The Java 3D API Questions and Answers

Michael F. Deering and Henry A. Sowizral

Index of Question Topics

1. How was the Java 3D API specification developed?
- Process
- Implementation Plans: platforms, dates, performance
- Compliance Testing

2. What technical features does the Java 3D API support?
- Immediate Mode
- View Model
- Tracker Model
- Capability Bits
- Vector Mathematics Library
- Float and Double
- Geometry Compression Format
- Triangles are Universal
- Hi-resolution Coordinates
- Scoping of Lights
- Sound Library
- Double-buffered, True-color, Z-buffered Rendering Model
- VRML
- Support Utilities and Applications
- Extensions and Updates

3. What features are not supported by the Java 3D API?
- Most Methods Final
- User Traversal of the Scene-graph
- Off-screen Rendering and Printing
- Shading Language

4. Who needs the Java 3D API?
- Markets
- Hardware Platforms

How was the Java 3D API specification developed?

Process

Q: What was the process used to design the Java 3D API?

A: Java 3D is part of the Java Media suite of APIs, which in turn is part of the overall Java API efforts. Java 3D is a joint collaboration between Intel, Silicon Graphics, Apple, and Sun. All four companies had advanced retained mode APIs under development and were looking at developing Java versions. These companies decided to pool their efforts to develop a single, compatible API that could truly be cross-platform.

This draft of the Java 3D API specification is a result of that partnership. Prior to this public review period, the specification was made available to Java licensees for review. After the public review period ends, the specification will be finalized. We have continued to refine the specification throughout this process and hope that few major decisions will need to be revisited. One of the purposes of this Q&A document is to point out some of the choices made by the Java 3D team and explain why they were made.

We encourage readers to send in comments on the Java 3D API specification through June 27, 1997. These comments will be reviewed for possible inclusion into future releases of the specification. We anticipate that the final Java 3D specification will be frozen and released by the end of the third quarter, 1997.

Implementation Plans

Q: What are the implementation plans for Java 3D?

A: JavaSoft will release implementations of Java 3D for JavaOS, MacOS, UNIX, and Windows. The initial reference implementations of the Java 3D API will be layered on top of existing lower-level immediate-mode 3D rendering APIs, specifically OpenGL, Direct3D, and QuickDraw3D. The initial Java 3D implementations will be written mostly in Java but will also take advantage of native methods.

We expect the initial Java 3D implementations to perform quite well because they will use existing, accelerated, low-level graphics APIs such as Direct3D, OpenGL, and Quickdraw3D.

Since all Java specifications are freely and openly available to the public, we anticipate that individuals or companies may choose to develop their own implementations of Java 3D. These implementations can be released at any time; however, any implementation that is based on a version of the Java 3D specification prior to the final 1.0 specification can at best be considered "beta."

Compliance Testing

Q: What about compliance testing?

A: JavaSoft will develop a compliance testing suite for certifying implementations as "Java 3D Compatible." The compliance test suite will ensure interoperability across platforms. The compliance suite will be developed in parallel with the reference implementations and final specification revisions.

What technical features does the Java 3D API support?

Immediate Mode

Q: Java 3D includes a fairly complete, generic, immediate mode. Why not a more complex one? Why not an even simpler one? Why not wrappers over OpenGL/Direct3D/QuickDraw3D?

A: The design of the Java 3D immediate mode enables cross-platform capability for all applications written using the Java 3D API. An application developer using immediate mode exclusively, whose main concern is performance and not inter-platform operability, should use the appropriate lower-level API rather than Java 3D. Otherwise, Java 3D's immediate mode provides the best compromise of features in a one-size-fits-all immediate-mode layer while still achieving reasonable performance. An even simpler immediate-mode API would severely limit possible performance-oriented optimizations; a more complex API could cause the underlying software and hardware models to diverge from one another. Another important constraint on Java 3D's immediate mode was the need to seamlessly interoperate with the retained mode API; otherwise we would have two divergent APIs.

