Evaluation of 3D Scene Graph APIs for Java

Thomas Winger

Master Thesis

May 2012

Faculty of Computer Sciences,
Østfold University College

Abstract

This thesis presents an evaluation of a set of 3D Scene Graph APIs for Java. The work consist mainly of two parts: Defining a methodology for comparing the APIs, and then applying the proposed methodology to the APIs.

An overview of the available 3D Scene Graph APIs in Java is presented, and a selection of these are chosen for the evaluation. The APIs subjected to the evaluation are Java 3D, Ardor3D and jMonkeyEngine3.

The proposed methodology focuses on the comparison on four different aspects. These are: *Project Management and Technical Infrastructure*, *System Architecture*, *System Features and Capabilities*, and *System Performance*.

The results from applying the evaluation method show that none of the APIs were superior to the others in all respects. The results identify strengths and weaknesses with each API, that indicate which use cases each API might be better suited for.

Keywords: Scene Graph, API, Evaluation, Java, 3D Graphics, OpenGL, Java3D, jMonkeyEngine3, Ardor3D

Acknowledgements

I wish to thank Børre Stenseth for the support, proof reading and critical discussions during this thesis, and throughout my years at Østfold University College. Thanks to Michael Louka for assisting me in defining the methodology, insightful discussions, and in proof reading this thesis. I also want to thank Tom-Robert Bryntesen for the inspiring technical discussions, and the assistance throughout this thesis.

I would also like to thank my friends, family and Mia for your continued support throughout my five years studying in Halden. Without your support and understanding, none of this would have been possible.

Prerequisites

This thesis covers many aspects related to computer science, with a special focus on computer graphics. It is not possible to go into detail on every subject covered in this thesis, therefore it is assumed that the reader has a basic understanding of 3D computer graphics, as well as a general knowledge about the programming language Java.

In the Background chapter, information about some of the most important aspects is given.

A glossary of words and expressions is included in Appendix A.

Table of Contents

Chapter 1

Introduction

3D Scene Graph APIs in Java are becoming increasingly powerful, and offer a wide range of features and capabilities. The constant increase in hardware capabilities, combined with improvements made to Java makes it an attractive platform for 3D graphics. Java has been used for rendering 3D graphics within various fields in the industry for many years, especially within computer-aided design (CAD) and research. Within the entertainment industry, commercially successful games have not been very common in Java. However, with the recent trend towards smaller independent game studios (indie), Java is becoming an attractive platform for game development. These indie studios often focus on rapid development and multi-platform support. The recent multi-million success game Minecraft [1] is an example of this.

The strength of 3D Scene Graph APIs is that they let the developers focus on the creation of content and management of the scenes, rather than the intricacies in a 3D graphics engine. This greatly reduces complexity, and reduces development time and costs.

1.1 Motivation

There are many different 3D Scene Graph APIs available in Java, and they offer a wide variety of different features and capabilities. This makes it difficult to distinguish the APIs from one another, because they can seem very similar at first glance. It is also important to keep in mind that many of the APIs might claim, or give the impression that they offer certain functionality, while they in reality do not.

[1] Minecraft is a 3D game that randomly generates the game content, and lets the players build anything they can imagine. Website: `http://www.minecraft.net/`

It is important that potential users who are considering such APIs are well informed, and aware of their differences, similarities and limitations. Some aspects of the software are hard to grasp from just examining them briefly. One such factor is related to the management of the software development. The management and structure of the project or community may be so bad, that main developers might leave, or the user community give up.

Another aspect related to this is the design of the APIs. The API may be designed in such a way that it is impossible to add significant new functionality and features without rewriting major parts of it; or perhaps it is written and designed so poorly, that it is impractical to extend, and might eventually collapse under its own weight.

To my knowledge there has not been published any previous work that examines, and compares various 3D Scene Graph APIs in Java. Therefore it is imperative that more research is done around this. This will help inform existing users and developers of the various APIs, and make them more aware of potential shortcomings or differences. It will also help new users when choosing which API to use, because they are able to make a more educated choice.

This thesis work has been done in cooperation with the Institute for Energy Technology (IFE) [1]. IFE is an international research institute, that specializes in research within energy and nuclear technology [2]. They use the 3D Scene Graph API Java 3D for their visualizations. IFE would like to change to a newer API due to various reasons, including limitations in Java 3Ds design, and also because Java 3D is not actively maintained any more. IFE is therefore interested in migrating to a newer API, and thus needed to compare the different APIs available in Java against each other.

