Dream SDK Coobook

How to write extensions for Ray Dream Studio™

Version 2.0.1
Wednesday, July 02, 199 7

The software described in this manual is furnished under a licensing agreement printed on the inside front cover of the Ray Dream Designer User Manual. It may only be used or copied in accordance with the terms of this license. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical or otherwise, without the prior express written permission of MetaCreations, Corp.
The information in this user guide is provided for informational use only, is subject to change without notice, and should not be construed as a commitment by MetaCreations, Corp. MetaCreations, Corp. assumes no responsibility or liability for any errors or inaccuracies that may appear in this user guide.

Licensee acknowledges that the DreamSDK Development Toolkit may contain bugs, errors and other problems that could cause system failures. Consequently, the DreamSDK is provided to Licensee “AS IS,” and MetaCreations disclaims any warranty or liability obligations to Licensee of any kind. Accordingly, Licensee acknowledges that any research or development that it performs regarding the DreamSDK or any product associated with the DreamSDK is done entirely at Licensee’s own risk.

LICENSEE ACKNOWLEDGES THAT METACREATIONS MAKES NO EXPRESS, IMPLIED, OR STATUTORY WARRANTY OF ANY KIND FOR THE PRODUCT INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY WITH REGARD TO PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE.

METACREATIONS SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF REVENUE, LOSS OF PROFITS, BUSINESS INTERRUPTION, LOSS OF INFORMATION OR DATA AND THE LIKE) ARISING OUT OF THE USE OF OR INABILITY TO USE THE PROTOTYPE EVEN IF METACREATIONS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Contents

Chapter 1 - Introduction

This chapter introduces the main concepts of the Dream SDK. It also gives important tips on how to use this documentation and will help you in making the main technical choices so you can have a smooth and exciting experience developing your 3D Components.

1.1.An Open Architecture for 3D Illustration and Animation

What this 3D Open Architecture is

The architecture described in this document was created with the following design requirements in mind:

Enable an extension mechanism for 3D Illustration and Animation applications to share major components such as shaders, renderers, light sources, atmospheric effects, animation effects, 3D primitives and more between applications

- Provide a transparent integration in the client application user interface
- Provide a Plug-and-Play component installation
- Provide a cross-platform solution (Windows and Macintosh)
- Remove the burden out of writing 3D Extensions. When a choice had to be made, the burden was put on the host application to facilitate the development of 3D Extensions.

These technical choices were made because 3D is a complex field generally speaking: anyone having any specific 3D needs has to invest in enormous programming efforts to achieve his 3D project.

For example, a researcher needing to make a 3D data visualization and simulation project will have to implement a 3D renderer, a 3D Database system, 3D manipulation tools and user interface, etc. Another example is an architect who has some needs to render his 3D buildings, but cannot find any on-the-shelves 3D program that can support his exotic 3D architecture file format.

Often the problem to solve is very simple to code, but the 3D tools around it are vast and complex.

This is why an open architecture is especially welcomed in the 3D field. Applications such as Ray Dream Designer¨ or Corel DreamTM 3D built on this architecture can be extended very easily this way. For example, adding a Spherical camera takes only a few hundreds lines of code and is about a one day project.

The other obvious advantage of programming 3D Components is to provide solutions that run in different client applications, thus expanding the potential market of your 3D Component to many users without being forced to choose a specific 3D application. This advantage is usual in open architectures, and OLE programmers are familiar with it.

What this 3D Open Architecture is not

- A rendering library
- A CAD oriented API

What we explain in this document is a way for 3D Applications and 3D Components to communicate together. As such, it describes both the client and server sides, and how they

interact.

Supported Platforms and Compilers

The supported platforms are MacOS PPC, Windows 95, Windows NT and Win32s.

The supported compilers are Code Warrior 7 on the Macintosh, and Microsoft Visual C++ on Windows.

On the Macintosh, the use of Steve Jasik's The Debugger or Metrowerks MetroNub debugger is highly recommended for debugging.

1.2.The 3D Pipeline

One could describe the main purpose of any 3D Application in these very simplistic steps:

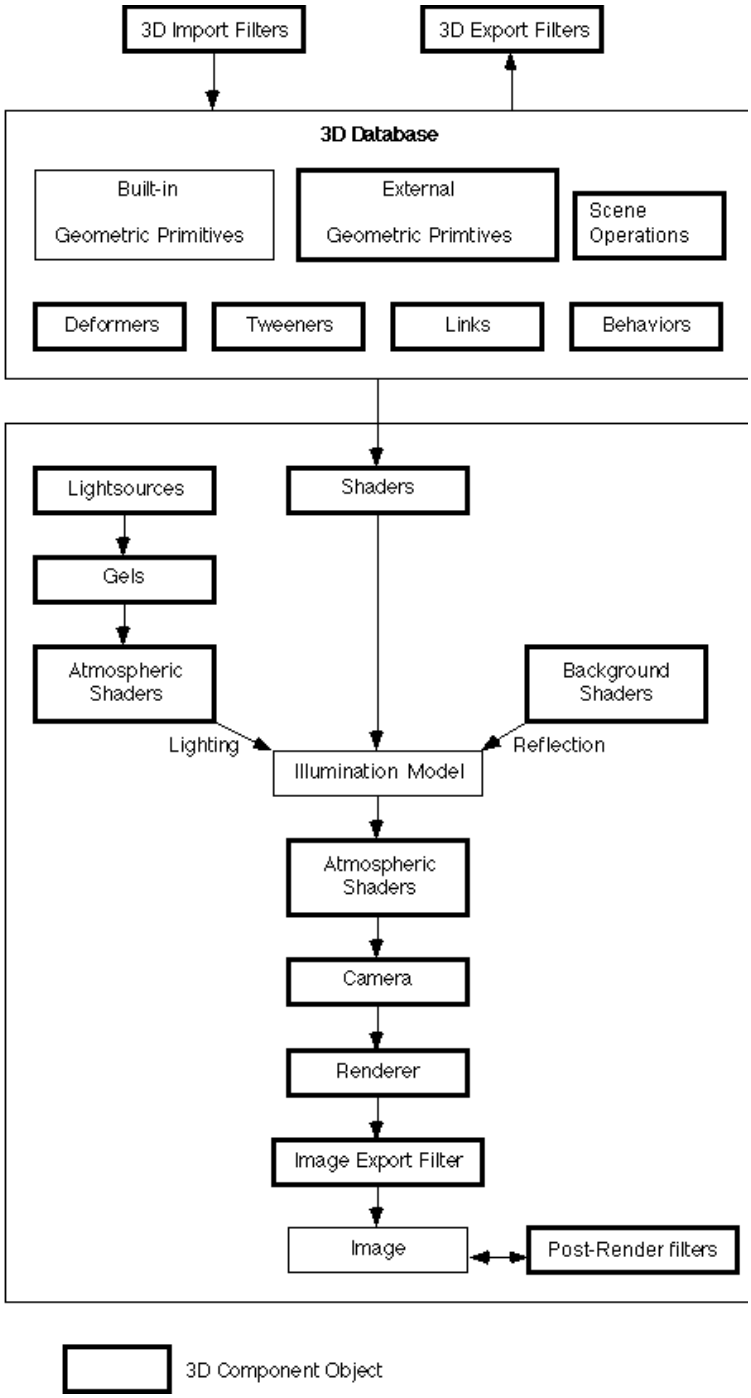
1. Create/Manipulate the 3D data
2. Render it in an image

However there is a long road to complete this process. This is why the 3D Open Architecture divides this pipeline in many little “bricks” that contribute one way or another to the main task. All these bricks are developed as external components, called the 3D Components or the 3D Extensions. Features can be added by simply adding new 3D Components. The other “bricks” will use and communicate with this new Components automatically.

Here is the list of all the things you can do:

1. 3D Database:
    - 3D Import filters
    - 3D Export filters
    - 3D Geometric Primitives
    - Deformers (deformations applied on a Geometric Primitive)
    - Tweeners (Animation interpolation objects)
    - Links (for mechanical dependencies between objects)
    - Behaviors (to add specific behaviors to elements in the 3D Database)
    - Scene Operations (commands that are added to the application as menu items)
2. Rendering:
    - Shaders
    - Light sources
    - Ambient Lights
    - Gels (put in front of light sources to change the light beam)
    - Atmospheric Shaders (fog...)
    - Reflected Background Shaders (for Environment mapping)
    - Backdrops (For 2D image composing)
    - Cameras (to define the type of 3D projection: conical, isometric, etc.)
    - Renderers (ray-tracing, Z-buffer, etc.)
    - Post-render filters (i.e. G-Buffers filters)

The way these 3D Components interact together in the 3D pipeline looks like this::



The 3D Pipeline

This figure gives a better view of the many possibilities offered by this architecture: it allows the extension of a complex process through the development of simple 3D Components.

3D Shell and 3D Extensions

The 3D Application is the client of the different services provided by the **3D Components**. Because the 3D Components add features, they are often called **3D Extensions**.

The 3D Application is in charge of organizing the data flow between the Components, and it also offers a set of services to the 3D Components to allow then to interact with its own internal data (like the 3D Database). The 3D Application is called the **3D Shell**, which is a better and more generic term.

1.3.Roadmap: How to use this documentation

You will find different types of chapters in this documentation, each type answering a specific need of the 3D Components developer:

- The **Cookbook chapters**: These chapters are step-by-step / how-to descriptions that will guide you through the main points of each subject. They often make references to the Toolkit examples and comment them widely. You will find those chapters most helpful to quickly understand the main concepts and tasks behind each 3D Extension, and will be especially appreciated by newcomers.
- A **theoretical chapter**, the “Database Overview” chapter. This chapter will give you the mathematical background and the definition of the different 3D terms used throughout the documentation. If you find anywhere an unknown 3D concept or term, chances are that it is explained in the “Database Overview” chapter.
- **Reference chapters**: “3D Components API Reference”, “3D Shell API Reference”, “Data Structures Reference” , etc. This is where you will find all the little picky details on each extension function, shell procedure or data structure.
- **Miscellaneous Appendices**: “Toolkit Libraries”, “Resources Reference”, “Managing the user interface of a 3D Component”, “File format”, etc. You use these chapters for some specific needs. They provide precious information on how to make the whole thing work.

Cookbook and Reference chapters are designed to be complementary: the Cookbook is used to get the general concepts and identify the steps to follow, and the Reference is used to get to the bottom of each call parameter. In many ways, one cannot live without the other.

Also the examples and the .h files in the Dream SDK are precious sources of information.

1.4.Using the COM dynamic linking

The communication between the 3D Shell and the 3D Components is done by using the Component Object Model (**COM**).

The choice was made to use the COM because COM is a very widely spread industry standard (COM is the low-level layer on which OLE is built), and is actively promoted by major industry players. So if you are already familiar with OLE, you will find it easier to be started with this documentation. COM offers a nice and clean C++ like interfacing, and it is highly recommended to read the excellent book “Inside OLE 2” by Kraig Brockschmidt from Microsoft Press to learn everything about it.

COM users should read the “Using the COM Dynamic Linking” appendix to learn all details about each technique and how to implement them.

1.5.How the whole thing works

Identifying Components at startup: Auto Plug-And-Play

When the 3D Shell is launched, it first identifies which 3D Components are available. To do this, it looks in its Extension directory and in all sub-directories of the Extension directory. On Windows, this will be the “ext” directory, on MacOS the “Ray Dream Extensions” folder.

On Windows:
All files with the “.rdx” extension are considered as Components.

On MacOS:
All files with the ‘RDEX’ file type are considered as Components.

Please note that there is nothing to register anywhere in the operating system to allow the identification of the Component. It is completely automatic. **Even if you develop a Windows extension, it is not necessary to register your Component in the Windows Registry Database.** This allows a true Plug-And-Play installation of Components. No problem for uninstalling, no full path names issues, no conflicts between different versions or languages, etc.

For each Component file found, the 3D Shell tries to find the corresponding resources. The resources are located in a “.dta” file next to the Component file on Windows, and in a “ data” file on MacOS. If no “.dta” file can be found on Windows, then it will try to look inside the “.rdx” file itself for any Windows resources. See the “Managing Your User Interface” appendix for more details on this. So far, let us just say that storing the Component resources in a “.dta” file is the preferred solution because it is a cross-platform solution and because the possibilities in terms of user interfaces are better.

Now that the resources are found, the 3D Shell looks for all ‘**COMP’** resources in the file (you can put several Components in the same file). The ‘COMP’ resource is the key to each Component. It identifies the name of the Component, the API version number it is based on, his own version number, and most important of all, its **Family ID** and its **Class ID**.

Family ID, Class ID...

Each Component belongs to a **Family**. The Component Family defines the kind of Component the 3D Shell is dealing with: a Shader, a Camera, an Export Filter, etc. All the items in bold in the “3D Pipeline” chart shown earlier are the available Families.

Each Family as a 4 letter code, its **Family ID**. For example, the Family ID for Shaders is ‘shdr’. Your Component must belong to an existing Family, otherwise the 3D Shell will not know what to do with it. Family IDs are described for each Component in the Cookbook and Reference chapters, and there is a general table in the “Managing the User Inter-

face" appendix.

In the COM terminology, a Family is strictly equivalent to an **Interface**.

Then Each Component has a **Class ID**. Its Class ID describes uniquely your Component. For example, in the Shader Family, there are different classes of shaders: the Checker Shader, the Marble Shader, the Wood Shader, etc. Each of this shader has a unique Class ID. Its Class ID is the Component's key to its data and instanciation process. This is how your Component will be able to store its private data in a Ray Dream Designer file and retrieve it later.

For this reason, if you intend to distribute your Component outside, even as a Freeware or Shareware, **it is vital that you register your Class ID from MetaCreations to ensure that it will be unique**. This Class ID will then be yours forever. This registration process is done by e-mail or fax, and it is free. See the "Managing the User Interface" appendix on how to register Class IDs.

In the COM terminology, a Class is strictly equivalent to a **Class** (did I hear anyone saying that all this was already looking familiar ?).

...And Instances

So far, this is all the 3D Shell does at startup. It simply identifies all available Components for later use. No Component code has been loaded or executed.

Then comes the time of instanciation. For example, a file is read, and it contains a 3D Object with a Marble Shader. The 3D Shell does not know anything about Marble Shaders. It just gets the Family ID and the Class ID. Looking up in its Component Class directory, it the **instanciate** a Marble shader by asking the Component implementing it to create one. The Shell just keeps on anonymous pointer on the Component **Instance**, and this will be good enough to communicate with it: reading/writing its data, handling the user interface, executing it, etc. More on this just after.

Note that each instance has its own data values. For example, if another object in the file has another Marble texture on it, then a new instance will be created in order to store the parameters of this second Marble. This is how you can have several Marbles with different vein spacing on different objects: they are different instances of the same Marble Class.

In the COM terminology, an Instance is strictly equivalent to an **Instance** (sorry COM-savvy readers, you got the point already...).

Communicating between the 3D Shell and the 3D Component

Once the Component has been instanciated, the 3D Shell will want to access the different Component services (i.e. routines). The 3D Shell is the orchestra director. The 3D Shell will call your Component routines when needed. "Do not call us, you will be called" could be the 3D Shell motto.

However, there are times when your Component needs additional services from the 3D Shell. For example, a 3D Import Filter will have to be able to create and manipulate data structures in the 3D Shell Database. Therefore, the 3D Shell offers a complete API to do all kind of things. When a routine of your Extension is called, you can then call back the Shell to complete your job.

How the Component User Interface and Public Data is managed by the 3D Shell

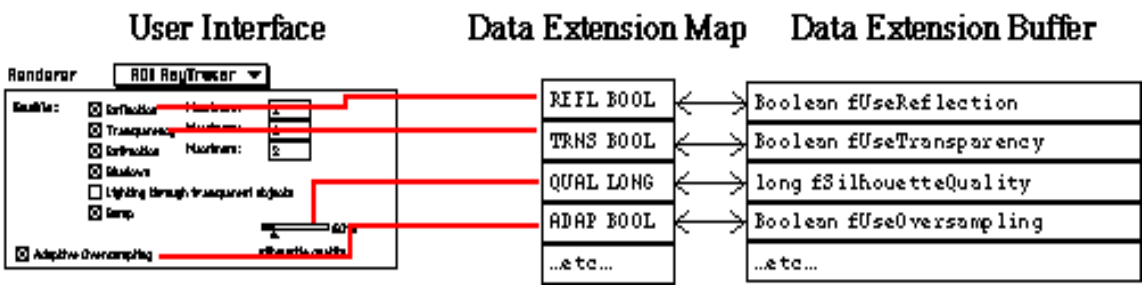
Often, the 3D Shell will want to show to the user the Component's User Interface so its parameters can be set before executing. The 3D Shell has unique way to do this so the user interface of your Component is seamlessly integrated in the Shell user interface, providing a consistent and integrated experience to the user.

The key to this UI integration is the '**pMAP'** resource. This resource is also the key to saving and reloading the Component's data.

The 'pMAP' resource describes a table called the **Data Extension Map**. During the initialization process, the Component told the Shell where its "public data" was stored: this is called the **Extension Data Buffer**, and it is located in the extension own RAM space. It is a record of the Component parameters (like the vein spacing values and the veins colors of the Marble Shader). The 3D Shell merely has an anonymous pointer on the Extension Data Buffer. The Data Extension Map will help the 3D Shell identifying the types and addresses of each data in the Extension Data Buffer, and build the relationship with the user interface.

Put simply, the Data Extension Map has two values for each entry: an ID to identify the user interface element (button, slider, etc.), and a type to identify the kind of data store in the Extension Data Buffer (Boolean, fixed, long, etc.).

This way, when the user changes something like the state of a check box, the Shell is able to compute the address of the corresponding item in the Data Buffer, and change it directly. Then it calls the Component and tells it that its data have been changed, so that the Component can react and update any internal pre-processed data.



Note that you do not have to write any code to handle your user interface. The Shell reads the resources describing your user interface from your Component file, and takes care of

everything: it reacts when the user clicks on buttons, sliders, etc. because it knows about all these user interface elements.

The other advantage of this mapping system of the 3D Component public data is that it provides the 3D Shell with a simple way to save the Component data in a file and retrieve it later without any knowledge of the Component's purpose in life. To store the Component data in the file, the Shell just writes the data ID and the value, and that's it. Later, after the Component is instanciated, the 3D Shell will write back this data in the Extension Data Buffer.

See the **I3DExDataExchanger** Interface and the "Managing the User Interface of a 3D Component" Appendix for all details. You will also find details on how to build the user interface itself (like using 'View' resources) and how to compile a .dta file.

Chapter 2 - Before writing a 3D Component

This chapter contains step-by-step / how-to descriptions that will guide you through the main points of each 3D Component. The Dream SDK examples are often referenced and commented. You will find this chapter most helpful to quickly understand the main concepts and tasks behind each 3D Extension, especially if you are a newcomer.

Here is what you need to do to be able to develop a 3D Component:

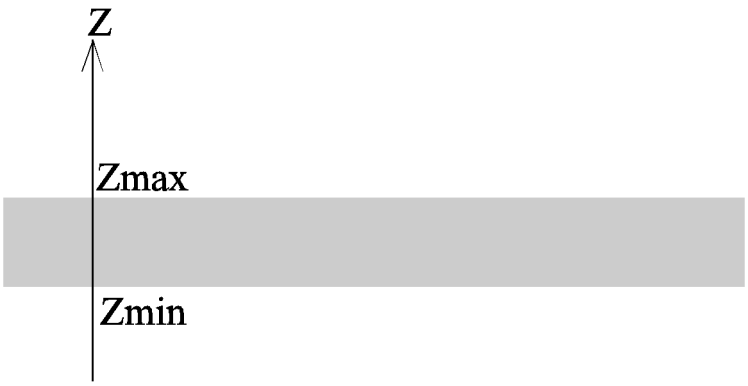
1. Read the Introduction chapter. Choose your platform(s).
2. If you have problems with some 3D terms, refer to the "Database Overview" chapter to be more familiar with the 3D concepts and the terminology.
3. You will probably need to read the appendix related to the user interface management and the one about the resources you will need to create.
4. Read the section in this chapter describing the 3D Component you are interested in, and make the appropriate technical/algorithmic decisions related to your own application.
5. Read the appendix describing the COM dynamic linking to learn more details on how to implement your 3D Component.
6. Read the Read-Me files in the Toolkit that correspond to your platform and compiler to learn details about building a project and a make file, compiling and linking, testing, etc.
7. Use the example in the toolkit as a framework to do your own 3D component. Each 3D Component has an example. Most of the time, these examples are ready to compile, saving you a lot of time and efforts as you start.
8. Read the "Toolkit Libraries" chapter if you need to learn about failure handling or math calculations.

