

# Honors Capstone Project

## Adding JOGL to Spiegel

Edward Dale

under the supervision of

Hans-Peter Bischof and Joe Geigel

May 5, 2006

### Abstract

The Spiegel<sup>1</sup> visualization framework allows one to break down a visualization task into a number of blocks that each perform an individual task. It currently uses Java3D<sup>2</sup>, an object-oriented wrapper around a number of graphics APIs, to do all rendering. This project will add a second rendering method using JOGL<sup>3</sup>, a thin Java wrapper around OpenGL. It will be tested by reimplementing existing visualization applications written in Spiegel and measuring the memory overhead and execution time of each method. Also, Spiegel will be further modularized to allow it to be deployed more easily using .JAR files.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>The JOGL Camera</b>	<b>3</b>
<b>3</b>	<b>Re-implementing Atoms3D</b>	<b>4</b>
<b>4</b>	<b>Modularizing</b>	<b>5</b>
<b>5</b>	<b>Results</b>	<b>6</b>
<b>6</b>	<b>Future Work</b>	<b>10</b>

---

<sup>1</sup><http://www.cs.rit.edu/~grapecluster/>

<sup>2</sup><https://java3d.dev.java.net/>

<sup>3</sup><https://jogl.dev.java.net/>

# 1 Introduction

Spiegel provides a pipeline architecture (Figure 1) that allows one to write blocks that accomplish little tasks and can be connected together to accomplish a greater task. At the end of this pipeline is a Camera block that renders the visualization. The input to the Camera block is in a format that the rendering subsystem can understand. This implies that both the Camera block and its input (a Visualizer block) are tied to the rendering subsystem. Currently, all Camera and Visualizer blocks are written for Java3D.

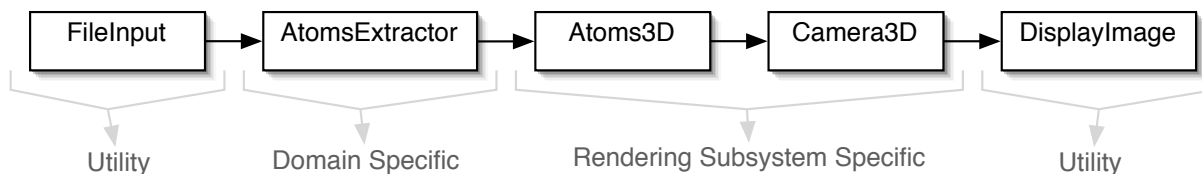


Figure 1: The Spiegel Pipeline

Java3D is an object-oriented wrapper around a number of graphics APIs. Rendering is done by building a scene graph modeling the scene, which is then traversed. Nodes in the tree include transformations, geometry, and appearance. This results in a potentially very large number of objects being created and added to the scene graph. An molecule with 1500 atoms would require 1500 transformations to position each atom, 1500 geometries to model the spheres, and 1500 appearances to modify the color and material of the atoms. These numbers skyrocket when visualizing a galaxy with tens of thousands of objects. There is the potential to share instances of these nodes, but this caused artifacts in the final rendering of the molecules. The output of Java3D Visualizers is a `BranchGroup` object which is a pointer to the root of a scene graph branch.

In recent years, there has been a desire to create an additional 3D API for Java that provides better access to the graphics hardware without the scene graph requirement. JSR (Java Specification Request) 231 is the result of that desire and JOGL (Java Bindings for OpenGL) is the current reference implementation for the specification. JOGL is a thin wrapper about OpenGL. OpenGL itself is modeled as a state machine and this shows in JOGL. Rendering in JOGL is done by modifying the state of the GL object using method calls. There are no additional objects necessary to do the rendering. In most cases, it would still be desirable to model the scene using some set of objects and JOGL allows one to do this in the best way possible.

