development* of plant components over time. For example, the division of a mother cell A into two daughter cells B and C can be described by the production $A \rightarrow BC$. In the case of multicellular organisms, L-system productions are applied in *parallel* to advance time consistently in all cells. The simulation is completed when the organism reaches a predefined *terminal age*, corresponding to a given number of derivation steps.

Chomsky grammars, in contrast, characterize the *structure* of plants, that is, the distribution of their features and components in space. The fact that organism A consists of parts B and C can again be expressed by a production, for example $A \rightsquigarrow BC$, but such a *decomposition rule* has a different meaning and functions in a different way than its L-system counterpart. Since non-overlapping substructures can be partitioned independently from each other, the decomposition rules may be applied sequentially. Furthermore, the appropriate condition for terminating a decomposition process is the reaching of *terminal symbols*, which represent components that cannot be divided further.

Our intended use of positional information is to capture the distribution of plant features and components in space. Consequently, the meaning and formal properties of productions used in this paper correspond with the definition of Chomsky grammars. In the big picture of a complete plant modeling software design, the switch from L-systems to Chomsky grammars amounts to a relatively minor modification of the code. Consequently, our modeling language, outlined below, expands, rather than replaces, features of the earlier purely L-system-based implementations of the `cpfg` modeling language [13, 28].

As in the case of L-systems, a branching structure is represented by a *bracketed string* of *modules* (symbols with associated parameters). Matching pairs of brackets enclose branches. Derivation begins with an initial string identified by the keyword `axiom`. *Context-free* productions are specified using the syntax

$$pred : \{block_1\} \;\; cond \;\; \{block_2\} \rightsquigarrow succ, \qquad (1)$$

where $pred$ is the predecessor (a single module), and $succ$ is the successor (a bracketed string of modules) [25]. The optional field $cond$ is the condition (logical expression) that guards production application. Fields $block_1$ and $block_2$ are sequences of C statements. The first block is executed before the evaluation of the condition. If the condition is true, the second block is also evaluated and the production is applied. For example, the rule

$$A(x) : \{y = x + 2;\} \;\; y \geq 5 \;\; \{z = y/3;\} \rightsquigarrow B(z)C(z+1) \quad (2)$$

can be applied to module A(4), subdividing it into modules $B(2)C(3)$.

The `cpfg` language also supports *context-sensitive* productions, in which the strict predecessor (module being replaced) $pred$ may be preceded by one or more modules constituting the *left context*, and/or followed by modules constituting the *right context*. These contexts are separated from the strict predecessor by symbols $<$ and $>$ respectively. For example, production

$$A(x) < B(y) > C(z) : x + z > 0 \rightsquigarrow M(y/2)N(y/2) \qquad (3)$$

decomposes module B into a pair of modules M and N, provided that module B appears in the context of modules A and C, and the sum of their parameters is greater then 0. In the scope of this paper, context is limited to query symbols, discussed later on.

In order to conveniently specify morphogenetic gradients inherent in the use of positional information, we have extended the `cpfg` modeling language with function calls of the form $\mathrm{func}(id, x)$. The integer number id is the identifier of a planar B-spline curve and the real number x is the function argument. Function plots are manipulated using the *interactive function editor* (Figure 2). It constrains the motion of the control points that define the function plots to guarantee that they assign a unique value y to each argument x.

The modeling language also supports function calls of the form $\mathrm{curveX}(id, s)$, $\mathrm{curveY}(id, s)$, $\mathrm{curveZ}(id, s)$, and $\mathrm{tanX}(id, s)$, $\mathrm{tanY}(id, s)$, $\mathrm{tanZ}(id, s)$, where id is the identifier of an arbitrary B-spline curve. These calls return coordinates of a point on the curve id and of the tangent vector at this point, given the arc-length distance s from the curve origin. The call $\mathrm{curveLen}(id)$ returns the total length of the curve.

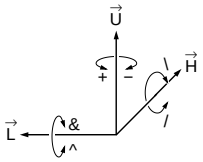


Figure 3: Controlling the turtle in three dimensions.

To create a graphical model, the derived string is scanned sequentially and reserved modules are interpreted as commands to a LOGO-style turtle [28]. At any point within the string, the *turtle state* is characterized by a position vector $\vec{P}$ and three mutually perpendicular orientation vectors $\vec{H}$, $\vec{L}$, and $\vec{U}$ that indicate the turtle's *heading*, the direction to the *left*, and the *up* direction (Figure 3). The coordinates of these vectors can be accessed using *query* modules of the form $?X(x, y, z)$, where X is the vector to be accessed, one of P, H, L, or U [26]. Module F causes the turtle to draw a line in the current direction, while modules f causes the turtle to move without drawing a line. Modules $+$, $-$, $\&$, $\wedge$, $/$, and $\backslash$ rotate the turtle around one of the vectors $\vec{H}$, $\vec{L}$, or $\vec{U}$, as shown in Figure 3. Many symbols are overloaded; for example, $+$ and $-$ denote the modules that rotate the turtle as well as the usual arithmetic operations. The length of the line and the magnitude of the rotation angle can be given globally or specified as parameters of individual modules. Branches are created using a stack mechanism: the opening square bracket pushes the current state of the turtle on the stack, and the closing bracket restores it to the last saved state. Other interpreted symbols will be introduced with the sample models.

4 Modeling curved limbs

The shape of curved limbs, such as stems and elongated leaves, is "vital in capturing the character of a species" [37]. In computer graphics, this was first recognized by Bloomenthal [5], who applied *generalized cylinders* to model tree branches. A generalized cylinder is obtained by sweeping a planar *generating curve*, which determines the organ's *cross section*, along a *carrier curve* [16] that defines the organ's *axis*. The generating curve may be closed, as is typically the case for stems, or open, as for thin leaves, and it may change size and shape while being swept [33]. It also must be properly oriented with respect to the carrier curve. The *Frenet frame* [34], which is frequently used for this purpose, creates well known problems along straight sections of the carrier curve and at inflection points, where it is not defined. It also twists 180° in the proximity of the inflection points [6]. To avoid these problems, we propose an alternative solution based on the use of turtle geometry. This solution subsumes the Frenet frame, as well as the twist-minimizing *parallel transport frame* [3, 6, 14], as special cases. The turtle frame was previously used by Jirasek *et al.* [15] in the context of biomechanical modeling of plant branches.

The carrier curve is defined as a sequence of infinitesimal turtle movements. Let s denote the arc-length distance of the turtle from the origin of this curve. To define a smooth curve, we specify functions $\omega_H(s)$, $\omega_L(s)$ and $\omega_U(s)$ that characterize the *rates* of turtle's rotations around the axes $\vec{H}\vec{L}\vec{U}$ as the turtle moves (we use the term "rate of rotation" although s is a spatial coordinate and not time). The infinitesimal rotations $d\Omega_H$, $d\Omega_L$ and $d\Omega_U$ between

curve points $\vec{P}(s)$ and $\vec{P}(s+ds)$ are then given by the equations:

$$d\Omega_H = \omega_H(s)ds, \quad d\Omega_L = \omega_L(s)ds, \quad d\Omega_U = \omega_U(s)ds. \qquad (4)$$

This specification yields a uniquely defined curve and moving reference frame (Appendix A.1). After replacing the infinitesimal increments ds by finite increments Δs, we obtain the following straightforward algorithm for modeling elongated plant organs:

Algorithm 1

```
1   #define ℓ              1.0   /* total axis length */
2   #define 𝒢              7     /* cross section ID */
3   #define Δs             0.02  /* turtle step */
4
5   #define ω_L(s)         func(1,s)
6   #define ω_U(s)         func(2,s)
7   #define ω_H(s)         func(3,s)
8   #define φ(s)           func(4,s)
9   #define width(s)       func(5,s)
10
11  Axiom: @#(𝒢) A(0,0)
12
13  A(s,φ): s ≤ ℓ
14        { ΔΩ_L = ω_L(s)Δs;
15          ΔΩ_H = ω_H(s)Δs;
16          ΔΩ_U = ω_U(s)Δs;
17          φ    = φ + φ(s)Δs; } ⇝
18              +(ΔΩ_L) &(ΔΩ_U) /(ΔΩ_H)
19              /(φ) #(width(s)) F(Δs) \(φ)
20              A(s + Δs,φ)
21
22  A(s,Φ): s > ℓ ⇝ ϵ
```

Following the implementation of generalized cylinders in the `cpfg` program [20], the generating curve is selected by expression $@\#(\mathcal{G})$ in the axiom (line 11). The generalized cylinder is created recursively by the first production (lines 13-20) as a sequence of slices of length Δs. The cross section size is defined by module # with the parameter $\mathrm{width}(s)$ (line 19), and is linearly interpolated between points s and $s+\Delta s$. The angles of turtle rotation are calculated according to Equation 4 in lines 14–16, and applied to the turtle in line 18. The order of rotations represented by the symbols $+$, $\&$ and $/$ in line 18 is arbitrary, since infinitesimal rotations commute. Function $\phi(s)$ (line 17) rotates the generating curve around the cylinder axis without affecting the shape of the axis. This is convenient when defining twisted organs. The second production (line 22) removes the apex A at the end of cylinder generation, by replacing it with the empty symbol ϵ. Figure 4 shows sample leaves and stems generated by this Algorithm, with all functions specified using the interactive function editor (Section 3).

