A Performance Evaluation of the Visualization Tool

Author: Sami Matilainen

Purpose

The purpose of this evaluation is to determine the source and impact on system performance of running the Visualization Interface with 16 C-routers.

Complexity Analysis

The total number of nodes N is a function of C (the number of C-routers):

$$N(C) = 28C$$

The total number of "xnode" and "ynode" nodes is also a function of C:

$$Xnode(C) = 16C$$

$$Ynode(C) = 16C$$

In the following I'm going to assume no quads are used.

In all of my tests I have used 6 categories and 5 dimensional data for the ynodestat data set. This means that the bars for the ynodestat data set are made up of 30 individual cylinder objects (note that this is identical to using ten categories and 3 dimensions as has been requested on the meetings). Since each cylinder is a 9-sided polygon (8 sides and top, no bottom) with a total of 24 triangles, you have 30*24=720 triangles for each xnodenode.

Additionally the xnodestat data set uses one-dimensional data with the same 6 categories as the ynodestat data. The number of triangles per node is then 24 as it consists of only one cylinder.

The node markers themselves are rendered as spheres with 112 triangles. The donut-shaped C-routers are an exception but there are so few of them so they can be approximated as having the same amount of triangles as the nodes.

The globe makes up a total of 8064 triangles. The Skybox is made up of 12 triangles and the quad used in the 2D-view is made up of 2 triangles.

With this we can calculate the total number of triangles in the globe view with the following formula:

$$TriG(C) = 15040C + 8076$$

In the plane view the equation looks like this:

$$TriP(C) = 15040C + 14$$

Seeing that the constants in both equations are negligible we may conclude that for each C-router that we add we have to push an extra 15000 triangles thru the OpenGL pipeline for each frame. To make things easier I will assume that each triangle is built up of 3 vertices. An approximation to the amount of vertices in the scene can then be given as:

$$Vert(C) = 45000C$$

In my tests I have used 4, 16 or 32 C-routers. With a 30 Hz refresh rate this amounts to 5.4 million, 21.6 million and 43.2 million vertices per second being pushed thru the pipeline. This can be compared to the GeForce 6600 GT (released in 2004), which is claimed to be able to handle over 300 million vertices per second (http://www.nvidia.com/page/geforce_6600.html).

Obviously there are other things that affect the performance of an OpenGL application such as lighting and shading. I have also not talked about the lines that make up the links/relations between the nodes. However, seeing that a 4 year old graphics card (which can be bought for about 1000 SEK today) can handle 10 times as many vertices as this application requires in the worst case I think we can safely conclude that the Visualization Tool is not limited by GPU performance.

One thing to note, however, is that the OpenGL pipeline doesn't operate solely on the GPU, which means that the CPU is still a possible bottleneck for rendering the 3D scene.

Previous Work

The Visualization Tool was tested once before (Prototype 5) and it was at that time discovered that the main bottleneck was the AnimatorTimer thread, which is part of the JOGL library. I created my own version of the AnimatorTimer and was able to reduce the CPU time required by the software by 65-70%. The optimized and cleaned up version of the Visualization Tool was named v 0.1.

Setup

I started the evaluation by doing some quick preliminary tests to determine what variables I should examine further and set up 3 tests.

For the first test I wanted to examine the load on the CPU by those parts of the software that are not part of rendering the 3D-scene. This could be achieved by commenting out the AnimatorTimer thread as it is the part which schedules calls to the Renderer.display() method which in turn renders the entire scene. It's important to understand that the rest of the software is completely unaffected by these changes, so the load on the system when running without the AnimatorTimer thread should be caused solely by the logic that handles the buffering, parsing and encoding of incoming data. No scheduled updates can happen, however, by resizing the window or the view panel it is still possible to call the Renderer.display() method and update the scene.

The second step was to comment out only Renderer.display() to determine how much time was spent in the sheduler (AnimatorTimer) itself, which was discovered to be the main bottleneck last time.

Both of these tests were run with the TICTest server running and with the TICTest server turned off. The TICTest server was configured to send as many messages as possible within the limits of standard process scheduling. The TICTest server sends one message for each xnode and ynode node in the system for every loop that it does and updates all values each time.

The last two tests were designed to test how big of an impact the amount of polygons had to the CPU.

Lastly, the results were also compared to the subjective experience of lag when navigating the scene and to the total CPU load on the system.