This project will create a new Camera block that uses JOGL to render scenes. Decoration blocks will be created similar to those that are already available for the Java3D Camera. Also, existing Spiegel visualization applications implemented with Java3D will be re-implemented using the new JOGL camera. Finally, changes will be made to Spiegel itself to modularize it, allowing it to be deployed with only the functionality needed.

## 2 The JOGL Camera

One of the main benefits of using JOGL for rendering is that it removes restrictions on how a scene must be modeled. There does, however, have to be a common interface that the JOGL Camera expects. Also, because of the state machine nature of JOGL, all method calls must be made on a single GL object that the Camera creates. These two requires inspired the design decision to use a callback pattern. The JOGL Camera has a single input that accepts multiple `Drawable` objects (Figure 2). To draw the scene, the Camera iterates through all of these objects and calls the `draw` method, passing it the GL object that it holds.

```
public interface Drawable {
    public void draw( GL gl );
}
```

Figure 2: The Drawable Interface

The creation of the GL object can be accomplished two different ways. A pbuffer (hardware-accelerated offscreen rendering) can be created which does all rendering into memory. This pbuffer can then be written to an image file and sent to an image display window at the absolute end of the pipeline. This has the advantage that it works the same as the Java3D camera. The downside is that this is one of the more experimental parts of the JOGL API and therefore may cause problems at some point.

The GL can also be retrieved by creating a `GLJPanel` or `GLCanvas`, which can be directly embedded into a Swing GUI. With this approach, the JOGL Camera is the last block in the pipeline. The downside of this method is that the current implementation causes the `init` method to be called numerous times, resulting in extra computation.

Regardless of how the GL is created, a JOGL Camera implementation has the same three basic methods: `init`, `display`, and `annotate`. The `init` method creates the GL object using one of the methods above. It then uses the block parameters to set up the projection and modelview matrices for correct 3d viewing. JOGL can also optionally wrap the GL object with one that debugs or traces the method calls made. Using the debug version, if any errors occur when making OpenGL method calls, an exception will be thrown. This has been immensely useful during the development of new code. The trace version will print out every OpenGL method call as it happens. In general, this has been far too verbose.

The `display` method simply loops through all of the `Drawable` inputs and calls their `draw` method. It also annotates the output and optionally writes the image to a file. JOGL provides a `Screenshot` class that makes it trivial to get an image of the rendering.

The final interesting method of the JOGL Camera is `annotate`. It displays the camera position

and name as well as any additional annotations that were given to the Camera as parameters. This takes some work because a new orthogonal projection must be created to be drawn in.

Initially, some lighting and enabling of OpenGL options was included in the JOGL Camera. However, after actually implementing some blocks that used the Camera, it was determined that it would be better if each block set up it's own lighting and options. This has the disadvantage that each block will spend some time resetting options to what they may have already been, but saves the trouble of relying on any kind of external state.

The implementation of the Cube and Axis decoration blocks was a direct copy of the existing Java3D versions. Instead of creating a `VertexArray` with the vertices, a `GL_LINES` primitive was created. The one optimization that was made is that the OpenGL calls are cached in a display list. The first time the decoration is rendered, a display list is created. Subsequent calls just invoke the display list. This is potentially a way to accelerate any JOGL blocks, at the expense of memory.

### 3 Re-implementing Atoms3D

As shown in Figure 1, the only rendering subsystem specific blocks to a visualization are the Visualizer and the Camera. The Camera was re-implemented using JOGL in the previous section, so in this section the Atoms3D Visualizer block will be re-implemented. This is possible because of the separation of concerns that Spiegel enforces.

The input to the current `Atoms3D` block is an `AtomIDMap`, which is a map of `Integers` to `PdbAtoms` containing all of the information about an atom. In order to make the JOGL version a drop-in replacement, it also takes an `AtomIDMap`. Internally, it iterates through this map and builds a list of `Sphere` objects, which contain all of the necessary information to render a sphere: position, color, and size. This is where the benefit of JOGL over Java3D really shows. Instead of having to model the molecule as a scene graph, it can be modeled as a list of spheres, the barest possible representation.