From the user's perspective, functions ω_L, ω_U, ω_H, ϕ and `width`, control bending, twist, and tapering of a generalized cylinder. Our experience confirms Barr's observation that such deformations are intuitive operations for modeling three-dimensional objects [1]. On the other hand, the user may prefer to specify the shape of an axis directly, for example as a spline curve. If this is the case, we *frame* it (*i.e.*, compute turtle's rotations $d\Omega_U$, $d\Omega_L$ and $d\Omega_H$) as follows.

Let $\vec{P}(s), s \in [0,\ell]$, be a given smooth curve. Assume that it has been framed by a moving turtle; the turtle's heading vector $\vec{H}$ thus coincides with the tangent vector $\vec{T}$ to the curve for all $s \in [0,\ell]$. Denote by $\vec{H}\vec{L}\vec{U}$ the turtle orientation at point $\vec{P}(s)$ of this curve

Figure 4: Leaves and stems of a herb lily (left) and tulip (right), modeled using Algorithm 1. The models are based on drawings in [38, pp. 56 and 58].

and by $\vec{H}' = \vec{H} + d\vec{H}$ the direction of the heading vector at point $\vec{P}(s+ds)$. Following [12], the infinitesimal rotation $d\vec{\Omega}$ that changes vector $\vec{H}$ to $\vec{H}'$ satisfies the equation $d\vec{H} = d\vec{\Omega} \times \vec{H}$, hence:

$$d\vec{H} = d\vec{\Omega} \times \vec{H} = (\vec{U}d\Omega_U + \vec{L}d\Omega_L + \vec{H}d\Omega_H) \times \vec{H} \qquad (5)$$

$$= (\vec{U} \times \vec{H})d\Omega_U + (\vec{L} \times \vec{H})d\Omega_L + (\vec{H} \times \vec{H})d\Omega_H \qquad (6)$$

$$= \vec{L}d\Omega_U - \vec{U}d\Omega_L + 0d\Omega_H. \qquad (7)$$

By taking dot products of the first and last expression with vectors $\vec{L}$ and $\vec{U}$, we obtain:

$$d\vec{H} \cdot \vec{L} = (\vec{H}' - \vec{H}) \cdot \vec{L} = \vec{H}' \cdot \vec{L} = d\Omega_U, \qquad (8)$$

$$d\vec{H} \cdot \vec{U} = (\vec{H}' - \vec{H}) \cdot \vec{U} = \vec{H}' \cdot \vec{U} = -d\Omega_L. \qquad (9)$$

By substituting $\vec{T}'$ for $\vec{H}'$ to emphasize that $\vec{T}'$ is a given tangent vector to the curve being framed, we obtain finally:

$$d\Omega_U = \vec{T}' \cdot \vec{L} \quad \text{and} \quad d\Omega_L = -\vec{T}' \cdot \vec{U}. \qquad (10)$$

Equations 10 constrain two rotational degrees of freedom. The third angle $d\Omega_H$ remains unconstrained, because it is multiplied by 0 in Equation 7. This implies that a moving turtle frame can be assigned to a given curve in different ways. In particular, if we set $\omega_H(s)$ in such a way that vector $\vec{L}$ (or $\vec{U}$) always lies in the osculating plane, we obtain the Frenet frame, and if $\omega_H(s) \equiv 0$, we obtain the parallel transport frame. We commonly use the latter, because it minimizes rotations of the reference frame around the axis of the generalized cylinder. The resulting algorithm for approximating and framing a given curve $\vec{P}(s)$ using a sequence of turtle motions is given below.

Algorithm 2

```
1   #define 𝒫   1       /* curve ID */
2   #define K   57.29   /* radians to degrees */
3
4   Axiom: A(0) ?U(0,0,0) ?L(0,0,0)
5
6   A(s) > ?U(u_x,u_y,u_z) ?L(l_x,l_y,l_z) : { s' = s + Δs } s' ≤ ℓ
7         { t'_x = tanX(𝒫,s'); t'_y = tanY(𝒫,s'); t'_z = tanZ(𝒫,s');
8           ΔΩ_L = K * (t'_x l_x + t'_y l_y + t'_z l_z);
9           ΔΩ_U = −K * (t'_x u_x + t'_y u_y + t'_z u_z); } ⇝
10              +(ΔΩ_U) &(ΔΩ_L) F(Δs) A(s')
```

The initial structure consists of apex A followed by query modules $?U$ and $?L$ (line 4). The parameter of the apex represents the current position of the turtle, measured as its arc-length distance from

the origin of curve $\mathcal{P}$. The production (lines 6 to 10) creates an organ axis as a sequence of generalized cylinder slices of length Δs, as in Algorithm 1 (functions controlling the orientation and size of the generating curve have been omitted here for simplicity). Specifically, rotations $\Delta\Omega_U$ and $\Delta\Omega_L$ are calculated by multiplying (dot product) the vectors $\vec{U}$ and $\vec{L}$ (lines 8 and 9) returned by the query modules $?U$ and $?L$ (line 6) with the tangent vector to the curve $\mathcal{P}$ returned by the tanX, tanY and tanZ function calls (line 7). The values $\Delta\Omega_U$ and $\Delta\Omega_L$ orient the next segment of the curve, represented by module F(Δs) in line 10. House-keeping productions that erase modules A, $?U$ and $?L$ at the end of the derivation have been omitted from this listing.

Figure 5: *Allum vineale* (field garlic), modeled using Algorithm 2 after the photograph in [4].

A sample application of Algorithm 2 is shown in Figure 5. Stems of a dry garlic plant have been modeled interactively, then framed using Algorithm 2 to orient the generating curve. Although the generating curve is circular in this case, its orientation is important for proper polygonization of the resulting generalized cylinders.

The turtle frame also plays an important role in orienting the organs and branches that are attached to an axis. Before discussing this in detail, we will consider the spacing of organs along an axis.

5 Organ spacing

We call points at which organs are attached to an axis the *nodes*, and the axis segments delimited by them the *internodes*. Let $\{s_i\}, i = 0, 1, \ldots$, be a sequence of node positions on an axis, and $\{l_i = s_{i+1} - s_i\}$ be the associated sequence of internode lengths (Figure 6a). It is straightforward to define the internode lengths using a function λ of the position of one of its incident nodes, for instance using the formula $l_i = s_{i+1} - s_i = \lambda(s_i)$. Unfortunately, with this definition function λ does not provide a robust control over the node distribution, because a small change in the position of the initial node s_0 may result in a totally different sequence of the nodes that follow. For example, if $s_0 = 0$, the function λ shown in Figure 6b will yield the sequence of node positions $\{s_i\} = 0, 1, 2, 3, \ldots$ (internode length equal to 1), but if $s_0' = 0.25$, the sequence of node positions will be $\{s_i'\} = 0.25, 0.75, 1.25, 1.75, \ldots$ (internode length 0.5).

To achieve a more stable behavior, we observe that $1/\lambda(s)$ can be interpreted as the local *density* of nodes, in the sense that the integer part of the integral

$$N(s_o, s) = \int_{s_0}^{s} \frac{ds}{\lambda(s)} \qquad (11)$$

represents the number of internodes between node s_0 and point s on the axis. Thus, given the initial node s_0, positions of the subsequent nodes correspond to the integer increments of the value of function N, that is, $N(s_o, s_{i+1}) = N(s_o, s_i) + 1$ (Figure 6c). The sequence of nodes $\{s_i\}$ defined this way is no longer critically sensitive to the initial node position s_0. Specifically, in Appendix A.2 we prove

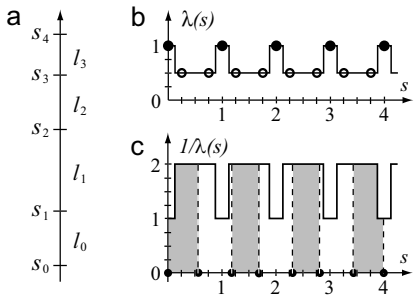


Figure 6: Partitioning an axis into segments. (a) The labeling of nodes and internodes. (b) Positional information represents the internode length. The same function $\lambda(s)$ generates very different node sequences (filled and empty circles), depending on the position of the initial node. (c) Positional information represents node density. Nodes are placed at the locations corresponding to the unit areas under the curve $1/\lambda(s)$. This definition leads to a more stable node spacing than (b).

that for any two node sequences $\{s_i\}$, $\{s_i'\}$ such that $s_0 < s_0' < s_1$, the elements of both sequences interleave: $s_i < s_i' < s_{i+1}$ for all $i = 0, 1, 2, \ldots$.

