Optimisation of a Graph Visualization Tool: Vizz3D

Johan Carlsson
Abstract

Vizz3D is a graph visualization tool developed at Växjö University. It is used to visualize different aspects of software systems in 3D, based on the static analysis of source code. It can optionally use Java3D or OpenGL as a graphics library.

In order to visualize huge 3D structures performance is very important. This comes from the fact that the structures must be redrawn with no delay when a user interacts with the system. If there were a delay the user would lose the cognitive orientation because his interaction and the feedback would not fit. Vizz3D was not capable to run huge visualizations fast enough, and therefore careful optimisation was essential. Additionally, the Vizz3D tool is just at the beginning of its software life cycle.

For optimisation, JOGL (Java Bindings for OpenGL) was chosen. The extension with a JOGL version was necessary since the GL4Java (OpenGL for Java) wrapper used for the implementation of Vizz3D is no longer supported. JOGL was therefore needed for assuring future maintainability.

The JOGL version of Vizz3D was optimised to be able to visualize huge graphs with acceptable performance. To determine what areas of Vizz3D that consumed most of its resources, the process of profiling were used. The system performance was improved according to several aspects: Computational performance, Scalability, Perceived performance, RAM footprint and Start-up time. The results were then evaluated by using benchmarking techniques. After optimisation, the performance of Vizz3D was improved a lot which led to that huge graphs now could be visualized with acceptable performance.
# Table of contents

1 **INTRODUCTION** ........................................................................................................... 1
  1.1 CONTEXT .......................................................................................................................... 1
  1.2 PROBLEM .......................................................................................................................... 2
  1.3 GOAL .................................................................................................................................. 2
  1.4 CRITERIA ............................................................................................................................ 2
  1.5 OUTLINE .......................................................................................................................... 3

2 **BACKGROUND INFORMATION** ................................................................................. 4
  2.1 THE OPENGL API ............................................................................................................. 4
    2.1.1 Software Implementation ............................................................................................. 4
    2.1.2 Hardware Implementation ......................................................................................... 4
    2.1.3 The OpenGL pipeline ................................................................................................. 5
    2.1.4 The OpenGL State Machine ...................................................................................... 5
    2.1.5 OpenGL wrappers ...................................................................................................... 5
    2.1.6 Alternatives to OpenGL ............................................................................................ 6
  2.2 VIZZ3D ............................................................................................................................ 7
    2.2.1 The Application ........................................................................................................... 7
    2.2.2 The Structure ............................................................................................................. 7
  2.3 SUMMARY ......................................................................................................................... 10

3 **ENSURING THE MAINTAINABILITY OF VIZZ3D** ...................................................... 11
  3.1 CREATING A JOGL SCENE ............................................................................................... 11
  3.2 THE IMPLEMENTATION OF VIZZJOGL ......................................................................... 11
  3.3 SUMMARY ......................................................................................................................... 13

4 **OPTIMISATION STUDY** ................................................................................................. 14
  4.1 MEASURING PERFORMANCE .......................................................................................... 14
    4.1.1 Profiling ...................................................................................................................... 14
    4.1.2 Benchmarking ............................................................................................................. 14
  4.2 PERFORMANCE ASPECTS ............................................................................................... 16
    4.2.1 Computational performance ...................................................................................... 16
    4.2.2 Scalability ................................................................................................................... 16
    4.2.3 Perceived performance ............................................................................................... 16
    4.2.4 RAM footprint ............................................................................................................ 17
    4.2.5 Start-up time .............................................................................................................. 17
  4.3 SUMMARY ......................................................................................................................... 17

5 **OPTIMISATION OF VIZZ3D** ......................................................................................... 18
  5.1 PROFILING ....................................................................................................................... 18
  5.2 COMPUTATIONAL PERFORMANCE ............................................................................... 19
    5.2.1 Display lists in OpenGL ............................................................................................. 19
    5.2.2 State changes in OpenGL ......................................................................................... 20
    5.2.3 Numerical errors in OpenGL ...................................................................................... 21
    5.2.4 Exceptions, Casts and Variables in Java ...................................................................... 21
    5.2.5 Loops, Switches and Recursion in Java ...................................................................... 26
    5.2.6 I/O, Logging and Console Output in Java ................................................................... 27
    5.2.7 Sorting in Java ............................................................................................................ 27
    5.2.8 Appropriate Data Structures and Algorithms in Java ............................................... 27
    5.2.9 Benchmarking the Computational performance ......................................................... 27
  5.3 SCALABILITY .................................................................................................................... 29
    5.3.1 Primitive quality in OpenGL ..................................................................................... 29
    5.3.2 Benchmarking the Scalability ..................................................................................... 29
5.4 PERCEIVED PERFORMANCE ................................................................. 31
  5.4.1 Animator class in OpenGL ........................................................... 31
  5.4.2 Threading in Java .............................................................................. 31
  5.4.3 Benchmarking the Perceived performance ..................................... 32

5.5 RAM FOOTPRINT .................................................................................. 33
  5.5.1 Object creation in Java ....................................................................... 33
  5.5.2 Strings in Java .................................................................................. 34
  5.5.3 Benchmarking the RAM footprint .................................................... 35

5.6 START-UP TIME .................................................................................... 36
  5.6.1 Progressbar update in Java ............................................................... 36
  5.6.2 Benchmarking the Start-up time ....................................................... 36

6 CONCLUSIONS AND FUTURE WORK ................................................. 38

  6.1 CONCLUSIONS ................................................................................... 38
    6.1.1 Maintainability ................................................................................ 38
    6.1.2 Performance ................................................................................... 38

  6.2 FUTURE WORK .................................................................................... 39

REFERENCES .............................................................................................. 40

APPENDIX I – PROFILING OUTPUT......................................................... 41
Figures

FIGURE 2.1 SOFTWARE IMPLEMENTATION OF OPENGL ............................................................. 4
FIGURE 2.2 HARDWARE IMPLEMENTATION OF OPENGL ................................................... 5
FIGURE 2.3 THE OPENGL PIPELINE. .................................................................................. 5
FIGURE 2.4 THE VIZZ3D GUI, JOGL VERSION. .................................................................... 7
FIGURE 2.5 THE VIZZ3D CORE STRUCTURE (BEFORE EXTENSION WITH A JOGL VERSION). ................................................................. 8
FIGURE 2.6 THE VIZZ3D VISUAL GRAPH STRUCTURE. ....................................................... 9
FIGURE 2.7 THE VIZZ3D INTERACTION GRAPH STRUCTURE (BEFORE EXTENSION WITH A JOGL VERSION). ................................................................. 10
FIGURE 3.1 THE VIZZ3D CORE STRUCTURE (AFTER EXTENSION WITH A JOGL VERSION). ........................................................................................................ 12
FIGURE 3.2 THE VIZZ3D INTERACTION GRAPH STRUCTURE (AFTER EXTENSION WITH A JOGL VERSION). ................................................................. 12
FIGURE 5.1 VIZZJOGL PROFILING OUTPUT. ................................................................. 18
FIGURE 5.2 COMPUTATIONAL PERFORMANCE RESULTS .............................................. 28
FIGURE 5.3 SCALABILITY RESULTS. ................................................................................ 30
FIGURE 5.4 RAM FOOTPRINT RESULTS. .......................................................................... 35
FIGURE 5.5 START-UP TIME RESULTS. .......................................................................... 37
1 Introduction

Developing visualization tools requires thinking of the systems performance. In 3D tools used for analysis of source code very much data must be handled due to the complexity. User interaction is essential for navigating and orienting in complex systems and this means that a fast feedback is needed to have a cognitive bridge between the input, result and the displayed system. This is difficult to do since systems are still not powerful enough or very expensive. This means that optimisation is necessary to use the existing and limited resources (computer power and memory) in the most efficient way. The optimisation is often not part of the initial development but can be applied later. There are several strategies for applying optimisations and for locating points that need to be optimised. To be able to improve the developed graph visualization tools further it is very important to ensure future maintainability. Extending the tool with a future supported graphics API can do this.

Vizz3D [1] is a graph visualization tool developed at Växjö University. The implementation is well structured. However, an extension of the available object-oriented implementation is necessary because the graphics libraries (GL) used for implementation are no longer supported. More functionality might be needed in the future to increase users understanding of the graphs. Examples can be rotation about three axis, support for shadows or new user-defined objects/shapes. In order to keep the tool maintainable in the future, Vizz3D needs to be extended with a more current API.

The performance of Vizz3D is sub-optimal. At the moment certain visualizations cannot be performed because the currently available hardware cannot satisfy the demands required by the implementation for complex visualization. For example, a self-analysis on the VizzAnalyzer [2] produces a graph with 654 nodes and 3417 edges that can be visualized very poorly. Other projects having millions lines of code will produce larger graphs that will not be able to be visualized. The implementation must be optimised to allow running Vizz3D on standard hardware with the performance needed for 3D visualization. Therefore existing performance bottlenecks must be identified and improved.

The purpose for solving the problem of this thesis is therefore to solve current shortcomings and to minimize future problems.

1.1 Context

VizzAnalyzer is a framework for reverse engineering allowing the integration and interaction of different analysis and visualization tools. It is used to analyse source code and visualize it with different tools. To allow different tools to integrate and work together a dynamic type system is used to control the data exchange between the tools, and a set of wrapper classes for handling their communication. Examples of tools that have been adapted to the framework are yED [3], a 2D graph editor, Wilma [3], a 3D graph tool and Crocopat [4], which is a tool for relational computation.

Vizz3D is another framework/tool, which has been adapted to the VizzAnalyzer framework. It is used to visualize data provided by the VizzAnalyzer (analysed code) in 3D graphics and it allows interaction with the graphical representation. The first implementation of Vizz3D used Java3D [5] as graphics API. In a course project during the spring semester of 2004 a version using OpenGL [6] as graphics API was created utilizing GL4Java [8] as a wrapper between Java and the native C OpenGL API. Hannes Ahlgren further extended this first (limited) version in his bachelor thesis [7] to one representing the same functionality as the original Java3D version.
1 Introduction

1.2 Problem

Vizz3D is used for visualizing large software systems in 2D/3D. The different program elements e.g. classes, interfaces, packages, methods, and attributes and their relations e.g. call, inheritance, contains, and implements relations are visualized by Vizz3D as interactive graphs consisting out of nodes and edges. The visualization of a node can be as simple as a sphere or cube, and as complex as a detailed shape representing a house. Since software systems are large the created graph can contain many thousand nodes and edges resulting in a resource consuming rendering effort. It is important that the visualization is performant, so that the user can interact with it without loosing the orientation and cognitive connection. Currently Vizz3D is not capable of visualizing huge graphs (software systems) with acceptable performance, since it exhausts the resources existing hardware can provide.

The problem of this thesis is therefore:

*Vizz3D lacks optimisation to visualize huge graphs with acceptable performance.*

This task is difficult to solve since Vizz3D is a complex software system lacking reusability, since the API used for visualization are not supported any more. The system must therefore be extended to allow future maintainability. Further software visualization itself is a complex area. To be successful state of the art technologies must be studied to be able to optimise the system.

1.3 Goal

As an in future maintainable version of Vizz3D is needed, the idea is to extend it with a more current API. With this extension the source code of Vizz3D will be reusable when adding new functionality to the system in the future. When the extension has been implemented it shall be optimised so that huge graphs (having for example 1000 nodes and 3000 edges) can be visualized with acceptable performance on existing hardware.

1.4 Criteria

1. The first goal of this thesis is to ensure the *maintainability* of Vizz3D for the future. The existing OpenGL version was implemented in a course project at the Växjö University. It was implemented using GL4Java as wrapper between Java and the native C OpenGL API that is no longer supported. Therefore Vizz3D needs to be extended with the more current wrapper implementation. The first goal is met when there is a new Vizz3D implementation/instance using a more current API, which has the same functionality as the currently used GL4Java version.

