A guide to key concepts and effective techniques

Java 3D Programming

Daniel Selman

MANNING

released at or

About this book

Java 3D is a client−side Java application programming interface (API) developed at Sun Microsystems for rendering interactive 3D graphics using Java. Using Java 3D you will be able to develop richly interactive 3D applications, ranging from immersive games to scientific visualization applications.

Who should read it?

Java 3D Programming is aimed at intermediate to experienced Java developers. Previous experience in graphics programming (OpenGL and Swing, for example) will be very useful, but it's not a prerequisite. No book stands alone and you should make good use of the many online resources and books listed in appendix B and the bibliography. Readers new to Java 3D should definitely download Sun's excellent (free) Java 3D tutorial. This book is intended to serve as a companion to the Sun API documentation and the Java 3D tutorial.

How is it organized?

The book has 18 chapters, plus three appendices and a bibliography. Each chapter is fairly self−contained or explicitly references related chapters, allowing you to focus quickly on relevant material for your problem at hand. I have ordered the material so that, if you were starting a project from scratch, progressing in the book would mirror the design questions you would face as you worked through your design study and development efforts. More commonly used material is, in general, closer to the beginning of the book.

Chapter 1 focuses on getting started with Java 3D, system requirements, running the examples in the book, plus a look at the strengths and weaknesses of Java 3D.

Chapter 2 introduces some of the fundamentals of 3D graphics programming, such as projection of points from 3D to 2D coordinates, lighting, and hidden surface removal.

Chapter 3 gets you started with Java 3D programming, from setting up your development environment and resources to running your first application.

Chapter 4 explains the fundamental data structure in Java 3D, the scenegraph. Aspects of good scenegraph design are described using an example application for discussion.

Chapter 5 is a reference to Java 3D's scenegraph nodes, along with usage instructions and examples.

Chapter 6 explains the elements of the Java 3D scenegraph rendering model and guides you in your choice of `VirtualUniverse` configuration.

Chapter 7 takes a step back and examines data models for 3D applications. Choosing a suitable data model involves understanding your interaction and performance requirements.

Chapter 8 is a reference to creating geometry to be rendered by Java 3D.

Chapter 9 covers the elements of the Java 3D `Appearance` class, used to control the rendered appearance of the geometric primitives in your scene.

Chapter 10 illuminates the Java 3D lighting model and shows you how to create powerful lighting for your scene.

Chapter 11 introduces the Java 3D behavior model, which allows you to attach code to the objects in your scene. Examples illustrate both keyboard and mouse behaviors for graphical user interfaces.

Chapter 12 expands upon the discussion of behaviors, covering the `Interpolator` behaviors, used to control geometry attributes using the `Alpha` class.

Chapter 13 describes how to write your own custom behaviors and register them with Java 3D for invocation. Example behaviors for debugging and complex physical animation as well as others are presented.

Chapter 14 explains how to increase the realism of your scenes by applying bitmaps to your geometry using the process of texture mapping.

Chapter 15 highlights some of the utility classes provided with Java 3D for operations such as triangulation and loading of input data.

Chapter 16 delves into more techniques valuable for interacting with 3D scenes, object interaction using the mouse for selection of 3D objects, and performing collision detection between 3D objects.

Chapter 17 shows, through example, how to build Java 3D applications that use the Swing packages for 2D user interface elements, and can be distributed as Java applets for use from a web browser.

Chapter 18 goes low−level to explain some of the implementation details of the Java 3D API. The aim is to give you a greater appreciation for what is going on behind the scenes and help you optimize your applications.

Appendix A cross−references all the examples by chapter and includes instructions for downloading, installing, and running the example code from the publisher's web site.

Appendix B includes a comprehensive listing of programming and graphics resources online. Print references are provided in the bibliography.

Appendix C explains the `Primitive` utility class, its geometry cache, and the `GeomBuffer` class, along with tips and caveats.

Source code

The book contains over 30,000 lines of example code, including some reusable library code that I hope will contribute to the collective understanding of the Java 3D community. Code of particular interest is shown in boldface. Appendix A contains a list of the example Java 3D applications and applets developed for this book, as well as detailed instructions for running the examples. The code itself is identified in the text by an initial reference to its location at http://www.manning.com/selman, the Manning web site for this book.

Typographical conventions

Italic typeface is used to introduce new terms.

`Courier` typeface is used to denote code samples as well as elements and attributes, method names, classes, interfaces, and other identifiers.

`Courier bold` typeface is used to denote code of special interest.

Code line continuations are indented.