Chapter 3 - Writing an AtmosphericShader

Family ID : ‘atmo’
COM Interface ID : IID_IDExAtmosphericShader
COM Interface file : I3DExAB.h

The atmospheric shader is used to create effects such as fog, clouds, etc.

We will develop a fog which are only between two altitudes.



The parameters will be the color of the fog, the two altitudes and the distance of visibility:

```
typedef struct AtmosData
{
  COLOR3D fColor;       // Color of the fog
  NUM3D   fZmin;        // Minimum altitude of the fog
  NUM3D   fZmax;        // Maximum altitude
  NUM3D   fVisibility;  // distance of Visibility
} AtmosData;
```

The visibility in the fog is the maximum distance from the camera to an object to see it. Beyond this distance everything is hidden by the fog.

So the formula to get the color of the light beam if the distance is less than the visibility is:

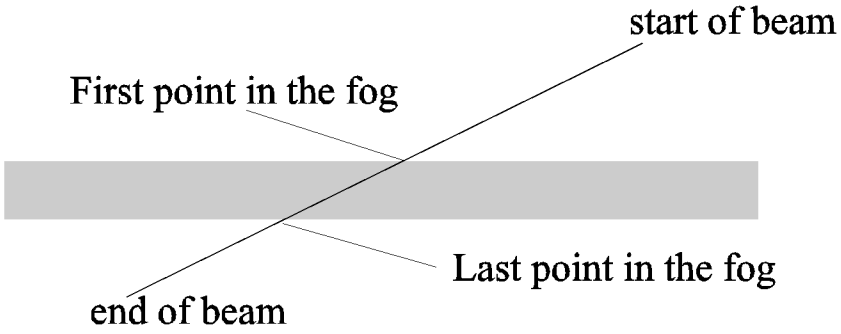
$$\text{Attenuation Factor} = 1 - \frac{\text{distance in the fog}}{\text{distance of Visibility}}$$

$$\textit{FilteredColor} = \textit{SourceColor} \times \text{Attenuation Factor} + \textit{FogColor} \times (1 - \text{Attenuation Factor})$$

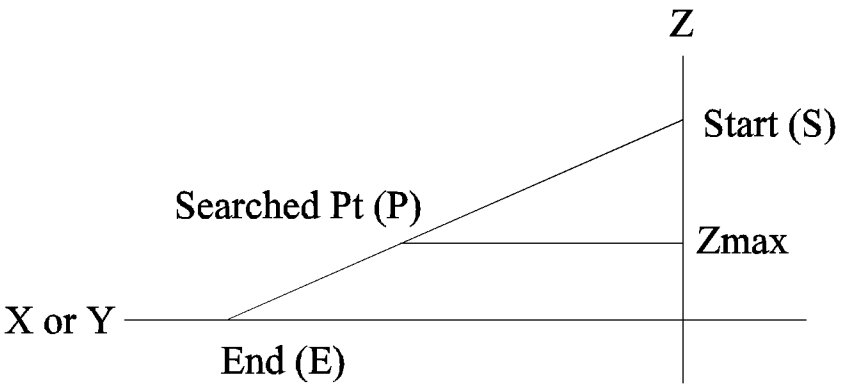
If the distance is higher than the visibility, the FilteredColor will be the FogColor.

1. Implement the **SegmentFilte r** function.

This function is called to create the atmospheric effect on a light beam between two points.
In our example, the fog is only between Zmin and Zmax, so for each light beam that does not go across the fog, the color is not affected.
In the other cases, you must find the starting point and the ending point in the fog of the light beam.



To find the coordinates of the First or Last point in the fog, you can use the following expressions:



$$X_P = X_S + (X_E - X_S) \times \frac{(Z\max - Z_S)}{(Z_E - Z_S)}$$

$$Y_P = Y_S + (Y_E - Y_S) \times \frac{(Z\max - Z_S)}{(Z_E - Z_S)}$$

Then you have the distance in the fog by calculating the norm of the vector LastPoint-FirstPoint.
So the C++ code will be:

```
HRESULT Atmos::SegmentFilter(THIS_ VECTOR3D* beg, VECTOR3D* end,
      COLOR3D* filterOut) {
  NUM3D    distanceInTheFog;
  VECTOR3D beamVector;
  VECTOR3D beaminfogbeg,beaminfogend;
  NUM3D    filtercoef,colorcoef;
```

```
if (((*beg)[2]>fData.fZmax)&&((*end)[2]>fData.fZmax)) { // the light
    beam is above the fog
  // Don't do anything
  }
else if (((*beg)[2]<fData.fZmin)&&((*end)[2]<fData.fZmin)) { // the
    light beam is under the fog
  // Don't do anything
  }
else { // The Light beam crosses the fog
  if ((*beg)[2]>fData.fZmax) {
    beaminfogbeg[2]=fData.fZmax;
    beaminfogbeg[0]=(*beg)[0]+((*end)[0]-(*beg)[0])/((*end)[2]-
    (*beg)[2])*(fData.fZmax-(*beg)[2]);
    beaminfogbeg[1]=(*beg)[1]+((*end)[1]-(*beg)[1])/((*end)[2]-
    (*beg)[2])*(fData.fZmax-(*beg)[2]);
    }
  else if ((*beg)[2]<fData.fZmin) {
    beaminfogbeg[2]=fData.fZmin;
    beaminfogbeg[0]=(*beg)[0]+((*end)[0]-(*beg)[0])/((*end)[2]-
    (*beg)[2])*(fData.fZmin-(*beg)[2]);
    beaminfogbeg[1]=(*beg)[1]+((*end)[1]-(*beg)[1])/((*end)[2]-
    (*beg)[2])*(fData.fZmin-(*beg)[2]);
    }
  else {
    beaminfogbeg=*beg; // the beginning point of the light beam is in
    the fog
    }
  if ((*end)[2]>fData.fZmax) {
    beaminfogend[2]=fData.fZmax;
    beaminfogbeg[0]=(*beg)[0]+((*end)[0]-(*beg)[0])/((*end)[2]-
    (*beg)[2])*(fData.fZmax-(*beg)[2]);
    beaminfogbeg[1]=(*beg)[1]+((*end)[1]-(*beg)[1])/((*end)[2]-
    (*beg)[2])*(fData.fZmax-(*beg)[2]);
    }
  else if ((*end)[2]<fData.fZmin) {
    beaminfogend[2]=fData.fZmin;
    beaminfogend[0]=(*beg)[0]+((*end)[0]-(*beg)[0])/((*end)[2]-
    (*beg)[2])*(fData.fZmin-(*beg)[2]);
    beaminfogend[1]=(*beg)[1]+((*end)[1]-(*beg)[1])/((*end)[2]-
    (*beg)[2])*(fData.fZmin-(*beg)[2]);
    }
  else {
    beaminfogend=*end; // the ending point of the light beam is in the
    fog
    }
  beamVector=beaminfogend-beaminfogbeg;
  distanceInTheFog=beamVector.GetNorm();
  filtercoef=kQuickFixOne-distanceInTheFog/fData.fVisibility;
  if (filtercoef<kQuickFixZero) {
    filtercoef=kQuickFixZero;
    }
  colorcoef=kQuickFixOne-filtercoef;
  filterOut->R=filterOut->R*filtercoef+fData.fColor.R*colorcoef;
  filterOut->G=filterOut->G*filtercoef+fData.fColor.G*colorcoef;
  filterOut->B=filterOut->B*filtercoef+fData.fColor.B*colorcoef;
  }
```

```
  return NOERROR;
  }
```

2. Implement the **DirectionFilte r** function.

This function is quite similar to the previous one, but you only have the beginning point of the light beam and its direction. The light beam is supposed to be infinite in this direction.

To determine the distance of the beam in the fog, you calculate the entry point and the exit point of the light beam, then you have the same formulae as **SegmentFilte r**.

```
HRESULT Atmos::DirectionFilter(THIS_ VECTOR3D* origin, VECTOR3D* direc-
      tion, COLOR3D* filterOut) {
  NUM3D    distanceInTheFog;
  VECTOR3D beamVector;
  VECTOR3D beaminfogbeg,beaminfogend;
  NUM3D    filtercoef,colorcoef;

  if (((*origin)[2]>fData.fZmax)&&((*direction)[2]>=kQuickFixZero)) {
    // Don't do anything
    }
  else if (((*origin)[2]<fData.fZmin)&&((*direction)[2]<=kQuickFixZero))
      {
    // Don't do anything
    }
  else {
    if (((*origin)[2]>fData.fZmax)||((*origin)[2]<fData.fZmin)) {
      beaminfogbeg[2]=fData.fZmax;
      beaminfogbeg[0]=(*origin)[0]+(*direction)[0]*(fData.fZmax-(*ori-
      gin)[2])/(*direction)[2];
      beaminfogbeg[1]=(*origin)[1]+(*direction)[1]*(fData.fZmax-(*ori-
      gin)[2])/(*direction)[2];
      beaminfogend[2]=fData.fZmin;
      beaminfogend[0]=(*origin)[0]+(*direction)[0]*(fData.fZmin-(*ori-
      gin)[2])/(*direction)[2];
      beaminfogend[1]=(*origin)[1]+(*direction)[1]*(fData.fZmin-(*ori-
      gin)[2])/(*direction)[2];
      }
    else if ((*direction)[2]>kQuickFixZero) {
      beaminfogbeg=*origin;
      beaminfogend[2]=fData.fZmax;
      beaminfogend[0]=(*origin)[0]+(*direction)[0]*(fData.fZmax-(*ori-
      gin)[2])/(*direction)[2];
      beaminfogend[1]=(*origin)[1]+(*direction)[1]*(fData.fZmax-(*ori-
      gin)[2])/(*direction)[2];
      }
    else if ((*direction)[2]<kQuickFixZero) {
      beaminfogbeg=*origin;
      beaminfogend[2]=fData.fZmin;
      beaminfogend[0]=(*origin)[0]+(*direction)[0]*(fData.fZmin-(*ori-
      gin)[2])/(*direction)[2];
      beaminfogend[1]=(*origin)[1]+(*direction)[1]*(fData.fZmin-(*ori-
      gin)[2])/(*direction)[2];
```

```
    }
  else { // (*direction)[2]=kQuickFixZero
    *filterOut=fData.fColor;
    return NOERROR;
    }
  beamVector=beaminfogend-beaminfogbeg;
  distanceInTheFog=beamVector.GetNorm();
  filtercoef=kQuickFixOne-distanceInTheFog/fData.fVisibility;
  if (filtercoef<kQuickFixZero) {
    filtercoef=kQuickFixZero;
    }
  colorcoef=kQuickFixOne-filtercoef;
  filterOut->R=filterOut->R*filtercoef+fData.fColor.R*colorcoef;
  filterOut->G=filterOut->G*filtercoef+fData.fColor.G*colorcoef;
  filterOut->B=filterOut->B*filtercoef+fData.fColor.B*colorcoef;
  }
return NOERROR;
}
```

Chapter 4 - Writing a Background

Family ID : ‘back’
COM Interface ID : IID_IDExBackground
COM Interface file : I3DExAB.h

The background interface is used to put a background in a scene behind every object. This background can be reflected on objects in the scene.

We will create a Sunset as an example. For this, you must have the West direction, the color of the sun, and of the sky colors at different points, and the size of the sun.

So the background data will be:

```
typedef BackData
{
  COLOR3D fSunColor;      // Color of the sun
  NUM3D   fSunDiameter;   // Sun Diameter in degrees
  NUM3D   fWestDirection; // Direction of West in degrees
  COLOR3D fZenithColor;   // Color in the Zenith
  COLOR3D fWestColor;     // Color in the West Direction (behind the sun)
  COLOR3D fEastColor;     // Color in the opposite direction
  COLOR3D fEarthColor;    // Color of the Earth
} BackData;
```

1. You have to implement only one function for the background effect:

```
HRESULT Sunset::GetBackgroundColor(THIS_ VECTOR3D* direction, COLOR3D*
      resultColor) {
  NUM3D in_sun;
  in_sun=(*direction)*fWestVector;
  if ((*direction)[2]<kQuickFixZero) { // You look the earth and not the
      sky
    *resultColor=fData.fEarthColor;
    }
  else if (in_sun>fSunLimit) { // You look directly at the sun
    *resultColor=fData.fSunColor;
    }
  else if (in_sun>kQuickFixZero) { // You look in the West Direction
    resultColor->Mode=0;
    resultColor->R   =fData.fWestColor.R*in_sun+fData.fZenith-
      Color.R*(kQuickFixOne-in_sun);
    resultColor->G   =fData.fWestColor.G*in_sun+fData.fZenith-
      Color.G*(kQuickFixOne-in_sun);
    resultColor->B   =fData.fWestColor.B*in_sun+fData.fZenith-
      Color.B*(kQuickFixOne-in_sun);
    }
  else { // You look in the East Direction
    resultColor->Mode=0;
    resultColor->R   =-fData.fEastColor.R*in_sun+fData.fZenith-
      Color.R*(kQuickFixOne+in_sun);
    resultColor->G   =-fData.fEastColor.G*in_sun+fData.fZenith-
```

```
    Color.G*(kQuickFixOne+in_sun);
  resultColor->B   =-fData.fEastColor.B*in_sun+fData.fZenith-
    Color.B*(kQuickFixOne+in_sun);
  }
return NOERROR;
}
```

Chapter 5 - Writing a Camera

Family ID : ‘came’
COM Interface ID : IID_IDExCamera
COM Interface file : I3DExCam.h

Cameras define the projection from the 3D world to the 2D screen on which the image is rendered. Like with a photographic camera, one can design different type of cameras with various lens, focal value, zoom effects, etc. Projection is not restricted to the conical projection (the natural projection in your eye), so isometric or fish-eye cameras can be designed.

In this part, we will develop two examples :

- a Spherical Camera, called *SphereCamera*, which can see all around.
- a Conical Camera, called *ConicCamera*, which is a simplified conical camera. You only have a lens with a focal of 50mm and a zoom.

1. Think about the parameters needed by your camera. For example, in a Spherical Camera, you need to know the aperture (360° correspond to all the 3D Space), and a Zoom factor to set the entire view in the production frame. Those parameters will be set by the user, so to help the shell modifying those parameters, create a structure like this one :

```
typedef struct CameraData {
   short fZoomCoef;
   short fAngle;
} CameraData;
```

2. You obtain the transformation data between the Global Coordinates System and the Screen Coordinates System (read the Database Overview chapter for more explanation about the different coordinates systems), with the function :

```
HRESULT I3DExCamera::SetTransform( TRANSFORM3D* transform);
```

You must copy the transformation data and not keep the pointer. The transformation data are organized in a matrix and a vector (Rotation and Translation).

The formulae to get the Screen Coordinates from the Global Coordinates is

$$(S) = [\mathbf{R}](c \cdot G) + \mathbf{T}$$

and because R is normalized the inverse transformation is

.

$$(G) = \frac{[\mathbf{R}]^t (S - \mathbf{T})}{c}$$

(S) is the position in the Screen Coordinates System,

(G) the same position in the Global Coordinates System,

T is the translation vector in Screen Coordinates,

R is the rotation matrix,

the factor c is here to respect the 3D Units and the Points Units. A 3D Unit is equal to 4 inches or 288 Points Unit.

You can find the transformation functions in C++ (*LocalToGlobal, GlobalToLocal, Local-ToGlobalVector, GlobalToLocalVector*) in the camera examples files.

To copy the transformation data in your object you only have to do this :

```
HRESULT SphereCamera::SetTransform(TRANSFORM3D* transform) {
fTransform=*transform; // copy the data in the field fTransform of your
      object
return NOERROR;
}
```

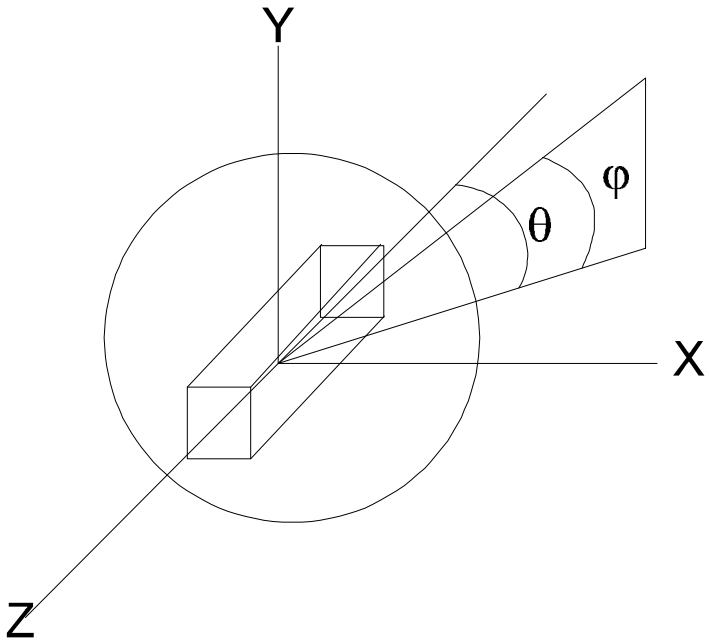
If you think you can make any preprocess calculations with the transformation data, do it in this function.

3. You have only two calls to implement for the Camera projection :

```
BOOLEAN I3DExCamera::Project3DTo2D(VECTOR3D* position, VECTOR2D* screen-
      Position, NUM3D* resultDistanceToScreen);
BOOLEAN I3DExCamera::CreateRay(VECTOR2D* screenPosition, VECTOR3D*
      resultOrigin, VECTOR3D* resultDirection);
```

The function Project3DTo2D gives the coordinates of the projection on the Screen of the 3D point, and the distance from the screen to the 3D point. If the 3D point is not in front of the camera, you will return FALSE.

In the SphereCamera the two screen coordinates will be the latitude and the longitude:



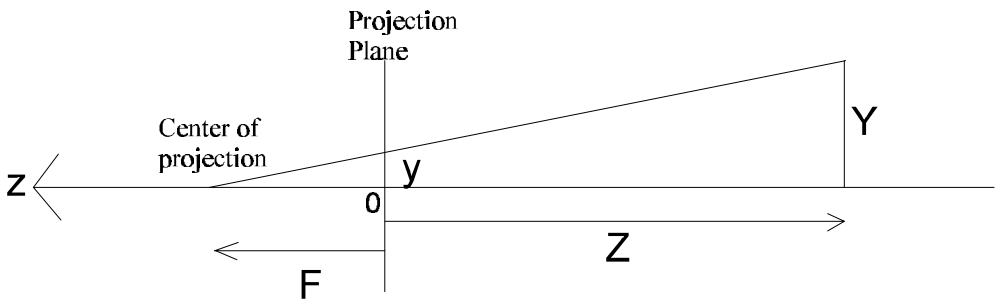
Spherical Projection convention

So for the Spherical projection you will have :

```
BOOLEAN SphereCamera::Project3DTo2D(THIS_ VECTOR3D* position, VECTOR2D*
      resultScreenPosition, NUM3D* resultDistanceToScreen) {
  VECTOR3D tempV;
  NUM3D    theta,phi,x2,z2,r;


  position->Normalize(tempV);

  // Get the Spherical coordinates : ro,theta,phi
  x2=tempV[0]*tempV[0];
  z2=tempV[2]*tempV[2];
  r=x2+z2;
  r.GetSquareRoot(r); // Must be calculate to get phi

  theta.DegreeSetFromSinCos(tempV[0],-tempV[2]);
  phi.DegreeSetFromSinCos(tempV[1],r);

  if (theta>kQuick180) {
    theta-=(kQuick180<<1); // theta : -180° to 180° (theta - 360)
    }
  if (phi>kQuick180) {
    phi-=(kQuick180<<1);   // phi   : -90°  to 90°
    }

  (*resultScreenPosition)[0]=theta*Coef;
  (*resultScreenPosition)[1]=phi*Coef;
  *resultDistanceToScreen=position->GetNorm();

  return TRUE; // a spherical camera can see everything in the scene
```

```
}
```

In the Conical Camera example, we use the standard definition of the conical projection :



So the formulae will be :

(where F is the focal length).