2. The second goal of this thesis is to improve the *performance* of Vizz3D by optimising OpenGL API calls and Java code (for example Object creation, Variables, Loops and Threading). The goal is met when the system performance according to the following aspects have been improved:

   2.1. Computational performance – How to reduce the needed computation time
   2.2. Scalability – How an application performs under heavy loads
   2.3. Perceived performance – How fast a user experiences an application
   2.4. RAM footprint – The amount of memory used
   2.5. Start-up time – The time it takes to launch an application
The aspects are sorted by order of importance. The Computational performance is very important for reducing the computation time in an application.

Scalability is important when the loads are heavy. Computational performance, RAM footprint and Perceived performance might influence the Scalability since an application needs to be fast and memory effective to scale well.

Perceived performance is important in interactive systems. Computational performance and Start-up time might influence the Perceived performance since the user feels if the application performs well.

RAM-footprint might be influenced by the Computational performance since some program elements used might be more memory effective than others. This aspect might be more important if the application consumes a high amount of memory.

The Start-up time is not crucial for the systems actual speed at run-time but a user might be frustrated if the time is too long.

1.5 Outline

Chapter 2 gives some background information about OpenGL and the Vizz3D graph system. This chapter shall help to understand some of the terminology used in the following chapters. Chapter 3 is about the implementation of the JOGL version of Vizz3D, which addresses the first goal of this thesis and describes in detail how the maintainability of Vizz3D was ensured for the future. Chapter 4 prepares chapter 5 by discussing how to evaluate performance and the performance aspects described in the criteria section. The techniques described here in theory are put into practice in the next chapter. Chapter 5 describes how the second goal was addressed and solved by using profiling and benchmarking techniques. Chapter 6 summarizes the effort done in the thesis and discusses how meeting the criteria defined solved the problem of this thesis. It further discusses future work. Appendix I contains a complete output of the profiling results described in chapter 5.
2 Background information

This chapter gives some brief background information that is needed to understand certain parts of the thesis.

It describes the OpenGL API since this will be used when extending Vizz3D with a JOGL version. It also describes the Java OpenGL wrappers GL4Java and JOGL, and discusses two alternatives to OpenGL (DirectX and Java3D). It further discusses briefly the Vizz3D system and its architecture focusing on the basic structure and functionality.

Most of the background information is gathered from OpenGL SuperBible [6] and the thesis Graph visualization with OpenGL [7].

2.1 The OpenGL API

OpenGL is a 3D graphics and modelling library developed by SGI, Silicon Graphics Inc [6]. It can be defined as a software interface to graphics hardware. It is not a programming language as Java or C. Instead it provides some pre-packaged functionality in an API (Application Program Interface). These functions are called from the ordinary code. OpenGL is available for most operating systems. Using a Java wrapper of OpenGL will lead to 3D applications almost independent of OS since Java is also available for most operating systems. However, the applications are still limited to the OS and hardware where OpenGL and Java are available.

2.1.1 Software Implementation

OpenGL can be implemented either by software or through hardware [6]. A software implementation can technically run anywhere as long as the system has the ability to display the generated image. For example, Windows applications usually call a Windows API called GDI (Graphics Device Interface). This is shown in fig 2.1. This is similar in other operating systems like UNIX. A software implementation takes graphics requests from an application and constructs a colour image of the 3D graphics. It then supplies this image to the GDI for display on the monitor.

2.1.2 Hardware Implementation

Usually a hardware implementation of OpenGL takes the form of a graphics card driver. This driver does not pass its output to the Windows GDI for display, instead the driver interfaces directly with the graphics display hardware (fig 2.2). A hardware implementation is usually much faster than a software implementation.
2.1.3 The OpenGL pipeline

The OpenGL pipeline describes what happens at OpenGL calls (fig 2.3). When an application makes OpenGL API function calls, the commands are placed in a buffer. After this, any transformation or lighting calculations are done if needed. Once this stage is complete, a rasterization is done. This step creates the colour image from the geometric and colour data. Then the image is placed in the frame buffer. This is the memory of the graphics display device, which means that the image is displayed on the screen.

2.1.4 The OpenGL State Machine

Each OpenGL command has an immediate effect based on the current rendering state. These states are flags that specify which features are on or off. Examples are “is lighting on or off” and “what is the fog’s density”. The states can be read and set by functions in the OpenGL API.

2.1.5 OpenGL wrappers

Programs written in C can easily call functions in the OpenGL API because they are written in C/C++. To be able to call these functions from Java an interface to C is needed. To make OpenGL available from Java in an easy way several wrappers have therefore been developed.

One wrapper is Jausoft’s GL4Java [8] (OpenGL for Java), that adds native OpenGL to the Java Virtual Machine (JVM). Its performance depends on the underlying JVM and OpenGL implementations. GL4Java consists of the Java Classes and the Native Library. The OpenGL calls goes from Java through the Java Native Interface (JNI) to the native OpenGL library.

Another way to use Java and OpenGL together is JOGL [9] (Java Bindings for OpenGL). It is designed to provide hardware-supported 3D graphics to Java applications and is part of a suite of open-source technologies initiated by the Game Technology Group at Sun Microsystems. JOGL started as “Jungle” and was developed...
by Ken Russel and Chris Kline. Russel is a Sun Microsystems employee working on the HotSpot Virtual Machine. Kline works for Irrational Games and both are very experienced with 3D graphics. One reason for JOGL being popular to use is that its development is supported by both Sun and SGI. Nowadays it provides full access to the APIs in the OpenGL 1.5 specification, and integrates with the AWT and Swing widget sets. This is very important, to make an application integrated and professional looking. The JOGL API version used in this thesis is 1.1.0-b10 together with the Java SDK version 1.4.2.

### 2.1.6 Alternatives to OpenGL

There are also other graphics API’s available. This section describes two popular ones.

One API is DirectX [13], which is developed by Microsoft. It is only available for the Windows operating system, which is a drawback if OS independence is wanted (OpenGL is available for most operating systems). DirectX gives access to hardware 3D graphics and sound features through software code. However, there is no direct Java2DirectX wrapper available.

Another graphics API is Java3D [5], which is developed by Sun. Java3D provides a set of object-oriented interfaces that support a high-level programming model to build 3D graphics. It can optionally use OpenGL or DirectX to do all low level calculations. One disadvantage is less control about the underlying graphics functions and lower performance through the additional abstraction layer. This means that OpenGL is faster than Java3D. Further Java3D is not supported any more, and does not support as many graphics functions.
2 Background information

2.2 Vizz3D

This section describes the Vizz3D system. The chapters focus on the basic functionality and structure and will not go into the smallest details. Most of the information in this chapter is gathered from the thesis Graph visualization with OpenGL [7].

2.2.1 The Application

Vizz3D is a graph visualization tool that lets the user interact with graphs. It can optionally use Java3D or OpenGL to do the visualization. The tool also allows manipulation and creation of new graphs. Examples of manipulations are adding and removing of nodes.

Vizz3D allows the user to use different metaphors and layouts so that the visualization of the graph can be altered. The graph can also be rotated, moved and zoomed to change the view. The visualization is shown in a GUI (Graphical User Interface). This is built up with a mainframe that contains a menu-bar, functionality icons, info panels and the visualization of the current graph, which is put on a canvas (fig 2.4).

Figure 2.4 The Vizz3D GUI, JOGL version.

2.2.2 The Structure

Vizz3D has an object oriented code structure. The main classes (VizzJ3d and VizzGL) are derived from a class named PlugIn. Every plug-in to VizzAnalyzer must use this class. The main classes initiate the MainFrame, creates a FileLoader that loads a graph and creates a GraphicsHandler. The GraphicsHandler then creates an MSceneGraph, which builds up a graph structure and adds it to the canvas (fig 2.5).
Every extension of Vizz3D has its own classes for handling of the visualization. For example, both VizzJ3d and VizzGL have an own GraphicsHandler class (GLGraphicsHandler and J3dGraphicsHandler). The work in the thesis Graph visualization with OpenGL [7] added a class named GraphicsHandlerAbstract, which is used to share common code, and a class named GraphicsHandlerInterface, which forces the respective GraphicHandlers to implement some methods.

There are three types of graphs that build up the visualization. When a user loads a graph a datagraph is created. This graph holds all the data but is also used for iterations for example. The second graph is a scenegraph. This is used by Java3D to build up the scene. OpenGL versions use it for making a visual graph and its metaphors and layouts. The third graph is the visual graph, which holds all the visual data, and is used when interacting with the nodes and edges.

There are two versions of the MLSceneGraph: J3dMLSceneGraph and GLMLSceneGraph. These create a graph structure through a VisualGraphInterface as either a J3dGraph or a GLGraph. The graph is an ordinary graph with nodes and edges, where the nodes visualize the classes in the analysed source code and the edges visualize their relations.

The work in the thesis Graph visualization with OpenGL [7] also introduced abstract classes to the visual graph to be able to use shared methods between VizzJ3d and VizzGL. An example can be found in the class VisualGraphAbstract where the methods to set and get variables, and the algorithms to add and remove nodes are identical. Figure 2.6 shows the present structure of the visual graph. This graph shows that GLGraph must implement all methods declared in GLGraphInterface, and VisualGraphAbstract must implement the methods declared in VisualGraphInterface. The nodes and edges also use this approach with the class named GLNode, which
implements the methods in \textit{GLNodeInterface} and the class \textit{GLEdge} that implements the methods in \textit{GLEdgeInterface}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{vizz3d-diagram.png}
\caption{The Vizz3D visual graph structure.}
\end{figure}

When it comes to the interaction it also has a complex structure (fig 2.7). The interaction part of Vizz3D consists of picking, transformations, rotation and many other things. When talking about the GL4Java version, the \textit{MLSceneGraph} creates a \textit{GLInteraction}. A \textit{GLAdvancedInteraction} is then created which handles the rendering of the graph. This means that the actual drawing of the scene is carried out in this class, and that it contains all the OpenGL or Java3D code. This class implements the predefined GL4Java class \textit{GLAnimCanvas} that is used to set up the canvas.
2.3 Summary

Chapter 2.1 described the OpenGL API since this will be used when addressing the first goal of the thesis, which was to ensure the maintainability of Vizz3D for the future. There are several graphics APIs available (for example OpenGL and DirectX). Since DirectX is available only for the Windows OS, using OpenGL instead will lead also to a fast system but which is in addition is available on a bigger variety of operating systems.

Using JOGL as the Java wrapper of OpenGL will lead to enhanced maintainability since it is maintained by Sun and SGI, and probably will be so in the future. The Java3D version of Vizz3D takes many resources compared to the GL4Java version, so using JOGL instead of Java3D will probably lead to better performance. This comes from the fact that OpenGL has no additional abstraction layer and that the rendering pipeline allows more control allowing for optimisation.

In chapter 2.2 the Vizz3D system and its architecture were described briefly to help understanding the implementation of the JOGL extension and the optimisation of it.
3 Ensuring the maintainability of Vizz3D

This chapter addresses the first goal of this thesis, which is to ensure the maintainability of Vizz3D for the future. It describes the implementation of a JOGL version of Vizz3D (VizzJOGL), which will later be optimised to address the second goal of the thesis.