View Model

Q: Java 3D provides a very sophisticated Virtual Reality based view model. Is all this complexity needed?

A: From the typical application's point of view, Java 3D's view model is quite simple: the application author places, orients, and scales a view platform object—period. Java 3D does the rest of the work. If the application only delivers an environment and leaves the details of viewpoint control to a browser, the application doesn't even need to provide a view platform object. If an application wants to port over existing code that uses a camera-based view model, it makes a single Java 3D call to establish the camera's parameters. Then, by moving the viewplatform instead of the camera, things function much as they did in the original system. Only the developers of browsers or sophisticated application packages need to worry about the additional details of Java 3D's view model; this is why the documentation of the view model is split into two parts: a chapter targeted at the ordinary application developer and an appendix for developers who need detailed knowledge of the view model.

Unfortunately, we cannot layer Java 3D's view model on top of the existing camera-based view model. Extending the Java vision of "write once, run anywhere" to modern viewing devices such as HMDs requires the implementation of the full Java 3D view-model semantics. The Java 3D view model simplifies to that of a camera-based model, not vice-versa. In our discussions with partners and potential users and after a through examination of the view model's semantics, we have reached the consensus that the view model's perceived complexity is necessary to extend the "write once, run anywhere" goal to include a "write once, view anywhere" goal.

JavaSoft's reference implementation will include full source code for the view model; this should address the concerns of Java 3D API implementors about coding complexity and understanding the semantics of the view model.

Tracker Model

Q: Why does Java 3D include support for a six-degree-of-freedom tracking model.

A: For performance reasons, the Java 3D view model needs to expose a six-degree-of-freedom head and hand-tracker as formal objects. Their exposure allows API implementors to optimize view model computations that rely on those trackers. Since Java 3D exposes such a device for head tracking, we generalized tracking to include additional channels of sensors, as well as simpler real-time continuous sensors such as joysticks.

AWT provides an abstraction of the most common desktop interaction peripherals: keyboards and mice. Java 3D uses these as is, rather than creating an incompatible I/O model. But for real-time critical devices, such as joysticks and six-degree-of-freedom devices, where AWT had not yet defined a mechanism, Java 3D introduces a new real-time class of I/O device. In real-time graphics systems, low latency can be more important than missing an event. An event more than a few 30th's of a second old may no longer be of any interest. The user's head position 1/10 of a second ago may not provide a useful approximation to the user's current head position. Java 3D provides access to the last "k" sensor values through a fixed-length circular buffer. An application can use those values to pick whatever value or values it wants to use in computing the sensor's "true" position and orientation.

Capability Bits

Q: Capability bits guard most of the modifiable state in Java 3D node objects. Their default value is access-not-granted ("don't touch that") for live scene-graphs. Why?

A: The vast majority of today's 3D environments are run-time environments, not editing environments. While in a typical editing situation it may be convenient to make most objects modifiable, such a default reduces the opportunity to optimize an application's runtime execution. If an application identifies only those objects that will change, Java 3D can perform optimizations over most of the scene-graph. Java 3D provides capability flags so that applications can specify this critical information.

Vector Mathematics Library

Q: The vector mathematics library is quite extensive, but why is it bundled with Java 3D? Why don't other Java packages use this code?

A: As stated in the Vector Mathematics appendix, we developed the set of math classes as part of the Java 3D process; however, JavaSoft has decided to make Java 3D's Vector objects and methods a separate package, extend them to include more functionality, and make them available to other evolving Java APIs. (In the process, some additional 2D vector and matrix classes may be defined.) However, to make the Vector Mathematics API available for review at the same time as the Java 3D APIs, it has been bundled with this release of the Java 3D API as an appendix.

Float and Double

Q: The vector mathematics library in particular, and many of the Java 3D interfaces in general, seem to give identical support for both single-precision and double floating-point formats.