The tests were run on two laptops with the following specs:

MacBook Pro:

Intel Core 2 Duo 2.4 Ghz
GeForce 8600M GT 256MB

HP Compaq nw8240:

Intel Core 2 Duo 2.0 Ghz
ATI Mobility Fire GL V5000 128MB

The MacBook Pro was hooked up to an external monitor so the load on the GPU was higher than that on the HP laptop. The Mac also ran the visualization Tool In a larger window. I tried best I could to limit the amount of background processes on both machines by not having any other applications open that were not needed for the tests.

The software used for the tests was a slightly modified version of the 0.7 release version.

Results

It was noted that the process managers in Mac OS X and Windows Vista reports a different amount of threads for the Visualization Tool. The Visualization Tool apparently uses 21 threads under Mac OS X and 20 threads under Windows Vista. This might be due to differences in the implementation of the Java API and/or JOGL.

In all tests, the load on the CPU is given as a percentage. 100% load would indicate that both processor cores would be utilized to their maximum. The process monitor tools on the Mac OS X reports the CPU load as a percentage alue where 200% load would indicate the same thing. In the graphs below I have adjusted for this so that all values are indicated with 100% being the maximum total load on both cores.

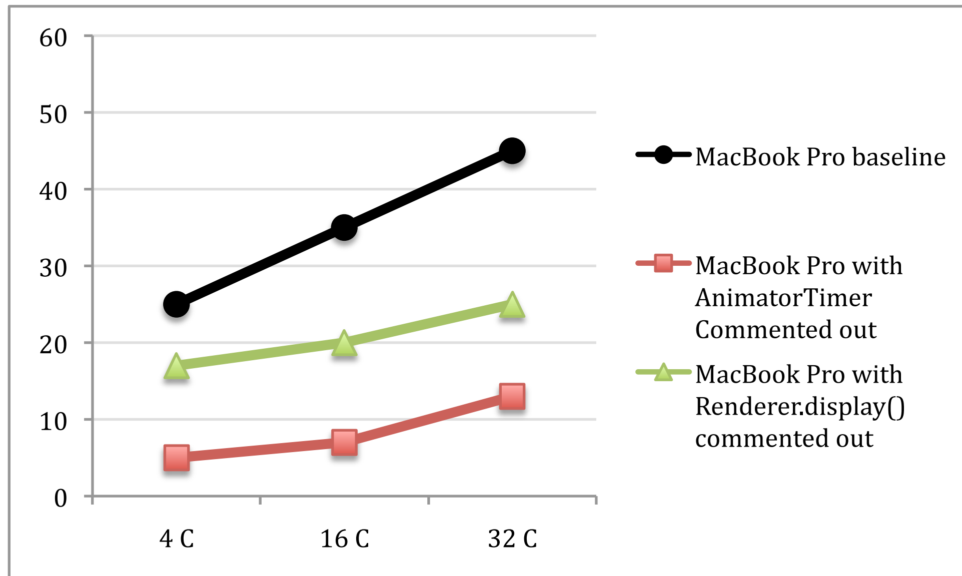
The baseline is the result of running the Visualization Tool with no modifications unless otherwise noted. The baseline tests on the MacBook Pro confirm that the load increases as a function of the number of C-routers. This increase seems to be linear but more samples are required to be able to claim this with certainty.