Specification of node spacing based on Equation 11 also has other useful properties. First, if $\lambda(s)$ has a constant value l between nodes s_i and s_{i+1}, then l is equal to the internode length:

$$\int_{s_i}^{s_{i+1}} \frac{ds}{l} = 1 \;\text{ implies }\; s_{i+1} - s_i = l. \qquad (12)$$

Second, if $\lambda(s)$ is a linear function, $\lambda(s) = as + b$, the length of consecutive internodes changes in a geometric sequence, $l_{i+1} = e^a l_i$ (proof in Appendix A.3). The ease of defining geometric sequences is important, because their approximations are often observed in nature (according to Niklas, they form the "null hypothesis" [22]).

The algorithm for placing nodes according to a given function $\lambda(s)$ is presented below.

Algorithm 3

```
1   Axiom: A(0,0)
2
3   A(s,a) : { s' = s + Δs }  s' ≤ ℓ
4         { a' = a + Δs/λ(s) ;
5           if (a' < 1) { flag = 0; }
6           else { a' = a' − 1; flag = 1; } } ⇝
7                 F(Δs) B(flag) A(s',a')
8
9   B(flag) : flag == 0 ⇝ ϵ
10  B(flag) : flag == 1 ⇝ @o
```

The initial structure consists of apex A (line 1). The first parameter represents the distance of the current point on the axis from the axis base, as in Algorithms 1 and 2. The second parameter represents the fractional part of the integral $N(0, s)$ given by Equation 11. The production in lines 3 to 7 creates the axis as a sequence of segments F of length Δs, separated by markers of potential node locations B. If the $flag$ is zero, module B is subsequently erased (line 9). When a exceeds 1, the $flag$ is set (line 6) to produce a node marked by symbols $@o$ (line 10).

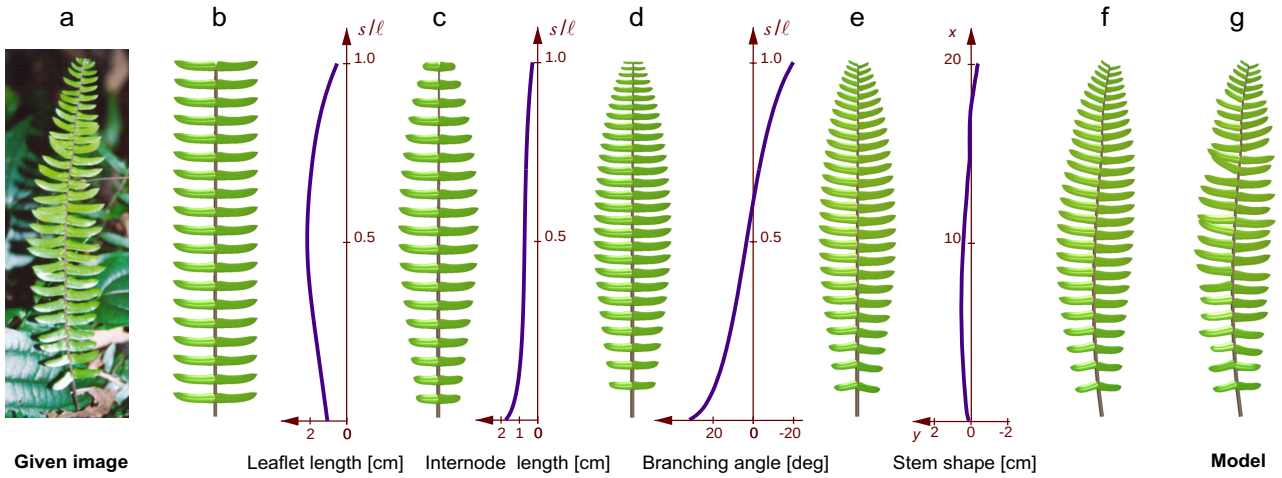


Figure 7: Using positional information to model a *Pellaea falcata* (sickle fern) leaf.

6 Modeling single-compound plant structures

We have combined the methods for framing and partitioning an axis into the following algorithm, which makes it possible to model a variety of *single-compound* structures (sequences of organs supported by a single stem). Definitions of graphical functions and constants used in previous algorithms have not been included. Secondary features, such as the randomization of values returned by functions, have also been omitted.

Algorithm 4

```
1   #define Φ   0          /* phyllotactic angle */
2
3   Axiom: A(0,0,0) ?U(0,0,0) ?L(0,0,0)
4
5   A(s,a,φ) > ?U(u_x,u_y,u_z) ?L(l_x,l_y,l_z) :
6              { s' = s + Δs } s' ≤ ℓ
7       { t'_x = tanX(P,s'); t'_y = tanY(P,s'); t'_z = tanZ(P,s');
8         ΔΩ_L = K * (t'_x l_x + t'_y l_y + t'_z l_z);
9         ΔΩ_U = −K * (t'_x u_x + t'_y u_y + t'_z u_z);
10          a   = a + Δs/λ(s)
11        if (a < 1) { flag = 0; }
12        else { a = a − 1; flag = 1; φ = φ + Φ; } } ⇝
13             +(ΔΩ_U) &(ΔΩ_L) #(stem_width(s))
14             F(Δs)B(s,φ,flag) A(s',a,φ)
15
16  B(s,φ,flag) : flag == 0 ⇝ ε
17  B(s,φ,flag) : flag == 1
18      { l = length(s); w = width(s); } ⇝
19            [ /(φ) [ +(brangle(s)) ~L(l,w) ]
20                   [ −(brangle(s)) ~L(l,w) ] ]
```

The key new element is the third production (lines 17 to 20), which inserts a pair of organs at the node. The organs are defined as instances of a predefined surface L, with the length, width and angle of insertion determined by functions of position s.

In order to present the operation of Algorithm 4 from a user's perspective, let us consider the process of modeling a *Pellaea falcata* (sickle fern) leaf. The photograph of the target structure is shown in Figure 7a. Construction begins with a generic single-compound (pinnate) leaf (b), which is generated when all graphically defined functions are set to their default constant values. The length of the leaflets is then modified as a function of their position on the stem (c). Since the leaf silhouette is determined by the extent of its component leaflets, this function controls the overall leaf shape. The next two functions define the lengths of the internodes (d) and the values of the branching angles between the stem and the leaflets (e). The stem shape is then established by manipulating a parametric curve (f). Finally, the branching angles and the leaflet lengths are randomized to capture the unorganized variation present in the original leaf (g). The model also makes use of functions that have not been shown in Figure 7, which define the taper of the stem and the width of the leaflets.

In the above example, the individual leaflets have been modeled as predefined surfaces L, scaled in length and width using functions of their position on the stem (lines 18 to 20 in Algorithm 4). Leaves, petals and similar organs can also be modeled as generalized cylinders with Algorithm 1. We use this technique in most

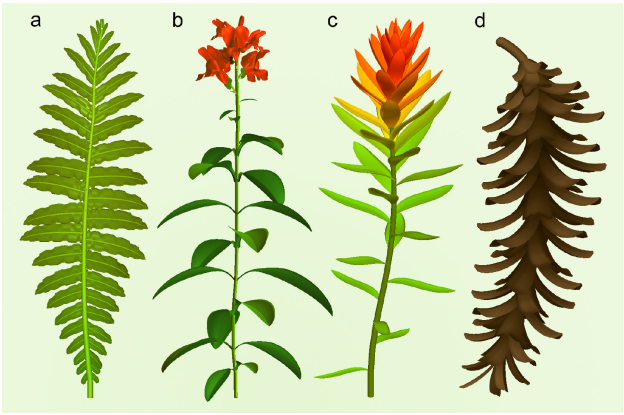


Figure 8: Plants and plant organs with different phyllotactic patterns: (a) *Blechnum gibbum* leaf with the distichous arrangement of leaflets, (b) *Antirrhinum majus* (snapdragon) plant with a decussate arrangement of leaves, (c,d) *Casilleja coccinea* (Indian paintbrush) plant and *Pinus strobus* (white pine) cone with spiral arrangements of leaves and scales.

models, because it allows us to define and manipulate organ shapes more easily. For example, the rippled surface of the *Blechnum gibbum* leaflets (Figure 8a) was obtained by randomly changing the shape, size and orientation of the generating curve.