Motivated by these factors, it is important to provide a study that thoroughly investigates and analyses the various 3D Scene Graph API alternatives in Java.

1.2 Research Objectives

The work in this thesis is an evaluation of a set of 3D Scene Graph APIs for Java [3]. The work consists of two parts. The first part involves finding, or defining a methodology to use for evaluating the APIs. This gives the first objective for this research:

1. Identify or develop a suitable methodology for evaluating 3D Scene Graph APIs.

[1] `http://www.ife.no/`

[2] IFE also do research within other fields, including oil and gas, architecture, electricity production, and transportation.

[3] The APIs that will be subject of this evaluation are discussed in Section 1.4.

The other part of this research consists of applying the chosen methodology to the evaluation of the APIs. This gives the second research objective:

2. Evaluate the APIs using the chosen methodology.

1.3 Methodology

The methodology used for the evaluation in this thesis uses an experimental approach. The methodology is explained in greater detail in Chapter 4. In short, no sufficient existing methodology was found, so it was necessary to define a methodology specific to this thesis. The proposed methodology combines some existing frameworks with some that were defined specifically for this evaluation. Together they are used to evaluate the different aspects of the API software systems as thorough and as detailed as required for this analysis. The four different layers are:

1. Project Management and Technical Infrastructure.
2. System Architecture.
3. System Features and Capabilities.
4. System Performance.

Each of these layers is investigated, using different methods specific to the layer. The results from each layer are then used to evaluate the API as a whole.

1.4 Selection of APIs

In order to do the evaluation as extensively and in-depth as the research objectives require, it was decided that the evaluation should be limited to only some of the 3D Scene Graph APIs available in Java. If I was to include too many APIs, there would not be enough time to do this as detailed as desired, and therefore I chose to prioritise quality over quantity.

The study in Section 3 investigates the most prominent 3D Scene Graph APIs available for use for Java application development. This study was done to highlight various traits and aspects of the software projects. This was used to decide which APIs should be included in the detailed evaluation. The selection was done in cooperation with IFE. The APIs that were chosen were *Java 3D*, *Ardor3D* and *jMonkeyEngine3*. Java 3D was chosen because it is widely used in the industry, including fields such as research, visualization and various CAD-fields (computer-aided design) and

was the standard scene graph API for Java development by Sun Microsystems. Another reason for choosing Java 3D is because IFE is currently using Java 3D for their visualizations, and they want a comparison of it against the other candidates.

Other candidates were chosen because they had some characteristics that were considered important. First and foremost they are still actively developed, opposed to many of the other APIs available in Java. This means that the results from this thesis may also help the developers to further improve their APIs. Ardor3D was chosen because they have profiled themselves to be clearly aimed at providing services for the industry, with a company backing them up. Their software is used by major companies, including NASA and Rolls Royce Marine. While the main focus for Ardor3D is towards industry, the API is also being used to develop games. jMonkeyEngine3 was chosen because it is cutting-edge in terms of technology, and features a fully shader-based architecture. jMonkeyEngine3 is geared mostly towards games.

By focusing the evaluation on these three APIs, the thesis is able to go much more into detail about each one of them.

1.5 Outline

Chapter 1, Introduction: Explains the motivation behind the thesis, the research objectives, selection of which APIs to evaluate and a brief summary of the methodology.

Chapter 2, Background: Gives a brief introduction to some fundamental concepts related to computer graphics and scene graphs. A look at related and previous work regarding evaluation of projects or other aspects, as well as some concepts used in the evaluation in this thesis.

Chapter 3, Investigation of 3D Scene Graph APIs: Investigates the available 3D Scene Graph APIs in Java. Investigates not only the three APIs that this evaluation will focus on, but also other 3D APIs available in Java.

Chapter 4, Evaluation Methodology: Explains the methodology used in this thesis in greater detail. The methodology consists of a combination of various frameworks, including project maturity, metrics extracted from design structure matrices, tables for various features and capabilities, and benchmarks for assessing performance.

Chapter 5, Testbed Design and Implementation: Explains the design and implementation of the benchmarks used for testing the performance of the APIs.

Chapter 6, 3D Scene Graph APIs Evaluation Results: Presents the results and findings from applying the research methods proposed in this thesis.

Chapter 7, Summary, Discussion, Future Work and Conclusions: Gives a summary of the work done in this thesis, discusses the findings and results, comments about future work, and concludes the work in this thesis.