How to use the book

I have tried to organize many of the topics in the book in an order appropriate for developers designing and building a new Java 3D application. I would suggest initially reading or skimming the chapters sequentially to get an overall feel for the design of your application, and then returning to specific chapters and examples for reference material as required. Please note that the example source code for the book is provided under the GNU General Public License (GPL) (http://www.gnu.org/licenses/licenses.html). I encourage you to modify and distribute the source code in accordance with the spirit of open source and the GPL license.

If you still need help or have questions for the author, please read about the unique Author Online support that is offered from the publisher's web site.

Author Online

Purchase of *Java 3D Programming* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to http://www.manning.com/selman. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the author some challenging questions, lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's web site as long as the book is in print.

CHAPTER 1

What is Java 3D and is it for me?

Java 3D is an application programming interface (API) developed at Sun Microsystems for rendering interactive 3D graphics using the Java programming language. Java 3D is a *client−side* Java API. Other examples of Sun client−side APIs include the Abstract Windows Toolkit (AWT) and Java Foundation Classes (JFC/Swing), which are both Java class libraries for building applications with a Graphical User Interface (GUI). The client−side Java APIs are in contrast to Sun's server−side APIs such as Enterprise Java−Beans (EJB) and the other components of Java 2 Enterprise Edition (J2EE).

Making 3D graphics interactive is a long−standing problem, as evidenced by its long history of algorithms, APIs, and vendors. Sun is not a major player in the 3D graphics domain, although its hardware has long supported interactive 3D rendering. The dominant industry standard for interactive 3D graphics is OpenGL, created by Silicon Graphics (SGI). OpenGL was designed as a cross−platform rendering architecture and is supported by a variety of operating systems, graphics card vendors, and applications. The OpenGL API is written in the C programming language, and hence not directly callable from Java. A number of open source and independent programming efforts have provided simple Java wrappers over the OpenGL API that allow Java programmers to call OpenGL functions, which are then executed in native code that interacts with the rendering hardware. One of the most popular is GL4Java, which you can find at http://www.jausoft.com/gl4java/.

However, there are few advantages to using a Java wrapper over OpenGL, as opposed to coding in C and calling OpenGL directly. Although programmers can use the more friendly Java APIs, they must incur the overhead of repeated calls through the Java Native Interface (JNI) to call the native OpenGL libraries.

Java 3D relies on OpenGL or DirectX to perform native rendering, while the 3D scene description, application logic, and scene interactions reside in Java code. When Sun set out to design Java 3D, although they did not have the resources or industry backing to replace OpenGL, they wanted to leverage more of Java's strengths as an object−oriented programming (OOP) language instead of merely delegating to a procedural language such as C. Whereas OpenGL's level of description for a 3D scene consists of lists of points, lines, and triangles, Java 3D can describe a scene as collections of objects. By raising the level of description and abstraction, Sun not only applied OOP principles to the graphics domain, but also introduced scene optimizations that can compensate for the overhead of calling through JNI.

1.1 Strengths

The foremost strength of Java 3D for Java developers is that it allows them to program in 100 percent Java. In any sizeable 3D application, the rendering code will compose only a fraction of the total application. It is therefore very attractive to have all the application code, persistence, and user interface (UI) code in an easily portable language, such as Java. Although Sun's promise of Write−Once−Run−Anywhere is arguably more of a marketing dream than a reality, especially for client−side programming, Java has made important inroads toward enabling application developers to write applications that can be easily moved between platforms. The platforms of most interest today are Microsoft Windows 98/NT/2000, Sun Solaris, LINUX, and Macintosh OS X.

Java has arguably become *the* language of networked computing and the Internet. High−level support for remote method invocation (RMI), object serialization, platform independent data types, UNICODE string encoding, and the security model all provide persuasive arguments for adopting the Java language for applications that are increasingly gravitating away from a desktop−centric worldview. Many of the state−of−the−art 3D graphics applications being built with Java 3D today are leveraging the strengths of Java as a language for the Internet.

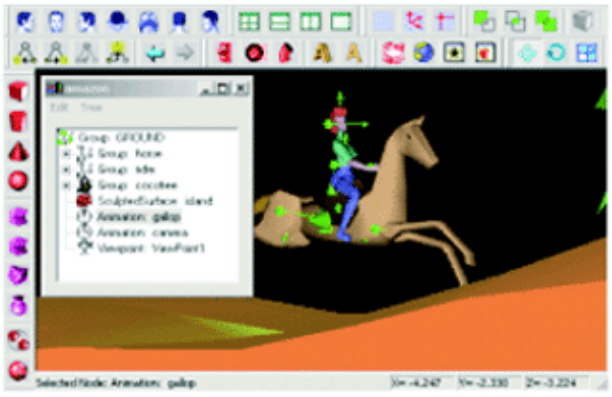
The Java 3D API itself has much to offer the application developer. By allowing the programmer to describe the 3D scene using coarser−grained graphical objects, as well as by defining objects for elements such as appearances, transforms, materials, lights, and so forth, code is more readable, maintainable, reusable, and easier to write. Java 3D uses a higher level scene description model, the *scenegraph*, which allows scenes to be easily described, transformed, and reused.