3.1 Creating a JOGL scene

The first thing to do when creating a JOGL scene is to set up the environment. In Vizz3D this happens in the JOGLAdvancedInteraction class. Any class that creates a JOGL scene must implement the class GLEventListener, which is a predefined class in the JOGL API. This means that the class must contain the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init()</td>
<td>This method is called immediately after the JOGL context is initialised for the first time. It is used to set variables like light, depth testing and background colour. Cullfacing is also set here, which means that only the exteriors of shapes are rendered. This saves lots of CPU power. After this method, the display method is called.</td>
</tr>
<tr>
<td>display()</td>
<td>This method is called to initiate JOGL rendering by the client. It is called over and over again to redraw all primitives. In the first un-optimised implementation of the JOGL version this method had been called manually (by a call to display() in the end of the display() method). Later after the optimisation, the predefined Animator class did this in an optimised way. (described in chapter 5.4.1).</td>
</tr>
<tr>
<td>reshape()</td>
<td>This method is called during the first repaint after the component has been resized. It sets up the volume, which is the scene space. It also specifies the field of view.</td>
</tr>
<tr>
<td>displayChanged()</td>
<td>This method is called when the display mode or the display device has changed. The method is however unimplemented in the current versions of JOGL, so it will never be called in VizzJOGL. The class that creates the JOGL scene must still contain the method since GLEventListener is a Java interface class.</td>
</tr>
</tbody>
</table>

3.2 The implementation of VizzJOGL

Prior implementing a JOGL extension of Vizz3D, it was necessary to study the already existing GL4Java version (VizzGL). During the study it turned out that the two OpenGL wrappers are programmed in a very similar way. This was not too surprising, since they both use OpenGL as graphics library. This finding allowed reusing much of the original VizzGL (GL4Java version) code in the VizzJOGL version.

The thesis Graph visualization with OpenGL [7] prepared Vizz3D for new visualization plug-ins, which means that new plug-ins just have to derive from, or implement some existing classes.

The first step of the implementation was trying to get the GL4Java version to work from another directory to be aware of all dependencies it had. When this was done there

---

1 A graphics scene built by using methods from classes in the JOGL API.
were three Vizz3D versions. Two that were built on GL4Java and one that was built on Java3D. This was a good starting point to begin with the porting of the GL4Java code to new JOGL code. The new Vizz3D core structure is shown in figure 3.1.

The structure in figure 3.1 is already described in chapter 2.2.2, but shows now also the extension with JOGL. New classes in the class diagram are VizzJOGL (the main class of the JOGL extension that initiates the MainFrame), JOGLGraphicsHandler (that handles the visualization) and JOGLMLSceneGraph (that creates a graph structure).

Since the Visual graph structure in Vizz3D consists of one GL and one J3d version the whole GL structure was reused when the JOGL version was implemented. This was possible since the visual graph is separated from the core structure. Therefore the visual structure still looks exactly as showed in figure 2.6 in chapter 2.2.2.

However, when it comes to the interaction part new specific classes for the JOGL version were needed. The new Vizz3D interaction structure is shown in figure 3.2.
The structure in figure 3.2 is already described in chapter 2.2.2, but shows now also the extension with JOGL. New classes in the class diagram are \textit{JOGLInteraction} (that handles some of the interaction) and \textit{JOGLAdvancedInteraction} (that handles the rendering of the graph).

\textit{VizzGL} derives from a predefined GL4Java class called \textit{GLAnimCanvas}. This class contains the methods described in chapter 3.1 and some more. When JOGL should be used instead, this derivation was removed from the class \textit{JOGLAdvancedInteraction}. This class now instead implemented the predefined class \textit{GLEventListener} from the JOGL API.

The implementation of the JOGL version was quite straightforward since the GL4Java version worked similar. The implementation was made easier from the fact that the GL4Java implementation is described in the thesis \textit{Graph visualization with OpenGL}.

One problem having a lot of impact on the solution came from the fact that JOGL doesn’t allow OpenGL commands in Java Listeners (KeyListener, MouseListener or MouseMotionListener), but GL4Java does. This limitation was overcome by implementing flags in these methods being checked in the display method. This means that if a mouse button is pressed to zoom the scene for example, a true or false flag is set to true. The flag is then checked in the display method, and if it is true the scene is redrawn to show the zoomed image. Examples of these flags are: zooming, picking and translation.

Since the implementation of the JOGL extension started out from the GL4Java version the two had the same functions after the portation was ready. The speed of the new version ended up to be 7 frames per second under the in chapter 5.2.9 described circumstances (how this was measured is described in chapter 4.2). This was exactly as fast as the GL4Java version.

\textbf{3.3 Summary}

This chapter described how the first goal of this thesis was addressed. It explained in detail how using JOGL as new OpenGL wrapper, replacing the current GL4Java, ensured the maintainability of Vizz3D for the future.

This means that the criterion for the first goal is met after the implementation, since the JOGL version has the same functionality as the currently used GL4Java version.
4 Optimisation study

Since optimisation is essential for the solution of the problem of this thesis some theory needs to be discussed and a strategy developed, before optimisation on the VizzJOGL implementation can be applied. This chapter is dedicated to the discussion of optimisation and performance.

First the evaluation of performance will be discussed; further performance aspects contributing to the overall performance are described.

4.1 Measuring performance

There are two basic analysis techniques for evaluating performance [10]:

- **Profiling** – determining what areas of the system that consumes most resources.
- **Benchmarking** – comparing two or more operations.

### 4.1.1 Profiling

Profiling is the process of finding the performance bottlenecks (hot spots) in a system [10]. There are many tools available, but most of them are commercial. Two examples are AppPerfect DevSuite [14] and JProfiler [15]. Commercial tools often provide a graphical user interface for statistics and automatic fixing of some kinds of problems. Java 2 SDK is equipped with some free basic profiling tools, which is activated from the command line.

Many applications spend most of their time in just a few methods. A profiler tool can help identify these hot spots so performance tuning can be done more effectively. Java 2 contains a heap profiler, hprof, which can be used to find out where the system spends its time.

To profile the application with the heap profiler, one invokes hprof and run the application from the command line [11]:

```
java – Xrunhprof:cpu=samples <MainClassName>
```

As the application runs, the profiler gathers data. This data is collected in a generated text file (java.hprof.txt). The generated text file will show which methods that take the most time.

The command line above means that the tool will take samples of the CPU execution. This means that the call stack will be sampled at regular intervals and the methods on the stack recorded. This regular recording identifies the method currently being executed. By accumulating the number of hits on each method, the resulting data identifies where the application is spending most of its time. For example, if 25% of stacks sampled show method dig() on the top, then dig() takes 25% of the running time. See chapter 5.1 for example of profiling output.

When profiling in this way, the most useful performance tuning technique is to target the top five or ten methods and choose the most obvious one to fix first. The reason for this is that once changing one thing, the profile tends to change.

### 4.1.2 Benchmarking

The process of comparing operations to produce qualitative results is called benchmarking [10]. The operations can for example be different algorithms that produce the same result under the same preconditions. Benchmarks typically measure the time it
takes to perform a particular task, but can also measure the amount of memory required for example. They can also be used to measure a system’s start-up time.

One benchmarking technique that is suitable in many situations is to add timing functionality to the software code. The java.lang.System class contains some very useful methods. One of them is currentTimeMillis, which returns the number of milliseconds that have elapsed since midnight, January 1, 1970. While the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger. For example, many operating systems measure time in units of tens of milliseconds, which means that the test-time must be longer to produce a reliable result. This method can be used to measure how long a specific task in a system takes to execute. This is simply done by storing the time before and after the section of code executes, and then calculating the elapsed time by subtracting. This is shown below:

```java
long startTime = System.currentTimeMillis();
"code to measure"
System.out.println(System.currentTimeMillis() – startTime);
```

Since Vizz3D is a graphics system, one available metric is frames per second (FPS) [6]. FPS is the number of times a screen can be updated per second. Typically, applications start to have the fast feedback needed to prevent a user for loosing the cognitive orientation due to a delay between the interaction and feedback at around 15 FPS or higher [6].

How FPS is measured in Vizz3D is shown below:

```java
fpsCount++;
currTime = System.currentTimeMillis();
if ((currTime - baseTime) > 1000) {
    FPS = (fpsCount * 1000) / (currTime - baseTime);
    baseTime = currTime;
    fpsCount = 0;
}
```

The function is called one time per frame. The number of frames past is hold by the variable fpsCount. The current time is hold by the variable currTime and the starting time by the variable baseTime. When 1 second has gone (1000 ms), the FPS number is counted by dividing the number of frames past (fpsCount) by the number of seconds past (currTime – baseTime). Then the baseTime is set to the current time and fpsCount is reset to 0.

This kind of simple benchmarks can be used to compare the performance of alternative solutions and algorithms. Just implement the different solutions and measure the time difference. However, to be able to compare the times before and after optimisation some requirements on the environment are needed. Examples of these requirements are the same preconditions (for example the same operating system), same test data (for example the same test graph to visualize). The test results also need to be stored to be able to show the effects of each single optimisation.
Another important use of benchmarks is that they enable the developer to track and analyse trends. As fixing bugs and adding features, the performance will probably change. By using benchmarks one can easily determine whether the system is getting faster or slower.

4.2 Performance aspects

Before evaluating the performance it is necessary to apply different optimisation techniques. Several aspects contribute to the overall performance of an application [10]:

1. Computational performance
2. Scalability
3. Perceived performance
4. RAM footprint
5. Start-up time

Some aspects of performance are applicable to client-side applications, some to server-side and some to both. Some of the theory described below applies also to other domains (for example embedded devices or parallel networks). But since the domain of this thesis is Personal Computers and similar servers the theory is restricted to that domain.

4.2.1 Computational performance

Most people think about computational performance when discussing system performance. It concerns characteristics such as:

- How many instructions are required to execute a statement?
- How can I restructure this specific method to gain speed?
- What algorithm shall I use here?

Most of the performance literature concentrates on computational performance. Which algorithms to use and how to build up the methods are key factors in the overall performance of a system.

4.2.2 Scalability

Scalability measures how systems perform under heavy loads. This is an often-neglected aspect of performance. A graph system might perform well when using small graphs, but poor when increasing the graph size. A system should be designed to accommodate its intended use. However in a graph system, the graph sizes might vary a lot.

4.2.3 Perceived performance

In some way perceived performance is the most important aspect of performance. This measures how fast the system *feels*, rather than how fast it really is. This is important for interaction applications since the user directly experiences if the system performs poorly.

There are many ways to improve the perceived performance of an application without actually making any of the code run faster. This can be to change the mouse cursor to a wait cursor when the application is busy, to introducing background threads that
performs work, or different working and gui threads allowing the application to respond to user interaction, and not freeze while performing some complex calculations.

4.2.4 RAM footprint

This part considers the amount of memory needed to run the application. This can be of crucial importance to the overall performance of a system.

Modern operating systems provide a virtual memory system where hard disk space can be used instead of physical RAM. However this force the operating system to page to virtual memory and therefore the application will perform poorly because the hard disk speed is much slower than the speed of the physical RAM.

Some systems perform well while they are developed, but poorly once deployed. This is a fact since the developers typically have better workstations than average users. If the software is going to be used on workstations with limited resources the systems must be designed for this.

The developed system is probably not the only program that should run on a user’s machine. It is therefore important that the application doesn’t consume all memory resources.

For measuring of how much RAM a system consumes Java provides some facilities. Two methods in the java.lang.Runtime class can be used. These methods look at the size of the virtual machine’s heap:

- `Runtime.totalMemory` returns the size of the heap used to allocate objects
- `Runtime.freeMemory` returns the amount of memory not being used in the object heap

To figure out how much memory that is used one just subtract the freeMemory from the totalMemory as shown below:

\[
\text{MemUsed} = \text{totalMemory} - \text{freeMemory}
\]

4.2.5 Start-up time

The amount of time it takes to launch a program can be critical. If developing an applet for example, the time to load the web page can’t be one minute. A user will probably not wait that long time and he will load another page instead.

4.3 Summary

This chapter discussed optimisation and performance. It therefore addressed the second goal of this thesis, which is to improve the performance of Vizz3D by optimising OpenGL API calls and Java code.