Constant Φ in Algorithm 4 controls *phyllotaxis*, or the arrangement of organs around the stem [28]. If $\Phi = 0$, organs are arranged in a planar *distichous* pattern, as in Figures 7 and 8a. If $\Phi = 90^\circ$, consecutive pairs of organs are issued in mutually perpendicular planes, forming a *decussate* pattern (Figure 8b). Finally, if $\Phi = 137.5^\circ$ (the golden angle), and only one organ is attached to each node (line 20 of Algorithm 4 is removed), a *spiral* phyllotactic pattern results (Figure 8c and d). Thus, a change in a single constant extends Algorithm 4 to three dimensions.

Figure 9: *Helichrysum bracteatum* (strawflower).

A distinctive feature of *Helichrysum bracteatum* (a strawflower, Figure 9) is the posture of petals (ray florets), which are more curved near the center of the flower head than on the outside. To capture this gradient, the position of the petals on the main axis of the flower head was used to interpolate between two curves that describe the extreme postures of the petals. A similar technique made it possible to control the shape of leaves and petals in the beargrass model (Figure 10). Photographs of the inflorescences that we used as a reference to construct this model are shown in Figure 11.

7 Compact phyllotactic patterns

In spiral phyllotactic patterns, the individual organs, *e.g.* petals, florets, or scales, are often densely packed on their supporting surface (the *receptacle*), as illustrated by the model of beargrass. Modeling such patterns using Algorithm 4 requires a coordinated manipulation of the radius of the receptacle, the size of the organs being placed, and their vertical displacement (corresponding to the internode length). In this section we facilitate the modeling process by relating the vertical displacement to the radius of the supporting surface and the size of organs. Both the radius and the organ size can be defined as functions of organ position on the receptacle, making it possible to capture a wide range of forms and patterns. The proposed model has the same generative power as the *collision-based* model of phyllotaxis introduced by Fowler *et al.* [11], but operates faster because it avoids the explicit detection of collisions between organs.

Vogel [35] provided the first mathematical description of phyllotactic patterns used for computer graphics purposes [28]. His model places equally sized organs on the surface of a flat disk, stating that the n-th organ will have polar coordinates:

$$\phi = n \cdot 137.5^\circ, \quad r = c\sqrt{n}, n = 1, 2, \ldots \tag{13}$$

where c is a constant. The angular displacement of 137.5° between consecutive organs is treated as empirical data, reproduced but not explained by the model. The formula for the radial displacement r is justified by two observations: (a) since organs are placed from the disk center outwards, the ordering number n of the organ placed at a distance r from the center is equal to the total number of organs

Figure 10: Model of *Xerophyllum tenax* (beargrass).

Figure 11: Photographs of *Xerophyllum tenax* inflorescences.

that occupy a disk of radius r, and (b) if all organs occupy the same area, the total number n of organs in a disk of radius r will be proportional to r^2, hence $r = c\sqrt{n}$.

Vogel's model abstracts from the shape of organs and places them in a disk according to the area they occupy. Lintermann and Deussen proposed a similar approximation to derive a formula for placing organs on the surface of a sphere [19]. Both approaches are subsumed by the model of Ridley [31], which operates on arbitrary surfaces of revolution. Our algorithm is based on Ridley's analysis.

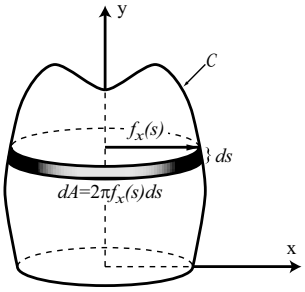


Figure 12: A receptacle.

Let $(f_x(s), f_y(s)), \quad s \in [0, L]$ be a parametric definition of a planar curve $\mathcal{C}$ that generates the receptacle when rotated around the y axis of the coordinate system (Figure 12). We assume natural parameterization of the curve $\mathcal{C}$, which means that parameter s is the arc-length distance of point $(f_x(s), f_y(s))$ from the origin of this curve. The area dA of the infinitesimal slice of the receptacle generated by the arc $[s, s + ds]$ is then equal to $2\pi f_x(s)ds$ (Figure 12). We denote by $\pi\rho^2(s)$ the area occupied by an organ placed on the receptacle at a distance s from the origin of the generating curve $\mathcal{C}$. As in the case of partitioning an axis into internodes (Section 5), we can interpret $1/\pi\rho^2(s)$ as the organ density at s. The integer part of the integral

$$N(0, s) = \int_0^s \frac{2\pi f_x(s)}{\pi\rho^2(s)} ds = \int_0^s \frac{2 f_x(s)}{\rho^2(s)} ds \qquad (14)$$

is then equal to the total number of organs placed in the portion $[0, s]$ of the receptacle. Consecutive organs are placed at locations that increment $N(0, s)$ by one. This leads to the following algorithm:

Algorithm 5

```
1   #define C      1              /* generating curve ID */
2   #define ℓ      curveLen(C)    /* length of curve C */
3   #define ρ(s)   func(2,s)      /* density function */
4   #define Δs     0.001          /* integration step */
5
6   Axiom: A(0,0)
7
8   A(s,a) : s < ℓ
9       { while( a < 1 && s < ℓ)
10        { x = curveX C,s);
11          a = a + (2x/ρ²(s))Δs;
12          s = s + Δs;
13        }
14      a = a − 1; y = curveY(C, s);
15    }
16    ⇝ [f(y)-(90)f(x)~O(ρ(s))] \(137.5) A(s,a)
```

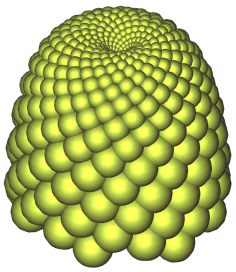


Figure 13: Example of a compact phyllotactic pattern generated using Algorithm 5.

The first parameter of module A represents the arc-length distance s of the current point from the base of the receptacle. The second parameter is the fractional part a of the integral $N(0, s)$ (Equation 14). The integration is performed incrementally by the *while* loop inside the production (lines 9 to 13). When the integral reaches 1, an organ O of radius $\rho(s)$ is placed at height y and distance x from the receptacle axis y (line 16). Consecutive organs are rotated with respect to each other by the golden angle 137.5° measured around this axis. A sample pattern generated by Algorithm 5 is shown in Figure 13.

Figure 14: Inflorescences of *Kniphofia* sp. (red-hot poker plant) generated using Algorithm 5: models of two developmental stages (top) and the photographs used as a reference (bottom).

Figure 15: A *Pinus banksiana* (Jack pine) cone.

In realistic models, we replace spheres O by models of plant organs, as in [11]. For example, Figure 14 shows two developmental stages of the inflorescence of *Kniphofia* sp. (red-hot poker plant), in which florets have been modeled using generalized cylinders. In Figure 15 the algorithm has been additionally modified to allow for a curved cone axis. This modification is equivalent to the deformation of a straight cone, performed as a post-processing step.

8 Modeling multiple-compound structures

Algorithms 4 and 5, introduced in the previous sections, have been illustrated using examples of single-compound monopodial structures, each consisting of a sequence of organs placed along an axis or on a receptacle. The same algorithms can also be used, how-

Figure 16: A photograph and a model of a *Spiraea* sp. twig. The arrangement of shoots on the twig and the arrangement of leaves and flowers in each shoot follow the spiral phyllotactic pattern. The approximately vertical posture of all shoots reflects strong orthotropism, which has been simulated by biasing the turtle's heading vector in the vertical direction as described in [28, page 58].

ever, to generate structures in which the main axis supports entire substructures. For example, the *Spiraea* sp. twig shown in Figure 16 was constructed using Algorithm 4 twice: first to place the flower-bearing shoots along the main stem, then to place the leaves and the flowers within each shoot. In this case, all shoots have been assumed equal, except for the different shoot axis shapes caused by their *orthotropism* (tendency to grow vertically). In general, however, the supported structures may vary in a systematic manner, reflecting a morphogenetic gradient along the main stem.

To capture this gradient, we assume that, given two branches of the same order, the shorter branch is identical (up to the effects of tropisms and random variation) to the *top* portion of the longer branch. This concept of *branch mapping* is supported by both biological arguments and simulation results.

Biologically, it is related to the fact that apical meristems, the main engines of plant development, are located at the distal ends of

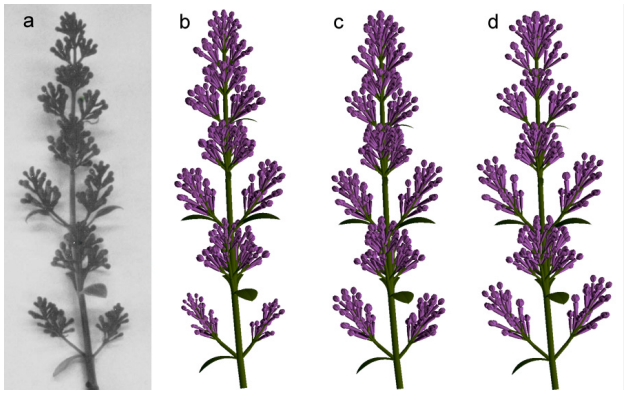


Figure 17: The effect of branch mapping. (a) An inflorescence of common lilac *Syringa vulgaris*. (b) Reconstruction of this inflorescence based on the measurements of all branches and flowers. (c) The same structure, all flowers assumed to be identical. (d) An approximate reconstruction based on branch mapping.

branches. Thus, if branch $\mathcal{B}$ develops over a shorter time or at a slower rate than an otherwise equivalent branch $\mathcal{A}$, branch $\mathcal{B}$ will resemble the *top* portion of $\mathcal{A}$.