An interesting consequence of the callback design of the JOGL Camera is that the visualizer blocks (ie `Atoms3D`) can implement `Drawable`. The design of `Atoms3D` is such that in the update method, it builds the sphere list. This is stored in an object-level variable and `this` is put on the output and sent down the pipeline to the JOGL Camera. Once in the Camera, the `draw` method of `Atoms3D` is called, which iterates through the sphere list and makes the necessary OpenGL calls for each sphere.

The only snag in the implementation of the JOGL `Atoms3D` block was in the lighting. Lighting was enabled to get the shading of the individual atoms correct. The lights were positioned and adjusted correctly, but the molecule was appearing completely white (Figure 3). The problem was eventually determined to be that although the spheres were being scaled down to size, their normals were not being scaled, so they were being lit far too much. This was resolved by enabling `GL_RESCALE_NORMAL`, which scales normals by a scaling factor derived from the modelview matrix.

Once this was done, everything fell into place and the molecule appeared correctly (Figure 4). It is anticipated that there will be more obscure problems such as this than with Java3D because of the level that JOGL operates at.

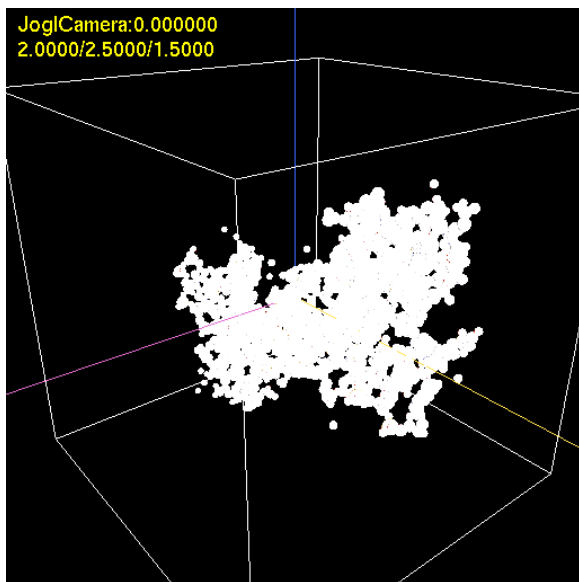


Figure 3: White molecule

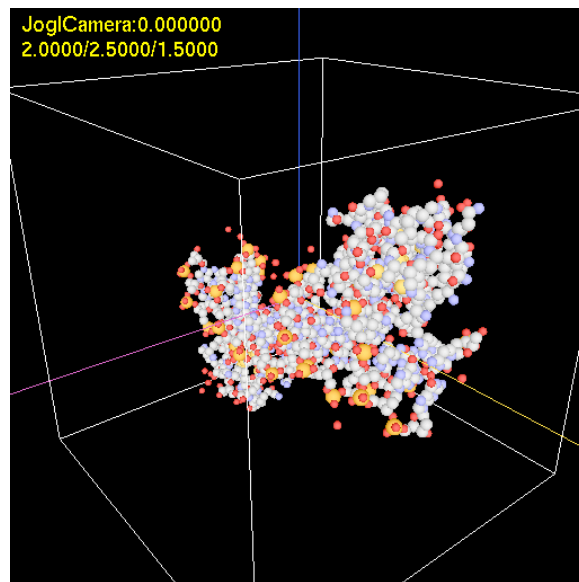


Figure 4: Colored molecule

## 4 Modularizing

Only partially related to the JOGL work is the work that was done to modularize Spiegel, allowing it to be partially deployed. Spiegel currently loads all its plugin from the `spiegel.viewcontrol.functions.plugins` package from the filesystem. This means that if someone wants to develop a plugin, it must be put into this package and Spiegel must be run directly, not from within a .JAR file. This is less than optimal in case third parties wish to develop plugins for the system. The goal of this part of the project was to make changes to Spiegel and then develop a new set of plugins for the new system without touching the Spiegel code.