Chapter 5 continues to discuss how the second goal is met by describing which optimisations that were applied on Vizz3D.
5 Optimisation of Vizz3D

This chapter applies the theory prepared in the previous chapters and describes the optimisation of the VizzJOGL implementation. The chapter addresses the second goal of this thesis, which is to improve the performance of Vizz3D by optimising OpenGL API calls and Java code.

First the optimisation strategies used will be described. Further the actual optimisations will be discussed in chapter 5.2 to 5.6. The optimisations applied are sorted by the performance aspects described in chapter 4.2.

5.1 Profiling

To decide which optimisation strategies to use when tuning a system it’s a good starting point to use a profiling tool to examine if there are some major bottlenecks to begin to optimise.

In this thesis the heap profiler tool from the Java 2 SDK was used since it is free. The heap profiler (hprof) was started from the command line (as described in chapter 4.1.1) with VizzJOGL as the main class. In this way the JOGL version of Vizz3D was chosen. When Vizz3D had started, a test graph file was loaded. After some seconds the system was stopped to limit the profiling output file size.

Three rows of the output when profiling the JOGL version of Vizz3D are shown in figure 5.1. Appendix I contains the complete profiling output.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Self</th>
<th>Accum</th>
<th>Count</th>
<th>Trace Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50.27%</td>
<td>50.27%</td>
<td>1588</td>
<td>sun.awt.windows.WToolkit.eventLoop</td>
</tr>
<tr>
<td>2</td>
<td>35.23%</td>
<td>85.50%</td>
<td>1113</td>
<td>net.java.games.jogl.impl.windows.WGL.NativeEventLoop</td>
</tr>
<tr>
<td>3</td>
<td>5.19%</td>
<td>90.69%</td>
<td>164</td>
<td>java.io.FileInputStream.open</td>
</tr>
</tbody>
</table>

Figure 5.1 VizzJOGL profiling output.

The most important column in figure 5.1 is self. This column shows that the awt eventloop took 50.27 % of the total running time. This comes from the fact that Vizz3D is a GUI application, and that the GUI is drawn over and over again.

The second row (rank 2) in the self-column shows that the JOGL event loop took 35.23 % of the total running time, and the third row shows that it took some time to open files. The reason that the third number (5.19%) is so high comes from the fact that the profiling was stopped only some seconds after loading a test graph file. This is due to the fact that the profiling file becomes too big if profiling longer time without being more expressive.

This profiling result shows directly that optimising the JOGL event loop seems the way to go. This was the first tuning technique that was used in this thesis: Improve major performance bottlenecks.

The profiling results showed no other class in Vizz3D that were especially slow. So when tuning the Java part there were no special methods to concentrate the optimisation on. However, a bottleneck was found in the time it took to load a graph, so some work was concentrated on improving that.

Since no other major bottlenecks were found when profiling Vizz3D, the second tuning strategy used in this thesis was: Apply general optimisation techniques.
5.2 Computational performance

This section describes the optimisations that improved the Computational performance of Vizz3D. Chapter 5.2.1 – 5.2.3 describes the optimisation done in the OpenGL part. Chapter 5.2.4 – 5.2.8 describes the optimisation done in the Java code part.

5.2.1 Display lists in OpenGL

The GL4Java version of Vizz3D just goes through the nodes and edges in a graph one after one when rendering them. To speed up the rendering two different display lists were used, one for the nodes and one for the edges.

A display list provides a good way to create a pre-processed set of OpenGL commands [6]. A display list is delimited with a glNewList/glEndList pair. To create the JOGL node list the code below was used in the class JOGLAdvancedInteraction:

```java
int dispListNode = gl.glGenLists(1);
gl.glNewList(dispListNode, GL.GL_COMPILE);
for (int i = 0; i < nodesSize; i++) {
    if (graph.getNode(i).isVisible()) {
        renderNode(i);
    }
}
gl.glEndList();
```

The first line creates a display list with the name dispListNode. Then the second line tells JOGL to begin compiling a list of JOGL commands. The second parameter to glNewList can be either GL.GL_COMPILE or GL.GL_COMPILE_AND_EXECUTE. The difference is whether to compile and store the commands or to compile, store and execute as they occur. The for-loop compiles all nodes to the list, and then the creation of the list is terminated with a call to glEndList. When the list is initialised, a call to glCallList(dispListNode) is needed one time every display loop to render the scene.

However, since Vizz3D is a graph application with the possibility to change the layout and other parameters, the node positions must be checked every display loop to see if the scene has changed since the last rendering. The checking is done by summing all node positions (oldNodePosSum) and comparing the result with the sum from the latest display loop (oldNodePosNumber) as shown below:
double oldNodePosSum = 0.0;
GLNodeInterface currNode;

for (int i = 0; i < nodesSize; i++) {
    currNode = graph.getNode(i);
    oldNodePosSum += currNode.getCurrentPosition().x;
    oldNodePosSum += currNode.getCurrentPosition().y;
    oldNodePosSum += currNode.getCurrentPosition().z;
}

if (oldNodePosSum != oldNodePosNumber) {
    oldNodePosNumber = oldNodePosSum;
    nodePosChanged = true;
}

If the node positions have changed, a new list must be initialised to call a list with correct node positions every loop. This is solved with a variable: nodePosChanged which is true if the node positions have changed and false otherwise (shown in the code listing above). If some binding is loaded which change some other node attribute (colour, shape, size) the some node must be moved to force a reinitialising of the display list. The same applies to the edges in the scene.

5.2.2 State changes in OpenGL

Restructuring OpenGL state-setting commands can also improve the performance [6]. The commands can be classified into two different categories, vertex-data and modal state-setting commands.

Vertex-data commands are calls that can occur between a glBegin/glEnd pair:

- glVertex – Specifies point, line and polygon vertices
- glColor – Stores a current colour
- glIndex – Updates the current colour index
- glNormal – Sets the current normal to the given coordinates
- glMaterial – Assigns values to material parameters
- glTexCoord – Specifies texture coordinates

Restructuring a program to eliminate some vertex-data commands will not significantly improve performance since the processing of these calls is very fast.

Modal state-setting commands are commands that:

- Turn on/off capabilities
- Change attribute settings for capabilities
- Define lights
- Change matrices

These calls cannot occur between a glBegin/glEnd pair.

Changes of these types are much more expensive to process than simple vertex-data commands. Minimizing or grouping modal state changes can optimise performance. Grouping the changes together, and the rendering primitives, will provide better performance than mixing them with primitives.
The lighting in Vizz3D must be called every time the graph is rotated to get the correct position. This is done by three calls to `glLightfv` for every node and edge. This was reduced to only one time per rendering frame instead which led to better performance. Calls to `glPushAttrib(GL.GL_LIGHTING_BIT)` and `glPopAttrib` was then reduced to one per node.

Calls to `glPushMatrix` and `glPopMatrix` for each edge were also removed, and that led to better performance when the number of edges was high. All these changes were applied in the class `JOGLAdvancedInteraction`.

### 5.2.3 Numerical errors in OpenGL

After timing a function that draws direction arrows a numerical error was spotted. The error was found in a class called `DrawingFunctions`, which contains methods that draw the graphs spheres, boxes, cylinders, cones and direction arrows on the edges. Before applying a layout the drawing of edge arrows took very long time because the edge vector numbers were bad. The arrows shall be rotated to match the edges. Calling the `glRotated` with an angle that is NaN (not a number) takes very long time and has no meaning at all (because there are no visible edges before calling a layout anyway). By adding a row that checked that the angle was a number the performance was increased when drawing the scene before calling a layout. The rows are shown below:

```java
if (!Double.isNaN(angle)) {
    gl.glRotated(angle, rotVector.x, rotVector.y, rotVector.z);
}
```

### 5.2.4 Exceptions, Casts and Variables in Java

Various programmatic elements, including exceptions, cast and variables might also have a performance cost [11].

When considering the cost of exceptions [11] states in a test that they generally use no extra time if no exception is thrown. But if an exception occurs and a catch block is executed there is a significant overhead. This overhead is mainly due to the cost of taking a snapshot of the stack. The snapshot is used to print a stack trace. This means that exceptions shouldn’t be thrown in the normal code path. They should only occur if an error has occurred.

Casts also have a cost. Some casts can be eliminated at compile-time. However, casts not resolvable at compile time must be executed at run-time. Primitive data type casts are quicker than object data typecasts because no tests are involved, but they still have an associated cost.

Variables might also be a concern. Local variables and method-argument variables are the fastest variables to access and update. This is a fact because local variables remain on the stack, so they can be manipulated directly. Static and instance variables are manipulated in heap memory. Temporary variables can also be used to restrict the number of times a method is being called. Instead of calling the method each time, just declare a temporary variable to store the value. There are also some techniques when using a variable in a loop. For example that it’s better to declare a variable before the loop instead of in the loop to avoid multiple declaring [11].

Controlling of the exceptions used in Vizz3D showed that they are only used when it’s absolutely necessary. No applicable optimisation technique was found.
When it comes to *casts*, one technique that was applied to the Vizz3D code was to use a temporary variable instead of repeatable casting. In the class *JOGLMLSceneGraph* the lines:

```java
if (e.isDirected()) {
    from = ((DirectedEdgeInterface) e).getFrom();
    to = ((DirectedEdgeInterface) e).getTo();
} else {
    from = ((UndirectedEdgeInterface) e).getFirstNode();
    to = ((UndirectedEdgeInterface) e).getSecondNode();
}
```

Were therefore changed to, by introducing the temporary variables `de` and `ude` to eliminate unnecessary casts:

```java
if (e.isDirected()) {
    DirectedEdgeInterface de = (DirectedEdgeInterface)e;
    from = de.getFrom();
    to = de.getTo();
} else {
    UndirectedEdgeInterface ude = (UndirectedEdgeInterface)e;
    from = ude.getFirstNode();
    to = ude.getSecondNode();
}
```

*Variables* concerned the technique using temporary variables to store values was used in Vizz3D. In the class *JOGLMLSceneGraph* this technique could be applied and the readability as a side effect improved. The old code was:

```java
GLNode n = new GLNode(sceneGraph, node, geom);
    n.setCurrentPosition(currentMetaphor.getXPos(),
        currentMetaphor.getYPos(), currentMetaphor.getZPos());
    n.setFinalPosition(currentMetaphor.getXPos(),
        currentMetaphor.getYPos(), currentMetaphor.getZPos());
    return n;
```

These lines were changed to, by introducing the temporary variables `xPos`, `yPos` and `zPos` to store the temporary values, saving 3 method calls per node:
GLNode n = new GLNode(sceneGraph, node, geom);

double xPos = currentMetaphor.getXPos();
double yPos = currentMetaphor.getYPos();
double zPos = currentMetaphor.getZPos();

n.setCurrentPosition(xPos, yPos, zPos);
n.setFinalPosition(xPos, yPos, zPos);

return n;

The technique that it’s better to declare a variable before the loop instead of in the loop was also used in Vizz3D. For example, when iterating through nodes in the class JOGLInteraction of Vizz3D the old code was:

```java
while (vni.hasNext())
    GLNodeInterface node = (GLNodeInterface) vni.next();
```

These lines were changed to, by moving the declaration of node out of the loop to avoid multiple declaring, saving 1 declaration per node:

```java
GLNodeInterface node;

while (vni.hasNext())
    node = (GLNodeInterface) vni.next();
```

And when filling the list of nodes and edges, a bit of the old code was:
NodeIterator ni = dataGraph.nodes();

while (ni.hasNext()) {
    ni.next();
    NodeInterface node = ni.getNode();
    TypeValues type = (TypeValues) node
    .getProperty(GraphProperties.TYPE);
    Color col = (Color) node
    .getProperty(GraphProperties.NODE_COLOR);
    GLMetaphorType shape = (GLMetaphorType) node
    .getProperty(VisualProperties.SCENE_NODE_SHAPE_GL);
}