A modeling example supporting the use of branch mapping is shown in Figure 17. An inflorescence of common lilac *Syringa vulgaris* (a) has been measured and reconstructed at three levels of accuracy: with all architectural information present (b), using the assumption that all flowers are identical (c), and using the assumption that shorter branches are identical to the top portions of the longer branches of the same order (d). Although reconstruction (d) is visually the least accurate, it still matches the real structure well.

Branch mapping makes it possible to define all branches of the same order using one set of functions. This concept is captured by the following algorithm.

Algorithm 6

```
1   Axiom: A(0,0)
2
3   A(o,s) : o < MAX && s < max_len[o]
4            { rel = s/max_len[o]; } ~>
5        #(int_width(o,rel)) F(int_len(o,rel))
6        [+(branch_ang(o,rel))
7             A(o + 1,max_len[o + 1] - branch_len(o,rel)) ]
8        [−(branch_ang(o,rel))
9             A(o + 1,max_len[o + 1] - branch_len(o,rel)) ]
10       /(90) A(o,s+int_len(o, rel))
11
12  A(o,s) : s ≥ max_len[o] ~> ~K
```

Algorithm 6 can be viewed as a recursive version of Algorithm 4, with the mechanism for creating curved axes removed for simplicity, and the internode length determined using point-sampled positional information as in Figure 6b for the same reason. Parameters o and s of the apices A represent the axis order and position along this axis, respectively. The array max_len$[o]$ specifies the length ℓ_{max} of the longest axis of each order $o <$ MAX. This value is used to represent positional information in relative terms, as a fraction rel of ℓ_{max} (line 4). This facilitates the specification of all functions, since they have fixed domain $[0, 1]$. Functions int_width(o, rel), int_len(o, rel), branch_ang(o, rel) and branch_len(o, rel) characterize morphogenetic gradients: the width and length of internodes, the branching angles at which the child branches are inserted, and the length of these child branches. All axes of the same order share the same set of functions. Within an axis of length ℓ, parameter s ranges from the initial value of $\ell_{max} - \ell$ (assigned to the newly created apices A in lines 7 and 9) to the maximum value of ℓ_{max} (condition in line 3). Thus, morphogenetic gradients along shorter axes are aligned with the distal portion of the longest axis of the same order, as required for branch mapping. Predefined flowers K are placed at the ends of the branches (line 12).

Examples of lilac inflorescences generated by Algorithm 6 are shown in Figure 18. Lilac inflorescences have decussate phyllotaxis. As was the case for Algorithm 4, a small modification of Algorithm 6 makes it possible to generate structures with spiral phyllotaxis. An example of the resulting structure — the inflorescence of an *Astilbe plant* — is shown in Figure 19.

Algorithm 6 can also be applied to approximate trees with clearly delineated branch axes (many young trees satisfy this criterion). If the axes of first-order branches are approximately straight and higher-order branches are relatively short, the outline of the tree crown is determined by the extent of the first-order branches and

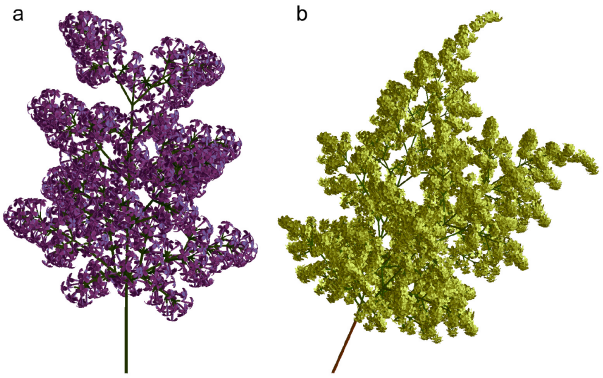


Figure 18: Inflorescences of two lilac species modeled using Algorithm 6: (a) *Syringa chinensis* CV. Rubra and (b) *Syringa reticulata*.

Figure 19: A photograph and a model of an *Astilbe x arendsii* CV. Diamant plant.

can easily be controlled by function branch_len$(0, rel)$ (Figure 20). In this sense, the use of positional information addresses the problem of progressing from silhouette to detail in the modeling process, exemplified by Figure 1c.

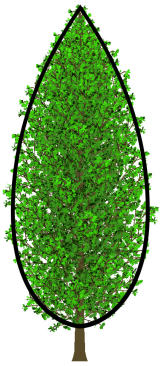


Figure 20: A generic tree model and its silhouette specifciation.

The problem of generating trees given their silhouettes occurs in several applications. One of them is the modeling and rendering of plant ecosystems. According to the approach proposed by Deussen *et al.* [10], the complexity of ecosystem modeling can be addressed by performing an individual-based simulation of the whole ecosystem, then replacing the coarse plant models used in this simulation with their detailed counterparts. The modeling method described in the present paper provides a means of creating plant models that match silhouettes determined at the ecosystem level (Figure 21).

9 Conclusions

We have explored the idea of plant modeling with functions that relate features of a plant to their positions along plant axes. Our experience confirms previous observations that this use of positional information is intuitive and well suited to the interactive modeling of plants. Visually important aspects of plant appearance — posture, the arrangement of components, and the overall silhouette — can easily be captured and controlled, while the procedural approach removes the tedium of specifying and placing each plant component individually. The algorithms are sufficiently fast to support interactive plant modeling on current personal computers.

We demonstrated the power of the modeling with positional information by recreating the form of several plants found in nature, presented on photographs, or depicted in drawings. The modeled structures range from individual leaves to compound herbaceous plants and trees.

The use of positional information is not limited to interactive modeling applications. We showed this by incorporating detailed tree models into a plant ecosystem model that only provided coarse characteristics of tree silhouettes. A related potential application is the automatic generation of plant models that match silhouettes of real trees, given their photographs [32].

At the technical level, our paper contributes: (a) a conceptual distinction between L-systems and Chomsky grammars as formal bases of developmental and structural plant models; (b) a generalized method for framing plant axes, free of the artifacts of the Frenet frame; (c) a robust method for spacing organs along plant axes; (d) an analytic method for generating phyllotactic patterns on arbitrary surfaces of revolution, based on Ridley's model; (e) the notion of branch mapping and its application to the modeling of compound plant structures; and (f) an example of the modeling system that integrates all of these concepts.

One open research problem is the use of constraints. In Algorithm 5 we introduced a relation between organ size and available space to constrain organ position in phyllotactic patterns. Many other relations have also been identified by biologists and can be applied to plant modeling [17]. By incorporating them into the algorithms we may further facilitate the modeling process. Specifically, constraints may reduce the number of parameters and functions that must be specified explicitly, while enforcing biological plausibility of the resulting structures.

Another interesting problem falls in the domain of interactive modeling techniques. In the present implementation, the user manipulates function plots, curves, and surfaces that are displayed separately from the model. A direct manipulation interface, in which the user would interact with the modeled structure itself, may lead to an even more intuitive modeling process.

A Appendices

A.1 Fundamental theorem of differential turtle geometry

The method for modeling curved limbs presented in Section 4 is based on the following extension of the fundamental theorem of differential geometry for three-dimensional curves [34, page 61] to the turtle reference frame.