Java 3D includes a view model designed for use with head−mounted displays (HMDs) and screen projectors. By insulating the programmer from much of the complex trigonometry required for such devices, Java 3D eases the transition from a screen−centric rendering model to a projected model, where rendering in stereo allows for greater realism.

Java 3D also includes built−in support for sampling 3D input devices and rendering 3D spatial sound. By combining all of the above elements into a unified API, Java 3D benefits from a uniformity of design that few other APIs can match.

Java 3D's higher level of abstraction from the mechanics of rendering the scene have also opened the field of interactive 3D graphics to a new class of audience, people who would typically have been considered 3D content creators. Think of 3D graphics creation as a spectrum, with resources and talents distributed across a variety of tasks, as illustrated in figure 1.1.

Content Creation



VRML

```
Material  {
   ambientColor  0.200 0.200 0.200
   diffuseColor  0.800 0.400 0.500
   shininess  0.000
}
Cube {
   width  2.000
   height  2.000
   depth  2.000
}
```

Java 3D Programming

```
universe = createVirtualUniverse();
Locale locale = createLocale( universe );
BranchGroup sceneBranchGroup = createSceneBranchGroup();
Background background = createBackground();
```

OpenGL Programming

```
glClearColor(1.0, 1.0, 1.0, 1.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 0.0, 0.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Vendor Specific Programming

Figure 1.1 Java 3D fills an important gap between VRML, which is centered around describing 3D content, and OpenGL, which is a C API for rendering points, lines, and triangles

Many new programmers have moved from Virtual Reality Modeling Language (VRML) into Java 3D. They are 3D content creation specialists; and they require the greater flexibility offered by a programming API, though they are reluctant to learn OpenGL and C. For this audience, Java 3D fills an important niche, allowing them to concentrate on content creation and application logic, without choking on the details of rendering and arcane programming syntax.

1.2 Weaknesses

Many of the strengths can be reversed and cited as weaknesses. For some programmers coming *from* OpenGL, there are some OpenGL features that are hard or impossible to achieve within Java 3D. Some of this audience may miss the total control they have over the scene and the rendering process. Many others, however, will quickly learn the mapping from OpenGL functions to Java 3D objects and will appreciate the productivity gains they can achieve using Java 3D.

Although Java 3D includes some clever optimizations, a skilled developer using OpenGL and native C code may be able to achieve higher performance than a Java programmer using Java 3D. If absolute rendering performance is the top−priority for your application then you may be better off using OpenGL or another native rendering API.

One particular problem, inherent in Java, which can be noticeable in performance−critical applications, is the impact of the Java garbage collector (GC). The Java runtime, the Java 3D runtime, and the application code all create objects. All these objects will eventually be garbage, and be collected by the Java Virtual Machine (JVM) GC. While the GC is running there may be an appreciable system slowdown, resulting in several rendered frames being dropped. If garbage collection occurs in the middle of a critical animation sequence, the realism of the rendered scene may be lowered for the user. However, with continued improvements in GC technology, faster hardware, and well−designed and implemented applications, such considerations are no longer prevalent.