EdgeIterator ei = dataGraph.edges();

while (ei.hasNext()) {
    ei.next();
    EdgeInterface edge = ei.getEdge();
    TypeValues type = (TypeValues) edge
    .getProperty(GraphProperties.TYPE);
    Color col = (Color) edge
    .getProperty(GraphProperties.EDGE_COLOR);
    GLMetaphorType shape = (GLMetaphorType) edge
    .getProperty(VisualProperties.SCENE_EDGE_SHAPE_GL);
    Integer line = (Integer) edge
    .getProperty(GraphProperties.EDGE SHAPE);
}

These lines were changed to, by moving the declarations out of the loop to avoid multiple declaring and also reusing the variables col, shape and type, saving 4 declarations per node and 5 per edge:
NodeIterator ni = dataGraph.nodes();
NodeInterface node;
Color col;
GLMetaphorType shape;
TypeValues type;

while (ni.hasNext()) {
  ni.next();
  node = ni.getNode();
type = (TypeValues) node
    .getProperty(GraphProperties.TYPE);
  col = (Color) node
    .getProperty(GraphProperties.NODE_COLOR);
  shape = (GLMetaphorType) node
    .getProperty(VisualProperties.SCENE_NODE_SHAPE_GL);
}

EdgeIterator ei = dataGraph.edges();
EdgeInterface edge;
Integer line;

while (ei.hasNext()) {
  ei.next();
  edge = ei.getEdge();
type = (TypeValues) edge
    .getProperty(GraphProperties.TYPE);
  col = (Color) edge
    .getProperty(GraphProperties.EDGE_COLOR);
  shape = (GLMetaphorType) edge
    .getProperty(VisualProperties.SCENE_EDGE_SHAPE_GL);
  line = (Integer) edge
    .getProperty(GraphProperties.EDGE_SHAPE);
}

It is also unnecessary to declare a new loop variable if one already exists. It is better to
reset the old one and reuse it. Some examples were found where one first uses a variable
i to iterate through the nodes and then declares a new variable j to iterate through the
edges in the same method.

Sometimes declaring a variable can be avoided. However, this is always a choice
between understandable code and optimised code. In the class JOGLGraphicsHandler
one example was found where the variable declaration could be removed. The original
code was:

InteractionInterface cgl;

cgl = (InteractionInterface) sceneGraph
    .createCanvas3D(100,100);

return cgl;
These lines was changed to the single line, saving 1 declaration:

```java
return (InteractionInterface) sceneGraph
    .createCanvas3D(100,100);
```

In the class `JOGLCamera` another example was found. The old code was:

```java
JOGLVector3 vPosition = new JOGLVector3(positionX, positionY, positionZ);
JOGLVector3 vView = new JOGLVector3(viewX, viewY, viewZ);
JOGLVector3 vUpVector = new JOGLVector3(
    (upVectorX, upVectorY, upVectorZ);

myPositionVector = vPosition;
myViewVector = vView;
myUpVector = vUpVector;
```

It is better to assign the variables directly unless it is not required in some special cases, where the original vector must be memorized for later reuse. The lines were changed to, saving 3 declarations and 3 assignments:

```java
myPositionVector = new JOGLVector3(positionX, positionY, positionZ);
myViewVector = new JOGLVector3(viewX, viewY, viewZ);
myUpVector = new JOGLVector3(upVectorX, upVectorY, upVectorZ);
```

5.2.5 Loops, Switches and Recursion in Java

Loops, Switches and Recursion are common code structures that also can be tuned [11]. Java applications spend much time in loops, and there are techniques to optimise them. Some of the techniques are straightforward, like removing code from the loop if it is executed only once, but others are more complicated. If switch statements or recursion are used in a class they might also be able to optimise.

One general guideline is to avoid using a method call in a loop termination test. [11] States that the overhead is significant. This technique was applied to the rendering class of VizzJOGL (`JOGLAdvancedInteraction`). When the nodes and edges shall be drawn or the node positions be checked they must be iterated through as shown below for the nodes:

```java
for (int i = 0; i < graph.nodesSize(); i++)
```

This was replaced with the following two lines, saving 1 method call per node and 1 per edge:

```java
int nodesSize = graph.nodesSize();
for (int i = 0; i < nodesSize; i++)
```
5.2.6 I/O, Logging and Console Output in Java

I/O directly to the computer memory is much faster than I/O to a disk or a network [11]. Examples of optimisation techniques are buffering and caching. If the application is logging any data or console output this of course also degrades performance. But the performance depends on which technique that is used.

In Vizz3D the I/O is about loading a graph from a file. This is handled by a graph package called grail. This does not belong to the actual Vizz3D system and therefore no work was done to try to optimise grail.

Logging is not used in Vizz3D and the console output is only used to print important information.

5.2.7 Sorting in Java

When it comes to sorting, the JDK system provides methods for arrays and collections [11]. These methods are adequate for all but the most specialized applications. If an optimisation is needed one can normally get enough improvement by implementing a standard sort method (such as quicksort) or better use the already implemented sorting algorithms provided by the Java API.

In Vizz3D no place was found where these techniques could optimise the performance since the data structure used are in an external graph package (grail), which was not part of the focus.

5.2.8 Appropriate Data Structures and Algorithms in Java

An important decision is which data structures or algorithms to choose for the application [11]. One should be aware of optimised alternative structures and algorithms, and always consider the possibility to switch to a different one, rather than always performance-tune the structure or algorithm already being used.

Every visual node and edge in Vizz3D is an object with its own properties. To store the objects a list interface is used. The Java 2 Collections framework is equipped with four possible lists: Vector, Stack, ArrayList and LinkedList. The Vector, Stack and ArrayList implementations are based on arrays, and LinkedList has an underlying implementation that consists of a doubly linked list. This leads to that the performance of LinkedList’s is worse than the others for most operations. Vector and ArrayList are implemented with an underlying Object[] array. The difference between them is the use of synchronization. None of the ArrayList methods are synchronized, while most of the Vector methods are. This makes ArrayList faster than Vector [11].

Choosing the right collection for Vizz3D was quite easy. What was needed was that the collection should be fast for indexed access (when accessing node properties) and also for insertion to the end of the list (when making the list of nodes). Overall the Vector and ArrayList implementations give the best performance for this tasks. The GL4Java version uses the Vector class, but since the ArrayList is faster this was used instead.

5.2.9 Benchmarking the Computational performance

During the optimisations in the Computational performance aspect, the method System.currentTimeMillis() was used to check if the changes had any direct effect. Using that method, changes that only had some milliseconds effect could be measured. However, when the optimisations should be evaluated after finish another evaluation method was used.
The most obvious method for evaluating the optimisations done under the Computational performance aspect is to use frames per second. This comes from the fact that Vizz3D is a graph application.

But to be able to measure the FPS number a specific test graph must be chosen. The graph chosen was a graph over the VizzAnalyzer source code (VA.gml), with 654 nodes and 3417 edges. The graph had the size of 1174 KB and contained the geometric types box and sphere. No textures were applied on the graph. This graph was very slow when loading into Vizz3D but still possible to handle. But it was chosen to test if the system scaled well (see chapter 4.2.2).

The test environment was a quite modern computer (Intel Pentium 4 CPU 2.60 GHz, 512 MB RAM, Nvidia GeForce FX 5200 128 MB graphics card), with no other program started than the ones started automatically. The operating system used was Windows XP.

To be able to compare the results the same settings were used when rendering the graph. The layout used was Circularizer\(^1\) with the radius of 10 (to fit the whole graph on the screen) without any rotation. The graph was shown in window mode and the screen resolution was 1024*768, with the colour depth at 32 bits.

Measuring the FPS number as described in chapter 4.1.2 while rendering the VizzAnalyzer graph before and after optimisation gave the results showed in figure 5.2. The figure shows that FPS number increased by 200 % from 7 to 21.

It might be strange that the FPS numbers were exactly 7 for all three Vizz3D versions before optimisation, but this is only a coincidence.

![Computational performance results](image)

**Figure 5.2** Computational performance results.

---

\(^1\) The graph formed as a circle.
5.3 Scalability

This section describes an optimisation that improved the Scalability of Vizz3D. However, the optimisations done in the Computational performance, RAM footprint and Perceived performance aspects does also contribute to the scalability since a system needs to be fast and memory effective to scale well.

5.3.1 Primitive quality in OpenGL

Reducing the quality of the primitives drawn may significantly improve the performance, especially when the number of primitives is high [6]. But of course one can’t reduce the quality too much. Vizz3D requires lots of primitives when the graphs are big. The primitives are drawn in the class `DrawingFunctions`. The functions `glutSolidSphere` and `glutSolidCone` has the number of slices and stacks as parameters. Slices means number of subdivisions around the Z axis (longitude) and stacks number of subdivisions along the Z axis (latitude). Reducing this number from 20 to 15 gave better performance, since it saves 25 % of the vertexes to be rendered. However, there is of course a quality loss, but it is only visible when zooming the scene much and the loss is also small. Applying this optimisation technique changes the tool/look. But since the initial quality was higher than necessary from the beginning this is not a problem. A better future solution could be to implement a Level of Detail (LOD) mechanism, which means that the primitive quality becomes better when zooming in and worse when zooming out. This comes from the fact that a distant object is rendered as a dot regardless how many points it has.

5.3.2 Benchmarking the Scalability

Since the optimisations done under the Computational performance, RAM footprint and Perceived performance aspects all contributes to the Scalability the most obvious method to evaluate the results of the optimisation is to use frames per second with a huge graph. As mentioned in chapter 5.2.9 the VizzAnalyzer graph was chosen since it was very slow when loading into Vizz3D but still possible to handle. See chapter 5.2.9 for the result graph.

Another method was also used to evaluate the scalability. This was to test how many nodes that actually could be drawn at the speed 15 FPS. The number 15 FPS was chosen since that is the number when systems start to have the fast feedback needed to prevent a user for loosing the cognitive orientation due to a delay between the interaction and feedback. This means that the evaluation method tests how huge graphs that can be drawn in Vizz3D and still be handled very smoothly. However, the JOGL version can still handle larger graphs but with slower speed than 15 FPS (for example 10 000 nodes at 6 FPS). A small program was implemented that generated test-graphs with an optional number of box shaped nodes, with only type and label as properties. The number of nodes drawn at 15 FPS before and after optimisation is shown in figure 5.3. The figure shows that the number of nodes that could be drawn increased by 103 % from 1790 to 3640. The JOGL version therefore performs at the same speed with a much heavier load than before, which shows that the scalability has improved after optimisation.
Scalability results

![Scalability results graph]

**Figure 5.3** Scalability results.
5.4 Perceived performance

This section describes the optimisations that improved the Perceived performance of Vizz3D. However, the optimisations done in the Computational performance and Startup time aspects do also contribute to the Perceived performance. 5.4.1 describes an optimisation done in the OpenGL part. 5.4.2 describes an optimisation from the Java code part.

5.4.1 Animator class in OpenGL

To efficiently use multi threading in VizzJOGL the predefined Animator class was added in the class JOGLAdvancedInteraction. An animator can be attached to a GLDrawable (for example the VizzJOGL canvas) to drive its display() method in a loop. To be as efficient as possible, it sets up the rendering thread for the drawable to be its own thread. Therefore it cannot be combined with manual repaints.