The screenPosition must be in Point Units so we use this conversion formulae:

$$L_{\text{in Pts}} = L_{\text{in mm}} \cdot \frac{288}{4} \cdot \frac{10}{254}$$

because 25.4mm is equal to 1 inch and 288 Points Units is equal to 4 inch.
So for the ConicCamera, you will have :

```
BOOLEAN ConicCamera::Project3DTo2D(THIS_ VECTOR3D* position, VECTOR2D*
      resultScreenPosition, NUM3D* resultDistanceToScreen) {
  NUM3D temp;

  *resultDistanceToScreen=-(*position)[2]; // Distance to the screen (-
      z) in Point Unit

  temp=kQuickFocal*kQuick288*kQuickmmTo3DUnit;  // Focal length in Point
      Unit
  if ((*position)[2]!=temp) {
    temp-=(*position)[2]; // Distance from the 3D Point to the Focal
      Point
    (*resultScreenPosition)[0]=(*position)[0]*kQuickFocal/temp*fQuick-
      Zoom;
    (*resultScreenPosition)[1]=(*position)[1]*kQuickFocal/temp*fQuick-
      Zoom;
    // Conical projection
    }
  else {
    (*resultScreenPosition)[0]=kQuickFixZero;
```

```
    (*resultScreenPosition)[1]=kQuickFixZero;
    // the Point is in the Focal Plane so it can't be projected on the
      screen
    }

  *resultScreenPosition*=fCoef; // Conversion in Point Unit

  if (*resultDistanceToScreen<=kQuickFixZero)
    return FALSE; // The point is behind the Camera
  else
    return TRUE;  // The point is in front of the Camera, it's visible.
  }
```

Now, we have to implement the function that allows Ray-tracing. It creates a ray from a screen point. As you see in the spherical projection figure, the ray always starts on the same point in the Spherical Camera example, that's why the *resultOrigin* will always be (0,0,0) in the Screen Coordinates System. In the SphericCamera, the *screenPosition* are the two angle theta and phi, so the resultDirection is very easy in the Screen Coordinates System :. And the CreateRay function will be :

$$\begin{cases} x = \sin(\theta) \cdot \cos(\varphi) \\ y = \sin(y = \dfrac{Y \cdot F}{F - Z} \cdot Zoom \\ z = -\cos(\theta) \cdot \cos(\varphi) \end{cases}$$

```
BOOLEAN SphereCamera::CreateRay(THIS_ VECTOR2D* screenPosition,
      VECTOR3D* resultOrigin, VECTOR3D* resultDirection) {
  VECTOR3D SpherePos;
  NUM3D    theta,phi,sintheta,costheta,sinphi,cosphi;

  SpherePos[0]=SpherePos[1]=SpherePos[2]=kQuickFixZero;
  LocalToGlobal(&fTransform,&SpherePos,resultOrigin);
  // Origin of the Ray is the center of the Sphere.

  theta=(*screenPosition)[0]/Coef;
  phi  =(*screenPosition)[1]/Coef;


  theta.DegreeGetSinCos(sintheta,costheta);
  phi.DegreeGetSinCos(sinphi,cosphi);

  SpherePos[0]=sintheta*cosphi;
  SpherePos[1]=sinphi;
  SpherePos[2]=-costheta*cosphi;
  // 3D Coordinates (x,y,z) from Spherical coordinates  (r=1,theta,phi)

  LocalToGlobalVector(&fTransform,&SpherePos,resultDirection);
  resultDirection->Normalize(*resultDirection);
  // Direction vector must be in Global Coordinates System and Normalized

  // Angle selection :
  if ((theta>QuickAngle)||(theta<-QuickAngle)) {
    return FALSE; // Not in front of the camera
    }
  if ((phi>(QuickAngle>>1))||(phi<-(QuickAngle>>1))) {
```

```
    return FALSE; // Not in front of the camera
    }

  return TRUE;
  }
```

For the Conical Camera, it is the opposite transformation of the Project3DTo2D function:

```
BOOLEAN ConicCamera::CreateRay(THIS_ VECTOR2D* screenPosition, VECTOR3D*
      resultOrigin, VECTOR3D* resultDirection) {
  VECTOR3D tempV;

  // Origin of the Ray is on the Screen in the Global Coordinates System
  tempV[0]=(*screenPosition)[0]/fCoef/fQuickZoom;
  tempV[1]=(*screenPosition)[1]/fCoef/fQuickZoom;
  tempV[2]=kQuickFixZero;
  LocalToGlobal(&fTransform,&tempV,resultOrigin);

  // Create the vector from the Center of projection to the screenPoint
  tempV[2]=-kQuickFocal;

  LocalToGlobalVector(&fTransform,&tempV,resultDirection);

  resultDirection->Normalize(); // the Direction vector must be normal-
      ized

  return TRUE;
  }
```

4. You can also add the function **Clip3D** to increase the speed of the Z-Buffer. This function have to cut a facet when it goes out of the screen and return a polygon. Because this kind of functions is very complicated, you will find a library that do it for a conic or isometric camera. You do not have to do it for a spherical camera because every point of the space is visible. So for the conic camera you will have :

```
ULONG ConicCamera::Clip3D(FACET3D* localFacet, VERTEX3D* localVertices,
            FACET3D* cameraFacet, VERTEX3D* cameraVertices,
            NUM3D* clipBox) {
   return ConicClip3D( kQuickFocal, localFacet, localVertices, cam-
      eraFacet, cameraVertices, clipBox);
   }
```

Chapter 6 - Writing a Deformer

Family ID : ‘defo’
COM Interface ID : IID_IDExDeformer
COM Interface file : I3DExDfr.h

A deformer gives you the possibility to deform objects with some geometric rules. For example, we will implement a deformer which can make a pyramid with a cube or an egg with a sphere.

1. We have to know the parameters used by our deformer. We can change the axis and we have to enter two different scaling values for the beginning and the end of the two perpendicular axis. So we will have a data structure like this one:

```
typedef struct DeformerData {
  long     fAxis;       // ID of the axis (AXEX AXEY or AXEZ)
  NUM3D    fUBegScale;  // First scaling of the U axis (if Z is selected,
      it is the X axis)
  NUM3D    fUEndScale;  // Last scaling of the U axis (if Z is selected,
      it is the X axis)
  NUM3D    fVBegScale;  // First scaling of the V axis (if Z is selected,
      it is the Y axis)
  NUM3D    fVEndScale;  // Last scaling of the U axis (if Z is selected,
      it is the Y axis)
  BOX3D    fBoundingBox;  // Bounding Box
  } DeformerData;
```

2. If, to deform the objects, you only have to change the coordinates of each point, you can implement only the **DeformPoints** function. But if your deformer can change the structure of the objects (add facets or patches for examples) you have to implement the **DeformFacets** and **DeformPatches** functions. In our case of the pyramidal deformer we only need the function **DeformPoints**, so we will return *ResultFromScode(E_NOTIMPL)* with the two other functions.
The scaling will be linear between the beginning and the end of the bounding box along the right axis. The code will be like this:

```
HRESULT Deformer::DeformPoint(THIS_ VECTOR3D* point,VECTOR3D* result)
{ short u,v,w;
  NUM3D wrelative;
  if(fData.fAxis==kAxisX)
  { u=1; // Y Axis
    v=2; // Z Axis
    w=0; // X Axis
  }
  else if (fData.fAxis==kAxisY)
  { u=2; // Z Axis
    v=0; // X Axis
    w=1; // Y Axis
  }
  else
  { u=0; // X Axis
```

```
    v=1; // Y Axis
    w=2; // Z Axis
  }
  if ((fData.fBoundingBox.fMax[w]-fData.fBoundingBox.fMin[w])==kQuick-
      FixZero)
  { (*result)=(*point);
    return NOERROR;
  }
  wrelative=((*point)[w] - fData.fBoundingBox.fMin[w])/(fData.fBounding-
      Box.fMax[w]-fData.fBoundingBox.fMin[w]);
  if ((fData.fUEndScale+fData.fUBegScale)==kQuickFixZero)
  { (*result)[u]=kQuickFixZero;
  }
  else
  {
    (*result)[u] = (*point)[u] * (fData.fUBegScale + wrelative *
      (fData.fUEndScale-fData.fUBegScale));
  }
  if ((fData.fVEndScale+fData.fVBegScale)==kQuickFixZero)
  { (*result)[v]=kQuickFixZero;
  }
  else
  {
    (*result)[v] = (*point)[v] * (fData.fVBegScale + wrelative *
      (fData.fVEndScale-fData.fVBegScale));
  }
  (*result)[w] = (*point)[w];
  return NOERROR;
}
```

3. As you see in this code, you have to know the bounding box before any deformation of a point. There is a function in the Deformer interface that allows this: **SetBBox**. This function is called before calling any deformation functions. So if you want to make any preprocessing calculations, you can do them in the functions **SetBBox** or **ExtensionDataChange d**. In our case, we do not make any preprocessing, so we have a very simple **SetBBox** function:

```
HRESULT Deformer::SetBBox(THIS_ BOX3D *bbox)
{
  fData.fBoundingBox=*bbox; // copy the bbox in a field of the object
      Deformer.
  return NOERROR;
}
```

Note: If you want to implement the functions **DeformFacets** or **DeformPatches**, you will have to use the **IShIterator** to get the different facets or patches one by one. Then you will have to return the facets or the patches with the callback function like in the Geometric Primitive.

Chapter 7 - Writing a Geometric Primitive

Family ID : ‘prim’
COM Interface ID : IID_IDExGeometricPrimitive
COM Interface file : I3DExPrim.h

Geometric primitive describe geometric objects (like a sphere, a cube, etc.).

This component needs to return a list a bicubic patches or facets. It can also give some information on its UV Space.

In this part, we will describe three different implementations of a Geometric Primitive :

- a 3D Star, based on facets with a special UV-Space

- a Teapot, based on bicubic patches

- a Sphere, based on Ray tracing definition

When you drop a geometric primitive in the scene, the shell can create a dialog to change some parameters of your object. We will use this feature for the 3D Star, to change the number of branches. So for this example, we will create a view and return its ID with the function **GetResID** (see I3DExDataExchanger interface for more details). Otherwise you return -1 with the function **GetResID**.

1. For the 3D Star, we have to create a data structure:

```
typedef struct StarData
```

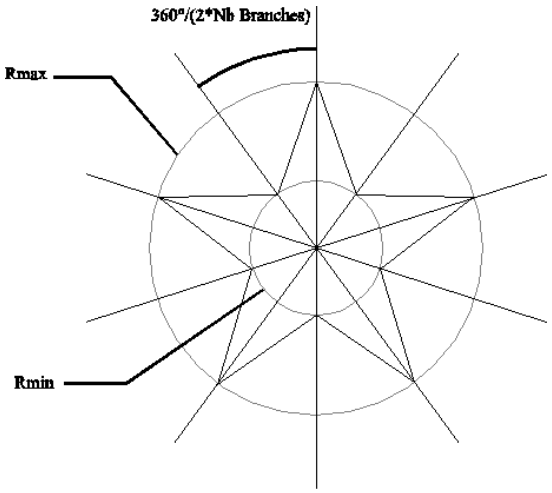
```
{
  short fNbBranches; // Number of Star's branches
} StarData;
```

2. For each primitive, you have to decide if you return bicubic patches of facets. The Z-buffer in the perspective window can not use the ray-tracing information so you have to create patches or facets for the Sphere too.

The shell knows your choice between facets and patches by calling the function **IsPatch-Based**:

```
BOOLEAN Facets::IsPatchBased()
{
  return FALSE; // The 3D Star is based on facets and not on Patches
}
```

3. Then you have to implement one of these two functions: **EnumPatches** and **EnumFacets**.
This Figure show you how to create the middle section of the Star.



How to construct a star

For the 3D Star we only use facets, so we implement **EnumFacets**:

```
HRESULT Facets::EnumFacets(THIS_ EnumFacetsCallback callback, void*
      privData, NUM3D fidelity)
{
  short i;
  FACET3D StarFacet;
  VECTOR3D v1,v2,normal;
  NUM3D angle,anglestep,radius1,radius2,radiusswap,sinus,cosinus;
  NUM3D k360=ShortToQuickFix(360);

  angle=kQuickFixZero;
  anglestep=(k360/ShortToQuickFix(fData.fNbBranches))>>1;
  radius1=ShortToQuickFix(10);
  radius2=ShortToQuickFix(4);
```

```
// Inferior Facets
// -- Common Vertex of each inferior facets
StarFacet.fVertices[0].fVertex[0]=kQuickFixZero;
StarFacet.fVertices[0].fVertex[1]=kQuickFixZero;
StarFacet.fVertices[0].fVertex[2]=ShortToQuickFix(-4);

for (i=0;i<(fData.fNbBranches*2);i++)
{
  // Variable information of the common vertex
  StarFacet.fUVSpace=0;
  StarFacet.fVertices[0].fu=kQuickFixOne;
  StarFacet.fVertices[0].fv=(angle+(anglestep>>1))/k360;
  // Second Vertex information
  angle.DegreeGetSinCos(sinus,cosinus);
  StarFacet.fVertices[1].fVertex[0]=cosinus*radius1;
  StarFacet.fVertices[1].fVertex[1]=sinus*radius1;
  StarFacet.fVertices[1].fVertex[2]=kQuickFixZero;
  StarFacet.fVertices[1].fu=kQuickFixOneHalf;
  StarFacet.fVertices[1].fv=angle/k360;
  // Last Vertex information
  angle += anglestep;
  angle.DegreeGetSinCos(sinus,cosinus);
  StarFacet.fVertices[2].fVertex[0]=cosinus*radius2;
  StarFacet.fVertices[2].fVertex[1]=sinus*radius2;
  StarFacet.fVertices[2].fVertex[2]=kQuickFixZero;
  StarFacet.fVertices[2].fu=kQuickFixOneHalf;
  StarFacet.fVertices[2].fv=angle/k360;
  // Swap the radius to "alternate" the point
  radiusswap=radius1;radius1=radius2;radius2=radiusswap;
  // Normal vector calculation
  v1=StarFacet.fVertices[1].fVertex-StarFacet.fVertices[0].fVertex;
  v2=StarFacet.fVertices[2].fVertex-StarFacet.fVertices[0].fVertex;
  normal=v2^v1; // Vectorial product
  normal.Normalize(); // Normalization
  StarFacet.fVertices[0].fNormal=normal; // the 3 points have the same
    normal vector
  StarFacet.fVertices[1].fNormal=normal;
  StarFacet.fVertices[2].fNormal=normal;
  // callback is used to give a facet to the shell
  callback(&StarFacet,privData);
}


// We use the same algorithm for the superior facets
// there is only the common vertex which is different

angle=kQuickFixZero;
radius1=ShortToQuickFix(10);
radius2=ShortToQuickFix(4);

// Superior Facets
StarFacet.fVertices[0].fVertex[0]=kQuickFixZero;
StarFacet.fVertices[0].fVertex[1]=kQuickFixZero;
StarFacet.fVertices[0].fVertex[2]=ShortToQuickFix(4);
```

```
  for (i=0;i<(fData.fNbBranches*2);i++)
  { // First-Common point
    StarFacet.fUVSpace=0;
    StarFacet.fVertices[0].fu=kQuickFixZero;
    StarFacet.fVertices[0].fv=(angle+(anglestep>>1))/k360;
    // Second point
    angle.DegreeGetSinCos(sinus,cosinus);
    StarFacet.fVertices[1].fVertex[0]=cosinus*radius1;
    StarFacet.fVertices[1].fVertex[1]=sinus*radius1;
    StarFacet.fVertices[1].fVertex[2]=kQuickFixZero;
    StarFacet.fVertices[1].fu=kQuickFixOneHalf;
    StarFacet.fVertices[1].fv=angle/k360;
    // Last point
    angle += anglestep;
    angle.DegreeGetSinCos(sinus,cosinus);
    StarFacet.fVertices[2].fVertex[0]=cosinus*radius2;
    StarFacet.fVertices[2].fVertex[1]=sinus*radius2;
    StarFacet.fVertices[2].fVertex[2]=kQuickFixZero;
    StarFacet.fVertices[2].fu=kQuickFixOneHalf;
    StarFacet.fVertices[2].fv=angle/k360;
    // radius swapping
    radiusswap=radius1;radius1=radius2;radius2=radiusswap;
    // Normal calculation
    v1=StarFacet.fVertices[1].fVertex-StarFacet.fVertices[0].fVertex;
    v2=StarFacet.fVertices[2].fVertex-StarFacet.fVertices[0].fVertex;
    normal=v1^v2;
    normal.Normalize();
    StarFacet.fVertices[0].fNormal=normal;
    StarFacet.fVertices[1].fNormal=normal;
    StarFacet.fVertices[2].fNormal=normal;
    // callback function
    callback(&StarFacet,privData);
  }

  return NOERROR;
  }
```

We do not use the *fidelity* parameter because each facets match perfectly the surface of the object. But for example with a sphere, if the *fidelity* is greater you must return more facets to better match the sphere.