Theorem. Let $\vec{H}(s)\vec{L}(s)\vec{U}(s)$ denote a moving reference frame defined on an interval $[0, \ell]$. Furthermore, let $\vec{H}(0)\vec{L}(0)\vec{U}(0)$ be the

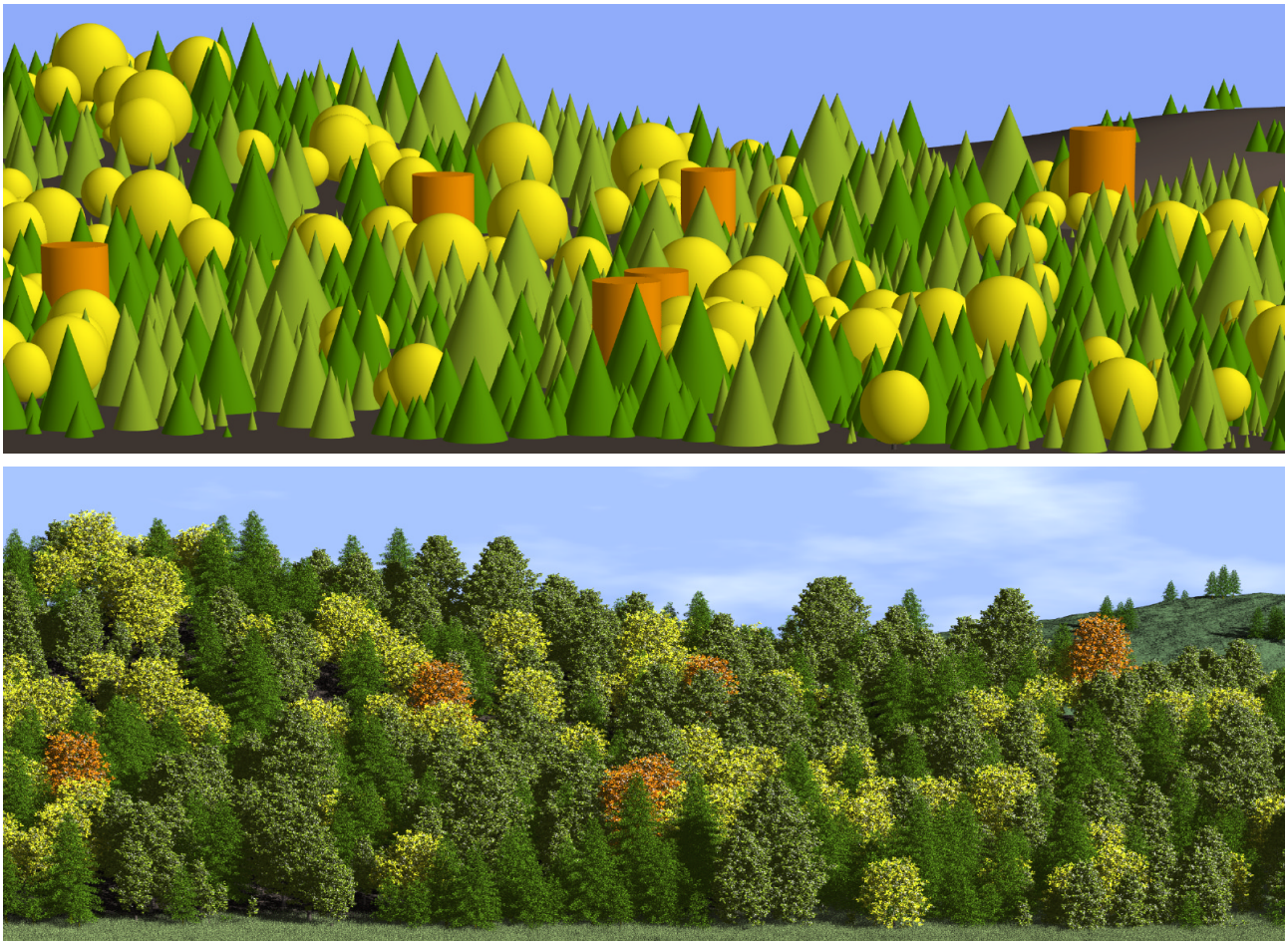


Figure 21: Visualization of an ecosystem simulation. Top: direct visualization. Bottom: realistic visualization. Tree silhouettes match the shapes coarsely defined at the ecosystem level.

initial orientation of this frame, and differentiable functions $\omega_H(s)$, $\omega_L(s)$ and $\omega_U(s)$ be its rates of rotation around the axes $\vec{H}(s)$, $\vec{L}(s)$ and $\vec{U}(s)$. The orientation of this frame is then uniquely defined for all $s \in [0, \ell]$. Moreover, given the initial frame position $\vec{P}(0)$, there is a unique differentiable curve $\vec{P}(s)$ for which s is the natural (arc-length) parameter, such that $\vec{H}(s)$ is tangent to $\vec{P}(s)$ for all $s \in [0, \ell]$.

Proof. Following [12], an infinitesimal rotation vector $d\vec{\Omega}$ acting on an arbitrary vector $\vec{A}$ changes it by $d\vec{A} = d\vec{\Omega} \times \vec{A}$. Thus, changes of the $\vec{H}\vec{L}\vec{U}$ reference frame due to the rotation rate vector $\vec{\omega} = \omega_H\vec{H} + \omega_L\vec{L} + \omega_U\vec{U}$ satisfy the system of equations:

$$\frac{d\vec{H}}{ds} = \vec{\omega} \times \vec{H}, \quad \frac{d\vec{L}}{ds} = \vec{\omega} \times \vec{L}, \quad \frac{d\vec{U}}{ds} = \vec{\omega} \times \vec{U}. \qquad (15)$$

Given the initial frame orientation $\vec{H}(0)\vec{L}(0)\vec{U}(0)$, vectors $\vec{H}(s)$, $\vec{L}(s)$ and $\vec{U}(s)$ are thus the unique solution to the initial value problem for the system of differential equations (15) in the interval $[0, \ell]$. Moreover, curve $\vec{P}(s)$ is given by the integral:

$$\vec{P}(s) = \vec{P}(0) + \int_0^s \vec{H}(s)ds. \quad \square \qquad (16)$$

A.2 Stability of node distribution

The fact that the distribution of nodes defined by integer values of Equation 11 does not depend critically on the choice of the initial node can be formally stated as follows.

Theorem. Consider a function λ such that $\lambda(s) > 0$ for all $s > 0$, and let $s_o, s'_0 > 0$ be two numbers. Using function N specified by Equation 11, define sequences $\{s_i\}$ and $\{s'_i\}$ such that $s_{i+1} = N(s_0, s_i) + 1$ and $s'_{i+1} = N(s'_0, s'_i) + 1$ for all $i = 0, 1, 2, \ldots$. If $s_0 < s'_0 < s_1$ then $s_i < s'_i < s_{i+1}$ for all $i = 0, 1, 2, \ldots$.

Proof by induction on i. The assumption $\lambda(s) > 0$ implies that $F(s) \equiv N(s_0, s)$ is an increasing function of the argument s. Thus, $s_i < s'_i < s_{i+1}$ implies $F(s_i) < F(s'_i) < F(s_{i+1})$, and therefore $F(s_i)+1 < F(s'_i)+1 < F(s_{i+1})+1$. By substituting $F(s_i)+1 = F(s_{i+1})$, $F(s'_i) + 1 = F(s'_{i+1})$, and $F(s_{i+1}) + 1 = F(s_{i+2})$, we obtain $F(s_{i+1}) < F(s'_{i+1}) < F(s_{i+2})$, hence $s_{i+1} < s'_{i+1} < s_{i+2}$. $\square$

A.3 Distribution of nodes defined by a linear function λ.

Theorem. Consider the sequence of nodes s_i defined by integer values of Equation 11, and let $\lambda(s) = as + b$. The length of consecutive internodes $l_i = s_{i+1} - s_i$ satisfies the equation $l_{i+1} = e^a l_i$ for $i = 0, 1, 2, \ldots$.

Proof. From Equation 11 we obtain:

$$1 = N(s_0, s_{i+1}) - N(s_0, s_i) \quad (17)$$

$$= \int_{s_i}^{s_{i+1}} \frac{ds}{as+b} = \frac{1}{a} \ln \frac{as_{i+1}+b}{as_i+b} \; . \quad (18)$$

Thus, $as_{i+1} + b = e^a(as_i + b)$ and, similarly, $as_{i+2} + b = e^a(as_{i+1}+b)$. By subtracting these equations sidewise and dividing by a we obtain $s_{i+2} - s_{i+1} = e^a(s_{i+1} - s_i)$, or $l_{i+1} = e^a l_i$. □.

Acknowledgments

We would like to thank: Lynn Mercer for contributing Figures 1b and c, Josh Barron for Figure 8a, Laura Marik for Figure 15, Enrico Coen for joint work on the snapdragon model (Figure 8b), Campbell Davidson for joint work on the lilac models (Figure 18), Christophe Godin for joint work on the decomposition rules, Bernd Lintermann and Oliver Deussen for a detailed demo of `xfrog`, and the referees for their insightful comments. The support of the Natural Sciences and Engineering Research Council of Canada, the International Council for Canadian Studies, the Alberta MACI project and the University of Calgary is gratefully acknowledged.

References

[1] A. H. Barr. Global and Local Deformations of Solid Primitives. Proceedings of SIGGRAPH 84, in *Computer Graphics*, 18, 3, July 1984, pages 21–30.

[2] D. Barthélémy, Y. Caraglio, and E. Costes. Architecture, Gradients Morphogénétiques et Age Physiologique ches les Végétaux. In J. Bouchon, Ph. De Reffye, and D. Barthélémy, editors, *Modélisation et Simulation de l'Architecture des Végétaux*, pages 89–136. INRA Editions, Paris, 1997.