The Java client−side APIs, and especially Java 3D, can be difficult to distribute to end users. While the biggest pool of end users run Windows, Sun has had limited success getting Java 2 (JRE 1.2) deployed on the Windows platform. Java 2 is required for Java 3D, although Microsoft's JVM does not support Java 2. This means that end users are required to download Sun's Java 2 implementation, install it, and then download Java 3D and install it, all prior to running your application. If you are deploying your application as an applet, the installation process is potentially more complex as some end users will have to manually copy or edit configuration files before they can view your applet. In addition a suitable version of OpenGL or DirectX must be installed and configured for the end user's hardware and drivers. This lengthy download and installation process can lead to frustration; I think we are some way from seeing mainstream software and games companies offering consumer−grade software products built using Java 3D, or even Java 2. Many modern end users expect the convenience of point−and−click installation and do not have the computer skills to set CLASSPATH variables or debug installation problems.

There is light at the end of the tunnel, however, as the Java WebStart project attempts to make installing and running SDK 1.2 Java applications as easy as running native applications—which may be just as well. At present it does not appear that Microsoft will be shipping *any* JVM with Windows XP.

At present, the biggest groups of Java 3D users appear to be computer scientists, businesspeople, hobbyists, game developers, and programmers. These early adopters are spearheading the deployment of Java 3D for mainstream applications.

1.3 System requirements (developer and end user)

Java is a resource−intensive development and deployment environment and creating interactive 3D graphics is still one of the most challenging tasks for modern PCs. Interactive 3D rendering requires hardware dedicated to 3D rendering, usually provided by third−party display hardware specially adapted for processing 3D scenes. Fortunately, 3D−display hardware has reduced in price radically over the past few years, and today's typical game PCs are able to exceed the capabilities of the expensive dedicated graphics workstations of just a few years ago.

For a realistic immersive 3D experience (first−person 3D games, for example), a consistently high frame rate is required, typically 30 frames per second (FPS) or higher. More powerful rendering hardware will be able to achieve higher frame rates, at higher screen resolutions and with higher resolution texturing, all of which contribute to the overall experience. Modern PCs could typically achieve reasonable frame rates without dedicated rendering hardware, however the processor must execute both application logic and rendering code—to the detriment of both.

Nonimmersive 3D applications (such as visualization or modeling) do not typically require as high a frame rate as immersive applications. However the application logic may become the limiting factor on frame rate, as complex calculations may be necessary prior to rendering every frame.

The frame rate that the end users see is determined by a number of factors:

- *Vertex or transform bound*—Ability of the display hardware to transform and display each vertex in the scene
- *Fill bound*—Ability of the display hardware to shade and texture the scene and push the resulting pixels to the screen
- *Application logic bound*—Ability of the application to prepare the scene for rendering

Different types of application will place different demands on those factors, and the type of application you are writing will often dictate the hardware requirements for development and end users.

The minimum requirements for most interactive 3D applications are:

- 500+ MHz main processor
- Dedicated 3D−display hardware, with at least 16 MB of texture memory. New 3D graphics cards are released regularly so you should research the latest cards within your budget. Ensure that the card has good OpenGL compatibility for use with Java 3D. The Java 3D mailing list is a good source of information on people's experiences with various graphics cards.
- 128+ MB of system RAM

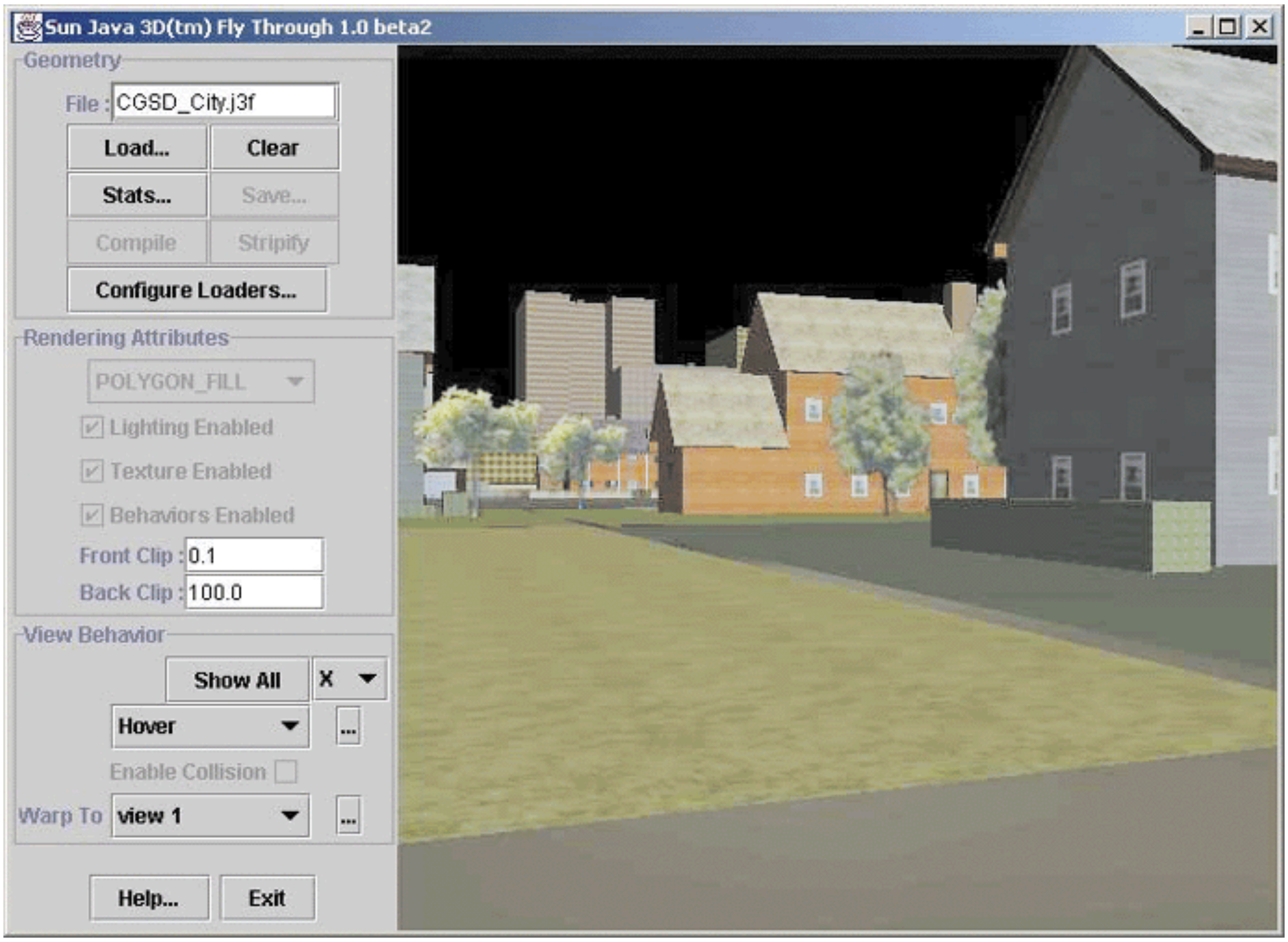
An important part of designing your application should be to set your performance targets. Gather requirements from your user base on typical available hardware and ensure that your application can perform adequately on your target machine configuration. You may need to test using several popular graphics cards to ensure compatibility and performance. You may need to try several driver versions to find the best drivers for your supported cards. Unfortunately, Write−Once−Run−Anywhere does not work out too well in the world of 3D graphics!