4. For the Teapot example which uses bicubic patches, we have a list of patches and we only call the callback function for each patches. The sphere example is different because we create each patches (one eighth of a sphere) with a mathematical formula.

```
// Teapot Example :

HRESULT Teapot::EnumPatches(THIS_ EnumPatchesCallback callback, void*
      privData)
{ short indexPatch;
  PATCH3D TeapotPatch;
  short uPatchIndex,vPatchIndex;
  VECTOR3D aVertex;
```

```
  TeapotPatch.fu[0]=kQuickFixZero;
  TeapotPatch.fu[1]=kQuickFixZero;
  TeapotPatch.fv[0]=kQuickFixZero;
  TeapotPatch.fv[1]=kQuickFixZero;
  TeapotPatch.fUVSpace=0;
   TeapotPatch.fReserved=0;


  for (indexPatch=0;indexPatch<NUM_PATCHES;indexPatch++)
  { for (uPatchIndex=0;uPatchIndex<4;uPatchIndex++)
    {
      for (vPatchIndex=0;vPatchIndex<4;vPatchIndex++)
      {
        aVertex[0]=DoubleToQuickFix(vertex[vertex_index[index-
      Patch][uPatchIndex][vPatchIndex]-1][0]);
        aVertex[1]=DoubleToQuickFix(vertex[vertex_index[index-
      Patch][uPatchIndex][vPatchIndex]-1][1]);
        aVertex[2]=DoubleToQuickFix(vertex[vertex_index[index-
      Patch][uPatchIndex][vPatchIndex]-1][2]);
        aVertex*=kTeapotSize;
        TeapotPatch.fVertices[uPatchIndex][vPatchIndex]=aVertex;
      }
    }
    callback(&TeapotPatch,privData);
  }
  return NOERROR;
   }



// Sphere Example :

// Functions to create the Patches of the Sphere
// -- Inverse a Patch
void InversePatch(PATCH3D *aPatch) {
  VECTOR3D swapPoint;

  for (short uu=0;uu<=1;uu++)
  {
    for (short vv=0;vv<4;vv++)
    {
      swapPoint=aPatch->fVertices[uu][vv];
      aPatch->fVertices[uu][vv]=aPatch->fVertices[3-uu][vv];
      aPatch->fVertices[3-uu][vv]=swapPoint;
    }
  }
}

// Create Default Patches of the Sphere
void MakeSpherePatch(PATCH3D *aPatch,NUM3D di,NUM3D dj,NUM3D dk)
{
  NUM3D magicFactor=DoubleToQuickFix(0.552284749830793398);
  NUM3D swapUV;

  aPatch->fVertices[0][3][0]=kQuickFixZero;
```

```
aPatch->fVertices[0][3][1]=kQuickFixZero;
aPatch->fVertices[0][3][2]=dk;

aPatch->fVertices[0][2][0]=magicFactor*di;
aPatch->fVertices[0][2][1]=kQuickFixZero;
aPatch->fVertices[0][2][2]=dk;

aPatch->fVertices[0][1][0]=di;
aPatch->fVertices[0][1][1]=kQuickFixZero;
aPatch->fVertices[0][1][2]=magicFactor*dk;

aPatch->fVertices[0][0][0]=di;
aPatch->fVertices[0][0][1]=kQuickFixZero;
aPatch->fVertices[0][0][2]=kQuickFixZero;

aPatch->fVertices[1][0][0]=di;
aPatch->fVertices[1][0][1]=magicFactor*dj;
aPatch->fVertices[1][0][2]=kQuickFixZero;

aPatch->fVertices[2][0][0]=magicFactor*di;
aPatch->fVertices[2][0][1]=dj;
aPatch->fVertices[2][0][2]=kQuickFixZero;

aPatch->fVertices[3][0][0]=kQuickFixZero;
aPatch->fVertices[3][0][1]=dj;
aPatch->fVertices[3][0][2]=kQuickFixZero;

aPatch->fVertices[3][1][0]=kQuickFixZero;
aPatch->fVertices[3][1][1]=dj;
aPatch->fVertices[3][1][2]=magicFactor*dk;

aPatch->fVertices[3][2][0]=kQuickFixZero;
aPatch->fVertices[3][2][1]=magicFactor*dj;
aPatch->fVertices[3][2][2]=dk;

aPatch->fVertices[3][3][0]=kQuickFixZero;
aPatch->fVertices[3][3][1]=kQuickFixZero;
aPatch->fVertices[3][3][2]=dk;

aPatch->fVertices[2][3][0]=kQuickFixZero;
aPatch->fVertices[2][3][1]=kQuickFixZero;
aPatch->fVertices[2][3][2]=dk;

aPatch->fVertices[1][3][0]=kQuickFixZero;
aPatch->fVertices[1][3][1]=kQuickFixZero;
aPatch->fVertices[1][3][2]=dk;

//middle of the Patch (control points)
aPatch->fVertices[1][2][0]=magicFactor*di;
aPatch->fVertices[1][2][1]=magicFactor*magicFactor*dj;
aPatch->fVertices[1][2][2]=dk;

aPatch->fVertices[2][2][0]=magicFactor*magicFactor*di;
aPatch->fVertices[2][2][1]=magicFactor*dj;
aPatch->fVertices[2][2][2]=dk;
```

```
aPatch->fVertices[1][1][0]=di;
aPatch->fVertices[1][1][1]=magicFactor*dj;
aPatch->fVertices[1][1][2]=magicFactor*dk;

aPatch->fVertices[2][1][0]=magicFactor*di;
aPatch->fVertices[2][1][1]=dj;
aPatch->fVertices[2][1][2]=magicFactor*dk;

// Patch's uv-space information
if (di>kQuickFixZero)
{
  if (dj>kQuickFixZero)
  {
    aPatch->fu[0]=kQuickFixZero;  // 0
    aPatch->fu[1]=kUmax>>2;       // to 90°
  }
  else {
    aPatch->fu[0]=(kUmax+kUmax+kUmax)>>2; // 270°
    aPatch->fu[1]=kUmax;                  // to 360°
  }
}
else
{
  if (dj>kQuickFixZero)
  {
    aPatch->fu[0]=kUmax>>2; // 90°
    aPatch->fu[1]=kUmax>>1; // to 180°
  }
  else
  {
    aPatch->fu[0]=kUmax>>1;               // 180°
    aPatch->fu[1]=(kUmax+kUmax+kUmax)>>2; // to 270°
  }
}

if (dk>kQuickFixZero)
{
  aPatch->fv[0]=kVmax>>1;  // 90°
  aPatch->fv[1]=kVmax;     // to 180°
}
else
{
  swapUV=aPatch->fu[0];
  aPatch->fu[0]=aPatch->fu[1];
  aPatch->fu[1]=swapUV;

  aPatch->fv[0]=kVmax>>1;
  aPatch->fv[1]=kQuickFixZero;
}

int signi=(di>kQuickFixZero?1:-1);
int signj=(dj>kQuickFixZero?1:-1);
int signk=(dk>kQuickFixZero?1:-1);
if (signi*signj*signk<0)
{
```

```
    InversePatch(aPatch);
  }

  aPatch->fUVSpace=0;
  aPatch->fReserved=0;
}

HRESULT Sphere::EnumPatches(EnumPatchesCallback callback, void* priv-
      Data)
{
  PATCH3D aPatch;

  MakeSpherePatch(&aPatch, kDefaultSphereRadius, kDefaultSphereRadius,
      kDefaultSphereRadius);
  callback(&aPatch,privData);
  MakeSpherePatch(&aPatch, -kDefaultSphereRadius, kDefaultSphereRadius,
      kDefaultSphereRadius);
  callback(&aPatch,privData);
  MakeSpherePatch(&aPatch, kDefaultSphereRadius, -kDefaultSphereRadius,
      kDefaultSphereRadius);
  callback(&aPatch,privData);
 MakeSpherePatch(&aPatch, -kDefaultSphereRadius, -kDefaultSphereRadius,
      kDefaultSphereRadius);
  callback(&aPatch,privData);
  MakeSpherePatch(&aPatch, kDefaultSphereRadius, kDefaultSphereRadius, -
      kDefaultSphereRadius);
  callback(&aPatch,privData);
  MakeSpherePatch(&aPatch, -kDefaultSphereRadius, kDefaultSphereRadius,
      -kDefaultSphereRadius);
  callback(&aPatch,privData);
  MakeSpherePatch(&aPatch, kDefaultSphereRadius, -kDefaultSphereRadius,
      -kDefaultSphereRadius);
  callback(&aPatch,privData);
 MakeSpherePatch(&aPatch, -kDefaultSphereRadius, -kDefaultSphereRadius,
      -kDefaultSphereRadius);
  callback(&aPatch,privData);

  return NOERROR;
}
```

5. Now, you have to give the Bounding Box of your object with the function **GetBBox**:

```
HRESULT Sphere::GetBBox(BOX3D* bbox)
{ bbox->fMin[0]=-kDefaultSphereRadius;
  bbox->fMax[0]=kDefaultSphereRadius;
  bbox->fMin[1]=-kDefaultSphereRadius;
  bbox->fMax[1]=kDefaultSphereRadius;
  bbox->fMin[2]=-kDefaultSphereRadius;
  bbox->fMax[2]=kDefaultSphereRadius;
  return NOERROR;
}


HRESULT Teapot::GetBBox(BOX3D* bbox)
```

```
{
   return ResultFromScode(E_NOTIMPL); // the bounding box is calculated
      by the shell
}


HRESULT Facets::GetBBox(BOX3D* bbox)
{ bbox->fMin[0]=-ShortToQuickFix(10);
  bbox->fMax[0]=ShortToQuickFix(10);
  bbox->fMin[1]=-ShortToQuickFix(10);
  bbox->fMax[1]=ShortToQuickFix(10);
  bbox->fMin[2]=-ShortToQuickFix(4);
  bbox->fMax[2]=ShortToQuickFix(4);
  return NOERROR;
}
```

The hot point is automatically at the coordinates (0,0,0) in the local coordinate system, so if you want to have an hot point not in the center of the object you have to translate the object in its local coordinate system.

6. After you have defined all the geometric calls, you can define the shading calls (i.e. the UV-Space). Because it is too difficult to define an UV Space on the teapot, we only define an UV-Space on the sphere and the 3D Star.
The UV Space are also use in the geometric calls, each point is given an uv-coordinate and an uv-space ID.
Then you have to give some more information about the uv-spaces. This will be done with the functions **GetUVSpaceCount, GetUVSpace** and **UV2XYZ**:
The first one gives the number of defined uv-spaces on the object.
The second one gives information about each UV Space of the object.
And the last one gives a method to calculate 3D coordinates with only uv-coordinates, if this function is not implemented, a standard interpolation of the facets or the patches will be used to calculate 3D coordinates.

```
ULONG Facets::GetUVSpaceCount()
{
  return 1; // the star is describe with only 1 UV-Space
}

HRESULT Facets::GetUVSpace(ULONG uvSpaceID, UVSpaceInfo* uvSpaceInfo)
{
  if (uvSpaceID == 0)
  {
    uvSpaceInfo->fMin[0] = kQuickFixZero; // u coordinate goes from 0 to
      1
    uvSpaceInfo->fMax[0] = kQuickFixOne;
    uvSpaceInfo->fMin[1] = kQuickFixZero; // v coordinate goes from 0 to
      1
    uvSpaceInfo->fMax[1] = kQuickFixOne;
    uvSpaceInfo->fWraparound[0] = FALSE;  // No Wrap around
    uvSpaceInfo->fWraparound[1] = FALSE;
    uvSpaceInfo->fIsFlatSurface = FALSE;  // the surface is not flat
  }
  return NOERROR;
```

```
}

// We use the default interpolation method to get all the coordinate of a
      point in UV Coordinates
HRESULT Facets::UV2XYZ(VECTOR2D* uv, BOOLEAN* inUVSpace, VECTOR3D*
      resultPosition) {
  return ResultFromScode(E_NOTIMPL);
}


ULONG Sphere::GetUVSpaceCount()
{
  return 1; // the Sphere is describe with only 1 UV-Space
}

HRESULT Sphere::GetUVSpace(ULONG uvSpaceID, UVSpaceInfo* uvSpaceInfo)
{
  if (uvSpaceID == 0)
  {
    // UV-Space is equivalent to Spherical Coordinates
    uvSpaceInfo->fMin[0] = kQuickFixZero;      // u coordinate goes from
      0 to 360
    uvSpaceInfo->fMax[0] = kUmax;
    uvSpaceInfo->fMin[1] = kQuickFixZero;      // v coordinate goes from
      0 to 180
    uvSpaceInfo->fMax[1] = kVmax;

    uvSpaceInfo->fWraparound[0] = TRUE;
    uvSpaceInfo->fWraparound[1] = FALSE;
    uvSpaceInfo->fIsFlatSurface = FALSE;  // the surface is not flat
  }
  return NOERROR;
}

HRESULT Sphere::UV2XYZ(VECTOR2D* uv, BOOLEAN* inUVSpace, VECTOR3D*
      resultPosition) {
  NUM3D phi,theta;
  NUM3D sinphi,cosphi;
  NUM3D sintheta,costheta;

  *inUVSpace=TRUE;
  if (((*uv)[0]<kQuickFixZero) || ((*uv)[0]>=kUmax)) *inUVSpace=FALSE;
  if (((*uv)[1]<kQuickFixZero) || ((*uv)[1]>=kVmax)) *inUVSpace=FALSE;

  if (*inUVSpace==TRUE)
  {
    phi=(*uv)[0];
    theta=(*uv)[1]-(kVmax>>1);

    // Spherical Coordinates To XYZ-Coordinates
    phi.DegreeGetSinCos(sinphi,cosphi);
    theta.DegreeGetSinCos(sintheta,costheta);

    (*resultPosition)[0]=cosphi*costheta;
    (*resultPosition)[1]=sinphi*costheta;
    (*resultPosition)[2]=sintheta;
```

```
    (*resultPosition)*=kDefaultSphereRadius;
  }

  return NOERROR;
}
```

7. If you can give the real coordinate of an intersection between a ray and the surface of the object, it can be better to implement the ray-tracing calls. In this case, the final image will be better because the object will not be approximated. This is very easy to implement for a sphere.
You have 3 functions: **RayHit**, **RayAllHits**, and **GetRayHitDetails**.
If you implement **RayHit**, you must implement **GetRayHitDetails**.
If you implement **RayAllHits**, you must implement the two other functions.

In the **RayHit** function you only have to calculate the position (P) and the parameter (t) of the intersection:

$$P = Origin + t \times Direction$$

```
HRESULT Sphere::RayHit(BOOLEAN* didHit, Ray3D* aR, RayHitParameters*
      RayHitParams, RayHit3D* hit)
{
  VECTOR3D OriginToCenter;
  NUM3D    DirectionNorm2;
  NUM3D    dotProduct;
  NUM3D    t;              // Sphere Center = Ray Origin + t * Ray Direc-
      tion
  VECTOR3D CH;             // position of the center projected on the ray
  NUM3D    distCH2;         // square of the distance of the projected
      point
  NUM3D    resT;
  NUM3D    delta,delta2;
  NUM3D    radius2=kDefaultSphereRadius*kDefaultSphereRadius;

  OriginToCenter=-aR->fOrigin;

  DirectionNorm2=aR->fDirection*aR->fDirection;
  dotProduct    =aR->fDirection*OriginToCenter;
  t=dotProduct/DirectionNorm2;

  CH=(aR->fOrigin)+(aR->fDirection)*t;
  distCH2=CH*CH;
  if (distCH2>radius2) *didHit=FALSE;
  else if (distCH2==radius2)
  {
    resT=t;
    *didHit=TRUE;
  }
  else
  {
```

```
    *didHit=TRUE;
    delta2=(radius2-distCH2)/DirectionNorm2;
    delta2.GetSquareRoot(delta);
    resT=t-delta;
    if (resT<=RayHitParams->tmin)
    {
      resT=t+delta;
    }
  }
  if (resT<=RayHitParams->tmin)
  { *didHit=FALSE;
  }

  if (resT>RayHitParams->tmax)
  { *didHit=FALSE;
  }

  if (*didHit==TRUE)
  { hit->fPosition=aR->fOrigin+aR->fDirection*resT;
    hit->ft=resT;
  }

  return NOERROR;
}

HRESULT Sphere::GetRayHitDetails(THIS_ RayHit3D* hit)
{ NUM3D phi,theta,rxy;
  NUM3D sinphi,cosphi,sintheta,costheta;

  hit->fNormal=hit->fPosition/kDefaultSphereRadius;

  phi.DegreeSetFromSinCos(hit->fNormal[1],hit->fNormal[0]);
  rxy=hit->fNormal[0]*hit->fNormal[0]+hit->fNormal[1]*hit->fNormal[1];
  rxy.GetSquareRoot(rxy);
  theta.DegreeSetFromSinCos(hit->fNormal[2],rxy);

  if (hit->fShouldSetUV)
  { hit->fUV[0]=phi;
    if (theta>kVmax)
    { theta = theta - (kVmax<<1);
    }
    hit->fUV[1]=theta+(kVmax>>1);
  }
  if (hit->fShouldSetIsoUV)
  {
    phi.DegreeGetSinCos(sinphi,cosphi);
    theta.DegreeGetSinCos(sintheta,costheta);

    hit->fIsoU[0]=-sinphi*kDefaultSphereRadius;
    hit->fIsoU[1]=cosphi*kDefaultSphereRadius;
    hit->fIsoV[2]=kQuickFixZero;
    hit->fIsoV[0]=cosphi*sintheta*kDefaultSphereRadius;
    hit->fIsoV[1]=sinphi*sintheta*kDefaultSphereRadius;
    hit->fIsoV[2]=costheta*kDefaultSphereRadius;
  }
```

```
  return NOERROR;
}
```

8. There is a private resource to add for a primitive extension. This resource is called ‘Cmpp’ (Component Private). For a primitive, this resource contains a short which are the icon Id of the Primitive. Do not forget that you have 3 icons to define : one 24x24 pixels for the hierarchy windows and two 16x16 (selected and not selected) for the toolbar. The first icon has the Id specified in the ‘Cmpp’ resource and the Id of the not selected and selected icon are respectively first Icon Id + 50 and first Icon Id + 100.

Because the 3D Shell do not support yet the Windows Icon resources. You have to use the default icon given with the application. To do this, you must not create a ‘Cmpp’ resource.

Chapter 8 - Writing a Light Source

Family ID : 'lite'
COM Interface ID : IID_IDExLightsource
COM Interface file : I3DExLit.h

Light sources define the lighting in a 3D scene. All kind of light sources can be designed: spot lights, bulb lights, distant lights (sun, moon), etc. Various features can be coded in a Light Source Extension, because the Light Source has complete control on the light intensity returned to the 3D Shell.

The user can combine lights with gels to make additional lighting effects (see the **I3DExLightsourceGe l** interface).

In this part, we will create a « BeamsLight ». It sends beams in all directions like in a night club:

This picture was made with only one BeamsLight in the center of the scene.

1. Like every extensions, you must think about the different parameters needed by your light source. For the BeamsLight, we have defined this data structure :

```
typedef struct LightData {
  short   fHorApertureAngle;    // Angular Limits of the light source in
      degrees
  short   fVerApertureAngle;    //
  NUM3D   fIntensity;           // Light source intensity
  short   fNbBeamsHorizontally; // Number of Beams Horizontally and Ver-
      tically
  short   fNbBeamsVertically;   //
  COLOR3D fLightColor;          // Default color
  short   fBeamAperture;        // Angular Limit of a single Beam
  } LightData;
```

2. Like the Camera extension, you will obtain the transformation data (from the Global System to the Lightsource Local Coordinates System) with the function:

```
HRESULT I3DExLightsource::SetTransform(TRANSFORM3D* transform);
```

The transformation data are organized in a vector (translation) and a matrix (rotation). The

formulae to get the Global Coordinates from the Local Coordinates and vice versa, are the same as written in the Camera Cookbook.
You can find the transformation functions in C++ in the BeamsLight example file (Beamlght.cpp).

To copy the transformation data in your object you only have to do this :

```
HRESULT BeamsLight::SetTransform(TRANSFORM3D* transform) {
  fTransform=*transform; // copy the data and not the pointer
  return ResultFromScode(S_OK);
   }
```

If you think you can make any preprocess calculations that use only the transformation data, do it in this function.

3. Implement the following function to tell the shell that you light source can be represented in the perspective display. For example a infinite light cannot be represented in the perspective display so the function will return FALSE otherwise return TRUE.

```
BOOLEAN BeamsLight::IsVisibleInPerspective(THIS) {
  return TRUE; // the source is not a distant light (like the sun)
               // so it can be in the 3D perspective display.
  }
```

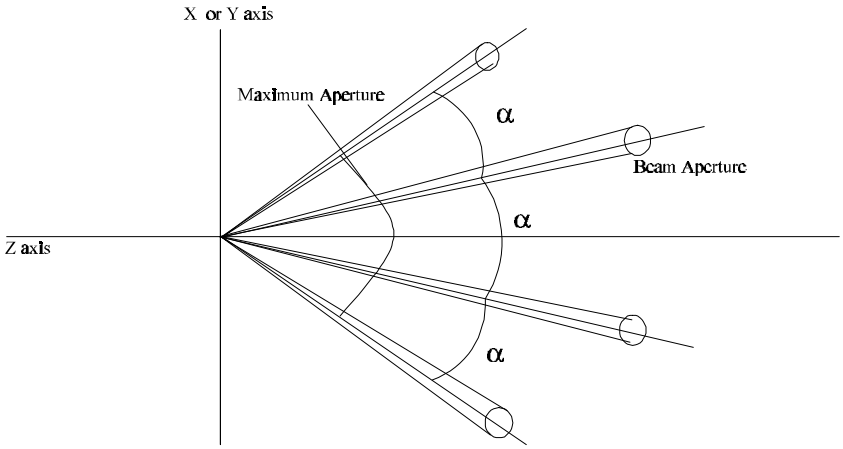
4. There are two important functions in the Lightsource Interface :

```
HRESULT I3DExLightsource::GetDirection(VECTOR3D* position,VECTOR3D*
      resultDirection, NUM3D* resultDistance);
BOOLEAN I3DExLightsource::GetColor(VECTOR3D* position, VECTOR3D* direc-
      tion, NUM3D distance,COLOR3D* result, BOOLEAN* callForShadowEf-
      fect);
```

The **GetDirection** function is most simple to implement. You only have to calculate a vector from the light source center to the 3D position passed to the procedure, return the length of this vector and normalize it :

```
HRESULT BeamsLight::GetDirection(VECTOR3D* position, VECTOR3D* resultDi-
      rection, NUM3D* resultDistance)
{ *resultDirection=fTransform.fT-(*position); // fT is the origin of the
      light source in the Global Coordinates System, given by SetTrans-
      form
  *resultDistance=resultDirection->GetNorm();
  *resultDirection/=(*resultDistance); // must be a unit vector
  return ResultFromScode(S_OK);
}
```

The second function will be as difficult as the light source effect.
For the BeamsLight, we want to have a lot of beams regularly spaced :



Repartition of the light beams

This figure shows how the BeamsLight will illuminate the scene. Each beam will be separated by the angle:

$$\alpha = \frac{\text{MaxAperture}}{NbBeams - 1}$$

If the NbBeams equals one, the beam will be on the Z-axis (with theta or phi equal to zero).