A: This decision reflects the state of the industry today. It is much easier to provide both single and double precision support than to force the issue. Realistically, most 3D graphics hardware only operates on single-precision floating-point numbers (or even just fixed-point). On the other hand, some applications, especially in the MCAD space, use double-precision values exclusively. Because of this, the vector mathematics library supports both formats and many of the Java 3D methods accept either format as a user convenience. Also, in some computations, particularly 3D transformations, while the final rendering occurs using a single-precision composite transform matrix, the compositing of the matrix components on some platforms is calculated in double-precision.

Geometry Compression Format

Q: What is the Java 3D geometry compression format?

A: Java 3D is a run-time API. It does not define an external file format. The geometry compression format is a special case. It is both a run-time binary format (for platforms with hardware support for compressed geometry), a format that applications can use at the file and network level, and one that can interact with APIs other than Java 3D. Like the vector mathematics package, we describe the geometry compression format in an appendix to ensure completeness for this release of the specification; it will later be broken out into a separate specification document.

One more optimization point: even machines that do not support the rendering of compressed geometry directly may use non-standard, optimized, internal formats for representing renderable geometry. When provided with a compressed geometry, these machines can transcode it directly into their own internal format without incurring the space cost or semantics loss that would occur if the compressed geometry were first decompressed into a canonical floating-point format.

Triangles are Universal

Q: Is the only area geometric primitive supported by Java 3D the triangle?

Java 3D provides a broad implementation base for developing applications targeted at many markets, while still allowing those applications to run compatibly on many different hardware platforms. The triangle serves as a very useful lowest-common-dominator primitive that ports nicely across most platforms. Higher-level primitives vary massively by application area. In many areas, no other primitives are used. In others, fully trimmed NURBS are the minimum next step. By limiting the directly-supported graphics shapes to triangles, applications can choose to add further functionality at higher-level abstractions.

We fully expect higher-level functionality in future versions of the Java 3D API.

High-resolution Coordinates

Q: Why are high-resolution coordinates supported? Why not something simpler? More complex?

A: To support large scale virtual worlds, applications need some form of standardized higher-resolution coordinate system. To address this need, high-resolution coordinates are supported in Java 3D. However, the computational cost associated with supporting anything other than fixed translations between high-resolution coordinate frames was deemed too high. The extension of high-resolution coordinates to include rotations is still under evaluation, and may be included in later releases.

Scoping of Lights

Q: Why are there so many complex scoping modes for lights?

A: Without full-radiosity calculations, lighting is an approximation that uses hints provided by the application. Java 3D supports four general classes of scoping hints: scoping through explicit on/off switches, scoping through hierarchical specification, scoping through bounding volume, and scoping through natural attenuation.

Sound Library

Q: Why is the 3D sound API defined within Java 3D rather than one of the other Java Media APIs?

A: Java 3D is the only Java API that includes support for headtrackers, a parameterization of an end-user's physical head, generalized 3D transformations, and other such concepts. Because of this, fully-spatialized audio really only makes sense within the context of Java 3D. However, Java's Sound API is used to provide general sound and MIDI support. This is another issue under continuous evaluation.

Double-buffered, True Color, Z-buffered Rendering Model

Q: Java 3D assumes a double-buffered, true-color, Z-buffered rendering model as minimum. Why?

A: We made this minimal rendering configuration decision after extensive discussion and debate. Market trends clearly show a shift towards these minimum rendering capabilities, even among low-cost game boards. Note also that true color does not necessarily mean 24-bit color—an API can still support a true-color model even with a 15-bit or 8-bit frame buffer.

A very important factor in the minimal-rendering-configuration debate was the realization that support for indexed color would result in two different APIs; the same was found to be true for non Z-buffered rendering techniques. Unfortunately, supporting indexed color or no Z-buffering would result in incompatible programs that generally would not operate across platforms with different capabilities.

VRML

Q: Java 3D and VRML both appear to support similar retained scene-graph style applications. Why both?