The Animator class provides two methods:

- start() - Starts the animator
- stop() - Stops the animator, blocking until the animation thread has finished

The animator is started when all data is read in, to run the display loop efficiently. This doesn’t lead to faster code when measuring the display method time, but the perceived performance becomes much faster. This means that a user experiences the system as faster because the drawing of the scene is done by its own thread, allowing other threads to do other work so that the system doesn’t freeze. Before exiting drawing, the stop method is called, so that the animator thread exits.

VizzJOGL runs in an AWT graphical user interface. When adding an animator the menus work quite slow because the CPU must schedule between the animator and AWT threads. To solve this problem the priority of the animator thread was changed to a lower value, by adding the line showed below in the init method:

```java
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
```

With this priority the AWT menus work at normal speed again.

When using the Animator class the application always tries to use 100% of the computer power. This leads to a very high frame rate when drawing small graphs. An interaction application doesn’t need to draw graphs at 80 FPS, it’s enough with 30 FPS. To solve this problem frame rate binding to 30 FPS was implemented. This was done by allowing the animator thread to sleep a specific time to always produce correct display loop time. When graphs are rendered the computer uses exactly the amount of power that it needs to render the scene at 30 FPS. If the rendered graph is even bigger (so that 100% of the CPU is not enough for rendering at 30 FPS) the frame rate will drop below 30 FPS.

5.4.2 Threading in Java

Using threads allows an application to do multiple things at the same time. Java’s extensive support for threads makes it quite easy to use multithreading. Introducing threads can greatly improve the perceived performance [11].

In Vizz3D threads are already used in an efficient way. An own thread is for example used when filling the lists of nodes and edges.
The most important usage of threads in Vizz3D was implemented during the optimisation of the OpenGL part. This was done using the predefined JOGL Animator class to run the display() method in a loop (described in chapter 5.4.1).

5.4.3 Benchmarking the Perceived performance

Since the optimisations done under this category just change how the user experiences (feels) an application there is no obvious way to evaluate this method by numbers. However, Vizz3D felt a lot faster after the optimisations had been applied and that meant that the optimisations had a positive effect.
5 Optimisation of Vizz3D

5.5 RAM footprint

This section describes the optimisations that improved the RAM footprint of Vizz3D. However, the optimisations done in the Computational performance aspect does also contribute to the RAM footprint.

5.5.1 Object creation in Java

Java handles low-level memory allocation and de-allocation automatically [11]. It also comes with a garbage collector. Java makes memory safe by preventing access to low-level memory handling routines. However, it does not prevent the use of excessive amounts of memory or cycling through more memory than intended. This means that memory leaks can occur by holding onto objects without releasing references. The garbage collector will stop reclaiming those objects, resulting in increasing amount of memory being used. Creating many objects costs time and CPU power. Under this category there are some general guidelines and standard techniques one should follow for using object memory efficiently, and some of these were applied to Vizz3D [11].

In frequently used routines creating objects should be avoided. Because such routines are called frequently, objects will be created frequently. Rewriting such routines will decrease memory usage.

This is a standard rule so no obvious error of this kind was found in Vizz3D. The system was searched for example for loops were unnecessary objects were created over and over again in the loop.

If using a collection (see 5.2.8 for more about choosing appropriate data structures), one shall precise it to be as big it will need to be. It is even better for the collection to be bigger than necessary than smaller, since growing is slow. For example, a Vector grows by creating a new larger Array object, copying all elements from the old array, and discarding it. Most collections have similar implementations for growing the collection beyond its current capacity.

In Vizz3D this guideline applies because the visual nodes and edges are stored in two ArrayLists in the class GLGraph. The ArrayLists are only changed either when adding or removing a node or an edge, or at initialising. However, since graphs vary very much in size it should be bad to precise the ArrayLists to same number every time. It should be better to precise it to the number of nodes and edges that were in the graph. This was solved by first declaring the variables and then initialising them to correct numbers in the constructor. Correct numbers are fetched from the grail graph package by the methods nodeCount() and edgeCount(). How this was done is shown below:

```java
nodeList = new ArrayList(dataGraph.nodeCount());
edgeList = new ArrayList(dataGraph.edgeCount());
```

Another specific technique was also applied to Vizz3D. When creating a new object all chained constructors are automatically called. Initialising variables more than once therefore causes some overhead (However, newer versions of the compiler can eliminate the redundant statement [11]). It’s good to be aware of the default values that Java initialises variables to:

- `null` for objects
- `0` for integer types (byte, short, int, long)
- `0.0` for float types (float and double)

The reinitialising was removed in every class of Vizz3D.
5.5.2 Strings in Java

*String* objects are immutable [11]. This means that once created they can’t be altered. Applying methods like `String.trim()` doesn’t actually change the string, instead it returns a changed copy. Strings are also final, which means they cannot be subclassed. Strings therefore have advantages and disadvantages when performance is concerned. Therefore the usage of them must be optimised. In the Vizz3D code strings are not used much, but some tuning techniques were examined.

There is a difference between compile-time resolution and run-time creation of strings. When compiling an application, strings are resolved to eliminate the concatenation operator if possible. This means that the line:

```java
String str = "Vizz" + "3" + "D";
```

Is compiled as:

```java
String str = "Vizz3D";
```

However, if a string involving the concatenation operator cannot be resolved at compile-time, it must be executed at run-time. This causes extra objects to be generated. An example of this is concatenation of a string variable with an unknown value. In this case it is better to use the `StringBuffer` class. Some examples of this were found in the Vizz3D code. For example, the line:

```java
String configFileName = Data.configDir+"VAconfigconfig.xml";
```

Could be rewritten as:

```java
String configFileName = 
(Data.configDir+"VAconfigconfig.xml").toString();
```

And the line:

```java
name = classname + " " + name;
```

Could be rewritten as:

```java
name = (new StringBuffer()).append(classname).append(" ").append(name).toString();
```

Rewriting the expressions like this is optimal, creating only two objects. However, since the examples found in Vizz3D are only called one time at initialisation the lines were left as before to keep the readability. If the lines should be found for example in a frequently used loop, this optimisation could be considered.

Another point is that using the `String` constructor forces the string to be created at run-time:

```java
String str = new String("Vizz3D");
```
However, no declaration like this was found while analysing the Vizz3D code (manually).

5.5.3 Benchmarking the RAM footprint

The memory was measured in the class `JOGLAdvancedInteraction` with the method described in chapter 4.2.4. Measuring the memory usage after optimisation of both the Java and OpenGL part gave a positive result. Before optimisation the memory while rendering the VizzAnalyzer graph (VA.gml) with the JOGL version was like the GL4Java version (19 MB). After optimisation the memory usage became 15 MB which was 21 % less than before. The Java3D plays in another league with its memory usage of 335 MB. Specifying “-Xms200m -Xmx400m” (defining the minimum and maximum heap size) on the command line is even necessary for the version to not die through “java.lang.OutOfMemoryError” while loading VA.gml. The memory usage for the VizzAnalyzer graph before and after optimisation is shown in figure 5.4 (The memory usages were consistent through the visualization and there were no peaks.).

![RAM footprint results](image.png)

**Figure 5.4** RAM footprint results.
5.6 Start-up time
This section describes an optimisation that improved the Start-up time of Vizz3D.

5.6.1 Progressbar update in Java
When thinking of Start-up time and Vizz3D one directly thinks of the time it takes loading a graph. When starting Vizz3D no graph is loaded, so this Start-up time is very fast.

When loading a graph in Vizz3D a JProgressBarDialog is used to see the status of the loading. This was found being a very big bottleneck. The dialog was temporary removed and the time was taken (using System.currentTimeMillis() as described in chapter 4.1.2) of the loading of the testgraph (VA.gml) with the system configuration as described in chapter 5.2.9:

- Time with progressbar: 2.33 s
- Time without progressbar: 0.17 s

The bottleneck comes from the updating of the dialog. It is updated every node and edge created. However, it is good to see the status of the loading if the graph takes minutes to load, so it was not desirable to remove it completely. The solution for this problem was to increase the progressbar only one time per hundred node or edge. This was done by a modulus test:

```java
if(i % 100 == 0)
    diag.increase();
```

Then the maximum progressbar number must be divided with 100.

5.6.2 Benchmarking the Start-up time
The start-up time was measured with the time method described in chapter 4.1.2. The time was started before building the scene and stopped when the scene had been created. The loading time for the VizzAnalyzer graph (VA.gml) after the Start-up time optimisation is shown in figure 5.5. The figure shows that the time decreased with 92% from 2.33 s to 0.18 s for the JOGL version.
Start-up time results

![Start-up time results](image)

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java3D (-24%)</td>
<td>13.3</td>
<td>10.1</td>
</tr>
<tr>
<td>GL4Java (-94%)</td>
<td>2.24</td>
<td>0.14</td>
</tr>
<tr>
<td>JOGL (-92%)</td>
<td>2.33</td>
<td>0.18</td>
</tr>
</tbody>
</table>

**Figure 5.5** Start-up time results.
6 Conclusions and future work

6.1 Conclusions
The problem tackled within this thesis is the lack of optimisation of Vizz3D to visualize huge graphs with acceptable performance.

The previous chapters have described that solving the problem was a success and that a maintainable and performance improved system came out from this thesis.

The solutions to the goals connected to the problem are summarized in the following sections.

6.1.1 Maintainability
The solving of the first goal is described in chapter 3. The goal was to ensure the maintainability of Vizz3D for the future.

To extend Vizz3D with a JOGL version when there already existed a GL4Java version proved quite straightforward, as they are very similar. The extension was also made easier since interfaces and abstract classes already were implemented in the thesis Graph visualization with OpenGL [7].

The criterion for the first goal was that the new JOGL version should have the same functionality as the currently used GL4Java version when ready. Chapter 3.2 discussed how the implementation of the new extension started out from the GL4Java version and that it therefore had the same functionality when it was ready.

After solving the first goal there were a new version of Vizz3D using JOGL as graphics API. This means that Vizz3D was now armed for future releases since it’s up to date with current technology.

6.1.2 Performance
The second goal was to improve the performance of Vizz3D by optimising OpenGL API calls and Java code. The goal was met when the system performance according to five performance aspects had been improved.

The optimisations that improved the Computational performance of Vizz3D are described in chapter 5.2. Several optimisations were done which led to that the FPS number was improved from 7 to 21 when rendering the VizzAnalyzer graph (see chapter 5.2.9 for more information about the VizzAnalyzer graph). This means that the FPS number was improved with 200 %. Before start optimising the system one could guess at an improvement of 2-3 times, so the result 200 % was as expected. This result led to that the visualizations now work much smoother when applying operations like rotation, zooming and translation to the graphs.

An optimisation that improved the Scalability is described in chapter 5.3. It was about reducing the primitive quality in OpenGL. Since some optimisations from the other performance aspects also contribute to the Scalability the most obvious method to evaluate the results was the frames per seconds method already used. When measuring the FPS number a huge graph (The VizzAnalyzer graph) was therefore chosen to test the scalability. Another evaluation method was also used. This was to test how many nodes that actually could be drawn at the speed 15 FPS. After optimisation 3640 nodes could be drawn with 1790 as starting number, which was an improvement by 103 %. However, the JOGL version can still handle larger graphs but with slower speed than 15 FPS (for example 10 000 nodes at 6 FPS). The two evaluation methods both showed that Vizz3D scaled well after the optimisations compared to prior.

The optimisations that improved the Perceived performance is described in chapter 5.4. Both optimisations were about threads. Since threading just change how the user...
experiences (feels) an application there is no obvious way to evaluate this method by numbers. But since Vizz3D felt a lot faster after the optimisations had been applied the optimisations had a positive effect.

The optimisations that improved the RAM footprint are described in chapter 5.5. Optimisations from the other aspects also contribute to the RAM footprint and after all had been applied the memory usage was 15 MB with 19 MB as starting value. This means that the JOGL version uses a very small amount of memory when visualizing graphs.