As mentioned, Spiegel loaded plugins from the filesystem. It discovered plugins by listing the contents of a the `spiegel/viewcontrol/functions/plugins` directory. This system breaks if plugins can be in arbitrary packages and/or located in .JAR files. To resolve the discovery problem, the `PluginLoader` now finds plugins by examining a file that contains a list of plugin classes. In addition, it looks at a system property called `spiegel.plugins`. This means that plugins can be added at runtime by setting a system property. With the name of the plugin classes, the `PluginLoader` uses `Class.forName` to load the plugin class and create an instance of it. This requires that the plugin classes be located in the classpath.

To allow Spiegel to be executed from a .JAR file, a package called One-Jar<sup>4</sup> was used. One-Jar allows an application, with all of its dependencies, to be packaged into a single .JAR. Normally, dependent .JAR files must be in the classpath, not in the .JAR file. One-Jar looks in the `libs` directory of the application .JAR file and loads any .JAR files it finds into the classpath. It then runs the main class of the `main.jar` file in the `main` directory. This solves the problem of having to have plugin classes in the classpath.

Spiegel can only load scripts from the filesystem, so another function of One-Jar was used to copy some files into the filesystem upon execution. Using this, the `Scripts` directory is copied in to the filesystem so Spiegel can access it.

To facilitate the building of the application .JAR file, an Apache Ant<sup>5</sup> build script was created that can build the system and create a .JAR file, including any additional packages (ie JOGL). Using the build script, one needs only to run `ant` from a fresh CVS checkout to produce a `spiegel.jar` file that can be executed anywhere. In addition, Jar Bundler<sup>6</sup> was used to allow the build script to also produce a Mac OS X .APP file that has an icon and can be executed from within Finder.

With the build script and Spiegel changes in place, develop was begun on the JOGL code. The JOGL portion of the project was developed in an separate CVS repository with the Spiegel classes on the classpath to enforce a separation of the two modules. In addition, a separate CVS repository was created for the atom plugins including the old plugins and the new JOGL plugin. In this way, Spiegel can optionally be deployed with JOGL or atom support by simply including the `jogl.jar` or `atoms.jar` files in the main `spiegel.jar` file.

## 5 Results

For a visual comparison of the two Atoms3D blocks, the 1D66 molecule from the Protein Databank<sup>7</sup> was visualized using each. The results from Java3D is available in Figure 5 and the results from JOGL is in Figure 6. Note that although the JOGL molecule is significantly lighter than the Java3D version, this doesn't have an affect on the correctness of the visualization.

In the subsequent experiments, subsets of size  $N$  of the original 1D66 molecule were used for testing.

An attempt was made to quantify the memory of each method, however it was impossible to get a consistent measurement without much more complex tools. Some attempt to infer the relative memory footprint will be made from the execution times below.

The execution time of the Atoms3D `update` method was measured by calling the `System.nanoTime()` method before and after its execution. The results of this measurement can be seen in Figure 7.

