Performance Guide for Java 3D

I - Introduction

The Java 3D API was designed with high performance 3D graphics as a primary goal. This document presents the performance features of Java 3D in several ways. It describes the specific APIs that were included for performance. It describes which optimizations are currently implemented in Java 3D. And, it describes a few tips and tricks that application writers can use to improve the performance of their applications.

II - Performance in the API

There are several things in the API that were included specifically to increase performance. This section examines a few of them.

- Capability bits

Capability bits are the applications way of describing its intentions to the Java 3D implementation. The implementation examines the capability bits to determine which objects may change at run time. Many optimizations are possible with this feature.

- isFrequent bits

Setting the isFrequent bit indicates that the application may frequently access or modify those attributes permitted by the associated capability bit.

This can be used by Java 3D as a hint to avoid certain optimizations that could cause those accesses or modifications to be expensive. By default, the isFrequent bit associated with each capability bit is set.

- Compile

There are two compile methods in Java 3D. They are in the BranchGroup and SharedGroup classes. Once an application calls compile(), only those attributes of objects that have their capability bits set may be modified. The implementation may then use this information to "compile" the data into a more efficient rendering format.

- Bounds

Many Java 3D object require bounds associated with them. These objects include Lights, Behaviors, Fogs, Clips, Backgrounds, BoundingLeafs, Sounds, Soundscapes, ModelClips, and AlternateAppearance.

The purpose of these bounds is to limit the spatial scope of the specific object. The implementation may quickly disregard the processing of any objects that are out of the spatial scope of a target object.

- Unordered Rendering

All state required to render a specific object in Java 3D is completely defined by the direct path from the root node to the given leaf. That means that leaf nodes have no effect on other leaf nodes, and therefore may be rendered in any order. There are a few ordering requirements for direct descendants of OrderedGroup nodes or Transparent objects. But, most leaf nodes may be reordered to facilitate more efficient rendering.

- OrderedGroup

OrderedGroup now supports an indirection table to allow the user to specify the order that the children should be rendered. This will speed up order update processing, eliminating the expensive attach and detach cycle.

- Appearance Bundles
A Shape3D node has a reference to a Geometry and an Appearance. An Appearance NodeComponent is simply a collection of other NodeComponent references that describe the rendering characteristics of the geometry. Because the Appearance is nothing but a collection of references, it is much simpler and more efficient for the implementation to check for rendering characteristic changes when rendering. This allows the implementation to minimize state changes in the low-level rendering API.

- NIO buffer support for Geometry by-reference
NOTE: Use of this feature requires version 1.4 of the JavaTM 2 Platform.

This provides a big win in both memory and performance for applications that use native C code to generate their geometric data. In many cases, they will no longer need to maintain two copies of their data (one in Java and one in C). The performance win comes mainly from not having to copy the data from their C data structures to the Java array using JNI. Also, since the array isn't part of the pool of memory managed by the garbage collector, it should speed up garbage collection.

III - Current Optimizations in Java 3D

This section describes several optimizations that are currently implemented in Java 3D. The purpose of this section is to help application programmers focus their optimizations on things that will complement the current optimizations in Java 3D.

- Hardware
Java 3D uses OpenGL or Direct3D as its low-level rendering APIs. It relies on the underlying OpenGL or Direct3D drivers for its low-level rendering acceleration. Using a graphics display adapter that offers OpenGL or Direct3D acceleration is the best way to increase overall rendering performance in Java 3D.

- Compile
The following compile optimizations are implemented in the Java 3D 1.2.1 and 1.3 release:

Scene graph flattening: TransformGroup nodes that are neither readable nor writable are collapsed into a single transform node.

Combining Shape3D nodes: Non-writable Shape3D nodes that have the same appearance attributes, not pickable, not collidable, and are under the same TransformGroup (after flattening) are combined, internally, into a single Shape3D node that can be rendered with less overhead.

- State Sorted Rendering
Since Java 3D allows for unordered rendering for most leaf nodes, the implementation sorts all objects to be rendered on several rendering characteristics. The characteristics that are sorted on are, in order, Lights, Texture, Geometry Type, Material, and finally localToVworld transform. The only 2 exceptions are to (a) any child of an OrderedGroup node, and (b) any transparent object with View's Transparency sorting policy set to TRANSPARENCY_SORT_GEOMETRY. There is no state sorting for those objects.

- View Frustum Culling
The Java 3D implementation implements view frustum culling. The view frustum cull is done when an object is processed for a specific Canvas3D. This cuts

down on the number of objects needed to be processed by the low-level graphics API.

- Multithreading
The Java 3D API was designed with multithreaded environments in mind. The current implementation is a fully multithreaded system. At any point in time, there may be parallel threads running performing various tasks such as visibility detection, rendering, behaviour scheduling, sound scheduling, input processing, collision detection, and others. Java 3D is careful to limit the number of threads that can run in parallel based on the number of CPUs available.