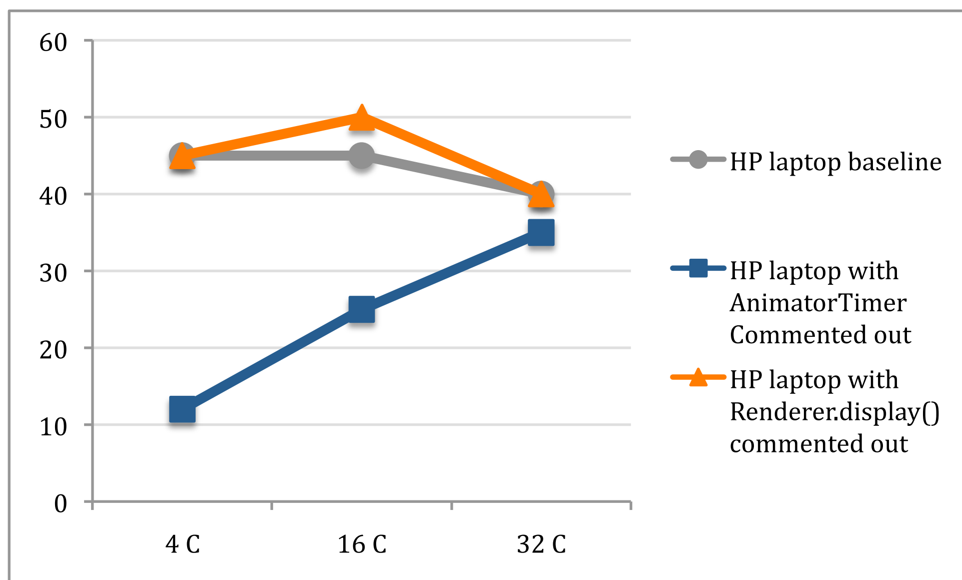
Interesting to note is that the baseline test on the HP laptop is constant around 40-50% CPU load. When investigating this further it would appear that the thread scheduling is much worse on the Windows Vista/XP implementation of JOGL and/or the JVM as the cores are not evenly loaded when running the Visualization Tool. On the Mac the load is perfectly balanced on the two cores. If we assume that the entire software is operating on one core only it means that the software is hitting the roof already with 4 C-routers. The subjective experience of lag also gives hints of this. We will see in all of the tests that this is indeed the limiting factor on the HP laptop.

It should also be noted that while the Visualization Tool is reporting a load of 40-50% the overall CPU load is reported as 80-100% and you can clearly see that there is an increase in the System process by about 20%. Then when you add the TICTest server you get to a full 100% on the HP laptop. The MacBook Pro is never taxed at 100% even when running both the TICTest server and 32 C-routers and several applications in the background. The experience of lag is slightly increased on the Mac in this type of situation.

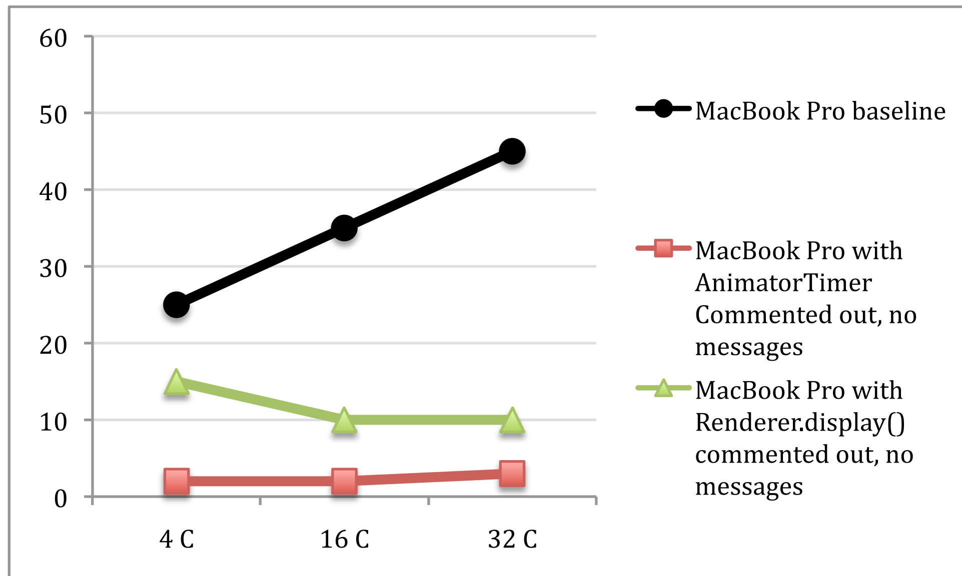
Looking at the first test clearly shows that by commenting out the AnimatorTimer have a large impact on the load on the CPU. This is confirmed by both the Mac and HP tests. This means that the AnimatorTimer and the Renderer