---

<sup>4</sup><http://one-jar.sf.net/>

<sup>5</sup><http://ant.apache.org/>

<sup>6</sup><http://informagen.com/JarBundler/index.html>

<sup>7</sup><http://www.rcsb.org/>

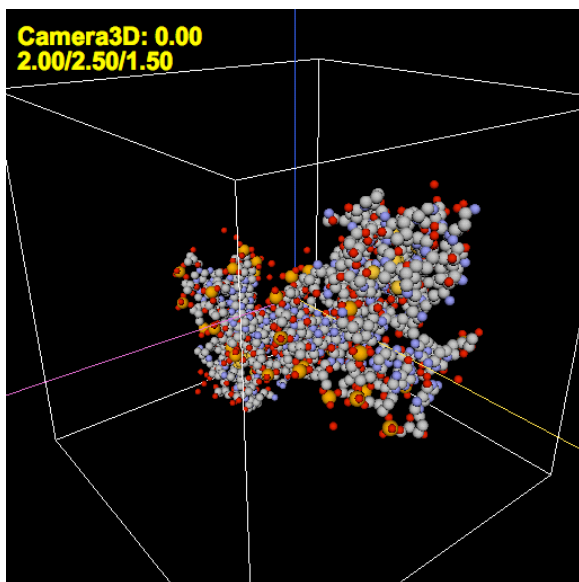


Figure 5: Java3D molecule

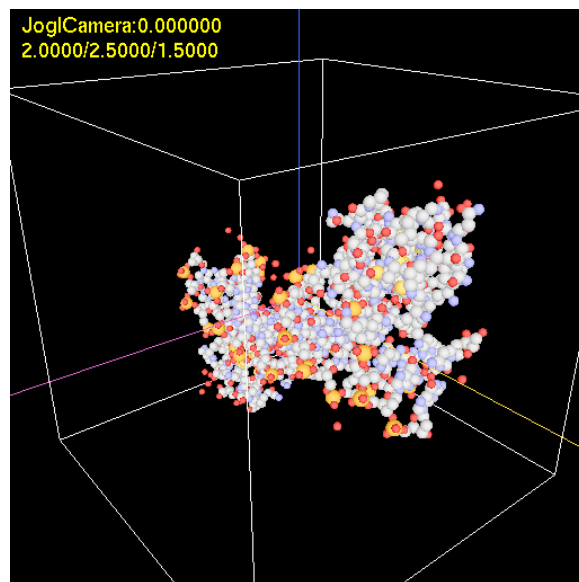


Figure 6: JOGL molecule

There is a drastic difference between the execution time of the two rendering methods. JOGL has a much lower, almost zero, slope compared to Java3D. This is consistent with the hypothesis that the scene graph approach of Java3D requires much more time and possibly memory compared to a more precise model such as the sphere list used by JOGL in this experiment. The large jump in the Java3D graph between 50 and 100 atoms can possibly be attributed to the Java garbage collector being executed. In this case, it would also seem that Java3D has a larger memory footprint because the garbage collector is consistently executed at a much lower number of atoms than JOGL.

The marked performance difference between JOGL and Java3D may seem like a lot, but this is code that will typically only be executed once per molecular visualization. After this code is executed, the camera can be manipulated to spin and rotate the molecule without causing Spiegel to re-execute the method. In a visualization where the visualizer is executed multiple times, this performance hit may be more detrimental.

The total execution time of each Camera's `update()` method was also measured using the same method. To perform this test, an `Orbit` block was connected to the Camera to cause it to orbit the molecule continuously and this was allowed to run for  $\sim 40$  timesteps for each input size and render method. The mean and standard error of each execution is presented in Figure 8. The execution times are much closer for smaller molecules, but Java3D starts to rise much faster than JOGL at around 250-500 atoms. This further confirms that JOGL is faster than Java3D for this experiment.

Also of note is the larger error in the Java3D measurements. This could possibly also be attributed to more frequent garbage collection causing more frequent execution time spikes. As mentioned previously, more frequent garbage collection would be indicative of a larger memory