Analyze the performance of your application using a tool such as OptimizeIt from VMGEAR (http://www.vmgear.com) to determine whether your frame rate is limited by your application logic or display hardware. Regular use of OptimizeIt will also help you to get the maximum performance from the JVM and increase garbage collection intervals.

1.4 EXPECTED PERFORMANCE

An important part of your application design is to estimate your expected performance and validate your design against your target machine configurations. Aim to build some simple prototypes that will allow you to extrapolate your finished application's performance. It is far easier to revise your designs at this stage than two weeks before completion.

For example, on my home machine—with an AMD 850 MHz processor, nVidia GeForce II Ultra (64 MB RAM) graphics card, and 256 MB RAM—I get about 35 FPS running the Java 3D Fly−Through example application (http://www.javasoft.com/products/java−media/3D/flythrough.html). The Fly−Through city scene (figure 1.2) is composed of 195,000 triangles, 4,115 `Shape3D` instances, and 1,238 `Appearance`s (uncompiled scenegraph).



File : CGSD_City.j3f
POLYGON_FILL
☑ Lighting Enabled
☑ Texture Enabled
☑ Behaviors Enabled
Front Clip : 0.1
Back Clip : 100.0
X
Hover
Enable Collision ☐
Warp To view 1

Figure 1.2 The Sun Java 3D example Fly−Through

1.4.1 Memory footprint

Java programs generally tend to require more memory than native programs. This is especially true of programs with a GUI using Swing/JFC. Java 3D can also have high memory requirements, especially if your application loads lots of large bitmaps for texture mapping objects, or defines complex geometry composed of many thousands of vertices.

To give you some idea of Java 3D's memory requirements, table 1.1 shows the total memory required for the Java 3D Fly−Through application. As you can see, bringing up the Swing application requires 25 MB, while opening the city scene pushes memory usage up to over 100 MB.

Table 1.1 Java 3D Fly−Through statistics

Working set	25 MB (no scene loaded)
Working set	108 MB (city scene loaded)

Memory usage will be an important component of your application performance. Performance will be extremely poor if your target users have less physical RAM available than the working set for your application. In this case, the operating system will have to page virtual memory to and from disk.

Another performance criterion that can be important for some applications is startup time. You should set targets for the startup time for your application. The JVM can take a considerable time to start, especially on slower machines with limited RAM. In addition, if you are loading large texture files or 3D object models, your startup time can become very significant. The RAM footprint of your application (including the JVM) and the available system RAM of the end user's computer are the most significant elements affecting startup time. You should take regular startup time measurements while you are in development to ensure that your end users are not frustrated every time they launch your application.

If you are deploying an applet, you should also be aware of the time required for it to download, as well as the graphics resources the applet requires for rendering. Texture images and 3D models can quickly become very large, so some download time targets based on typical end user bandwidth will also prove very useful.

As a reference, I measured the startup time of the Java 3D Fly−Through application. As you can see in table 1.2, launching the application took a very respectable 3 seconds, while loading the 3D content took 14 seconds. Fourteen seconds is a long time, and necessitates some form of progress indicator to reassure users that progress is occurring!

Table 1.2 Java 3D Fly−Through statistics

Start−up time	3 seconds
Loading city scene	14 seconds

1.5 Running the examples

By now, you are probably itching to see Java 3D in action. Please refer to appendix A for a list of the example Java 3D applications and applets developed for this book, as well as detailed instructions for running the examples.

1.6 Summary

Straddling the worlds of content creation and scripting on the one hand and low−level pipeline−based rendering programs on the other, the Java 3D API fills an important gap in 3D graphics APIs. With careful design and implementation, performance of Java 3D applications can rival native OpenGL applications and will exceed JNI−based Java wrappers over OpenGL.