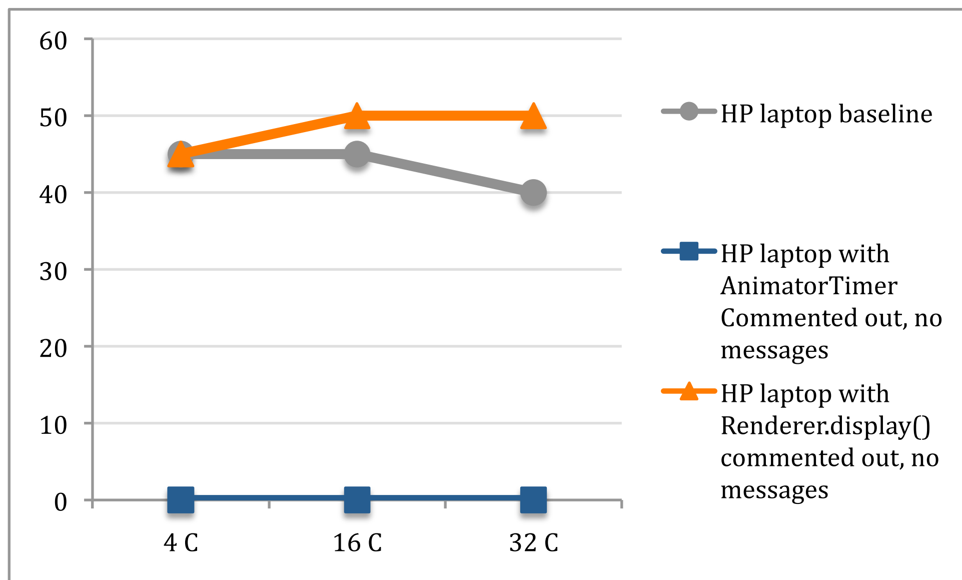
are still the largest bottleneck of the software. Notable is that the scheduling itself is such a large contributor as seen by the tests with the Renderer.display() method commented out.





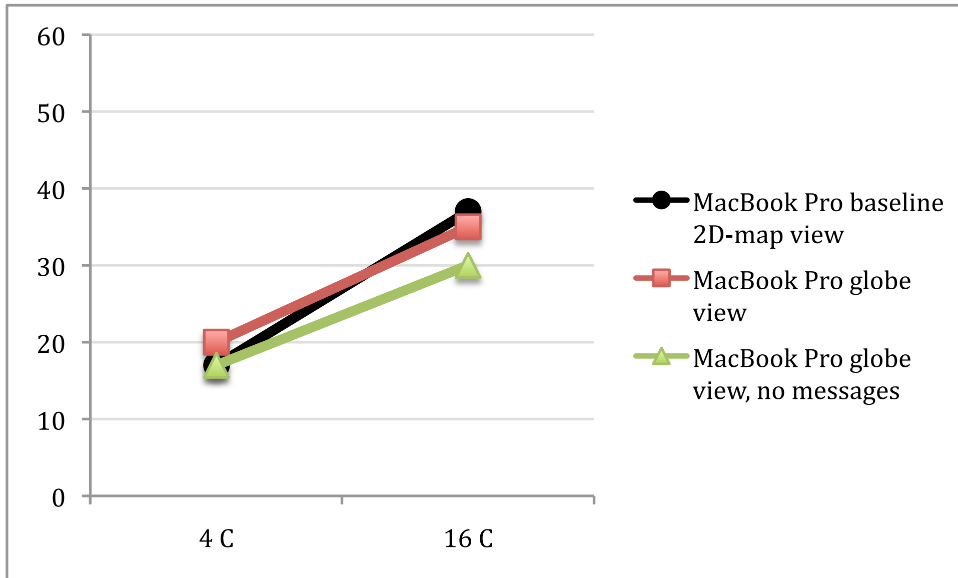
The following two charts show the same tests as run with the TICTest server not running.