An optimisation that improved the Start-up time is described in chapter 5.6. It was about updating the JProgressBarDialog. After this optimisation the Start-up time was 0.18 s with 2.33 s as starting value. This means that the graph loading now is 92 % faster than prior the optimisation and that the waiting time now is a lot shorter.

After solving the second goal the performance of Vizz3D was improved a lot. To improve the performance further an automated optimisation technique could be used. Since both the first goal and second goal now was met the problem of this thesis was now solved, meaning that Vizz3D can visualize huge graphs with acceptable performance.

### 6.2 Future work

The present implementations work well, with the JOGL version as the one with the best performance. However, as any implementation it can be improved. A goal one could pursue is to increase the users understanding of the graphs by adding more functionality to the system. A new function can for example be rotation around three axes, which will lead to better interaction and orientation in the graphs.

Other implementations might also be considered. The API Xith3D [12] is already considered. This API has features nearly identical to Java3D and its scenegraph. However, Xith3D also allows native calls to OpenGL with the help of the JOGL implementation.

Vizz3D could probably also be improved by automate optimisations. A system could for example be used that suggests the improvements done manually in this thesis. The improvement of the FPS number with 200 % was as expected, but an automation of the optimisations could lead to further improvement of the FPS number so that Vizz3D can visualize really huge graphs (having for example 50 000 nodes and 100 000 edges).
References


[12] Xith3D homepage. 05-09-16 http://xith.org/


Appendix I – Profiling output

This appendix contains the complete profiling output before optimisation. A part of the output is described in chapter 5.1.

CPU SAMPLES BEGIN (total = 3159) Tue Apr 05 15:48:34 2005
rank self acc  count trace method
1 50.27%  50.27%  1588  4728 sun.awt.windows.WToolkit.eventLoop
2 35.23%  85.50%  1113 25520 net.java.games.jogl.impl.windows.WGL.NativeEventLoop
3  5.19%  90.69%   164  200 java.io.FileInputStream.open
4  2.18%  92.88%    69   312 java.io.WinNTFileSystem.getBooleanAttributes
5  0.57%  93.45%    18  25825 sun.awt.windows.WComponentPeer.nativeHandleEvent
6  0.22%  93.67%    7 11545 java.lang.Thread.setPriority0
7  0.22%  93.89%    7  25814 java.lang.Object.wait
8  0.16%  94.05%    5  18630 sun.awt.windows.WGlobalCursorManager.findHeavyweightUnderCursor
9  0.16%  94.21%    5  25823 net.java.games.jogl.impl.windows.WindowsGLImpl.glCallList
10 0.16%  94.37%     5   76 java.util.zip.ZipFile.open
11 0.16%  94.52%     5  4604 sun.awt.windows.Win32SurfaceData.initDDraw
12 0.16%  94.68%     5  25902 sun.awt.windows.WToolkit.shutdown
13 0.13%  94.81%     4  25844 net.java.games.jogl.impl.windows.WGL.SwapBuffers
14 0.13%  94.94%     4  11368 java.io.FileInputStream.open
15 0.09%  95.03%     3  6718 sun.awt.AppContext.get
16 0.06%  95.09%     2 12702 sun.awt.Win32GraphicsDevice.enumDisplayModes
17 0.06%  95.16%     2  410 java.lang.String.toLowerCase
18 0.06%  95.22%     2  18630 java.awt.MediaTracker.setDone
19 0.06%  95.28%     2  17013 java.io.WinNTFileSystem.getBooleanAttributes
20 0.06%  95.35%     2  156 java.lang.StringBuffer.expandCapacity
21 0.06%  95.41%     2  11143 sun.awt.image.ImageWatched.newInfo
22 0.06%  95.47%     2   88 java.util.zip.ZipFile.getEntry
23 0.06%  95.54%     2  309 java.lang.ClassLoader.defineClass0
24 0.06%  95.60%     2  17430 java.io.WinNTFileSystem.getBooleanAttributes
25 0.06%  95.66%     2  3210 java.util.zip.ZipFile.getEntry
26 0.06%  95.73%     2   91 java.lang.StringBuffer.toString
27 0.06%  95.79%     2  25881 java.lang.Thread.sleep
28 0.06%  95.85%     2  19 java.lang.ClassLoader.findBootstrapClass
29 0.06%  95.92%     2   89 java.lang.StringBuffer.<init>
30 0.06%  95.98%     2  11318 java.lang.Thread.setPriority0
31 0.06%  96.04%     2  11088 java.io.FileInputStream.close0
32 0.06%  96.11%     2  10955 java.util.Vector.<init>
33 0.03%  96.14%     1  22710 sun.awt.shell.Win32ShellFolder2.getDisplayNameOf
34 0.03%  96.17%     1  1912 java.lang.Class.getSuperclass
35 0.03%  96.20%     1  18371 java.awt.Component.setBackground
36 0.03%  96.23%     1  5478 sun.awt.AppContext.get
37 0.03%  96.26%     1  2718 java.math.BigInteger.<init>
38 0.03%  96.30%     1  8443 javax.swing.plaf.basic.BasicListUI$PropertyChangeHandler.propertyChange
<table>
<thead>
<tr>
<th>Line</th>
<th>Percentage</th>
<th>Time</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>0.03%</td>
<td>1 17026</td>
<td>java.lang.Class.forName0</td>
</tr>
<tr>
<td>40</td>
<td>0.03%</td>
<td>1 3461</td>
<td>java.lang.StringBuffer.expandCapacity</td>
</tr>
<tr>
<td>41</td>
<td>0.03%</td>
<td>1 25801</td>
<td>java.awt.EventQueue.setCurrentEventAndMostRecentTimeImpl</td>
</tr>
<tr>
<td>42</td>
<td>0.03%</td>
<td>1 25879</td>
<td>visgraph.gl.sceneGraph.GLGraph.getNode</td>
</tr>
<tr>
<td>43</td>
<td>0.03%</td>
<td>1 12634</td>
<td>sun.awt.AppContext.get</td>
</tr>
<tr>
<td>44</td>
<td>0.03%</td>
<td>1 3096</td>
<td>sun.security.provider.SHA.computeBlock</td>
</tr>
<tr>
<td>45</td>
<td>0.03%</td>
<td>1 16222</td>
<td>java.lang.String.&lt;init&gt;</td>
</tr>
<tr>
<td>46</td>
<td>0.03%</td>
<td>1 6591</td>
<td>sun.awt.AppContext.get</td>
</tr>
<tr>
<td>47</td>
<td>0.03%</td>
<td>1 7035</td>
<td>sun.awt.windows.WWindowPeer.&lt;clinit&gt;</td>
</tr>
<tr>
<td>48</td>
<td>0.03%</td>
<td>1 311</td>
<td>java.net.URLClassLoader.findClass</td>
</tr>
<tr>
<td>49</td>
<td>0.03%</td>
<td>1 11187</td>
<td>java.awt.MediaTracker.setDone</td>
</tr>
<tr>
<td>50</td>
<td>0.03%</td>
<td>1 100</td>
<td>java.util.jar.JarFile.hasClassPathAttribute</td>
</tr>
<tr>
<td>51</td>
<td>0.03%</td>
<td>1 18604</td>
<td>java.awt.image.DirectColorModel.getRGB</td>
</tr>
<tr>
<td>52</td>
<td>0.03%</td>
<td>1 2814</td>
<td>java.lang.Throwable.fillInStackTrace</td>
</tr>
<tr>
<td>53</td>
<td>0.03%</td>
<td>1 12621</td>
<td>java.awt.RepaintManager.currentManager</td>
</tr>
<tr>
<td>54</td>
<td>0.03%</td>
<td>1 3879</td>
<td>java.lang.StringBuffer.toString</td>
</tr>
<tr>
<td>55</td>
<td>0.03%</td>
<td>1 12740</td>
<td>java.awt.plaf.basic.BasicLabelUI.getPreferredSize</td>
</tr>
<tr>
<td>56</td>
<td>0.03%</td>
<td>1 433</td>
<td>java.util.HashMap.hash</td>
</tr>
<tr>
<td>57</td>
<td>0.03%</td>
<td>1 18245</td>
<td>sun.awt.windows.Win32D3DRenderer.doDrawRectD3D</td>
</tr>
<tr>
<td>58</td>
<td>0.03%</td>
<td>1 15547</td>
<td>sun.awt.windows.Win32SurfaceData.invalidateSD</td>
</tr>
<tr>
<td>59</td>
<td>0.03%</td>
<td>1 25527</td>
<td>net.java.games.jogl.impl.windows.WGL.ChoosePixelFormat0</td>
</tr>
<tr>
<td>60</td>
<td>0.03%</td>
<td>1 3895</td>
<td>java.io.WinNTFileSystem.list</td>
</tr>
<tr>
<td>61</td>
<td>0.03%</td>
<td>1 245</td>
<td>java.lang.String.replace</td>
</tr>
<tr>
<td>62</td>
<td>0.03%</td>
<td>1 6</td>
<td>java.lang.ClassLoader.findLoadedClass</td>
</tr>
<tr>
<td>63</td>
<td>0.03%</td>
<td>1 197</td>
<td>sun.misc.URLClassPath$7.getInputStream</td>
</tr>
<tr>
<td>64</td>
<td>0.03%</td>
<td>1 17903</td>
<td>sun.awt.windows.WGlobalCursorManager.findHeavyweightUnderCursor</td>
</tr>
<tr>
<td>65</td>
<td>0.03%</td>
<td>1 11087</td>
<td>java.io.WinNTFileSystem.getBooleanAttributes</td>
</tr>
<tr>
<td>66</td>
<td>0.03%</td>
<td>1 1221</td>
<td>sun.security.x509.X500Name.parseDER</td>
</tr>
<tr>
<td>67</td>
<td>0.03%</td>
<td>1 3095</td>
<td>java.io.PushbackInputStream.read</td>
</tr>
<tr>
<td>68</td>
<td>0.03%</td>
<td>1 17960</td>
<td>java.awt.Component.location</td>
</tr>
<tr>
<td>69</td>
<td>0.03%</td>
<td>1 26</td>
<td>java.lang.String.concat</td>
</tr>
<tr>
<td>70</td>
<td>0.03%</td>
<td>1 10941</td>
<td>sun.awt.image.ImageRepresentation.setPixels</td>
</tr>
<tr>
<td>71</td>
<td>0.03%</td>
<td>1 23701</td>
<td>sun.awt.shell.Win32ShellFolder2.getEnumObjects</td>
</tr>
<tr>
<td>72</td>
<td>0.03%</td>
<td>1 16211</td>
<td>java.lang.String.substring</td>
</tr>
<tr>
<td>73</td>
<td>0.03%</td>
<td>1 11034</td>
<td>sun.awt.image.ImageWatched.newInfo</td>
</tr>
<tr>
<td>74</td>
<td>0.03%</td>
<td>1 5409</td>
<td>java.swing.JTable.createDefaultColumnsFromModel</td>
</tr>
<tr>
<td>75</td>
<td>0.03%</td>
<td>1 25766</td>
<td>net.java.games.jogl.impl.windows.WindowsGLImpl.glPushAttrib</td>
</tr>
<tr>
<td>76</td>
<td>0.03%</td>
<td>1 17561</td>
<td>javax.swing.event.EventListenerList.add</td>
</tr>
<tr>
<td>77</td>
<td>0.03%</td>
<td>1 23997</td>
<td>java.util.Hashtable.put</td>
</tr>
<tr>
<td>78</td>
<td>0.03%</td>
<td>1 11793</td>
<td>java.lang.Object.wait</td>
</tr>
<tr>
<td>79</td>
<td>0.03%</td>
<td>1 11460</td>
<td>sun.awt.image.ImageWatched.newInfo</td>
</tr>
<tr>
<td>80</td>
<td>0.03%</td>
<td>1 20753</td>
<td>java.io.Win32FileSystem.prefixLength</td>
</tr>
<tr>
<td>81</td>
<td>0.03%</td>
<td>1 25590</td>
<td>java.awt.windows.WGlobalCursorManager.findHeavyweightUnderCursor</td>
</tr>
<tr>
<td>82</td>
<td>0.03%</td>
<td>1 7094</td>
<td>sun.awt.windows.WComponentPeer.setFont</td>
</tr>
<tr>
<td>83</td>
<td>0.03%</td>
<td>1 15483</td>
<td>sun.awt.windows.WGlobalCursorManager.findHeavyweightUnderCursor</td>
</tr>
</tbody>
</table>
Appendix I – Profiling output