When you have a direction, you have to find the nearest beam (You will not compare each beam, it is NbBeamsHorizontally*NbBeamsVertically comparisons).
First, you have to get the Local Coordinate of the direction vector. For this, you can use the function *GlobalToLocalVector.* Then it is easier to have the angular expression of the direction vector.
That's why you transform the direction vector into spherical coordinates θ and φ (like in the SphericCamera example). Then you can use this formulae to get the beam index :

$$Index_{Hor} = \textbf{Nearest Integer of } ((\frac{\theta}{MaxAperture} + \frac{1}{2}) \times (NbBeams - 1))$$

And the same formulae for the Vertical Index
Be sure that the index is between 1 and *NbBeams*(Horizontally or Vertically)
The 2 index allow you to determinate the nearest main direction beam with these formulae:

$$\theta = MaxApertureHor \times \left(\frac{Index_{Hor}}{NbBeamsHor - 1} - \frac{1}{2} \right)$$

$$\varphi = MaxApertureVec \times \left(\frac{Index_{Ver}}{NbBeamsVer - 1} - \frac{1}{2} \right)$$

$$V = \begin{cases} x = \sin(\theta) \cdot \cos(\varphi) \\ y = \sin(\varphi) \\ z = \cos(\theta) \cdot \cos(\varphi) \end{cases}$$

When you have the two vectors in the same coordinates system, you use the dot product to get the small angle between the main direction of the beam and the given direction. Because these 2 vectors are normalized we have :

$$\cos(angle\_beam\_direction) = \vec{D} \cdot \vec{V}$$

To know if the given direction is in the light beam (i.e. the *angle_beam_direction* is bellow the beam aperture angle divided by 2), you compare the cosine of the beam aperture divided by 2, and the cosine of *angle_beam_direction*. If the first one is bigger, the point is not in the beam light. So the implementation of the **GetColor** function will be :

```
BOOLEAN BeamsLight::GetColor(VECTOR3D* position, VECTOR3D* direction,
      NUM3D distance, COLOR3D* result, BOOLEAN* callForShadowEffect) {
  VECTOR3D localVector,refDir;
  VECTOR3D projVector;
  NUM3D    angle,nearest_dir,hor,ver,theta,phi,r;
  NUM3D    costheta,sintheta,cosphi,sinphi;
  NUM3D    cosdifferentialAngle;
  NUM3D    angleLimit=ShortToQuickFix(360);

  *callForShadowEffect=TRUE; // We always want shadows for this light-
      source

  GlobalToLocalVector(&fTransform,direction,&localVector);

  // Initialize result to default color (intensity included)
  *result=fData.fLightColor;
  result->R*=fData.fIntensity;
  result->G*=fData.fIntensity;
  result->B*=fData.fIntensity;

  // Nearest direction vector determination

  if (fData.fNbBeamsHorizontally!=1) {
    // direction is calculated in the Spherical Coordinates (see Spheri-
      cal Camera)
    // -- Horizontal Determination
    angle.DegreeSetFromSinCos(localVector[0],localVector[2]);
    if (angle>(angleLimit>>1)) {
      angle-=angleLimit;
```

```
    }
  if ((angle>fHorAng+fBeamAngle)||(angle<-fHorAng-fBeamAngle)) return
    FALSE; // the point is outside the maximum horizontal aperture

  nearest_dir=(angle/(fHorAng<<1)+kQuickFixOneHalf)*ShortToQuick-
    Fix(fData.fNbBeamsHorizontally-1);
  hor=Qfloor(nearest_dir);
  if (nearest_dir-hor>kQuickFixOneHalf) hor+=kQuickFixOne;
  if (hor>ShortToQuickFix(fData.fNbBeamsHorizontally-1))
       hor=ShortToQuickFix(fData.fNbBeamsHorizontally-1);
  if (hor<kQuickFixZero) hor=kQuickFixZero;
  // hor must be between 0 and fNbBeamsHorizontally-1
  theta=(fHorAng<<1)*(hor/ShortToQuickFix(fData.fNbBeamsHorizontally-
    1)-kQuickFixOneHalf);
  }
else {
  theta=kQuickFixZero;
  }
if (fData.fNbBeamsVertically!=1) {
  // -- Vertical Determination
  r=localVector[0]*localVector[0]+localVector[2]*localVector[2];
  r.GetSquareRoot(r);
  angle.DegreeSetFromSinCos(localVector[1],r);
  if (angle>(angleLimit>>1)) {
    angle-=angleLimit;
    }
  if ((angle>fVerAng+fBeamAngle)||(angle<-fVerAng-fBeamAngle)) return
    FALSE;
    // the point is outside the maximum vertical aperture
  nearest_dir=(angle/(fVerAng<<1)+kQuickFixOneHalf)*ShortToQuick-
    Fix(fData.fNbBeamsVertically-1);
  ver=Qfloor(nearest_dir);
  if (nearest_dir-ver>kQuickFixOneHalf) ver+=kQuickFixOne;
  if (ver>ShortToQuickFix(fData.fNbBeamsVertically-1)) ver=Short-
    ToQuickFix(fData.fNbBeamsVertically-1);
  if (ver<kQuickFixZero) ver=kQuickFixZero;
  // ver must be between 0 and fNbBeamsVertically-1
  phi=(fVerAng<<1)*(ver/ShortToQuickFix(fData.fNbBeamsVertically-1)-
    kQuickFixOneHalf);
  }
else {
  phi=kQuickFixZero;
  }
// nearest direction vector

// Spherical Coordinates of the direction vector
theta.DegreeGetSinCos(sintheta,costheta);
phi.DegreeGetSinCos(sinphi,cosphi);
refDir[0]=sintheta*cosphi;
refDir[1]=sinphi;
refDir[2]=costheta*cosphi;
// Direction comparison
cosdifferentialAngle=refDir*localVector;    // dot product to get the
    cosinus of the angle
if (cosdifferentialAngle<fBeamLimit) return FALSE; // the point is out-
    side the Beam
```

```
  return TRUE;
  }
```

This functions return TRUE if the light illuminates the point, otherwise it returns FALSE. The *callForShadowEffect* returned value is set to tell the renderer that the light source creates shadows, and that the renderer must call ShadowEffect to obtain the special effect of the shadows.

6. Now, you have to implement the **ShadowEffect** function :

```
HRESULT BeamsLight::ShadowEffect(NUM3D distance, COLOR3D* result);
```

This function allows you to create smoother shadow by reducing the color intensity if the distanceThru is not too big rather than perfect shadow that are always black. In the BeamsLight example, we only create pure black shadows. So if this function is called, you modify the result color to give no color :

```
HRESULT BeamsLight::ShadowEffect(NUM3D distance, COLOR3D* result)
{
  result->R=kQuickFixZero;
  result->G=kQuickFixZero;
  result->B=kQuickFixZero;
   return ResultFromScode(S_OK);
}
```

Chapter 9 - Writing a Light Source Gel

Family ID : ‘gel’
COM Interface ID : IID_I3DExLightsourceGel
COM Interface file : I3DExGel.h

The user can combine lights with Gels to make additional lighting effects. A Gel can be thought as a colored slide that is put in front of the light source to modify the color and intensity of the light beam.

In this part, we will create a Gel that create a star with a number of branches from 3 to 30.

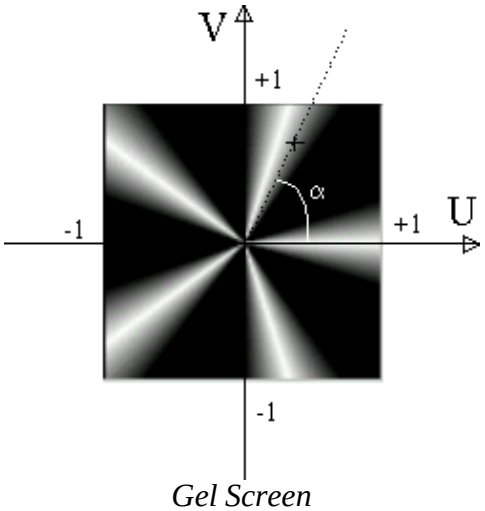
1. The data structure for the exchange with the shell will be :

```
typedef structure GelData {
  short fNbBranches;
}
```

2. You only have to implement one special function for the Gel Interface: **GetGelValue**

```
BOOLEAN GelLight::GetGelValue(VECTOR2D* gelScreenPosition, COLOR3D*
      result);
```

The gel screen is like a slide in front of a light source. This gel screen dimensions are -1.0 to +1.0 in each direction.
To have the star effect, we must have the polar coordinates of the gelScreenPosition:



Gel Screen

When you have this angle, you have to find the branch of the star. An easy way to that is to use a modulo formula:

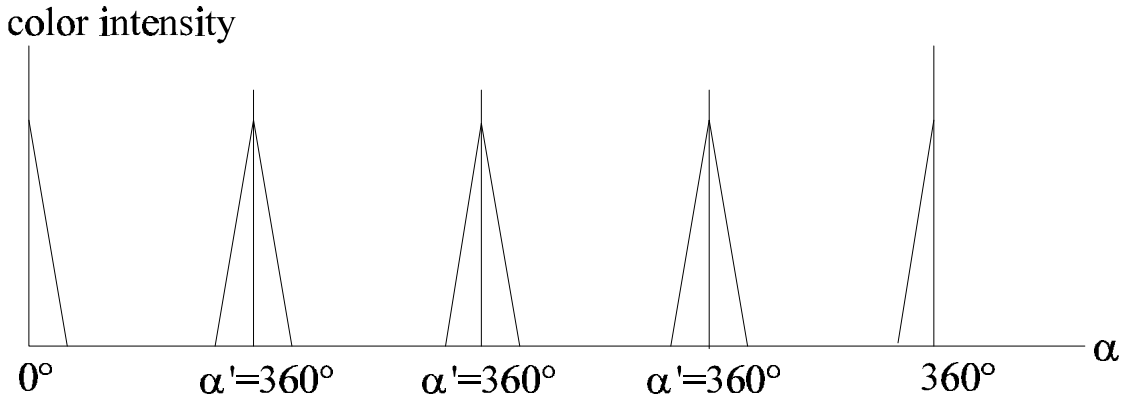
$$' = (\times nbBranches)\%360$$

- α' is an angle between 0 and 360°
- *nbBranch* is the number of branches of the star

Then, if you use a conversion function between α' and the color factor like this one:

Transformation function

you will obtain the peaks periodically spaced in the α-Space:



Example of α-peaks with *nbBranches*=4

The implementation of the function will be:

```
BOOLEAN GelLight::GetGelValues(VECTOR2D* gelScreenPosition,COLOR3D* result)
{ NUM3D alpha;
  NUM3D k360=ShortToQuickFix(360);
  NUM3D graylevel;
  // Get the angular position of the gelScreenPosition
  alpha.DegreeSetFromSinCos((*gelScreenPosition)[1],(*gelScreenPosi-
      tion)[0]);
  alpha*=ShortToQuickFix(fData.fNbBranches);

  // Calculate alpha*nbBranches modulo 360°
  while (alpha>k360) alpha-=k360; // Modulo 360

  // peaks function
```

```
  if (alpha<(k360>>2)) {
    graylevel=((k360>>2)-alpha)/(k360>>2);
    }
  else if (alpha>(k360-(k360>>2))) {
    graylevel=(alpha-(k360-(k360>>2)))/(k360>>2);
    }
  else {
    graylevel=kQuickFixZero;
    }

  result->Mode=0;
  result->R=graylevel;
  result->G=graylevel;
  result->B=graylevel;

  return TRUE;
}
```

Chapter 10 - Writing a Motion Link

Family ID : 'link'
COM Interface ID : IID_I3DExMotionLink
COM Interface file : I3DExLnk.h

The **I3DExMotionLin k** Interface defines motion links between Tree Elements. It allows to define "mechanical" constrains (degrees of freedom) between one Tree Element and its father.

In order to animate the link, you have to put each freedom value (one for each degrees of freedom) in the "pMAP", even if they not appear in the User Interface.

We will create a link that acts like a screw. So we only have one degree of freedom, the number of turns, and we have to know the axis and the step of the screw.

1. First we will describe the needed data structure:

```
typedef struct ScrewLinkData {
  long     fAxis;  // ID of the axis (AXEX AXEY or AXEZ)
  NUM3D    fStep;  // Step of the screw (1 turn -> translation of fStep)
  NUM3D    fFreedomValue; // number of turns
  } ScrewLinkData;
```

2. The 3D Shell has to know the number of degrees of freedom. Therefore it uses the function **GetNbrFreedom( )**. In our case, we only have one degree (the number of turns):

```
short ScrewLink::GetNbrFreedom(THIS) {
  return 1;
  }
```

3. Because the freedom value can be limited, you have to give the range of the values. This range is defined as a range of increment/decrement around the value. In our case, we do not want to limit the number of turns so we will always return the maximum range of increment and decrement:

```
HRESULT ScrewLink::GetFreedomRange(THIS_ short index, NUM3D* min, NUM3D*
      max) {
  if (index==1) {
    *min = -ShortToQuickFix(32767);
    *max =  ShortToQuickFix(32767);
    }
  return NOERROR;
  }
```

4. The shell does not know how to increment each freedom degrees (it cannot directly modify the value) so you have to implement the function **IncrementFreedomValu e**:

```
HRESULT ScrewLink::IncrementFreedomValue(THIS_ short index, NUM3D*
      increment) {
  if (index==1) {
    ThisP->fData.fFreedomValue += *increment;
    }
  return NOERROR;
  }
```

5. The most important thing to create a link is to give the rotation matrix and translation vector due to the freedom values:

```
HRESULT ScrewLink::GetTransform(THIS_ TRANSFORM3D* transform) {
  short u,v,w;
  NUM3D alpha,cosa,sina;
  NUM3D altitude;
  NUM3D m33[3][3];

  altitude = ThisP->fData.fFreedomValue * ThisP->fData.fStep;
  alpha = ThisP->fData.fFreedomValue * kQuickFixTwoPi;

  // Axis permutation:
  if (ThisP->fData.fAxis==kAxisX) {
    u=1;v=2;w=0;
    }
  else if (ThisP->fData.fAxis==kAxisY) {
    u=2;v=0;w=1;
    }
  else if (ThisP->fData.fAxis==kAxisZ) {
    u=0;v=1;w=2;
    }
  transform->fT[u] = kQuickFixZero;
  transform->fT[v] = kQuickFixZero;
  transform->fT[w] = altitude;
  alpha.GetSinCos(sina,cosa);
  m33[u][u]=cosa; m33[u][v]=sina; m33[u][w]=kQuickFixZero;
  m33[v][u]=-sina; m33[v][v]=cosa; m33[v][w]=kQuickFixZero;
  m33[w][u]=kQuickFixZero; m33[w][v]=kQuickFixZero; m33[w][w]=kQuickFix-
      One;
  transform->fR = *(MATRIX3D*)&m33;

  return NOERROR;
  }
```

Note: When the freedom values are initialized the function **GetTransfor m** must return the identical matrix for the rotation and a null vector for the translation.

6. To allow the inverse kinematics mechanism to work with your link, you have to implement the **GetTransfromPartialDerivat e** function. A partial derivative transformation is a transformation where each element is derived by a freedom degree. So you have as much Partial Derivative as freedom degrees:

$$\underline{R} = \begin{bmatrix} \underline{i_x} & \underline{j_x} & \underline{k_x} \\ \underline{i_y} & \underline{j_y} & \underline{k_y} \\ \underline{i_z} & \underline{j_z} & \underline{k_z} \end{bmatrix}$$

$$\underline{T} = \begin{bmatrix} \underline{T_x} \\ \underline{T_y} \\ \underline{T_z} \end{bmatrix}$$

In our case the derivatives are easy to calculate:

```
HRESULT ScrewLink::GetTransformPartialDerivate(THIS_ short index,
      TRANSFORM3D* transform) {
  short u,v,w;
  NUM3D alpha,cosa,sina;
  NUM3D m33[3][3];
  if (ThisP->fData.fAxis==kAxisX) {
    u=1;v=2;w=0;
    }
  else if (ThisP->fData.fAxis==kAxisY) {
    u=2;v=0;w=1;
    }
  else if (ThisP->fData.fAxis==kAxisZ) {
    u=0;v=1;w=2;
    }
  if (index==1) {
    transform->fT[u] = kQuickFixZero;
    transform->fT[v] = kQuickFixZero;
    transform->fT[w] = fData.fStep;
    alpha = kQuickFixTwoPi * fData.fFreedomValue;
    alpha.GetSinCos(sina,cosa);
    m33[u][u]=-kQuickFixTwoPi * sina; m33[u][v]=kQuickFixTwoPi * cosa;
      m33[u][w]=kQuickFixZero;
    m33[v][u]=-kQuickFixTwoPi * cosa; m33[v][v]=-kQuickFixTwoPi * sina;
      m33[v][w]=kQuickFixZero;
    m33[w][u]=kQuickFixZero; m33[w][v]=kQuickFixZero; m33[w][w]=kQuick-
      FixZero;
    transform->fR = *(MATRIX3D*)&m33;

    }
  return NOERROR;
  }
```

Chapter 11 - Writing a Scene Operation

Family ID : ‘scop’
COM Interface ID : IID_I3DExSceneOperation
COM Interface file : I3DExScO.h

The Scene Operation Interface defines a new feature of the 3D Shell. The user will directly access to this function in the ‘Arrange’ menu of the 3D Shell. It allows you to do everything you want in the scene (create new objects, rotate or translate an object or a list of objects).

In this cookbook, we will describe a Scene Operation that creates a stair with every selected object. To do this, we have to duplicate and translate an object.

1. To make a stair, we have to know the number of steps and the relative position of each step. So the data structure will be :

```
typedef struct SceneOpData {
  short fNbStep;
  NUM3D fDx;
  NUM3D fDy;
  NUM3D fDz;
  } SceneOpData;
```

2. There is two call to implement for a Scene Operation. The first one allows you to prepare the data before the shell show the setup dialog of your Scene Operation. This call is **I3DExSceneOperation::Prepar e** :

```
HRESULT SceneOp::Prepare(THIS_ I3DShScene *scene, I3DShTreeElement
      *tree, long index, long total) {
  return NOERROR;
  }
```

I3DExSceneOperation::Prepare() will be called repeatedly for each tree-root in the selection (a selected Tree Element with all its sub-tree elements selected will represent only one selection). If there is nothing selected, Prepare() will still be called once.

In our case, we have nothing to prepare, so we do nothing.

3. The second call is called by the 3D Shell for each tree-root in the selection, like Prepare(). In this function, we shall duplicate the Tree Element and translate it to make the stairs.

```
Boolean SceneOp::DoIt(THIS_  I3DShScene *scene, I3DShTreeElement *tree,
      long index, long total) {
  I3DShTreeElement *newStep;
  TREETRANSFORM3D stepTransform;
  short iStep;
```

```
if (!tree) return FALSE;

for (iStep=0;iStep<ThisP->fData.fNbStep;iStep++) {
  newStep = tree->Clone(TRUE);
  if (!newStep) return FALSE;
  tree->InsertRight(newStep);
  newStep->GetGlobalTransform(&stepTransform);
  stepTransform.fT[0]+=ShortToQuickFix(iStep)* fData.fDx;
  stepTransform.fT[1]+=ShortToQuickFix(iStep)* fData.fDy;
  stepTransform.fT[2]+=ShortToQuickFix(iStep)* fData.fDz;
  newStep->SetGlobalTransform(&stepTransform);
 }
 return TRUE;
}
```

4. To tell the Shell when it can use the scene operation, there is a ‘Cmpp’ (Component Private) resource. This resource contains a long which each bit have the following signification :

bit 31 : the Scene Operation can be use when there are more than 31 selected items.
bit 30 : the Scene Operation can be use when there are 30 selected items.
bit 29 : the Scene Operation can be use when there are 29 selected items.
…
bit 1 : the Scene Operation can be use when there is only one tree element selected.
bit 0 : the Scene Operation can be called when there is no selection.

This resource can be created on both cross-Platform and Windows-only sides, but for the Windows resource do not forget to put the long in the Motorola format (Highest byte first).

Chapter 12 - Writing a Shader

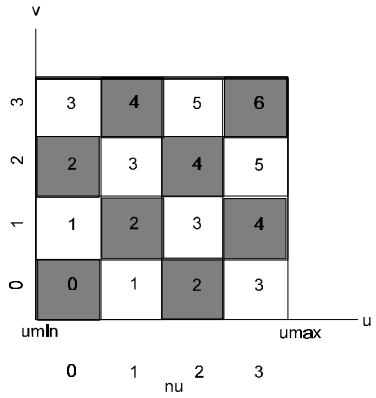
Family ID : ‘shdr’
COM Interface ID : IID_I3DExShader
COM Interface file : I3DExSha.h

Shaders are very powerful 3D Components used to give photorealistic appearance to the 3D objects surfaces. They define the shading values at each point of an object surface: color, shininess, reflectivity, etc. used to compute the illumination on the surface.