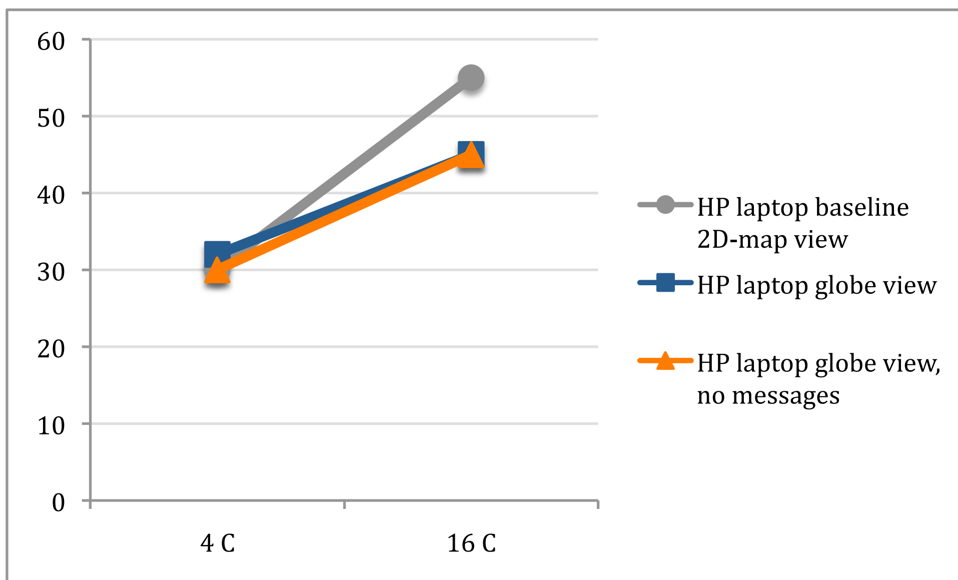




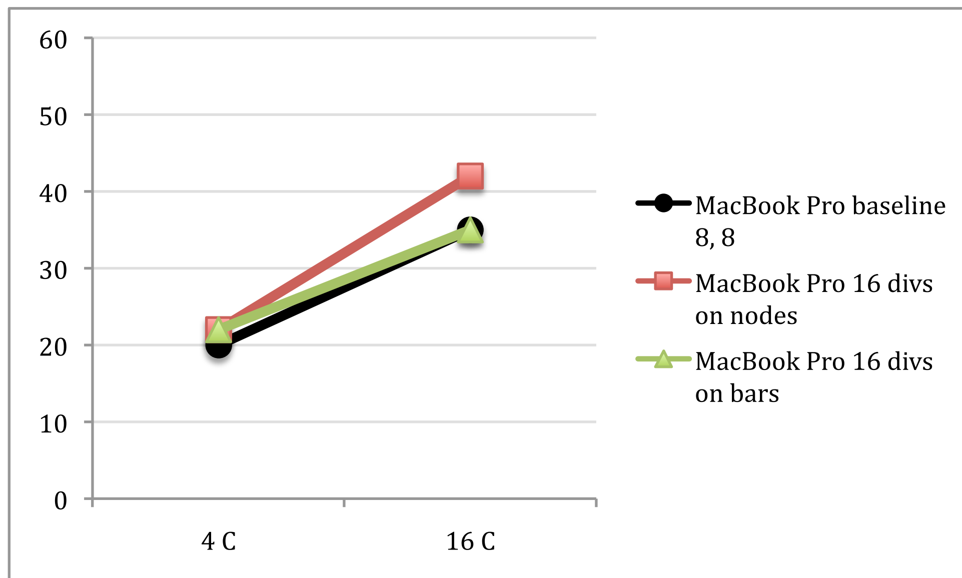
Here the result of turning of AnimatorTimer is even larger and the HP laptop is showing figures around 0%! This means that the buffering, parsing and storing of incoming messages has a marginal impact on CPU load.

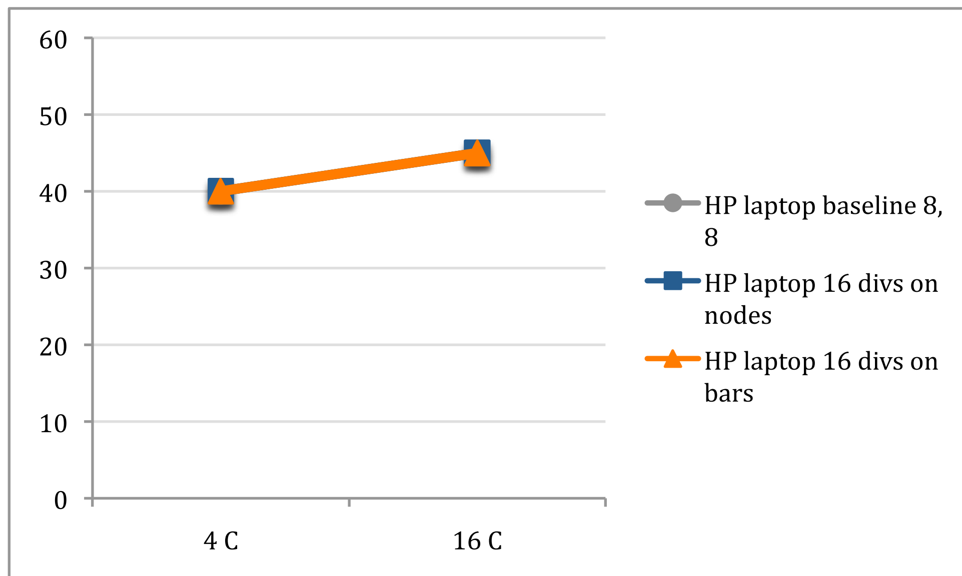
As expected, the globe view has no significant effect on the CPU load.