As a Java API, Java 3D is relatively mature, first appearing at the end of 1998. But compared to OpenGL, which first appeared in the early 1990s, Java 3D is still an upstart. For example, OpenGL contains an extension facility that allows vendors to write proprietary extensions to the API—a feature that is not yet implemented in Java 3D, though it is rumored to be appearing in Java 3D 1.4. The Architecture Review Board

(ARB) controls additions to OpenGL—while Java 3D may be placed under the Java Community Process (JCP), allowing experts and vendors to influence the direction of the API.

Java 3D is the right choice if you want to program 3D applications using Java. Just as Java introduced many useful abstractions over C++ and includes a rich library of standard APIs, Java 3D introduces abstractions over OpenGL/Direct3D and includes many features that will bring your applications to market faster. Java 3D can be frustrating at times—abstraction is not always a good thing—but it will save you time as you leverage years of API development by Sun’s engineers. While absolute performance is sometimes a requirement, 3D graphics hardware, processor, and memory availability are advancing so rapidly that any disparity between Java/Java3D and C/OpenGL is shrinking for all but the most memory−intensive applications.

CHAPTER 2

3D graphics programming

3D graphics programming is a fairly complex topic, worthy of a book unto itself (and there are many), but this introduction should serve as a good roadmap for further reading and give an appreciation for what Java 3D and your OpenGL or DirectX drivers are doing behind the scenes. In this chapter, I describe some of the fundamental underlying graphics techniques that allow a computer to transform a 3D scene description into a rendered image.

I'll explain much of the needed terminology; however, if you need more information, I recommend the online 3D graphics glossaries from Mondo Media (http://www.mondomed.com/mlabs/glossary.html), 3Dgaming.com (http://www.3dgaming.com/fps/techshop/glossary/), and Chalmers Medialab (http://oss.medialab.chalmers.se/dictionary/).

2.1 Learning 3D graphics programming

Given the enormous variety of teaching and learning styles, there probably is no *best* way of teaching 3D graphics programming. I learned 3D graphics programming by experimenting. I wrote my first 3D graphics program about 10 years ago. It was written in C and ran on my venerable Intel 80386 with a whole 256 KB of RAM! Needless to say, it didn't use Java 3D or OpenGL. The program was a modified port of a simple BASIC program that I "borrowed" from a simple little BASIC programming book. I later ported the program to run on Solaris using the GKS rendering API. The program was a very simple wire frame 3D model viewer and editor. You could load 3D shapes described using ASCII text files and then display them on screen. You could also interactively rotate the shapes about one axis. Times have certainly changed.

The interesting thing about my first 3D effort is that I built upon my general programming knowledge and some simple 2D rendering techniques, such as drawing a line to the screen. That's what we'll do here. In this chapter, we will turn the clock back 10 years and build some sections of that program all over again, this time using Java, Java 2D, and some of the Java 3D utilities. This should remove some of the mystery from the operations performed by 3D graphics libraries like Java 3D and OpenGL. At the end of the day, we are simply converting from 3D coordinates to 2D coordinates and drawing a bunch of points and lines. We can use the source code as a basis for introducing the basics of 3D graphics programming and highlight some of the fundamental operations that a graphics library such as Java 3D provides.

By looking at the example, you'll see the additional operations that a real graphics API provides, and that our homegrown, primitive API does not.

To begin, look at the output from a simple Java 3D program and compare it with the test−bed application MyJava3D. Figure 2.1 was rendered by a simple Java 3D program (the LoaderTest example), which loads a