There are two types of Shaders: Shaders and Sub-Shaders. Shaders take care of everything: they take all the input parameters and return all the shading values (see **DoShade**). The advantage is to have complete control on the shading process, the disadvantage is that the shader is more difficult to program and not very flexible. Sub-Shaders are more like little bricks that can be combined together by the user to « build » is own shader. Sub-Shaders return either a value (**GetValue**), a color (**GetColor**) or a vector (**GetVector**). They are more simple to program and much more flexible.

This Cookbook explains how to build two Sub-Shaders: one that returns a value (a checker), and one a color (a Rainbow shader).

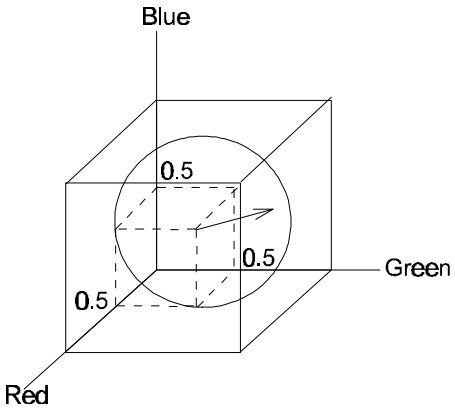
1. Before writing any code, you have to know the parameters specific to your shader and needed from the Shell to perform its calculations (point position, Normal, UV Coordinates, etc...).
In the Checker example, we need to know the numbers of squares horizontally and vertically, called nbSquareU and nbSquareV.



To create the Checker effect, we only need the UV Coordinates from the shell. We use the fact that if each number in the square is the addition of the integer part of nu and nv which are calculate with this formulae :, the even and odd numbers are arranged like in a checker.

$$n_u = \frac{(u - u_{\min})}{(u_{\max} - u_{\min})} \cdot nbSquareU$$

In the Rainbow example, we choose to use the Normal (in Local or Global Coordinates) to make out a color in the RGB-Cube.

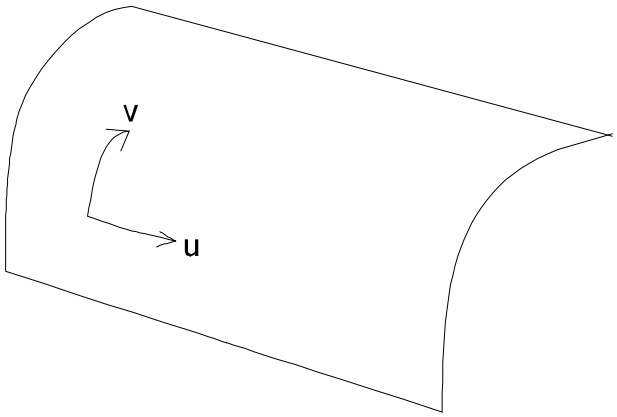


When you have your data structure to communicate with the shell, and when you have the resources, you can begin to implement your new shader.

2. You have to tell the Shell what your shader needs. To do that, you use two functions called **GetShadingFlags** and **DependsOnAppliedExtent**.

HRESULT I3DExShader::GetShadingFlags(ShadingFlags* theFlags);
BOOLEAN I3DExShader::DependsOnAppliedExtent(void);

if you create a shader which depends on the UV-Space, return TRUE with **DependsOnAppliedExtent** function.



U-V Coordinates on a Surface

GetShadingFlags allows the 3D Shell to learn which parameters the shader will need to perform the shading calculations. This way, only the minimal number of parameters is calculated (for more details on the ShadingFlags structure, see the descriptions of the data structure).

Because the Checker depends on the UV-Coordinates the implementation will look like:

```
BOOLEAN CheckerShader::DependsOnAppliedExtent(THIS) {
   return TRUE;
   }
```

```
HRESULT CheckerShader::GetShadingFlags(THIS_ ShadingFlags* theFlags) {
  theFlags->NeedsUV  = TRUE;  // the Checker uses UV Coordinates
  theFlags->CallOnce = FALSE; // the Checker is called for each point
  return NOERROR;
   }
```

But the Rainbow Shader uses only the Normal in the Global or Local Coordinates System:

```
BOOLEAN RainbowShader::DependsOnAppliedExtent(THIS)
{
  return FALSE; // The Rainbow doesn't use the UV Space but only the Nor-
      mal Vectors
}

HRESULT RainbowShader::GetShadingFlags(THIS_ ShadingFlags* theFlags)
{
   theFlags->fNeedsNormalLoc = TRUE;
   theFlags->fNeedsNormal = TRUE;
   theFlags->CallOnce = FALSE; // the Rainbow is called for each point
  return NOERROR;
}
```

3. A shader can be implemented in one of 3 ways : by value, color, or vector. You must implement one of them and return *ResultFromScode(E_NOTIMPL)* for the others.
The 3 functions are :

```
HRESULT I3DExShader::GetValue(NUM3D *result, ShadingIn* theShadingIn,
      ShadingElem* theShadingElem);

HRESULT I3DExShader::GetColor(COLOR3D *result, ShadingIn* theShadingIn,
      ShadingElem* theShadingElem);

HRESULT I3DExShader::GetVector(VECTOR3D *result, ShadingIn* theShadin-
      gIn, ShadingElem* theShadingElem);
```

which return respectively a value, a color, and a vector.
A Checker have only two states (black or white) and it can be used with the operator Mix for example. It only returns 0.0 or 1.0 as a Value. So the **GetColor** and **GetVector** will be like this:

```
HRESULT CheckerShader::GetColor(THIS_ COLOR3D*, ShadingIn*, Shadin-
      gElem*) {
   return ResultFromScode(E_NOTIMPL);
   }

HRESULT CheckerShader::GetVector(THIS_ VECTOR3D*, ShadingIn*, Shadin-
      gElem*) {
   return ResultFromScode(E_NOTIMPL);
   }
```

And the function **GetValue**:

```
HRESULT CheckerShader::GetValue(THIS_ NUM3D* result, ShadingIn* theShad-
```

```
      ingIn, ShadingElem* theShadingElem) {
   NUM3D tempu=ShortToQuickFix(CheckerPublicData.nbSquareU)/( theShadin-
      gElem-> fShaderBox.fRight-theShadingElem->fShaderBox.fLeft);
  NUM3D tempv=ShortToQuickFix(CheckerPublicData.nbSquareV)/( theShadin-
      gElem-> fShaderBox.fTop-theShadingElem->fShaderBox.fBottom);
  QuickWide nu = QuickMulWide(theShadingIn->fUV[0]-theShadingElem-
      >fShaderBox.fLeft, tempu);
  QuickWide nv = QuickMulWide(theShadingIn->fUV[1]-theShadingElem-
      >fShaderBox.fTop, tempv);
  if (!((QWFloor(nu)+QWFloor(nv)) & 0x00000001)) // Even Number ?
  { *result = ShortToQuickFix(0);
  }
  else
  { *result = ShortToQuickFix(1);
  }
  return NOERROR;
   }
```

On the contrary, the Rainbow returns a color, so you have:

```
HRESULT RainbowShader::GetValue(ShadingIn*,ShadingElem*,NUM3D*) {
   return ResultFromScode(E_NOTIMPL);
   }

HRESULT RainbowShader::GetVector(ShadingIn*,ShadingElem*,VECTOR3D*) {
   return ResultFromScode(E_NOTIMPL);
   }

HRESULT RainbowShader::GetColor(THIS_ COLOR3D* result, ShadingIn*
      theShadingIn, ShadingElem* theShadingElem){
   NUM3D temp=ShortToQuickFix(RainbowPublicData.fIntensity);
   temp/=ShortToQuickFix(100);
  result->Mode = 0; // RGB color mode
  if (RainbowPublicData.fModeLocalOrGlobal==1) {
    result->R    = (((theShadingIn->fNormalLoc[0])*temp)>>1)+kQuickFix-
      OneHalf;
   result->G = (((theShadingIn->fNormalLoc[1])*temp)>>1)+kQuickFixOne-
      Half;
   result->B    = (((theShadingIn->fNormalLoc[2])*temp)>>1)+kQuickFix-
      OneHalf;
   }
  else {
   result->R    = (((theShadingIn->fNormal[0])*temp)>>1)+kQuickFixOne-
      Half;
   result->G = (((theShadingIn->fNormal[1])*temp)>>1)+kQuickFixOneHalf;
   result->B    = (((theShadingIn->fNormal[2])*temp)>>1)+kQuickFixOne-
      Half;
    }
  return NOERROR;
  }
```

4. Implement the comparison of two shader:

```
BOOLEAN I3DExShader::IsEqualTo(I3DExShader* aShader);
```

You will compare your shader with the shader pointed by *aShader*.

If the data of the two shaders are equal, you return TRUE, else you return FALSE.
For the checker, we only have to compare the number of squares Horizontally and Vertically (we gather all the CheckerShader data in a structure called **CheckerPublicDat a**)

```
BOOLEAN CheckerShader::IsEqualTo(I3DExShader* aShader)
{ return ((CheckerPublicData.nbSquareU==((CheckerShader*)aShader)-
      >CheckerPublicData.nbSquareU)
         &&(CheckerPublicData.nbSquareV==((CheckerShader*)aShader)-
      >CheckerPublicData.nbSquareV));
}
```

You can make the type-cast because the shell verifies that the *aShader* pointer is a pointer of the same type.

Remarks and improvements:
In the Checker example, each time the function GetValue is called, two factors (tempu and tempv) are recalculated but don't change if the numbers of squares do not. So it is a good idea to make preprocess calculations. But each time the number of squares changes, you have to recalculate the factors. When the Shell changes a data, it calls **Extension-DataChange d**. To create a shader that allows preprocessing, add some private data (fPreprocessed, and fMul[2] which are boolean and the two preprocessed factors, for example), then change the **ExtensionDataChange d** function like this :

```
HRESULT COMShader::ExtensionDataChanged(THIS) {
  fPreprocessed=FALSE;
  return NOERROR;
  }
```

and the new **GetValue** function will be:

```
HRESULT CheckerShader::GetValue(ShadingIn* theShadingIn, ShadingElem*
      theShadingElem, NUM3D* result) {
   if (!fPreprocessed)
  { fMul[0]=ShortToQuickFix(CheckerPublicData.nbSquareU)/( theShadin-
      gElem-> fShaderBox.fRight-theShadingElem->fShaderBox.fLeft);
    fMul[1]=ShortToQuickFix(CheckerPublicData.nbSquareV)/( theShadin-
      gElem-> fShaderBox.fTop-theShadingElem->fShaderBox.fBottom);
    fPreprocessed=TRUE;
  }
  QuickWide nu = QuickMulWide(theShadingIn->fUV[0]-theShadingElem-
      >fShaderBox.fLeft, fMul[0]);
  QuickWide nv = QuickMulWide(theShadingIn->fUV[1]-theShadingElem-
      >fShaderBox.fBottom, fMul[1]);
  if (!((QWFloor(nu)+QWFloor(nv)) & 0x00000001)) // Even Number ?
  { *result = ShortToQuickFix(0);
  } else
  { *result = ShortToQuickFix(1);
  }
  return NOERROR;
}
```

Chapter 13 - Writing a Tree Behavior

Family ID : ‘treb’
COM Interface ID : IID_I3DExTreeBehavior
COM Interface file : ‘I3DExTbh.h’

The **I3DExTreeBehavio r** Interface is used to add any kind of additional behavior to a Tree Element. This is a little like multiple inheritance in C++: new features are added to an object.

In this cookbook, we will made a behavior that align three objects, by moving the tree element on which the behavior is applied, on the line made by the two other objects.

1. Our behavior have to know the two other object, so we have to know the names of the objects to find them in the scene. And to set the position of the tree element, we have to know a relative position. So we will have a data structure shared with the 3D shell like this:

```
typedef struct BehaviorData {
  char fNameObject1[256]; // name of the first object to align to
  char fNameObject2[256]; // name of the second object to align to
  NUM3D fRelPos;         // relative position between the two objects (0
      first, 1 second)
  } BehaviorData;
```

2. There is only one function for a Tree Behavior called **Apply**. This function is called when the TimeLine changes or then the scene changes. Even if you only have one tree element as a parameter, you can get all the scene and all the other element. That’s how we can get the two other objects to perform the alignment:

```
HRESULT Behavior::Apply(THIS_ I3DShTreeElement* tree) {
  TREETRANSFORM3D tr1;
  TREETRANSFORM3D tr2;
  TREETRANSFORM3D tr;
  VECTOR3D hp1;
  VECTOR3D hp2;
  VECTOR3D hp;
  I3DShTreeElement *tree1,*tree2;
  I3DShScene *scene;
  // Get the scene
  scene = tree->GetScene();
  if (!scene) return NOERROR; // abort any modifications if you can not
      get the scene

  // Search tree element 1 by name
  tree1 = scene->GetTreeElementByName(fData.fNameObject1);
  if (!tree1) return NOERROR; // abort because object 1 not found
  tree1->GetGlobalTransform8(&tr1);

  // Search tree element 2 by name
```

```
tree2 = scene->GetTreeElementByName(fData.fNameObject2);
if (!tree2) return NOERROR; // abort because object 2 not found
tree2->GetGlobalTransform8(&tr2);

// Set the new position of tree
tree->GetGlobalTransform8(&tr);
tr.fT = tr1.fT + (tr2.fT - tr1.fT) * fData.fRelPos ;
tree->SetGlobalTransform8(&tr);
return NOERROR;
}
```

Chapter 14 - Writing a Tweener

Family ID : 'twee'
COM Interface ID : IID_I3DExTweener
COM Interface file : 'I3DExTwn.h'

A Tweener is used to interpolate between two keyframes. When the shell has two keyframes and wants to calculate an image between those keyframes, it calls the Tweener to get a value, and it uses this value to interpolate linearly the keyframes. For example, if the Tweener returns 0.0, the resulting keyframe will be the first one, if it is 1.0, it will be the last one. You can use every value between 0.0 and 1.0 to get a frame that will be between the keyframes or use other values to get a frame outside the segment defined by the 2 keyframes.

In this cookbook, we will describe how to make an oscillator which slows down. The physical formula of that kind of oscillator is :

$$f(t) = \cos(\frac{t}{T}) \cdot \exp(-r \cdot t)$$

Where T is a pseudo-period and r, a factor that express the slow down.
To have the first frame when the animation begins, we have to return 0.0 and to have the last frame at the end we have to return 1.0. These conditions will be met if we use the following formulae :

$$f(t) = 1 - \cos(a \cdot t) \cdot \exp(-r \cdot t)$$

$$\text{with } a = 2 \ (NbOscillations + \frac{1}{4})$$

To create a Tweener, you only have to implement one specific function: **DoTweening**.

1. As you see in the formulae, we only have to use two parameters, the *r* coefficient and the number of oscillations between the two frames. So we will have a data structure shared with the shell like this :

```
typedef struct TweenerData {
  short fNbOsc;
  NUM3D fExpCoef;
  } TweenerData;
```

2. To avoid calculating *a* each time the Shell wants a value, you can preprocess it in the function **ExtensionDataChange d**:

```
HRESULT Tweener::ExtensionDataChanged(THIS) {
  fCosCoef = 2*M_PI*(fData.fNbOsc+0.25);
  QuickFixToDouble(fData.fExpCoef,fExpCoef);
  return NOERROR;
```

```
}
```

3. Now, to implement the main function of the Tweener, we have to return the result in *lambda* and we have three long to calculate the relative time *t* :

```
HRESULT Tweener::DoTweening(THIS_ NUM3D* lambda, long time, long time1,
      long time2) {
  long delta = time2 - time1;
  float t, value;
  if (delta==0) {
    *lambda=0;
    return NOERROR;
    }
  t = (1.0*time - time1) / delta;

  value = (1-cos(t*fCosCoef)*exp(-t*fExpCoef));

  *lambda = DoubleToQuickFix(value);

  return NOERROR;
  }
```

Chapter 15 - Writing a 3D Export Filter

Family ID : ‘3Dou’
COM Interface ID : IID_I3DExExportFilter
COM Interface file : ‘I3DExIO.h’

Writing a 3D Export Filter is the right strategy when one wants to get 3D Data out of Ray Dream Designer. It is a much easier choice than trying to read the Ray Dream Designer file format.

The toolkit example is a DXF exporter. It is based on COM. It just involves replacing the calls to QueryInterface() by the appropriate type-casting.

1. In this very simple example, we do not have any user interface, so I3DExExportFilter::Prepare() does not do anything, and I3DExExportFilter::WantsOptionDialog() returns FALSE.

2. In this exporter, the basic algorithm is this:

```
Create the output file                          Instanciate a IShFileStream
For each object instance in the scene:          I3DShScene::GetInstanceByIndex()
  Get the 3D object referenced by the instance I3DShInstance::Get3DObject()
    If it is a Primitive                        QueryInterface(IID_I3DShPrimitive)
      Get its global Transformation             I3DShTreeElement::GetGlobalTransform()
      If the Primitive if Patch based           I3DShPrimitive::IsPatchBased()
        Convert the patches to facets           ConvertPath2Facets()
        Transform the facets in Global coordinates
        Write the facets to the output file
      Else if the Primitive is Facets based
        Transform the facets in Global coordinates
        Write the facets to the output file
```

The code contains details that have to do with the DXF file format. These details are not very exciting to cover here.

The main point in the exporter example is the heavy use of the QueryInterface() call. Make sure you are familiar with the I3DShObject, I3DShPrimitive, I3DShTreeElement and I3DShInstance interfaces.

3. Make sure you build the right ‘Cmpp’ (Component Private) resource. It contains the information necessary to the 3D Shell to display your file format extension and name in the Save As dialog.

Please refer to the “The Component Private resources (“Cmpp”)” section in the “Managing the User Interface” Appendix for all details.

Chapter 16 - Writing a 3D Import Filter

Family ID : ‘3Din’
COM Interface ID : IID_I3DExImportFilter
COM Interface file : ‘I3DExIO.h’

Writing a 3D Import Filter is the right strategy when one wants to import 3D Data in Ray Dream Designer. It is a much easier (and compatible) choice than trying to create a file with the Ray Dream Designer file format.

The toolkit example is a facets importer, based on COM. The file format is text based and the extension on PC is .eas for “Easy”. This file can be created with any text editor.

An easy file is a set of surfaces display as followed:

number of points
first point
second point
...
last point

After the last surface, a zero must be add to specify the end of the file, followed by a space or a carriage return to prevent reading failure. For example, a square will be defined as follow:

```
4     number of points
0 0 0 first point (x, y, and z)
0 1 0 second point
1 1 0 third point
1 0 0 last point

0     no more surfaces, don’t forget to add a space or a carriage
   return
```

We will focus here in the steps involved to create a 3D object and insert it in the scene.

1. Find where you need to insert your 3D data

At first, you get a pointer on the scene and a pointer on the Tree Element under which to insert your data. Often, this Tree Element will be NULL. In this case, you need to insert under the Scene Tree Root. It is a sensible thing to call I3DShScene::CreateTreeRoo-tIfNone() to make sure there is one, just in case.

In the following code example, we choose to create a group if *fatherTree* is provided. This is a choice we make. The variable *topTree* is where we shall insert our data in the end:

```
HRESULT TEasyImporter::DoImport(THIS_ char* fullPathName, I3DShScene*
      scene, I3DShTreeElement* fatherTree)  {

 I3DShTreeElement* topTree;// Where we shall put everything
 .......
```

```
if (fatherTree == NULL)  {
    I3DShGroup*   topGroup;
    scene->CreateTreeRootIfNone();
    topGroup = scene->GetTreeRoot();
    topGroup->QueryInterface(IID_I3DShTreeElement, (LPVOID*) &top-
    Tree);
    topGroup->Release();
    }
else  {
    gShellUtilities->CoCreateInstance(CLSID_StandardGroup, NULL,
    CLSCTX_INPROC_SERVER, IID_I3DShTreeElement, (LPVOID*) &topTree);
    topTree->SetScene(scene);
    fatherTree->InsertLast(topTree);
    }
    scene->CreateRenderingCameraIfNone(IDTYPE('c', 'o', 'n', 'i'),
    (fatherTree == NULL));// Create a conical rendering camera if
    none, and a Distant light
       if we not importing in an existing scene

    DoReadEasyFile(stream, scene, topTree); // here we import the
    object
    ...
```

2. Create a default Rendering Camera and Light Source

Because the Easy format does not have the notion of cameras and light sources, we shall use a built-in API call designed just for that:

```
scene->CreateRenderingCameraIfNone(IDTYPE('c', 'o', 'n', 'I'), (father-
    Tree == NULL));
```

This will create a conical camera (we use the Conical Camera Class ID ‘coni’), and a default Distant Light if we are not importing in an existing scene (in this case, the second parameter is TRUE).