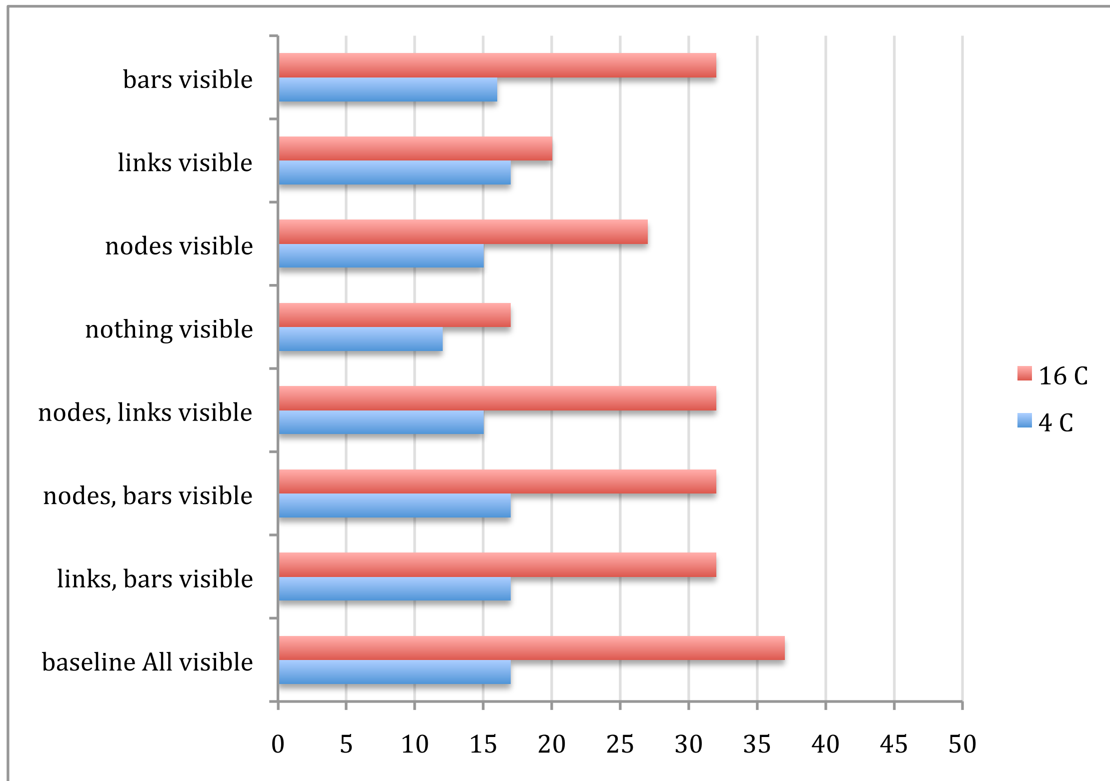
By varying the amount of “divs” (slices/stacks in primitives) I was able to increase the amount of polygons drastically without increasing the number of nodes. Interesting to note is that this increase in polygons had very little impact on the overall CPU load. This verifies what we were able to analytically conclude in the analysis that the amount of polygons in the Visualization Tool is much lower than what can be handled by a decent GPU. The reason I can not make a conclusion about this on the HP laptop is because all results are within the 40-50% region which we had discovered earlier seemed to be the maximum capacity of the CPU.