<table>
<thead>
<tr>
<th></th>
<th>Percent</th>
<th>Total</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td>0.03%</td>
<td>18803</td>
<td>java.lang.String.replace</td>
</tr>
<tr>
<td>85</td>
<td>0.03%</td>
<td>2222</td>
<td>java.lang.Class.forName0</td>
</tr>
<tr>
<td>86</td>
<td>0.03%</td>
<td>6727</td>
<td>java.util.HashMap.addEntry</td>
</tr>
<tr>
<td>87</td>
<td>0.03%</td>
<td>12815</td>
<td>java.awt.GridBagLayout.&lt;init&gt;</td>
</tr>
<tr>
<td>88</td>
<td>0.03%</td>
<td>2586</td>
<td>java.util.Locale.getDefault</td>
</tr>
<tr>
<td>89</td>
<td>0.03%</td>
<td>430</td>
<td>java.util.HashMap.addEntry</td>
</tr>
<tr>
<td>90</td>
<td>0.03%</td>
<td>2077</td>
<td>sun.security.pkcs.PKCS9Attribute.&lt;cinit&gt;</td>
</tr>
<tr>
<td>91</td>
<td>0.03%</td>
<td>21066</td>
<td>java.io.DataInputStream.readUTF</td>
</tr>
<tr>
<td>92</td>
<td>0.03%</td>
<td>22699</td>
<td>java.lang.String.toLowerCase</td>
</tr>
<tr>
<td>93</td>
<td>0.03%</td>
<td>11106</td>
<td>java.io.FileInputStream.close0</td>
</tr>
<tr>
<td>94</td>
<td>0.03%</td>
<td>6304</td>
<td>sun.awt.font.NativeFontCase.registerFonts</td>
</tr>
<tr>
<td>95</td>
<td>0.03%</td>
<td>2870</td>
<td>java.io.ByteArrayOutputStream.toByteArray</td>
</tr>
<tr>
<td>96</td>
<td>0.03%</td>
<td>11726</td>
<td>java.util.zip.Inflater.inflateBytes</td>
</tr>
<tr>
<td>97</td>
<td>0.03%</td>
<td>24366</td>
<td>sun.awt.windows.WGlobalCursorManager.findHeavyweightUnderCursor</td>
</tr>
<tr>
<td>98</td>
<td>0.03%</td>
<td>5047</td>
<td>java.awt.peer.BasicLookAndFeel.initComponentDefaults</td>
</tr>
<tr>
<td>99</td>
<td>0.03%</td>
<td>12042</td>
<td>vizz3d.common.gui.guipanels.action.ActionDelegator.&lt;init&gt;</td>
</tr>
<tr>
<td>100</td>
<td>0.03%</td>
<td>4700</td>
<td>java.awt.EventQueue.&lt;init&gt;</td>
</tr>
<tr>
<td>101</td>
<td>0.03%</td>
<td>11089</td>
<td>java.awt.image.Raster.createPackedRaster</td>
</tr>
<tr>
<td>102</td>
<td>0.03%</td>
<td>11188</td>
<td>java.util.Vector.elements</td>
</tr>
<tr>
<td>103</td>
<td>0.03%</td>
<td>10884</td>
<td>java.lang.Thread.setPriority0</td>
</tr>
<tr>
<td>104</td>
<td>0.03%</td>
<td>11020</td>
<td>java.util.Vector.elements</td>
</tr>
<tr>
<td>105</td>
<td>0.03%</td>
<td>18219</td>
<td>java.lang.Object.clone</td>
</tr>
<tr>
<td>106</td>
<td>0.03%</td>
<td>18608</td>
<td>java.awt.image.ImageRepresentation.setPixels</td>
</tr>
<tr>
<td>107</td>
<td>0.03%</td>
<td>9471</td>
<td>sun.awt.windows.WComponentPeer.endValidate</td>
</tr>
<tr>
<td>108</td>
<td>0.03%</td>
<td>23703</td>
<td>sun.awt.shell.Win32ShellFolder2.getAttributes0</td>
</tr>
<tr>
<td>109</td>
<td>0.03%</td>
<td>2868</td>
<td>java.io.PushbackInputStream.&lt;init&gt;</td>
</tr>
<tr>
<td>110</td>
<td>0.03%</td>
<td>3421</td>
<td>java.lang.Class.forName0</td>
</tr>
<tr>
<td>111</td>
<td>0.03%</td>
<td>2470</td>
<td>sun.security.util.DerValue.toByteArray</td>
</tr>
<tr>
<td>112</td>
<td>0.03%</td>
<td>12201</td>
<td>java.io.WinNTFileSystem.getLength</td>
</tr>
<tr>
<td>113</td>
<td>0.03%</td>
<td>20</td>
<td>java.lang.Throwale.fillInStackTrace</td>
</tr>
<tr>
<td>114</td>
<td>0.03%</td>
<td>2817</td>
<td>java.util.Hashtable.put</td>
</tr>
<tr>
<td>115</td>
<td>0.03%</td>
<td>12737</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>0.03%</td>
<td>25818</td>
<td>net.java.games.jogl.impl.GLContextStack.push</td>
</tr>
<tr>
<td>117</td>
<td>0.03%</td>
<td>10197</td>
<td>javax.swing.SwingUtilities.appContextGet</td>
</tr>
<tr>
<td>118</td>
<td>0.03%</td>
<td>16227</td>
<td>java.lang.String.&lt;init&gt;</td>
</tr>
<tr>
<td>119</td>
<td>0.03%</td>
<td>11625</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>0.03%</td>
<td>3941</td>
<td>java.lang.StringBuffer.toString</td>
</tr>
<tr>
<td>121</td>
<td>0.03%</td>
<td>25769</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>0.03%</td>
<td>17798</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>0.03%</td>
<td>14563</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>0.03%</td>
<td>23940</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>0.03%</td>
<td>25882</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>0.03%</td>
<td>17798</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.03%</td>
<td>99.11%</td>
<td>1 411 java.lang.String.toLowerCase</td>
</tr>
<tr>
<td>---</td>
<td>-------</td>
<td>--------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>127</td>
<td>0.03%</td>
<td>99.15%</td>
<td>1 11186 java.lang.Object.wait</td>
</tr>
<tr>
<td>128</td>
<td>0.03%</td>
<td>99.18%</td>
<td>1 24368 vizz3d.jogl.visengine.JOGLMLSceneGraph.&lt;init&gt;</td>
</tr>
<tr>
<td>129</td>
<td>0.03%</td>
<td>99.21%</td>
<td>1 7074 sun.awt.windows.WComponentPeer.setZOrderPosition</td>
</tr>
<tr>
<td>130</td>
<td>0.03%</td>
<td>99.24%</td>
<td>1 163 java.lang.String.substring</td>
</tr>
<tr>
<td>131</td>
<td>0.03%</td>
<td>99.27%</td>
<td>1 23899 sun.awt.windows.WComponentPeer.findHeavyweightUnderCursor</td>
</tr>
<tr>
<td>132</td>
<td>0.03%</td>
<td>99.30%</td>
<td>1 639 sun.security.util.ManifestDigester.&lt;init&gt;</td>
</tr>
<tr>
<td>133</td>
<td>0.03%</td>
<td>99.34%</td>
<td>1 11132 java.util.Vector.elements</td>
</tr>
<tr>
<td>134</td>
<td>0.03%</td>
<td>99.37%</td>
<td>1 15826 sun.awt.windows.WComponentPeer.findHeavyweightUnderCursor</td>
</tr>
<tr>
<td>135</td>
<td>0.03%</td>
<td>99.40%</td>
<td>1 176 java.lang.StringBuffer.toString</td>
</tr>
<tr>
<td>136</td>
<td>0.03%</td>
<td>99.43%</td>
<td>1 12202 java.util.zip.ZipFile.getEntry</td>
</tr>
<tr>
<td>137</td>
<td>0.03%</td>
<td>99.46%</td>
<td>1 2172 java.util.jar.ManifestFastInputStream.peek</td>
</tr>
<tr>
<td>138</td>
<td>0.03%</td>
<td>99.49%</td>
<td>1 9079 sun.awt.font.FontDesignMetrics.handleCharWidth</td>
</tr>
<tr>
<td>139</td>
<td>0.03%</td>
<td>99.52%</td>
<td>1 6246 java.util.Hashtable.put</td>
</tr>
<tr>
<td>140</td>
<td>0.03%</td>
<td>99.56%</td>
<td>1 18135 sun.awt.windows.WComponentPeer.endValidate</td>
</tr>
<tr>
<td>141</td>
<td>0.03%</td>
<td>99.59%</td>
<td>1 25300 sun.awt.windows.WComponentPeer.hide</td>
</tr>
<tr>
<td>142</td>
<td>0.03%</td>
<td>99.62%</td>
<td>1 10942 java.util.Vector.elements</td>
</tr>
<tr>
<td>143</td>
<td>0.03%</td>
<td>99.65%</td>
<td>1 11007 sun.awt.image.ByteComponentRaster.&lt;init&gt;</td>
</tr>
<tr>
<td>144</td>
<td>0.03%</td>
<td>99.68%</td>
<td>1 13424 sun.awt.AWTAutoShutdown.getInstance</td>
</tr>
<tr>
<td>145</td>
<td>0.03%</td>
<td>99.71%</td>
<td>1 18084 java.awt.EventQueue.postEvent</td>
</tr>
<tr>
<td>146</td>
<td>0.03%</td>
<td>99.75%</td>
<td>1 19892 sun.awt.shell.Win32ShellFolder2.getNextChild</td>
</tr>
<tr>
<td>147</td>
<td>0.03%</td>
<td>99.78%</td>
<td>1 201 sun.misc.Resource.getBytes</td>
</tr>
<tr>
<td>148</td>
<td>0.03%</td>
<td>99.81%</td>
<td>1 3211 java.util.zip.ZipFile.getEntry</td>
</tr>
<tr>
<td>149</td>
<td>0.03%</td>
<td>99.84%</td>
<td>1 23927 java.lang.Object.wait</td>
</tr>
<tr>
<td>150</td>
<td>0.03%</td>
<td>99.87%</td>
<td>1 23674 java.awt.Container.getInsets</td>
</tr>
<tr>
<td>151</td>
<td>0.03%</td>
<td>99.90%</td>
<td>1 25883 sun.awt.windows.WComponentPeer.findHeavyweightUnderCursor</td>
</tr>
<tr>
<td>152</td>
<td>0.03%</td>
<td>99.93%</td>
<td>1 16220 java.lang.StringBuffer.toString</td>
</tr>
<tr>
<td>153</td>
<td>0.03%</td>
<td>99.96%</td>
<td>1 2164 java.util.HashMap.&lt;init&gt;</td>
</tr>
<tr>
<td>154</td>
<td>0.03%</td>
<td>100.00%</td>
<td>1 152 sun.misc.URLClassPath$FileLoader.getResource</td>
</tr>
</tbody>
</table>

CPU SAMPLES END