- Space versus time property
By default, Java3d only builds display list for by-copy geometry. If an application wishes to have display list build for by-ref geometry to improve performance at the expense of memory, it can instruct Java3d by disable the j3d.optimizeForSpace property to false. For example :

java -Dj3d.optimizeForSpace=false MyProgram

This will cause Java3d to build display list for by-ref geometry and infrequently changing geometry. See also : Part II - isFrequent bits, and Part IV - Geometry by reference.

IV - Tips and Tricks

This section presents a few tips and tricks for an application programmer to try when optimizing their application. These tips focus on improving rendering frame rates, but some may also help overall application performance.

- Move Object vs. Move ViewPlatform
If the application simply needs to transform the entire scene, transform the ViewPlatform instead. This changes the problem from transforming every object in the scene into only transforming the ViewPlatform.

- Capability bits
Only set them when needed. Many optimizations can be done when they are not set. So, plan out application requirements and only set the capability bits that are needed.

- Bounds and Activation Radius
Consider the spatial extent of various leaf nodes in the scene and assign bounds accordingly. This allows the implementation to prune processing on objects that are not near. Note, this does not apply to Geometric bounds. Automatic bounds calculations for geometric objects is fine. In cases such as the influencing or scheduling bounds encompass the entire scene graph, setting this bounds to infinite bounds may help improve performance. Java3d will shortcircuit intersection test on bounds with infinite volume. A BoundingSphere is a infinite bounds if it's radius is set to Double.POSITIVE_INFINITY. A BoundingBox is a infinite bounds if it's lower(x, y, z) are set to Double.NEGATIVE_INFINITY, and it's upper(x, y, z) are set Double.POSITIVE_INFINITY. Bounds computation does consume CPU cycles. If an application does a lot of geometry coordinate updates, to improve performance, it is better to turn off auto bounds compute. The application will have to do the bounds update itself.

- Change Number of Shape3D Nodes
In the current implementation there is a certain amount of fixed overhead associated with the use of the Shape3D node. In general, the fewer Shape3D

nodes that an application uses, the better. However, combining Shape3D nodes without factoring in the spatial locality of the nodes to be combined can adversely affect performance by effectively disabling view frustum culling. An application programmer will need to experiment to find the right balance of combining Shape3D nodes while leveraging view frustum culling. The .compile optimization that combines shape node will do this automatically, when possible.

- Geometry Type and Format
Most rendering hardware reaches peak performance when rendering long triangle strips. Unfortunately, most geometry data stored in files is organized as independent triangles or small triangle fans (polygons). The Java 3D utility package includes a stripifier utility that will try to convert a given geometry type into long triangle strips. Application programmers should experiment with the stripifier to see if it helps with their specific data. If not, any stripification that the application can do will help. Another option is that most rendering hardware can process a long list of independent triangles faster than a long list of single triangle triangle fans. The stripifier in the Java 3D utility package will be continually updated to provided better stripification.

- Sharing Appearance/Texture/Material NodeComponents
To assist the implementation in efficient state sorting and allow more shape nodes to be combined during compilation, applications can help by sharing Appearance/Texture/Material NodeComponent objects when possible.

- Geometry by reference
Using geometry by reference reduces the memory needed to store a scene graph, since Java 3D avoids creating a copy in some cases. However, using this feature prevents Java 3D fromcreating display lists (unless the scene graph is compiled), so rendering performance can suffer in some cases. It is appropriate if memory is a concern or if the geometry is writable and may change frequently. The interleaved format will perform better than the non-interleaved formats and should be used where possible. In by-reference mode, an application should use arrays of native data types; referring to TupleXX[] arrays should be avoided. See also : Part III - Space versus time property.

- Texture by reference and Y-up
Using texture by reference and Y-up format may reduce the memory needed to store a texture object, since Java 3D avoids creating a copy in some cases. When a copy of the by-reference data is made in Java3D, users should be aware that this case will use twice as much memory as the by copy case. This is because Java3D internally makes a copy in addition to the user's copy to the reference data. Currently, Java3D will not make a copy of texture image for the following combinations of BufferedImage format and ImageComponent format (byReference and Yup should both be set to true):

On both Solaris and Win32 OpenGL:

BufferImage Format	**ImageComponentFormat**
BufferedImage.TYPE_CUSTOM of form 3BYTE_RGB	ImageComponent.FORMAT_RGB8 or ImageComponent.FORMAT_RGB
BufferedImage.TYPE_CUSTOM of form 4BYTE_RGBA	ImageComponent.FORMAT_RGBA8 or ImageComponent.FORMAT_RGBA
BufferedImage.TYPE_BYTE_GRAY	ImageComponent.FORMAT_CHANNEL8

On Win32/OpenGL:

BufferImage Format	**ImageComponentFormat**
BufferedImage.TYPE_3BYTE_BGR	ImageComponent.FORMAT_RGB8 or ImageComponent.FORMAT_RGB

On Solaris/OpenGL:

BufferImage Format	**ImageComponentFormat**
BufferedImage.TYPE_4BYTE_ABGR	ImageComponent.FORMAT_RGBA8 or ImageComponent.FORMAT_RGBA

- Drawing 2D graphics using J3DGraphics2D
The J3DGraphics2D class allows you to mix 2D and 3D drawing into the same window. However, this can be very slow in many cases because Java 3D needs to buffer up all the data and then composite it into the back buffer of the Canvas3D. A new method, drawAndFlushImage, is provided to accelerate the drawing of 2D images into a Canvas3D. To use this, it is recommended that an application create their own BufferedImage of the desired size, use Java2D to render into their BufferedImage, and then use the new drawAndFlushImage method to draw the image into the Canvas3D.

This has the advantage of only compositing the minimum area and, in some cases, can be done without making an extra copy of the data. For the image to not be copied, this method must be called within a Canvas3D callback, the specified BufferedImage must be of the format BufferedImage.TYPE_4BYTE_ABGR, and the GL_ABGR_EXT extension must be supported by OpenGL. If these conditions are not met, the image will be copied, and then flushed.

The following methods have also been optimized : all drawImage() routines, drawRenderableImage(), draw(Shape s), fill(Shape s), drawString(), drawLine() without strokeSet to copy only the minimum affected region without the restriction imposed in drawAndFlushImage method.

- Application Threads
The built-in threads support in the Java language is very powerful but can be deadly to performance if it is not controlled. Applications need to be very careful in their threads usage. There are a few things to be careful of when using Java threads. First, try to use them in a demand driven fashion. Only let the thread run when it has a task to do. Free running threads can take a lot of CPU cycles from the rest of the threads in the system - including Java 3D threads. Next, be sure the priority of the threads are appropriate. Most Java Virtual Machines will enforce priorities aggressively. Too low a priority will starve the thread and too high a priority will starve the rest of the system. If in doubt, use the default thread priority. Finally, see if the application thread really needs to be a thread. Would the task that the thread performs be all right if it only ran once per frame? If so, consider changing the task to a Behaviour that wakes up each frame.

- Java 3D Threads
Java 3D uses many threads in its implementation, so it also needs to implement the precautions listed above. In almost all cases, Java 3D manages its threads efficiently. They are demand driven with default priorities. There are a few cases that don't follow these guidelines completely.

- Behaviours

One of these cases is the Behaviour scheduler when there are pending WakeupOnElapsedTime criteria. In this case, it needs to wake up when the minimum WakeupOnElapsedTime criteria is about to expire. So, application use of WakeupOnElapsedTime can cause the Behaviour scheduler to run more often than might be necessary.

- Sounds
The final special case for Java 3D threads is the Sound subsystem. Due to some limitations in the current sound rendering engine, enabling sounds cause the sound engine to potentially run at a higher priority than other threads. This may adversely affect performance.

- Threads in General
There is one last comment to make on threads is general. Since Java 3D is a fully multithreaded system, applications may see significant performance improvements by increasing the number of CPUs in the system. For an application that does strictly animation, then two CPUs should be sufficient. As more features are added to the application (Sound, Collision, etc.), more CPUs could be utilized.

- Switch Nodes for Occlusion Culling
If the application is a first-person point of view application, and the environment is well known, Switch nodes may be used to implement simple occlusion culling. The children of the switch node that are not currently visible may be turned off. If the application has this kind of knowledge, this can be a very useful technique.

- Switch Nodes for Animation
Most animation is accomplished by changing the transformations that effect an object. If the animation is simple and repeatable, the flip-book trick can be used to display the animation. Simply put all the animation frames under one switch node and use a SwitchValueInterpolator on the switch node. This increases memory consumption in favour of smooth animations.

- OrderedGroup Nodes
OrderedGroup and its subclasses are not as high performing as the unordered group nodes. They disable any state sorting optimizations that are possible. If the application can find alternative solutions, performance will improve.

- LOD Behaviors
For complex scenes, using LOD Behaviors can improve performance by reducing geometry needed to render objects that don't need high level of detail. This is another option that increases memory consumption for faster render rates.

- Picking
If the application doesn't need the accuracy of geometry-based picking, use bounds-based picking. For more accurate picking and better picking performance, use PickRay instead of PickCone/PickCylnder unless you need to pick line/point. PickCanvas with a tolerance of 0 will use PickRay for picking.

- D3D user only
Using Quad with Polygon line mode is very slow. This is because DirectX doesn't support Quad. Breaking down the Quad into two triangles causes the diagonal line to be displayed. Instead Java 3D draws the polygon line and does the hidden surface removal manually.

Automatic texture generation mode Eye Linear is slower because D3D doesn't support this mode.