A: VRML 1.0 and 2.0 are mainly file formats. VRML 2.0 is aimed at a general Internet market and additionally includes some support for programming-language-independent runtime API calls. Java 3D is specifically a Java language API, and is *only* a runtime API. Java 3D does not define a file or network format of its own. It is designed to scale well and provide support for markets that require higher levels of performance than VRML can provide; specifically real-time games, sophisticated mechanical CAD applications, and real-time simulation markets. In this sense, Java 3D provides a lower-level, underlying platform API. We expect that many VRML implementations will be layered on top of Java 3D.

We have talked with VRML developers about their expectations for an underlying, platform-independent 3D graphics API. We have modified a number of features and some semantics of Java 3D to track the evolving VRML specifications and avoid gratuitous differences. (For example, both systems use meters as the default physical units.) Given the parallel, ongoing evolution of VRML and Java 3D, it is not possible for these two APIs, each with different goals to keep in lockstep at all times; this is why we expect VRML environments to layer on top of Java 3D. Also, in discussion with potential Java 3D users, especially in the MCAD domain, they expect support for features in Java 3D not presently included in VRML.

Support Utilities and Applications

Q: Will JavaSoft provide support utilities and applications?

A: Java 3D includes a number of objects and methods, including several methods described as “helping functions,” that are part of the Java 3D specification. We will also provide other capabilities such as loaders and example programs that are not officially part of the Java 3D specification. (Over time, some of these may become part of the official specification. We will always clearly label those capabilities that are formally part of the specification.)

We expect the graphics community to write a wide variety of applications in and for Java 3D. While companies will keep most of their software proprietary, some developers will likely follow the Internet community model and release software into the public domain. Such applications may be quite helpful to the Java 3D community.

Extensions and Updates

Q: What about extensions, how will the process work?

A: This specification is aimed at the initial Java 3D API release. We have already identified a number of features and extensions, that, while of general interest, have been tentatively placed beyond the scope of the initial API release. (Several examples of such items are documented within the specification.)

Other features are of more limited interest, such as common “laws of physics” for simulations. We expect that such features will be best handled as additional APIs on top of Java 3D. (By definition, these features are beyond the scope of the initial specification.)

After initial release, requests for features and extensions will be considered for additional minor and major releases as merited; no explicit schedule or time frame is in place for these updates. We understand the importance of stable and relatively long-lived semantics for true platform interoperability.

What features are not supported by the Java 3D API?

Most Methods Final

Q: While all the Java 3D node classes may be subclassed, most of the pre-defined methods are declared final. Why?

A: Most of the pre-defined final methods provide access to internal state. Without direct access to an object's internal representation, an application that extended those methods would cut itself off from the object's internal state information—a situation with very little value for an application. Applications can, however, subclass most of Java 3D's classes, add instance variables, add new methods to provide added semantics, and use the object's final methods to access its original functionality. This technique provides one way to achieve VRML support.

User Traversal of Scene Graphs

Q: The Java 3D rendering model assumes that Java 3D controls the traversal of the scene graph. Why aren't there many hooks for an application to gain control over traversal or to insert immediate-mode render calls?

A: This is probably the one area where Java 3D takes a principled stand. Modern software systems use internal representations of 3D scenes that have very little to do with the original hierarchical scene graph defined by the application. A good analogy is the relationship between the source text of a C program and the highly-optimized machine code generated by parallelizing supercomputer compilers. A correspondence between the source and machine code exists, but the equivalent machine code statements might not exist (sharing common subexpressions) and might not execute in the original order.

Today's very large virtual worlds require some form of separate spatial hierarchical structure to represent a 3D scene's object geometry and location. Typically, far less than 1% of the objects in a virtual world are visible in a given image frame. Only the visible objects need to be involved in the generation of a given frame. Unfortunately, a hierarchical scene-graph traversal semantic that requires traversing *every* node in the scene graph "just in case" or to accumulate side effects creates unnecessary and indeed overwhelming computational costs—a cost no longer viable. The VRML 2.0 standard took the same position on this issue: the traversal order of the hierarchical scene graph is left up to the underlying implementation. In Java 3D nearly the only things connected by the hierarchy are the nested coordinated transforms. As in VRML 2.0, we have pushed nearly all of the state information down into the leaves.