Lightwave OBJ file and renders it to the screen. Figure 2.2 was rendered in MyJava3D using AWT 2D graphics routines to draw the lines that compose the shape.

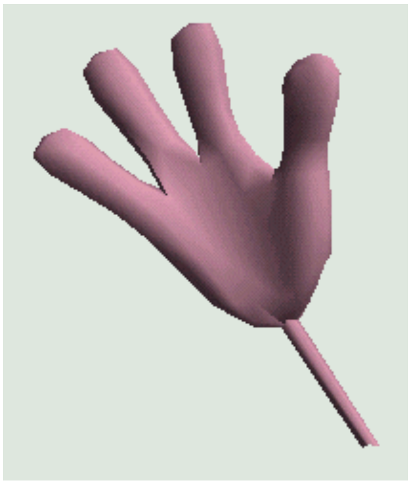


Figure 2.1 Output of a simple Java 3D application (LoaderTest)

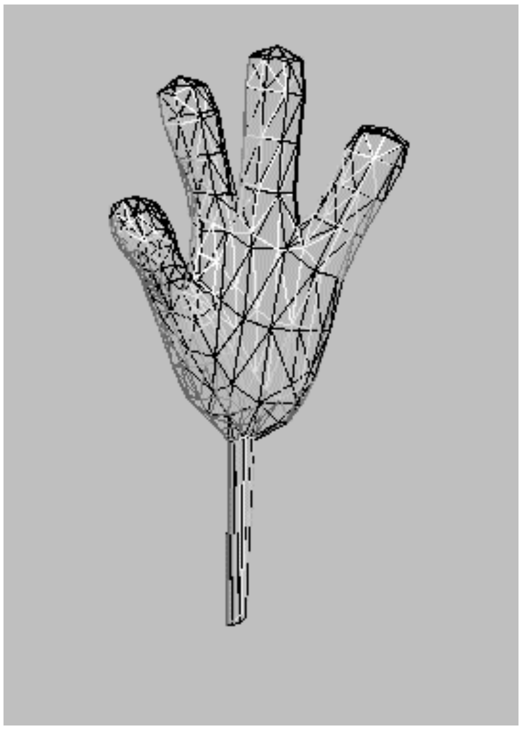


Figure 2.2 Output rendered by MyJava3D—a wire frame version of the same hand used for figure 2.1

The Java3D−rendered image is certainly superior. I’ll compare the two images in detail later in this chapter. However, the wire frame version (just lines) that was rendered using MyJava3D is also useful.

Note how the triangular surfaces that compose the 3D model are visible in figure 2.2. The model is composed of hundreds of points, each positioned in 3D space. In addition, lines are drawn to connect the points, to form triangular surfaces. The illusion of a solid 3D shape in figure 2.1 has now been revealed—what appeared to be a solid shape is in fact a hollow skin. The skin of the shape is described using hundred of points, which are then drawn as solid triangles. Java 3D filled the interior of the triangles while MyJava3D merely drew the outer lines of each triangle.

Consider the simplest series of operations that must take place to convert the 3D model data into a rendered image:

1. Load the 3D points that compose the vertices (corners) of each triangle. The vertices are indexed so they can be referenced by index later.
2. Load the connectivity information for the triangles. For example, a triangle might connect vertices 2, 5, and 7. The actual vertex information will be referenced using the information and indices established in step 1.
3. Perform some sort of mathematical conversion between the 3D coordinates for each vertex and the 2D coordinates used for the pixels on the screen. This conversion should take into account the position of the viewer of the scene as well as perspective.
4. Draw each triangle in turn using a 2D graphics context, but instead of using the 3D coordinates loaded in step 1, use the 2D coordinates that were calculated in step 3.
5. Display the image.

That's it.

Steps 1, 2, 4, and 5 should be straightforward. Steps 1 and 2 involve some relatively simple file I/O, while steps 4 and 5 use Java's AWT 2D graphics functions to draw a simple line into the screen. Step 3 is where much of the work takes place that qualifies this as a 3D application.

In fact, in the MyJava3D example application, we cheat and use some of the Java 3D data structures. This allows us to use the existing Lightwave OBJ loader provided with Java 3D to avoid doing the tiresome file I/O ourselves. It also provides useful data structures for describing 3D points, objects to be rendered, and so on.

2.2 Projecting from 3D world coordinates to 2D screen coordinates

Performing a simple projection from 3D coordinates to 2D coordinates is relatively uncomplicated, though it does involve some matrix algebra that I shan't explain in detail. (There are plenty of graphics textbooks that will step you through them in far greater detail than I could here.)

There are also many introductory 3D graphics courses that cover this material online. A list of good links to frequently asked questions (FAQs) and other information is available from 3D Ark at http://www.3dark.com/resources/faqs.html. If you would like to pick up a free online book that discusses matrix and vector algebra related to 3D graphics, try Sbastien Loisel's *Zed3D, A compact reference for 3D computer graphics programming*. It is available as a ZIP archive from http://www.math.mcgill.ca/~loisel/.

If you have some money to spend, I would recommend picking up the bible for these sorts of topics: *Computer Graphics Principles and Practice*, by James Foley, Andries van Dam, Steven Feiner, and John Hughes (Addison−Wesley, 1990).

2.2.1 A simple 3D projection routine

Here is my simple 3D−projection routine. The `projectPoint` method takes two `Point3d` instances, the first is the input 3D−coordinate while the second will be used to store the result of the projection from 3D to 2D coordinates (the *z* attribute will be 0). `Point3d` is one of the classes defined by Java 3D. Refer to the Java 3D JavaDoc for details. Essentially, it has three public members, *x*, *y*, and *z* that store the coordinates in the three axes.

From AwtRenderingEngine.java