[3] R. L. Bishop. There Is More Than One Way to Frame a Curve. *Amer. Math. Monthly*, 82(3):246–251, March 1975.

[4] H. Bjornson. *Weeds*. Chronicle Books, San Francisco, 2000.

[5] J. Bloomenthal. Modeling the Mighty Maple. Proceedings of SIGGRAPH 85, in *Computer Graphics*, 19, 3, July 1985, pages 305–311.

[6] J. Bloomenthal. Calculation of Reference Frames Along a Space Curve. In A. Glassner, editor, *Graphics Gems*, pages 567–571. Academic Press, Boston, 1990.

[7] T. E. Burk, N. D. Nelson, and J. G. Isebrands. Crown Architecture of Short-rotation, Intensively Cultured *Populus*. III. A Model of First-order Branch Architecture. *Canadian Journal of Forestry Research*, 13:1107–1116, 1983.

[8] N. Chomsky. Three Models for the Description of Language. *IRE Trans. on Information Theory*, 2(3):113–124, 1956.

[9] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, and C. Puech. Plant Models Faithful to Botanical Structure and Development. Proceedings of SIGGRAPH 88, in *Computer Graphics* 22, 4, August 1988, pages 151–158.

[10] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz. Realistic Modeling and Rendering of Plant Ecosystems. Proceedings of SIGGRAPH 98, Annual Conference Series, July, 1998, pages 275–286.

[11] D. R. Fowler, P. Prusinkiewicz, and J. Battjes. A Collision-based Model of Spiral Phyllotaxis. Proceedings of SIGGRAPH 92, in *Computer Graphics*, 26, 2, July 1992, pages 361–368.

[12] H. Goldstein. *Classical Mechanics*. Addison-Wesley, Reading, 1980.

[13] J. S. Hanan. *Parametric L-systems and Their Application to the Modelling and Visualization of Plants*. PhD thesis, University of Regina, June 1992.

[14] A. J. Hanson. Quaternion Gauss Maps and Optimal Framings of Curves and Surfaces. Technical Report 518, Computer Science Department, Indiana University, Bloomington, IN, 1998.

[15] C. Jirasek, P. Prusinkiewicz, and B. Moulia. Integrating Biomechanics into Developmental Plant Models Expressed Using L-systems. In H.-Ch. Spatz and T. Speck, editors, *Plant Biomechanics 2000*, pages 615–624. Georg Thieme Verlag, Stuttgart, 2000.

[16] J. J. Koenderink. *Solid Shape*. MIT Press, Cambridge, 1993.

[17] P. Kruszewski and S. Whitesides. A General Random Combinatorial Model of Botanical Trees. *Journal of Theoretical Biology*, 191(2):221–236, 1998.

[18] B. Lintermann and O. Deussen. XFROG 2.0. www.greenworks.de, December 1998.

[19] B. Lintermann and O. Deussen. Interactive Modeling of Plants. *IEEE Computer Graphics and Applications*, 19(1):56–65, 1999.

[20] R. Měch. *Modeling and Simulation of the Interactions of Plants with the Environment using L-systems and their Extensions*. PhD thesis, University of Calgary, October 1997.

[21] R. Měch and P. Prusinkiewicz. Visual Models of Plants Interacting with their Environment. Proceedings of SIGGRAPH 96, Annual Conference Series, August, 1996, pages 397–410.

[22] K. J. Niklas. *Plant Allometry: The Scaling of Form and Process*. The University of Chicago Press, Chicago, 1994.

[23] P. Oppenheimer. Real Time Design and Animation of Fractal Plants and Trees. Proceedings of SIGGRAPH 86, in *Computer Graphics*, 20, 4, August 1986, pages 151–158.

[24] W. F. Powell. *Drawing Trees*. Walter Foster Publishing, Inc., Laguna Hills, CA, 1998.

[25] P. Prusinkiewicz, J. Hanan, and R. Měch. An L-system-based Plant Modeling Language. Lecture Notes in Computer Science 1779, pages 395–410. Springer-Verlag, Berlin, 2000.

[26] P. Prusinkiewicz, M. James, and R. Měch. Synthetic Topiary. Proceedings of SIGGRAPH 94, Annual Conference Series, July, 1994, pages 351–358.

[27] P. Prusinkiewicz, R. Karwowski, R. Měch, and J. Hanan. L-studio/cpfg: A Software System for Modeling Plants, 2000. Lecture Notes in Computer Science 1779, pages 457–464. Springer-Verlag, Berlin, 2000.

[28] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[29] W. T. Reeves and R. Blau. Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. Proceedings of SIGGRAPH 85, in *Computer Graphics*, 19, 3, July 1985, pages 313–322.

[30] W. R. Remphrey and G. R. Powell. Crown Architecture of *Larix laricina* Saplings: Quantitative Analysis and Modelling of (nonsylleptic) Order 1 Branching in Relation to Development of the Main Stem. *Canadian Journal of Botany*, 62(9):1904–1915, 1984.

[31] J. N. Ridley. Ideal Phyllotaxis on General Surfaces of Revolution. *Mathematical Biosciences*, 79:1–24, 1986.

[32] T. Sakaguchi. Botanical Tree Structure Modeling Based on Real Image Set. SIGGRAPH 98 Conference Abstracts and Applications, 1998.

[33] J. M. Snyder and J. T. Kajiya. Generative Modeling: A Symbolic System for Geometric Modeling. Proceedings of SIGGRAPH 92, in *Computer Graphics*, 26, 2, July 1992, pages 369–378.

[34] I. Vaisman. *A First Course in Differential Geometry*. Marcel Dekker, New York, 1984.

[35] H. Vogel. A Better Way to Construct the Sunflower Head. *Mathematical Biosciences*, 44:179–189, 1979.

[36] J. Weber and J. Penn. Creation and Rendering of Realistic Trees. Proceedings of SIGGRAPH 95, Annual Conference Series, August, 1995, pages 119–128.

[37] K. West. *How to Draw Plants. The Techniques of Botanical Illustration*. Timber Press, Portland, OR, 1997.

[38] E. Wunderlich. *Botanical Illustration in Watercolor*. Watson–Guptill, New York, 1991.

Virtual Plants: Modelling and Real-time Rendering

Oliver Deussen

Faculty of Computer Science
Dresden University of Technology, Germany

deussen@inf.tu-dresden.de
www.inf.tu-dresden.de/cgm

Content

→ Modelling of plants using rule-based object generation

→ joint work with Bernd Lintermann

ZKM Center for Arts and Media Technology Karlsruhe, Germany

www.zkm.de

→ Real-time rendering of complex plant scenes

→ joint work with Carsten Coldiz (Dresden) as well as

→ Marc Stamminger and George Drettakis

Reves/INRIA, Sophia Antipolis, France

www-sop.inria.fr/reves/index.gb.html

Plant Modelling

SAN ANTONIO SIGGRAPH 2002

→ procedural plant modelling methods

- → algorithms that can be parameterised
- → more or less specific
- → often intuitive parameters (e.g. age, vigour)
- → example: AMAP/netFX

→ rule-based plant modelling methods

- → L-Systems, iterated function systems (plant images)
- → general modelling scheme
- → abstract representation for plants

xfrog - A New Modelling Paradigm

SAN ANTONIO SIGGRAPH 2002

→ combination of rule-based and procedural plant modelling

→ a plant is represented by a structure graph

Nodes: components that encapsulate data and algorithms

Component types:

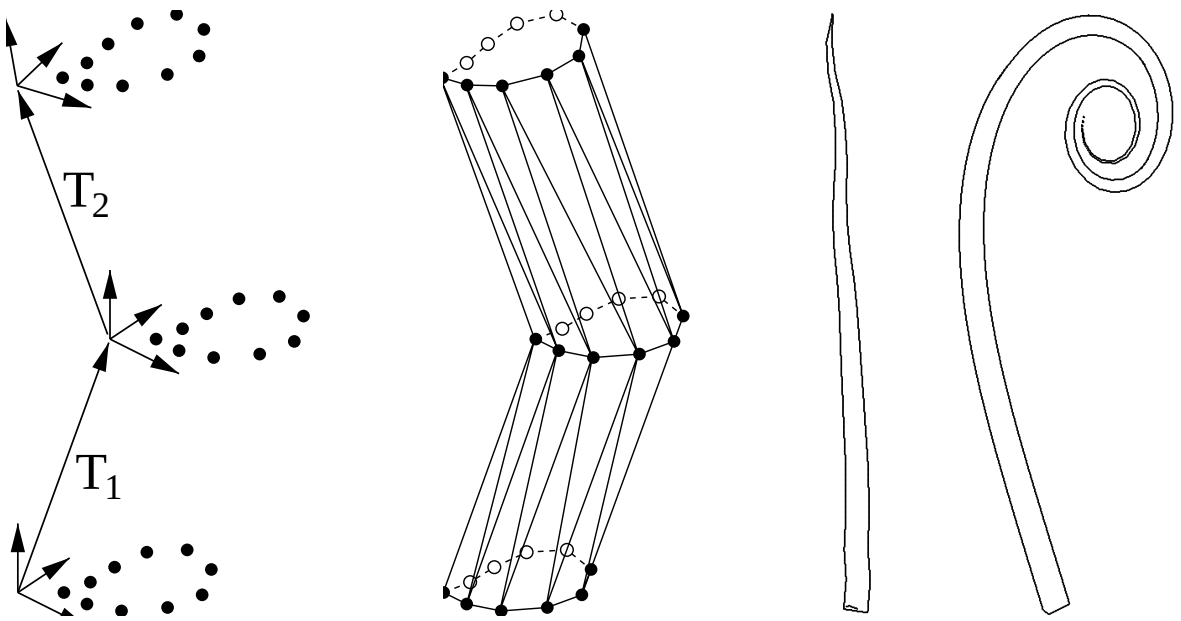
1. generation of geometry
2. multiplication of geometry
3. global modelling

Links: generation rules

Generation of Geometry

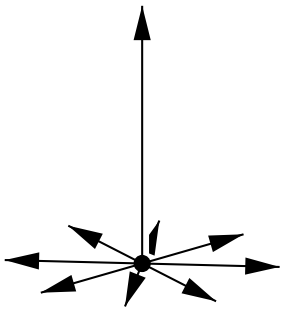
→ leaves: textured surfaces (triangles, quads or triangulated surfaces)

→ branches: generalized cylinders (inspired by Todd and Latham):

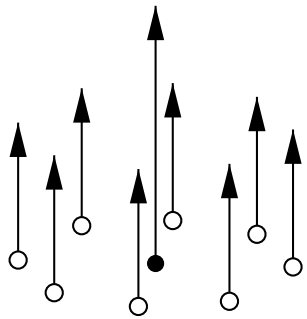


Multiplication of Geometry

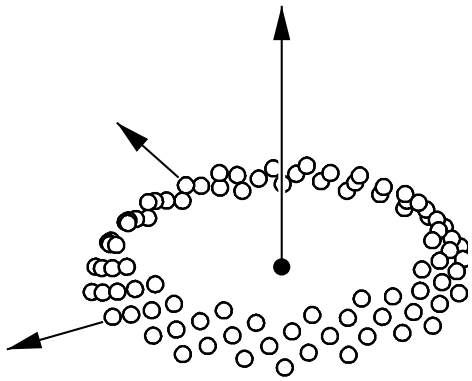
→ three types of multiplication:



Hydra component



Wreath component



Phiball component

→ problem: plant structure is represented by graph structure and by multiplication components

⇒ geometry generation in two stages

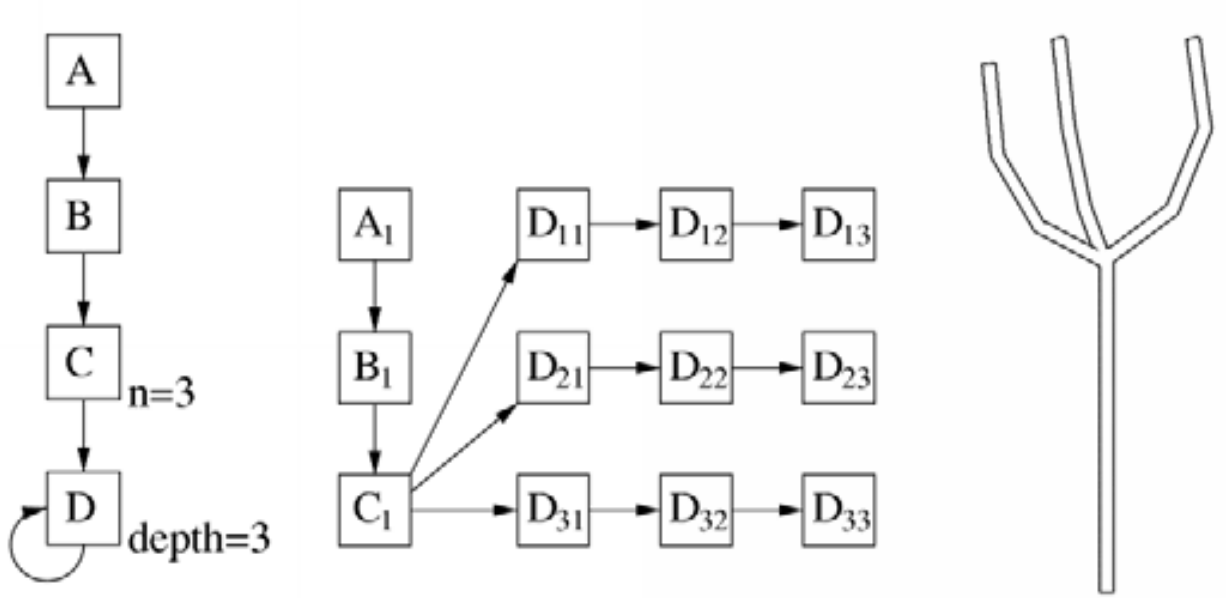
Multiplication of Geometry (cont.)

A root component

B component that generates a cylinder

C multiplication component, number of copies: 3

D component that generates a cylinder, recursion depth: 3



Plant Example I

Plant Example II

Plant Example III

Further Examples I

Further Examples II

Further Examples III

Video

Ecosystems I

Ecosystems II

Ecosystems III

Image: Bernd Lintermann

Ecosystems IV

Ecosystems V

Image: Bernd Lintermann

Real-time rendering of virtual landscapes

Problems With Complex Plant Scenes

- unnecessary amount of variety
  - → approximate instancing
    - → reduces the amount of different models needed in a scene
- over-specification
  - → simple primitives like points and lines
    - → reduces the amount of vertices to be processed
  - → visibility culling/occlusion culling
    - → unseen models are not transferred to the graphics processor
- too complex plant models
  - → level-of-detail algorithms
    - → each plant is shown in a complexity that is sufficient for its actual size

An Example

$\longrightarrow$ 16,7 million polygons

$\longrightarrow$ about one million pixels

$\Rightarrow$ bunches of polygons per pixel

LOD-Representation for Plants

SAN ANTONIO SIGGRAPH 2002

→ static level-of-detail:

- → model is represented by several different representation that are blended
- → visible popping artefacts can occur
- → all representations must be stored

→ dynamic level-of-detail:

- → model representation is computed individually for each size on the screen
- → popping artefacts are reduced
- → memory efficient
- → representation must be computed for each image

Representation by Points and Lines

SAN ANTONIO SIGGRAPH 2002

$\longrightarrow$ compact objects (leaves, petals) $\Rightarrow$ point representation

$\longrightarrow$ thin objects (branches, special leaves) $\Rightarrow$ line representation

$\longrightarrow$ point and line representations are shuffled

$\longrightarrow$ ... and stored in vertex arrays

$\Rightarrow$ variing amount of data can be used to represent a plant

$\Rightarrow$ blending between polygonal model and approximation is done successively on the basis of individual leaves

A Point Representation

Polygones 13,000 Points 6,500 3,250 1,625

Single Plant Example

SAN ANTONIO SIGGRAPH 2002

→ small triangles: vertex rate counts

→ standard way of approximation: replace polygons by points if projected area can be displayed more efficiently by point approximation

→ display can be tuned by point splat size and replacement threshold

standard display

Single Plant Example (cont.)

threshold reduced: points visible

threshold reduced, splat size increased

Line Representation

1,500 plants, original model size 12 million triangles, now displayed with 10-15 Hz

Importance Reduction

$\longrightarrow$ important parts of plants (blossoms, petals) are not so much reduced

$\Rightarrow$ quality can be increased without much more data

Complex Scene Examples

100 million polygones/8 Hz

70 million polygones/4 Hz

$\longrightarrow$ here: scene hierarchy used to enhance LOD representation

Video

Further Readings

→ B. Lintermann, O. Deussen:
Interactive Modelling of Plants
IEEE Computer Graphics and Applications, 19(1), pp. 56–65, 1999

→ O. Deussen, P. Hanrahan, M. Pharr, B. Lintermann, R. Měch, P. Prusinkiewicz:
Realistic Modelling and Rendering of Plant Ecosystems
SIGGRAPH 98 Conference Proceedings, pp. 275–286

→ O. Deussen, C. Colditz, M. Stamminger, G. Drettakis:
Interactive Visualization of Complex Plant Ecosystems
to be published (see: www.inf.tu-dresden.de/cgm)