Java 3D still provides techniques that applications can use to gain partial or full control of the traversal order, if needed. The ordered group node requires that Java 3D render its children in a specified linear order, if the node is rendered at all. Java 3D's mixed-immediate and retained-mode rendering causes Java 3D to automatically render many portions of the scene, while letting the application access the rendering loop so that it can render items in any way it wants.

Shading Language

Q: Java 3D provides layers of rendering abstraction: pure immediate mode, mixed immediate retained mode, retained mode, and compiled-retained mode. What about support for even higher quality graphics semantics: a shading language (like RenderMan™)?

A: The shading language used in Pixar's RenderMan works precisely because it *is* its own language with its own semantics. RenderMan is 'C- like', with many quite important differences. To do the same thing in Java 3D would require, in effect, a new non-standard dialect of Java—a direction that Java does not necessarily want to follow.

There are other ways to add shading language support to an API such as Java 3D, though in general they are more cumbersome. We are interested in supporting an even more sophisticated shading model than Java 3D's lowest common denominator, and this is an area under continuous active investigation. However, it was felt such support would be "a bridge too far" for the first release of Java 3D. Thus, we will continue to re-examine this issue to see if it warrants inclusion in future extensions or revisions of the Java 3D specification.

Off-screen Rendering and Printing

Q: What about support for off-screen rendering and printing?

A: Unfortunately, both off screen rendering support and printer support can require implementation-specific or hardware-specific features. While both off-screen rendering and printing support are considered important, they are *not* a part of the initial Java 3D specification.

Who needs the Java 3D API?

Markets

Q: What are the markets for Java 3D?

A: We designed Java 3D as a high-level, platform-independent, 3D graphics programming API amenable to very high-performance implementations, but still effective across a wide range of platforms. Java 3D is targeted at supporting a gamut of environments from small virtual universes through very large ones (lots of objects, most not visible at any given time). Java 3D also supports graceful scaling that keeps pace as the speed and capability of the underlying 3D hardware rendering engines increase by orders of magnitude over time.

Java 3D targets a variety of application domains, including providing efficient support for loading and supporting the various VRML formats on top of Java 3D. MCAD environments and applications also present an important application domain for Java 3D.

Real-time 3D games present one of the markets Java 3D supports. Java 3D includes many features that enable the development of platform-independent, high-performance games. These same features are ideal for the "higher end" gaming community: location-based entertainment, flight simulation, and the general visual simulation market.

Java 3D is *not* an authoring environment, and does not provide built-in authoring tools. It is aimed at providing the fastest possible run-time environment. However, we anticipate independent software suppliers will build authoring systems that generate Java 3D applications as output.

Hardware Platforms

Q: What specific hardware platforms is Java 3D aimed at?

A: Java 3D is aimed at a wide range of 3D-capable hardware and software platforms, from low-cost PC game cards and software renderers at the low end, through mid-range workstations, all the way up to very high-performance, specialized, 3D image generators. Because the initial implementations of Java 3D will be layered on OpenGL, Direct3D, and QuickDraw3D, Java 3D applications will run on any Java-enabled platform that supports one of these lower-level, immediate-mode 3D APIs. These APIs in turn have optimized implementations that provide support for various specific graphics hardware subsystems.

It is expected that Java 3D implementations will provide useful rendering rates on most modern PC's, and certainly will if their frame buffers have some 3D acceleration.

Of course, a very sophisticated Java 3D application aimed at real-time walk-throughs of an MCAD data base for, say, an oil platform, will quite likely not have adequate performance if run on a low-end PC.

On mid-range workstations, Java 3D is expected to provide applications with nearly full-speed hardware performance.

Finally, we designed Java 3D to scale as underlying hardware platforms increase in speed over time. Tomorrow's 3D PC game accelerators will support more complex virtual worlds than today's high-priced workstations; as the general level of 3D acceleration available on the mass market increases, Java 3D scales to meet it.