Appendix A: A glossary of words and expressions used throughout the thesis.

Chapter 2

Background

This chapter gives an introduction to the main concepts relevant to this thesis, as well as looking into some previous and related work.

Section 2.1 presents some related work. This includes various methods for investigating aspects of APIs, as well as some benchmarks that investigates the performance of 3D Scene Graph APIs. Section 2.2 explains what real-time 3D graphic libraries are, and which are the standards in the industry today. Section 2.3 explains the two main methods for rendering computer graphics, immediate- and retained-mode. Section 2.4 explains what a scene graph is, and how that relates to a 3D Scene Graph API. Section 2.5 explains shaders, which lets programmers program parts of the rendering pipeline. Section 2.6 describes the different techniques used for rendering stereoscopic images. Section 2.7 describes design structure matrices, which is a way to represent the connections between parts of a system. Section 2.8 describes various licenses that software are distributed under.

2.1 Related Work

To my knowledge, there have not been done any previous work that does a comprehensive and thorough evaluation of various 3D Scene Graph APIs for Java. There have been some related work, that touches upon some of the aspects covered in the evaluation in this thesis. This is mainly informal discussions on various discussion boards and blogs on the internet.

In this section we will present some related work that touches upon some of the aspects related to the evaluation of an API. Section 2.1.1 will look at previous work that evaluates open source projects. Section 2.1.2 will look at previous work that investigates and compares the performance between different solutions.

2.1.1 Evaluation of Open Source Projects

Open Source Software (OSS) differ in many ways from proprietary software. One of the main differences is that OSS is often developed by volunteers, opposed to hired people who are working for a firm. The teams working on proprietary software often work at the same place, which lets them meet each other physically. With OSS the contributors may be located all around the globe.

There are both successful and unsuccessful OSS projects(and also proprietary). Finding out who is worth ”betting the money on” is not easy. This section will present some studies that presents methods for evaluating the successfulness of various OSS projects. This may help in choosing the ”right one”.

Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code

MacCormak et al [88] attempts to characterize the difference in design between proprietary software and OSS in their study. They use Design Structure Matrices (DSMs) to define metrics to compare the structures of the different designs. They compare the architectures of the Mozilla web browser (known now as Mozilla Firefox) (proprietary) and the Linux Operating System Kernel (OSS), with emphasis on the modularity of the two products.

Their study aims to answer two research questions. First, they look at whether there are differences between software products that are developed with different organizational modes, This means OSS versus proprietary. Secondly, they seek to examinate the re-design of a product's architecture, where the goal was to make it more modular. For this they investigate the Mozilla web browser before and after the major re-design done in the fall of 1998.

In order to study this, they use Design Structure Matrices (DSM) [1] to analyse the designs of the software products. The costs of the dependencies between the elements in the software is used to measure the modularity of the two. The dependencies between the elements are function calls between the source files. They define two metrics that captures the costs, that highlights different. These metrics are the *propagation cost*, and the *clustered cost*. The former is the degree of coupling in the system, relating to the dependency between various elements, i.e. a change to A affects B. The longer the dependency chain, the higher grading are given to the cost value. The latter metric refers to the dependency cost, taking into account which module the element (A) belongs to, in regard to it's dependency to the other element (B). A closer hierarchical location results in a lower

[1] Design Structure Matrices is explained in further detail in Section 2.7

grading, than one with a greater distance.

The results from the comparison between Linux and Mozilla showed that Mozilla had a much higher *propagation cost* than Linux. Mozilla had 17.35%, opposed to 5.16% in Linux. This means that Linux is much more loosely-coupled than Mozilla. A change to a source file in Mozilla has the chance to impact three times as many files on average, as in Linux. The *clustered cost* of Mozilla was 70% higher than that of Linux. They also found that the dependency density (measured per 1000 source file pairs) for Linux was 40% higher than that of Mozilla. This means that there were more dependencies in Linux. This might seem contradictory with the clustered cost, but this is because there is a larger hierarchical distance between the dependencies in Mozilla, than in Linux. They also noted that the source files of Mozilla contained 50% more functions per file than Linux, although the lines of code did not differentiate much.

The results from the comparison between the Mozilla web browser before and after the re-design, showed that there were significant changes to the software. The architecture became much more modular than before. The number of source files, as well as the dependencies between the source files, dropped significantly. This resulted in the propagation cost dropping from 17.35% to 2.76%. This means that changes to source files have the potential to impact 80% fewer source files on average than before the re-design. This is also seen in the clustered cost, which was reduced by 30% of its previous value.