Of course, if your own requirements are more sophisticated, you can create any type of standard camera and light sources, and place them where you want in 3D.

3. Create your objects and put them in the scene:

3.1. Create the appropriate Primitive. Depending on your needs, you will create a Polygon List if you just have raw list of triangles, a Polygon Array if your have more ordered data, or even a Patch Array if you have high-level surfaces. Polygon and Patch Arrays are the best choices because they provide a natural way to calculate Normals and Texture Coordinates (UV Space).
In our case, we just need to use Polygon Lists without normals. The API provides a call to calculate Normals. In general, it is slow and memory consuming, and it will never give a perfect result, especially if your 3D Data is not perfectly accurate. But the easy format is quite simple, it is just a set of triangles.
If you can also provide Texture Coordinates, then this is really great for the user. Not providing UV values will force the user to choose a Projection Mapping mode (Spherical,

Cylindrical or Planar) that may not fit well your object (not mentioning the fact that Projection Mapping is not easy to understand).

```
I3DShPolygonList* surface;

//-- Create the primitive:
gShellUtilities->CoCreateInstance(CLSID_StandardPolygonList, NULL,
      CLSCTX_INPROC_SERVER, IID_I3DShPolygonList, (LPVOID*) &sur-
face);
surface->Init(FALSE /*no normals*/, FALSE /*no UV space*/);
```

3.2. Fill in your Primitive with your 3D data.

```
surface->PreAllocateFacets(nbPoints - 2/*nb of facets*/);
FailOSErr(ReadStreamVector3D(stream, &firstPoint));// read the
first point
FailOSErr(ReadStreamVector3D(stream, &lastPoint));// read the sec-
ond point

for (i=2; i<nbPoints; i++) {
  secondPoint[0]=lastPoint[0];
  secondPoint[1]=lastPoint[1];
  secondPoint[2]=lastPoint[2];
  FailOSErr(ReadStreamVector3D(stream, &lastPoint));
  // add a facet with the first point, the previous point and the
current point
  surface->AddFacet (SetEasyFacet(firstPoint, secondPoint, last-
Point));
  }
surface->CalcNormals(ShortToQuickFix(30));// angles under 30
degrees are
      smooth
surface->QueryInterface(IID_I3DShObject, (LPVOID*) &object);
surface->Release();
COLOR3DdefColor;
defColor.Mode=0;
defColor.R=kQuickFixOne;
defColor.G=kQuickFixZero;
defColor.B=kQuickFixZero;
defColor.A=kQuickFixOne;
object->SetSimpleShading(&defColor, kQuickFixOne, kQuickFixOne,
kQuickFixZero, kQuickFixZero);
```

3.3. Give it a **unique** name. This is important because Primitives are referenced by their name in Ray Dream Designer files.

```
sprintf(objName,"Easy %i",++counter);
object->SetName(objName);
```

3.4. Create an Instance, and hook it to your Primitive.

```
gShellUtilities->CoCreateInstance(CLSID_StandardInstance, NULL,
CLSCTX_INPROC_SERVER, IID_I3DShInstance, (LPVOID*) &instance);
```

```
instance->Set3DObject(object);
instance->QueryInterface(IID_I3DShTreeElement, (LPVOID*) &instan-
ceTree);
instanceTree->SetScene(scene);
instanceTree->CenterHotPointOnElement();
instance->Release();
```

3.5Put the Instance in the tree and calculate its 3D orientation and positioning.

```
topTree->InsertLast(instanceTree);
```

The main point in the exporter example is the heavy use of the QueryInterface() call. Make sure you are familiar with the I3DShObject, I3DShPrimitive, I3DShTreeElement and I3DShInstance interfaces.

4. Make sure you build the right ‘Cmpp’ (Component Private) resource. It contains the information necessary to the 3D Shell to display your file format extension and name in the Open and Import dialogs.

Please refer to the “The Component Private resources (“Cmpp”)” section in the “Managing the User Interface” Appendix for all details.

Chapter 17 - Writing a Post Render Filter

Family ID : ‘post’
COM Interface ID : IID_ I3DExPostRenderer
COM Interface file : ‘I3DExPos.h’

A Post Render Filter is a good way to add an effect on the final image. The toolkit example is a sandy filter that convert the result image in eight colors.

Two methods must be implemented: Filter and GetBufferNeeds. GetBufferNeeds is used to set the informations we will need during the filtering.

```
enum {kDistance=1, kPositionX=2, kPositionY=4, kPositionZ=8,
         kNormalX=0x10, kNormalY=0x20, kNormalZ=0x40,
         kAlpha=0x80, kIndex=0x100, kSurfaceU=0x200, kSurfaceV=0x400,
   kShaderColor=0x800,kShaderSpecularColor=0x1000,
kShaderSpecularSizeValue=0x2000,kShaderAmbientValue=0x4000,
kShaderLambertValue=0x8000,kShaderReflectionColor=0x10000, kShaderTran-
      parencyColor=0x20000,kShaderRefractionValue=0x40000,
         kShaderGlowColor=0x80000};    // For flags

void Sand::GetBufferNeeds (THIS_ /*RenderFilterNeeds&*/long* needs, /
      *TExternalRenderer*/void* renderer) {
   *needs = 0;
   }
```

This method is called before the rendering. The application prepare a buffer to stock during the rendering the informations you are asking for. For example, if you need to know which object is seen at a given point and where is this point on his UV space, you just need to set *needs* to:

```
*needs = kIndex | kSurfaceU | kSurfaceV;
```

In this example, we don’t need any of those informations, so, we just set *needs* to 0.

To process the filtering, we need to get access to the Offscreen of the image. The Offscreen is composed of chunks that are square pieces of the Offscreen used to store partially the picture on disk. In this example, we also need to create an Offscreen to draw a check on screen to show the progress of the filtering.

The shell provides the scene, so we can get informations on the objects, the image resulting from the rendering, a buffers which contain all the informations you asked for. It also gives you access to the graphic device, useful to draw the check. The renderer is not yet available.

```
void Sand::Filter (THIS_ I3DShScene* theScene, IShRasterOffscreen*
theImage, IUnknown** theBuffers, IShGraphicDevice* gd,
TExternalRenderer*/void* renderer, void* renderHelper) {
```

First of all, we will create the check to be drawn on screen to display the progress. This check fit exactly a chunk and will be displayed before each chunk to be computed.

```
ULONG depth;
RECT3D BRect;
BRect.top = 0;
BRect.left = 0;
BRect.right = 32000;
BRect.bottom = 32000;

BufferChunk* aChunk=0;
RECT3D* aClip=new RECT3D;
RECT3D* cursorClip=new RECT3D;
ULONG chunkH, chunkV, rowBytes;

// getting the dimensions of a chunk
theImage->GetChunkInfo(&chunkH, &chunkV, &rowBytes, &depth);
if (depth == 32) { // we need colors in 32 bits
   // creating the cursor
   IShRasterOffscreen* theCursor;
   sandShellUtilities->CoCreateIn-
   stance(CLSID_StandardRasterOffscreen,0,0,
            IID_IShRasterOffscreen,(void**)&theCursor);
   theCursor->InitOffscreen(chunkH, chunkV, depth);
   // we will get the first chunk that is the only one chunk
   IEnumChunk* FirstChunk = theCursor->EnumChunks(&BRect);
   FirstChunk->Next(1,&aChunk,0);
   theCursor->GetChunkRect(aChunk, cursorClip);
   theCursor->LockChunk(aChunk);
   long* PCurChunk;
   // now, we will fill the chunk with a black and yellow checker
   PCurChunk = (long*)(theCursor->GetChunkData(aChunk));
   short i,j;
   for (i=0; i< chunkV; i++) {
      for (j=0; j< chunkH; j++) {
         *(PCurChunk++)=(((i^j)&8)==0?0x0000000:0x0FFFF00); // square
   black or
                  yellow each 8 pixels
         }
      }
   theCursor->UnlockChunk(aChunk);
```

Ones the checker have been created with the size of a chunk, we will iterate on each chunk of the image to process the filtering.

```
   IEnumChunk* iter = theImage->EnumChunks(&BRect);
   for (; (iter->Next(1,&aChunk,0)) == NO_ERROR; ) {
      theImage->GetChunkRect(aChunk, aClip);
      gd->DrawOffscreen(theCursor, *cursorClip, *aClip); // draw the
   checker
      theImage->LockChunk(aChunk);
      long* data = (long*)theImage->GetChunkData(aChunk);
      // iterate on each pixel of the chunk
      for (short vv = aClip->top; vv< aClip->top + chunkV; vv++) {
         theImage->SetPosition(aClip->left, vv);
         for (short hh = aClip->left; hh< aClip->left + chunkH; hh++)
   {
```

```
                long color;
                // get the color
                theImage->Get(&color);
                // transform this color
                (*data) = RandomColor(color & 0xFF) + (Random-
    Color((color>>8) & 0xFF)<<8) +
                                RandomColor((color>>16) & 0xFF)<<16);
                // move to the next pixel
                theImage->GoRight();
                data++;
                } // for hh
             } // for vv
          theImage->UnlockChunk(aChunk);
          gd->DrawOffscreen(theImage, *aClip, *aClip); // draw the com-
    puted chunk
          } // for iter
    theCursor->Release();
    } // if depth
}
```

The most important function here is Get that gives you the information you need for the current position on the offscreen. theBuffers is a pointer on a table that contain the buffers for each informations you asked for. The first one is the color buffer, always available. This call returns the color of the current pixel:

```
theImage->Get(&color);  // color (0x00RRGGBB)
```

To get the others informations, we need to get an interface on I3DShRasterBufferGetPut that allows you to “navigate” thru the buffer like an offscreen.

```
I3DShRasterBufferGetPut* theDistance;
theBuffers[0]->QueryInterface(IID_ISHRasterBufferGetPut, theDistance);
...
theDistance->Get(&distance); // distance
```

Now, you’ve got to iterate on each chunk of your buffer to get the distance on each point as we did for the color. You need to keep iterating on the chunks of the image to set the new colors. Same way to get the others informations:

```
theBuffers[0]     // distance
theBuffers[1]     // position X
theBuffers[2]     // position Y
theBuffers[3]     // position Z
theBuffers[4]     // normal X
theBuffers[5]     // normal Y
theBuffers[6]     // normal Z
theBuffers[7]     // alpha
theBuffers[8]     // index
theBuffers[9]     // surface U
theBuffers[10]    // surface V
theBuffers[11]    // shader color
theBuffers[12]    // shader specular color
theBuffers[13]    // shader specular size value
```

```
theBuffers[14]    // shader ambient value
theBuffers[15]    // shader lambert value
theBuffers[16]    // shader reflection color
theBuffers[17]    // shader transparency color
theBuffers[18]    // shader refraction value
theBuffers[19]    // shader glow color
```

Alpha is the mask, 0 mean background, MaxLong mean object and intermediate values for the frontier. Index, surface U and surface V are used to locate the point in UV spaces.

17.1.Writing a Renderer

The API for this section does exist, but is not documented yet. The documentation will come in a future version of the DreamSDK. If you need more information, please contact MetaCreations directly.

Chapter 18 - DataBase Overview

Ray Dream Designer stores all the data of currently open documents in a global database. Third party extensions can access this database to get these or modify these data. The database contains a list of scenes. To each open document corresponds one scene.

18.1.Scene

A *Scene* is mostly comprised of 2 structures:

- An *Object List*, that contains all the *Objects* used in the scene.
- A *Scene Tree* that contains the hierarchical structure of the scene. This defines the positions and relationships between *Tree Elements*. These tree elements can be object instances, light sources, cameras and groups.

Objects

Objects contains the geometric information defining its surface regardless of its position. It also contains a reference to its shading information. The Objects are located in the Objects Browser window in Ray Dream Designer.

An Object can be of two types: a *Primitive* or a Scene.

Primitives are the basic objects that are located in the « Basics » section of the Objects Browser. There are different classes of primitives. Ray Dream Designer defines a certain number of default internal primitives that can be used by extensions. Those primitives will always be there. They are the Free Form Primitive, the Polygon Array, the Patch Array, the Polygon List (list of triangular facets), the Flat Primitive, the 3D Text, etc. Some other primitives are external primitives (coded in regular extensions) like the Cube, the Sphere, the Isocaedra, etc. Third party developer can define new classes of primitives.

A Scene is also considered an Object, because it can be instanciated in another Scene, like if it was a complex object. This is sometimes called « Scene Instancing ».

Scene Tree and Tree Elements

The scene tree is a hierarchical structure containing *Tree Elements*.
There are four types of tree elements: Object Instances, Light Sources, Cameras and Groups. Those elements are detailed below. All these elements have 2 things in common:

- a positioning in 3D space (orientation, location, scaling, etc...)
- a name

Object Instances

An object instance is a tree element that has a reference to an Object in the object list. Therefore there can be multiple instances in a scene of a same object. This object can be either a Primitive or another Scene.
Shading can be « overriden » at the instance level, thus surpassing any shading information the original object may have.

Lights

A *Light Source* is a tree element. There can be as many lights as needed in a scene. Most

Light Sources are implemented as Extensions.

Cameras

A *Camera* is another type of tree element. Like lights, there can be many cameras in a scene. One of them will be the *Rendering Camera*, i.e. the camera the final image will be rendered from. Most Cameras are implemented as Extensions.

Groups

A *Group* is a tree element used to gather other tree elements together. It can be open or closed.

18.2.Coordinate System s

Global Coordinate Syste m

When you look at the Perspective window in Ray Dream Designer, the axis of the Global Coordinate System are organized like this:

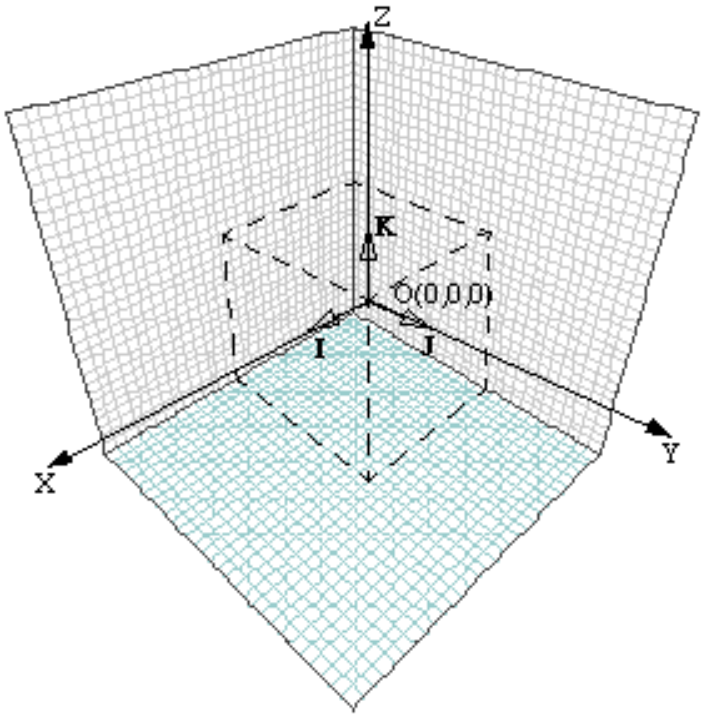


Figure 1. The Global Coordinate System

The projection of the (0, 0, 0) origin falls in the middle of each plane of the Working Box (the origin is *not* the far corner of the box). The **I**, **J**, **K** vectors are the unit vectors of the X, Y and Z axis.

Working Box Coordinate Syste m

The Working Box Coordinate System is defined by the position of the Working Box. As the Working Box can be moved and rotated in Ray Dream Designer, this coordinate system can be useful in some complex scenes. However, this system is never used when dealing with Extensions, so we will just mention it here.

Local Coordinate System (or Object Coordinate System)

The Local Coordinate System (also sometimes called the Object Coordinate System) define the system of an object or a group. Its axis, origin and scale depends on the positioning of the object in space.

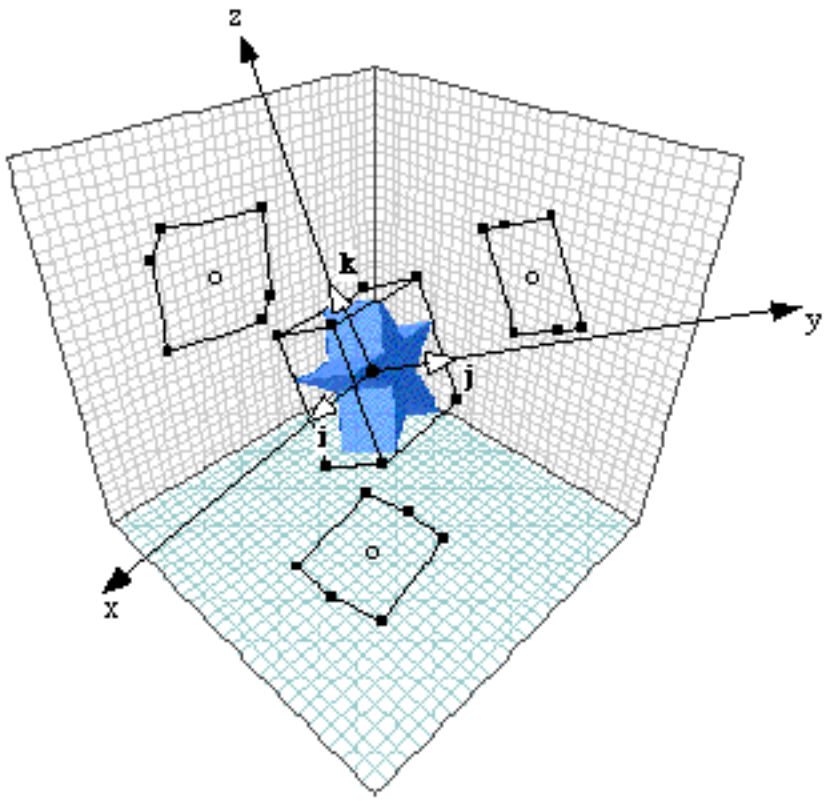


Figure 2. The Local Coordinate System of an object in a scene

The **i**, **j** and **k** vectors are the unit vectors of the x, y and z axis.

Screen Coordinate Syste m

When a rendering occurs, the 3D data is projected onto the screen through the rendering camera. The axis of the 3D coordinate system attached to the screen is like this:

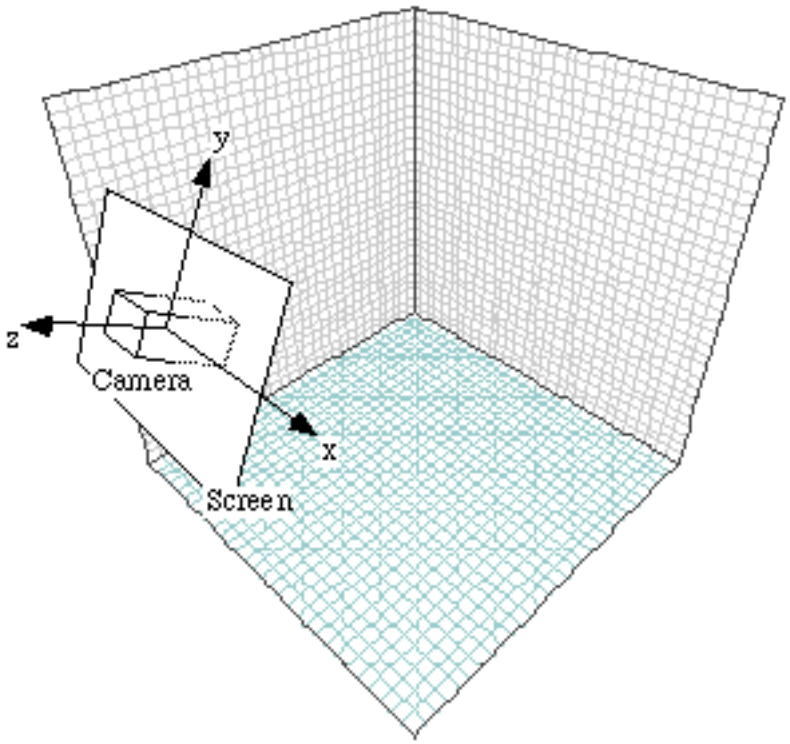


Figure 3. The Screen Coordinate System

As one can see, the objects seen by the camera have a negative z coordinate. The screen and the Screen Coordinate System are centered on the camera's center.