```
private int xScreenCenter = 320/2;
private int yScreenCenter = 240/2;
private Vector3d screenPosition = new Vector3d( 0, 0, 20 );
private Vector3d viewAngle = new Vector3d( 0, 90, 180 );
private static final double DEG_TO_RAD = 0.017453292;
private double modelScale = 10;

CT = Math.cos( DEG_TO_RAD * viewAngle.x );
ST = Math.sin( DEG_TO_RAD * viewAngle.x );
CP = Math.cos( DEG_TO_RAD * viewAngle.y );
SP = Math.sin( DEG_TO_RAD * viewAngle.y );

public void projectPoint( Point3d input, Point3d output )
{
 double x = screenPosition.x + input.x * CT − input.y * ST;
 double y = screenPosition.y + input.x * ST * SP + input.y * CT * SP
    + input.z * CP;
 double temp = viewAngle.z / (screenPosition.z + input.x * ST * CP
    + input.y * CT * CP − input.z * SP );

 output.x = xScreenCenter + modelScale * temp * x;
 output.y = yScreenCenter − modelScale * temp * y;
 output.z = 0;
}
```

Let’s quickly project some points using this routine to see if it makes sense. The result of running seven 3D points through the projectPoint method is listed in table 2.1.

```
CT: 1
ST: 0
SP: 1
CP: 0
```

Table 2.1 Sample output from the projectPoint method to project points from 3D−world coordinates to 2D−screen coordinates

WX	**WY**	**WZ**	**SX**	**SY**
1	1	0	250	30
−1	1	0	70	30
1	−1	0	250	210
−1	−1	0	70	210
0	0	0	160	120
1	1	1	255	25
−1	−1	1	65	215

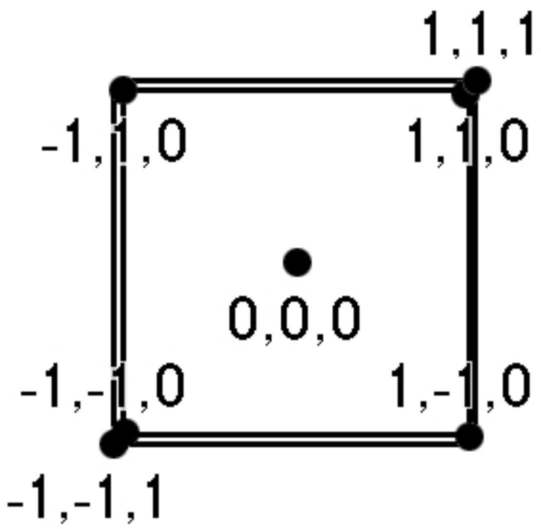


Figure 2.3 The positions of some projected points

Plotting these points by hand using a 2D graphics program (figure 2.3), you can see that they seem to make sense. Projecting the point 0,0,0 places a point at the center of the screen (160,120). While you have symmetry about the corners of the cube, increasing the Z−coordinate appears to move the two opposing corners (1,1,1 and −1,−1,1) closer to the viewer.

Taking a look at the `projectPoint` function again, you can see it uses the following parameters:

- Input point *x*, *y*, and *z* coordinates
- Center of the screen
- Sin and cosine of the viewer's angle of view
- Distance of the screen from the viewer
- Model scaling factor

This very simple projection function is adequate for simple 3D projection. As you become more familiar with Java 3D, you will see that it includes far more powerful projection abilities. These allow you to render to stereo displays (such as head−mounted displays) or perform parallel projections. (In parallel projections, parallel lines remain parallel after projection.)

2.2.2 Comparing output

Look at the outputs from MyJava3D and Java 3D again (figure 2.4). They are very different—so Java 3D must be doing a lot more than projecting points and drawing lines:

- Triangles are drawn filled; you cannot see the edges of the triangles.
- Nice lighting effect can be seen in the curve of the hand.
- Background colors are different.
- Performance is much better—measured by comparing the number of frames rendered per second.