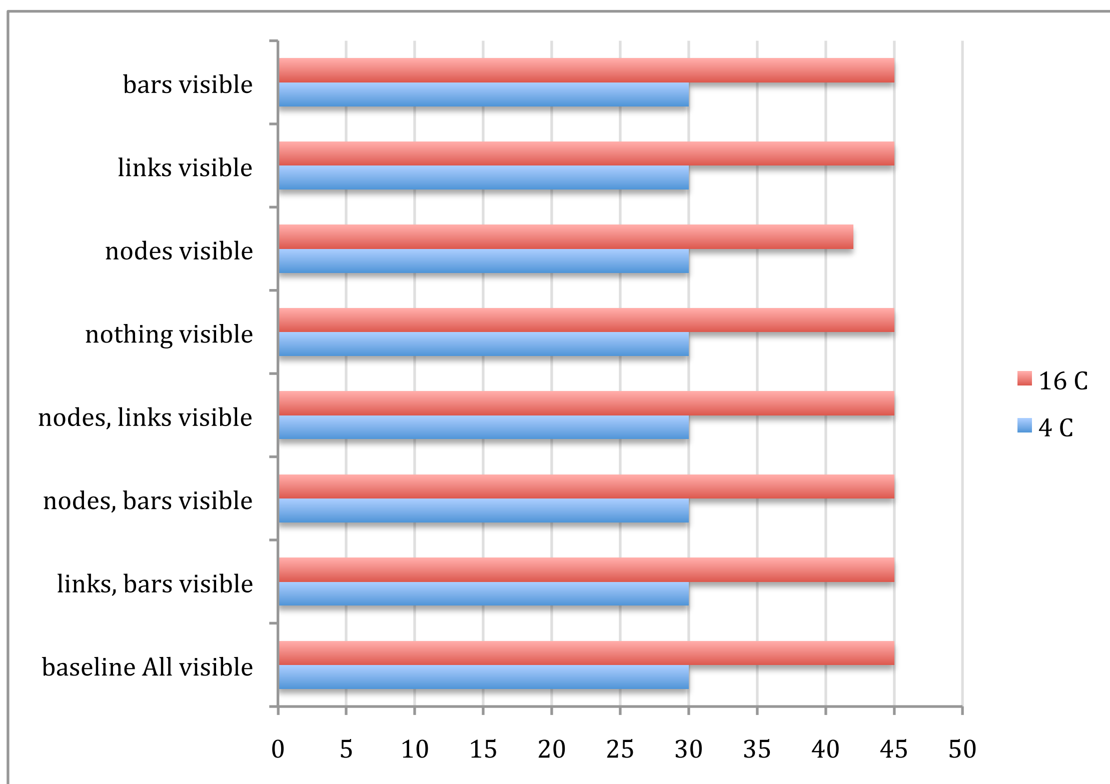




Seeing that the impact is so small I would recommend using 16 divs at least on the bars as it makes them look much smoother.

The final test (Mac results on the top figure) shows that showing/hiding nodes have an impact on performance. The CPU limitations on the HP laptop are quite obvious in these tests.





Summary of Conclusions and Recommendations

The GPU should not be the limiting factor if the PC running the Visualization Tool is relatively new.

The real problem on the HP laptop seems to be the poor thread scheduling which puts almost all load on one CPU core. A better CPU is therefore recommended. This, however, should not be interpreted as a "fact" as it can not be determined exactly how much more CPU time the Visualization Tool would need if it was given more elbowroom to grow.

The impact on the CPU from an increase in polygons is negligible. Therefore I recommend turning on the "eye candy" and setting at least the bar divs to 16 instead of the current 8.

The main bottleneck of the Visualization Tool is still in the software logic. Namely in two places. The AnimatorTimer, which is part of the JOGL API and the Renderer which was written by me, but which makes a large amount of calls to the OpenGL environment and pipeline. It would be very difficult to optimize these further without digging ever deeper into the (horribly complex) JOGL code. Seeing that the OpenGL pipeline doesn't operate solely on the GPU also limits the amount of optimizations that can be done in the Renderer logic.

Buying a better PC for the Visualization Tool is a gamble seeing that the thread scheduling is so poor on the Windows platform. A new MacBook Pro is almost guaranteed to run the software without hiccups.

Further Study

I recommend doing these tests on a more powerful PC. Preferably it should have a more powerful CPU. It would be interesting to run the tests on a machine with a really good CPU and swap in two different GPUs (one weak and one strong) and see if this has any noticeable difference on performance.

A complexity analysis of both the AnimatorTimer and the Renderer logic would be nice. I'm pretty sure the Renderer implementation is quite straightforward with the occasional doubly nested for-loop but I may have missed something important there.