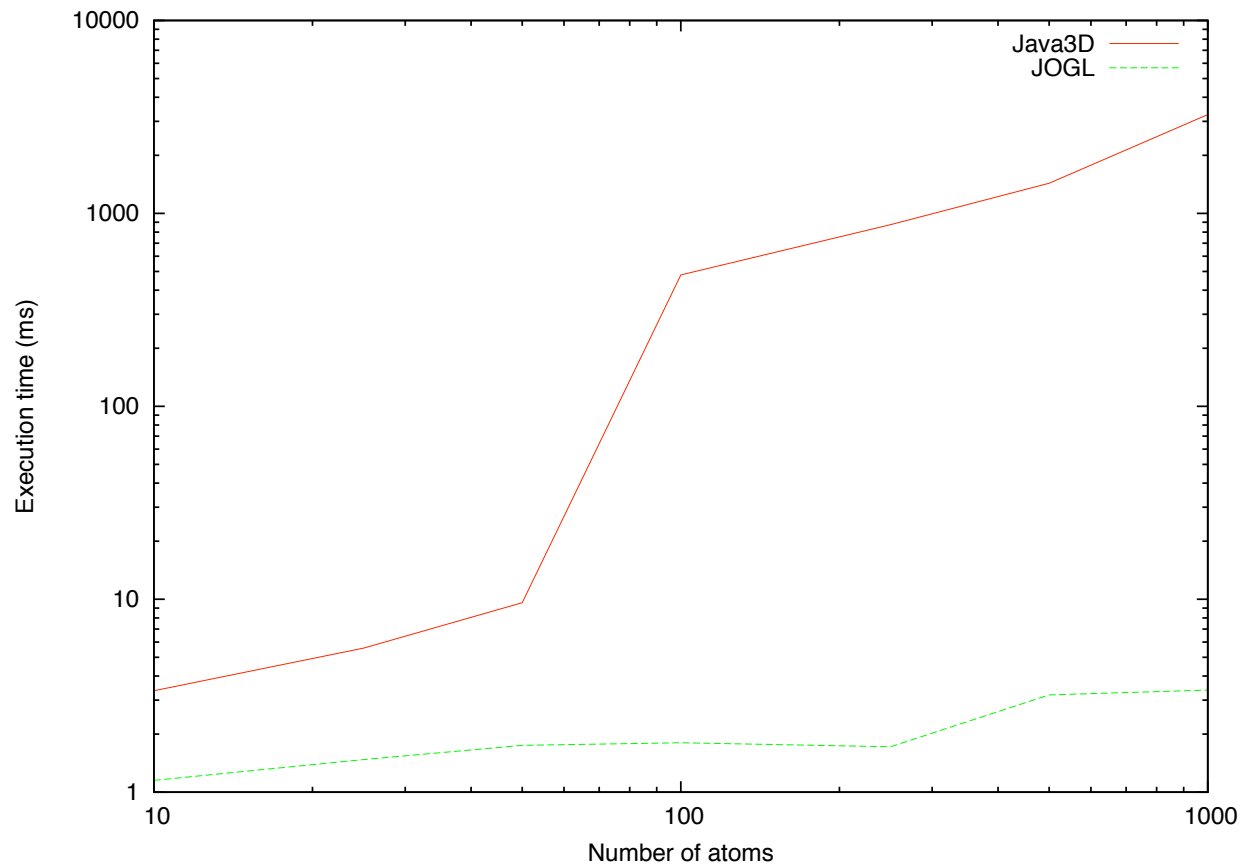


Figure 7: Atoms3D update() timing

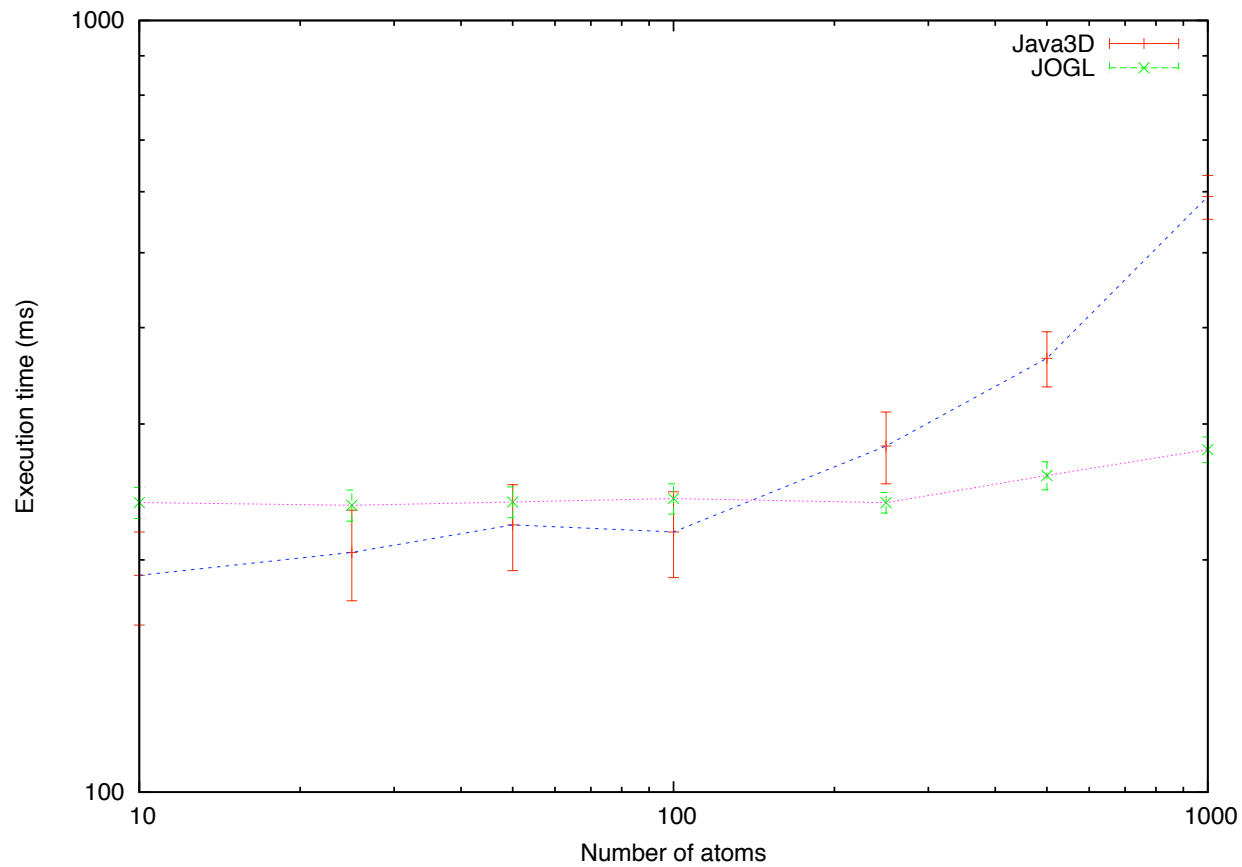


Figure 8: Camera update() timing

footprint.

From the results shown and discussed above, it can be inferred that Java3D has a higher memory footprint than JOGL for the simple model that has been implemented here. Moreover, the JOGL executes asymptotically faster than Java3D In Spiegel, where galaxies containing tens or hundreds of thousands of particles are regularly visualized, the memory savings provided by JOGL is likely the more important advantage, but the potential time savings could be very useful also. A more conclusive result could be determined using a profiler or other more advanced tool, however, it is believed that the methods used above provide enough information to draw conclusions.

## 6 Future Work

Spiegel has recently started using threads much more. Each block now executes in it's own thread. This presents the potential problem of multiple threads trying to change the state of the OpenGL state machine. This problem has not been encountered yet, but it is very possible once more complex visualization applications start being developed with the JOGL Camera. At some point, there may have to be some kind of locking mechanism to prevent multiple threads from affecting each other.

A JOGL plugin superclass could be created that adds the capability of any `Drawable` to be accelerated by caching it's OpenGL calls in a display list. This capability could also be put in the JOGL Camera. The two types of JOGL Cameras (`pbuffer` and `GLJPanel/GLCanvas`) share a large amount of duplicate code. Ideally, a `JoglCameraBase` could be created to extract this commonality.

The work on modularizing Spiegel has really just begun. Ideally, Spiegel would be completely modularized with only the visualization framework itself in it's own module. Modules could be created for the Java3D blocks (camera and decoration) and the existing astrology blocks. Potentially, the only plugins that would be in the `spiegel` module would be utility plugins such as `ConstantString` and `Orbit`. This would allow the core visualization framework to be stabilized at some point and a public API stabilized.