The Screen Coordinate System versus the Camera's Coordinate System

There is a slight difficulty here. The camera's aim is along the y axis of its transformation. As a result, here is the Local Coordinate System of the camera in the previous figure:

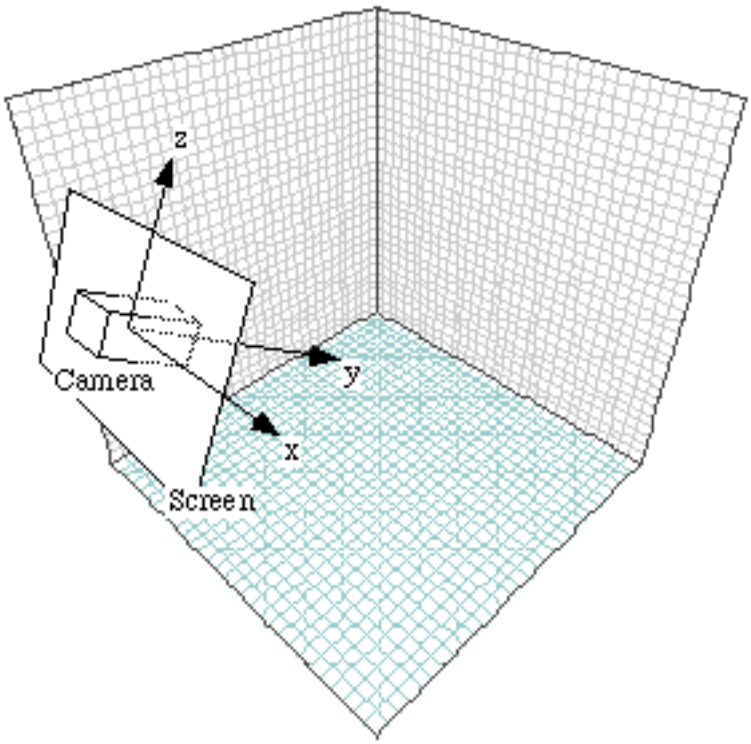


Figure 4. The camera's Local Coordinate System

You do not need to worry too much about this difference, because transformations calculated by the 3D Shell takes this into account. It is important only if you intend to position a camera in a scene.

The screen pixels space

The Screen Pixels Space is the actual Pixels coordinate system used to render the final image. Its unit system is in Points, as opposed to all other coordinate systems which are in 3D units. As a result :
1 3D unit = 288 points
because 1 inch = 72 points. More on the units business is covered later.
It is oriented with its vertical axis going down, and its (0, 0) origin in the top left corner of the image:

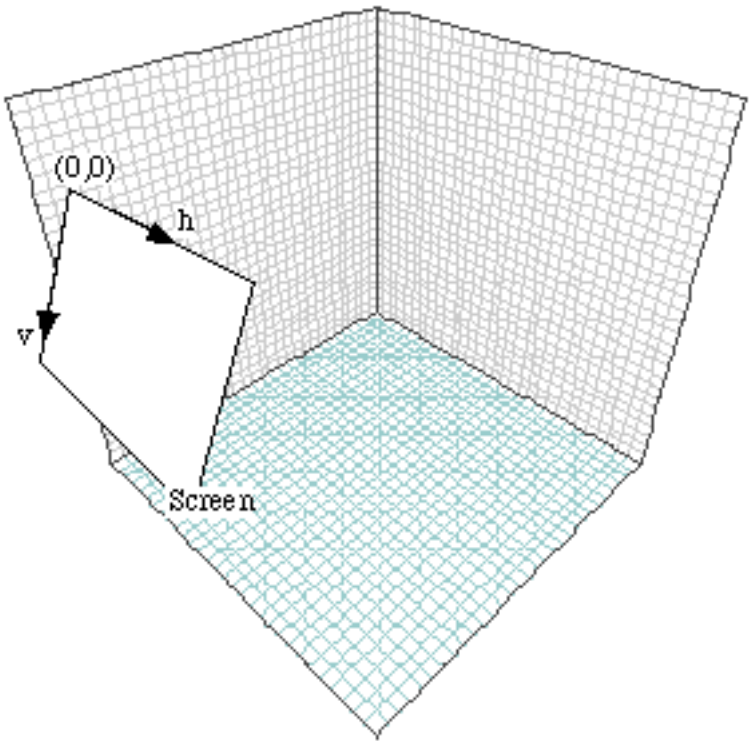
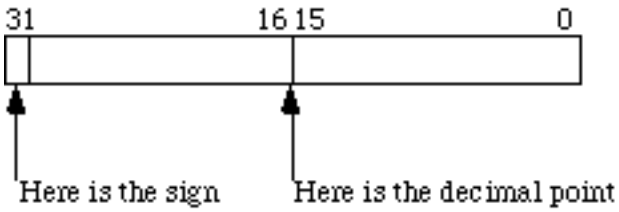


Figure 5. The Screen Pixels Space

18.3.Geometry

Geometric data type: 32-bit fixed poin t

All geometric data is in 32-bit fixed point format. This is similar to the `FIXED` data type in Windows, and identical to the `Fixed` data type on the Macintosh. As far as the C compiler is concerned, it is a `long`. The type is called `NUM3D` in the Dream SDK.



Examples of `NUM3D` values:

1.0	`0x00010000`
0.5	`0x00008000`
-1.0	`0xFFFF0000`
-1.5	`0xFFFE8000`

Range is limited to ± 32767.99998, accuracy is limited to 1.5259E-5 (=1/65536).

To convert from a NUM3D to a double:

```
double Num3DToDouble(NUM3D f) {
      return (f / 65536.0);
}
```

To convert from a double to a NUM3D:

```
FIXED DoubleToNUM3D(double d) {
      return (long) (d * 65536.0);
}
```

If you use C++, it is highly recommended to use the QuickMath library provided in the development toolkit. It defines a C++ class called `QuickFix` that matches the `NUM3D` type, and that contains all kind of optimized operators and functions. You will save a lot of time and efforts by using this library, and your code will be easier to read. See the development toolkit notes for details on how to use this library on your specific platform.

Units System

The units system used in Ray Dream Designer is defined as follows:

```
1 3D unit = 4 inches
```

All geometric data uses the 3D units. The various units shown by Ray Dream Designer in

the Geometry palette are just handled at the user interface level.

The advantage of using a fixed system like this is that there is no problem of data conversion into different units. As all geometric data is stored in 32 bit fixed-point format, this convention also gives the best range of values for typical scenes built by users: the maximum is ± 3.3 kilometers and the minimum is 1.5 µm. Of course, those are the maximum and the minimum of the accuracy you can get theoretically, but be careful not to overflow those values when making calculations. Because of this, it is reasonable to limit yourself to values well within this range. Try not going further than 0.5 kilometers, or you will overflow quickly. Likewise, try not going below 1 millimeter, or you will lose accuracy.

Tree Elements Transformatio n

Each tree element (object instance, light source, group or camera) is defined relatively to its parent in the tree. Depending of the API procedure you call, you will get the transformation parameters that define the attitude of the object in space in one format or another. Whatever callback you use, you will get the following data:

- A 3x3 rotation matrix **R** (or the 3 vectors that define it)
- A translation vector **T**
- A uniform scaling factor **s**

To transform a point from the Local Coordinate System of this tree element into the Coordinate System of its parent, the following formula is used:

$$M = s[R]m + T$$

with m the point in local coordinates, and M the same point in the parent's coordinates.

To make it easier to read, here is the same equation in expanded format for each x, y and z coordinate:

$$M_x = s(R_{ix} \times m_x + R_{jx} \times m_y + R_{kx} \times m_z) + T_x$$

$$M_y = s(R_{iy} \times m_x + R_{jy} \times m_y + R_{ky} \times m_z) + T_y$$

$$M_z = s(R_{iz} \times m_x + R_{jz} \times m_y + R_{kz} \times m_z) + T_z$$

With :

- $m(m_x, m_y, m_z)$ point in local coordinates
- $M(M_x, M_y, M_z)$ point in parent's coordinates
- $T(T_x, T_y, T_z)$ position of the tree element origin in parent's coordinates
- s uniform scaling factor
- R rotation matrix (see below)

More on the rotation matrix:

If you consider the 3 **i**(i_x, i_y, i_z), **j**(j_x, j_y, j_z) and **k**(k_x, k_y, k_z) vectors of the Local Coordinates System, you can write easily the **R** matrix by putting each vector in each column like this:

$$[R]=\begin{bmatrix} i_x & j_x & k_x \\ i_y & j_y & k_y \\ i_z & j_z & k_z \end{bmatrix}$$

For additional information on transformation matrices and Tree Transformations, see the Data Structure Reference chapter.

Geometry basics

When you deal with objects, there are several concepts to define that are related to the object's shape.

Surface Point

Ray Dream Designer deals with surfaces, not volumes. An object is made of surfaces that can be arbitrary complex. A point P on a surface is made of its x, y and z coordinates in the Object Coordinate System.

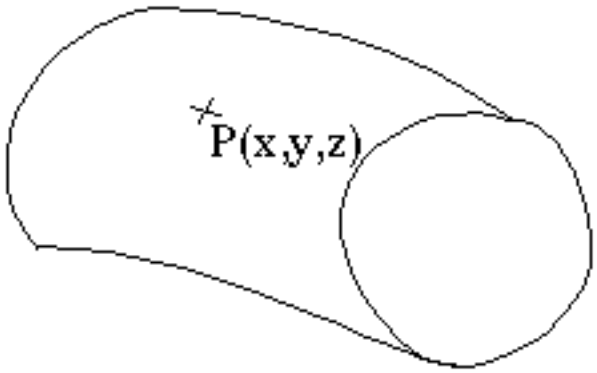


Figure 6. A point on a surface

Surface Normal

The Surface Normal at a point is the vector perpendicular to a plan that would be tangent to the surface at that point. The Normal is very important in 3D computer graphics, because it is heavily used for shading (especially in Phong and Ray Tracing shadings). Normals are usually normalized (their length is equal to 1), and always point outward.

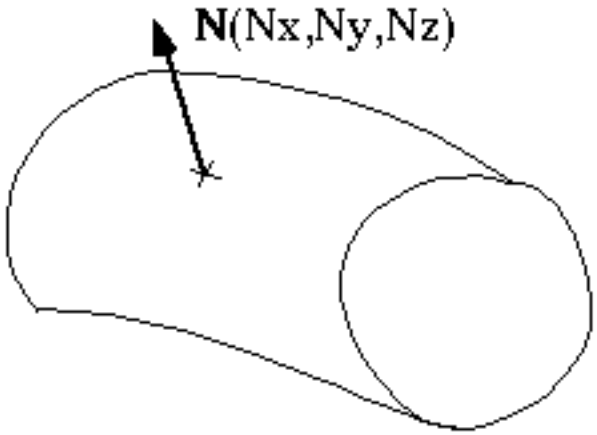


Figure 7. A Normal to the surface

Note that :

$$N_x^2 + N_y^2 + N_z^2 = 1$$

u,v Space (Texture Coordinate System)

Wrapping a texture (such as a texture map) on an object surface involves finding the correspondence (the "mapping") between a 2D space (the image) and a 3D space (the object surface). The typical rendering question is: "I have a point P(x, y, z) on my surface, where should I look up in the texture map ?".
When one deals with simple shapes (sphere, cube, cylinders, etc.), it is easy to find a mapping. When one deals with complex objects (list of facets or patches), the matter becomes much more difficult. To solve this, most packages use a technique called "projection mapping": an intermediate imaginary surface surrounding the object is used, and the texture is placed on it. Then the surface color at a point is calculated by using the part of the texture that the point is facing.
The problem with this is that you can get awkward results when the object is very unlike the intermediate surface. Unwanted deformation is a typical problem: you want the texture to shrink or enlarge only where the object surface does.

Parametric mapping - a better solution

The Ray Dream Designer free-form modeler is in fact capable of generating what we call "UV spaces", and the architecture of Ray Dream Designer can keep this information down to the facets or bicubic patches level, in order to allow direct mapping (also called "Parametric Mapping").
The idea is this: because objects are built by combining 2D curves, it is possible to generate a 2D space that will behave topologically like the object surface.

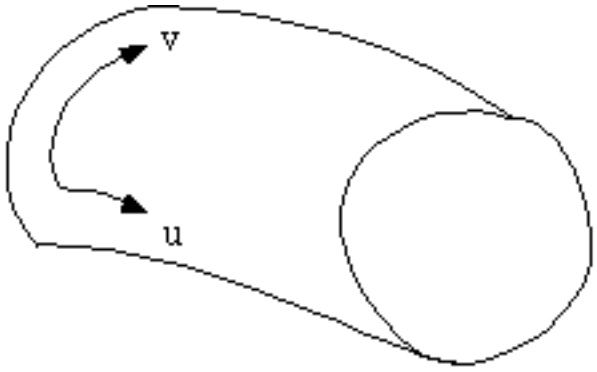


Figure 8. A typical u,v space on the side surface of an object

Every time you deal with a 3D point on the surface, you will be able to calculate the u,v coordinates for this point (provided that the object supports UV spacing). Say you need to know if the point is covered by a Paint Shape. The location of the paint shape is known by the u,v coordinates of its corners. The test becomes a simple 2D test: "are the u,v coordinates of the point inside a 2D rectangle ?".

Most of the time you get the u,v values from the 3D Shell. The only case where you have to generate them is when you develop a geometric primitive extension.

If the object has some surface discontinuities, then several u,v spaces are generated for the object. For example if your extrude a closed 2D curve, you will get 3 u,v spaces: one for the side surface, one for the back face and one for the front face. This is because it is not possible to have a u,v continuity across these different parts of the object (think of the problem of a napkin on a table). Each UV space has a number (0, 1, 2...), called a **UV Space ID**.

Shading

The shading can be applied on an object by giving general properties (color, reflection...) to the object or by mapping a texture on it. There is 5 ways of mapping a texture on an object.

- the parametric mapping,
- the box mapping,
- the cylindrical mapping,
- the spherical mapping,
- the pass thru mapping.

The parametric mapping is the mapping using the UV Space information as seen in the last section. This is the better way of mapping, but because all objects don’t have necessarily a UV Space, four other ways of mapping are provided.

The box mapping, the cylindrical mapping and the sphere mapping are obtained by projecting a texture on a cube, cylinder or a cube on the object.

The pass thru mapping is obtained by vertically projecting a planar texture on the object.

Facets

There are two kinds of low level geometric data used for rendering, exporting, etc.: **facets** and **bicubic patches**.

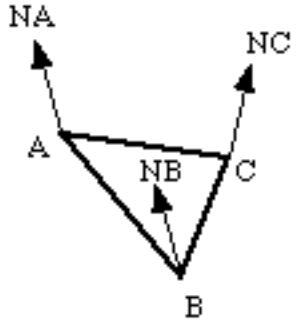


Figure 9. A 3D facet

Facets are triangles. Each vertex contains the (x, y, z) coordinates of the point, the Normal at this point, and the u,v coordinates at this point. The facet also stores the u,v Space ID to which it belongs.

```
typedef struct VERTEX3D {
  VECTOR3D    fVertex;    // x, y, z vertex coordinates
  VECTOR3D    fNormal;    // Nx, Ny, Nz normal values at that vertex
  NUM3D       fu,fv;      // Texture u,v values at that vertex
} VERTEX3D;

typedef struct FACET3D{
  VERTEX3D  fVertices[3];    // The facet three vertices
  short     fUVSpace;        // UV Space ID this facet belongs to
  short     fReserved;       // Reserved - 0
  } FACET3D;
```

Interpolating in a facet

Interpolating a point in a facet is a common exercise in 3D. If you call the vertices A, B and C, their Normals NA, NB, NC, and their u,v values uA, vA, uB, vB, and uC, vC, then:

from:

$$P = \alpha A + \beta B + \gamma C$$

you can interpolate:

$$N = \alpha NA + \beta NB + \gamma NC$$

Don't forget to renormalize the normal:

$$N_{nnit} = \frac{N}{\sqrt{N^2}}$$

Likewise:

$$u = \alpha uA + \beta uB + \gamma uC$$
$$v = \alpha vA + \beta vB + \gamma vC$$

Of course, this is an approximation of the real values, but facets are already approximations of the real surface. As long as facets are small enough, this works just fine.

Bicubic Patches

Bicubic patches are more interesting geometric entities. They can describe more complex surfaces than facets, and are more resolution independent. For example Extrusions created by Ray Dream Designer's Free-Form modeler generate bicubic patches. Turn on the Wireframe display: what you see are the boundaries of the patches.

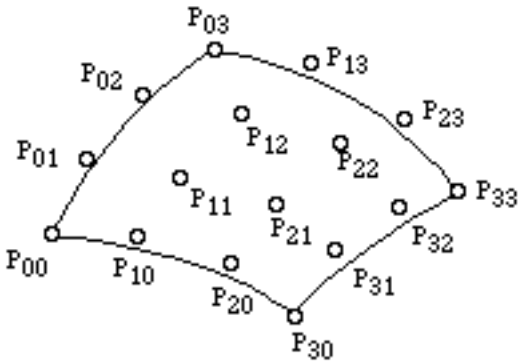


Figure 10. A bicubic patch

A patch is made of 16 3D points. You can think of a bicubic patch as a cubic B_zier curve that has its 4 vertices moving along 4 other B_zier curves.

```
typedef struct PATCH3D{
  VECTOR3D  fVertices[4][4];    // The patch 16 vertices
  NUM3D     fu[2];              // u values at the patch boundaries
  NUM3D     fv[2];              // v values at the patch boundaries
  short     fUVSpace;           // UV Space ID this patch belongs to
  short     fReserved;          // Reserved - 0
  } PATCH3D;
```

One can think of a bicubic Bézier patch as a Bézier curve moving on 4 other perpendicular Bézier curves. Let's apply this concept to calculate a point on the surface.

The patch can be defined as a parametric surface of two normalized parameters, tu and tv.

$$\text{Bicubic Patch} = S(t_u , t_v)$$
$$0.0 \le t_u \le 1.0$$
$$0.0 \le t_v \le 1.0$$

First consider the 4 Bézier curves defined by C0=(P00, P10, P20, P30), C1=(P01, P11, P21, P31), C2=(P02, P12, P22, P32), and C3=(P03, P13, P23, P33), and calculate on each

a point at t_u:

$$P_n(t_u) = P_{0n}(1-t_u)^3 + 3P_{1n}(1-t_u)^2 t_u + 3P_{2n}(1-t_u)t_u^2 + P_{3n}t_u^3$$

Then calculate the point at tv on the Bézier curve (P0(t_u), P1(t_u), P2(t_u), P3(t_u)) by re-using the same formulae as above. This is the result.

Patch normals

A nice thing about patches is that you do not have to worry about storing normals: they are automatically defined by the patch geometry. So this means that you do not have to calculate them for the 3D Shell. For the Mathematics savvy reader, let's remind that normals are defined as:

$$N(t_u, t_v) = \frac{\partial S}{\partial t_u} \times \frac{\partial S}{\partial t_v}$$

Patch u,v Space

The u and v values are constant along the patch boundaries.
P00, P01, P02, P03: u[0] P00, P10, P20, P30: v[0]
P30, P31, P32, P33: u[1] P03, P13, P23, P33: v[1]
(u, v) values at any point on the patch is calculated from these boundaries values by doing a Bézier interpolation.

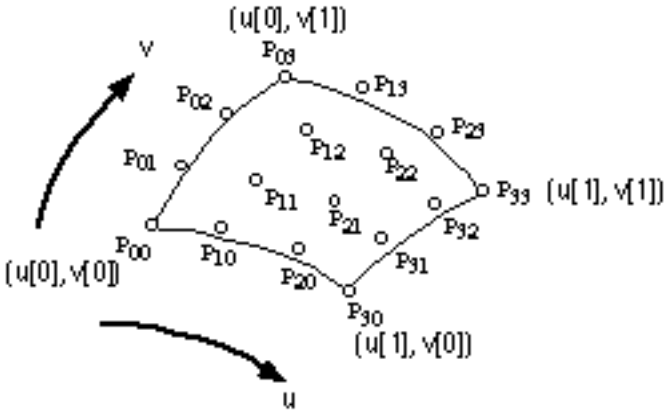


Figure 11. A patch (u,v) Space

To learn more about Bézier bicubic Patches

Read the excellent book « Introduction to Computer Graphics » by Folley - Van Dame - , published by Addison-Weisley.
Also take a look at source of the examples of the Dream SDK, like the Teapot primitive or the patch deformer.

Index

Numerics

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W