Comparing the Linux Kernel against the Mozilla web browser after the re-design, shows that the propagation cost of Mozilla dropped to half to that of Linux. The clustered cost also dropped to about half to that of Linux. The density of dependencies however, dropped only to 35% to that of Linux. This was surprising given the drastic drop of both the propagation and the clustered cost. The authors speculates that Mozilla's advantage in modularity comes from the reduction in the number of dependencies, rather than a superior configuration in terms of the pattern of distribution.

Their study indicates that a products architecture is dependent on the organizational structure of the development teams. The Linux Kernel was proven to be much more modular at a source file level than the Mozilla web browser. They also showed that Mozilla was able to significantly improve the modularity by a re-design of the architecture. This shows that it is possible to make proprietary software developed by a co-located team more modular, even though it was a conscious and deliberate decision by the development team.

Software Process Maturity and the Success of Free Software Projects

In this paper Michlmayr [90] seeks to determine if the maturity of different OSS projects differs between successful and unsuccessful projects. To investigate this, 80 projects from SourceForge [1] was chosen. Half of the projects were successful, and the other half were unsuccessful. The successfulness of the projects selected for the study was determined by looking at the download statistics from SourceForge. This was done with the assumption that successful projects would have more downloads than unsuccessful projects.

In order to evaluate what determined the successfulness of the projects, Michlmayr created a general mechanism to evaluate the process maturity of any OSS project. The mechanism he proposes focuses on important components in distributed projects, that are related to the maturity of the processes in the projects. In particular, the coordination and the communication between the developers of the project. The assessment method is grouped into five categories, which each takes different aspects of OSS projects into account. Each category consist of some questions which can be answered by a *yes* or *no*, or in some cases by a grade of zero, one or two points (the more points, the higher maturity). The five categories include: *1) Version Control*, *2) Mailing Lists*, *3) Documentation*, *4) Systematic Testing*, and *5) Portability*.

The results show that free software and open source projects profit from the use of mature processes. They show that the use of *version control* tools were more widely used in successful projects than in unsuccessful, and also the majority of repositories were available publicly. The use of version control helps multiple people work together in an efficient distributed way. It may also attract more volunteers because the users can see what needs to be done. The use of *mailing lists* also seemed to be a key factor in the success of a project, aiding in much of the coordination process of the projects. Mailing list archives also replaced documentation to some degree. The availability of *documentation* did not seem to give any clear indication of whether it was successful or not, however it seems that user documentation is a more important factor than developer documentation. This was supported by the fact that most developers would seek knowledge from the source code and follow discussions on the mailing lists. In regards to *systematic testing* it seems that successful projects make more use of release candidates, which can be taken as an indication of a well defined release plan for the project. Defect tracking systems was more used in successful projects, which serve a crucial role in receiving feedback from users, and lets them analyse and prioritize it accordingly. The presence of automated testing suites did not seem to impact the successfulness of the

[1] SourceForge is a free web-based source code repository that helps developers control and manage OSS development. Website: `http://sourceforge.net/`

projects. This might be because users often tracks them down and reports them through a defect issue tracking system. The last category, which was *portability* was very good in both the successful and unsuccessful projects, which is speculated to be because of the Unix philosophy [1]. The diverse nature of the participating volunteers is also believed to be a contributing factor to the portability of the software.

The author is certain that both version control and mailing lists are key factors in successful open source projects. Various testing strategies is also important. The presence of documentation was not proven to be of any major significance to the successfulness of the project. It was proven however that user documentation is more important than developer documentation.

In essence it seems that as a result of the nature of OSS projects, the successfulness of a project is often dependent on the presence of good tools for communication and coordination.

Producing Open Source Software

In the book *Producing Open Source Software* [85], Karl Fogel touches upon many of the same principals used for evaluating the successfulness of a OSS project as mentioned in the previous research. Fogel places emphasis on two key components of a successful OSS project, which is the *technical infrastructure* and the *social and political infrastructure*.

The technical infrastructure covers most of the tools mentioned in the previous paragraph, such as version control, mailing lists and bug tracking. In addition he talks about the use of real-time chat systems for direct communication, where users and developers can ask questions with instant responses. Other tools such as RSS-feeds are mentioned, as well as the use of Wikis. He makes the point that it is equally important that the tools are being used correctly, than just the mere presence of them. Tools should not be misused, like for example keeping discussions to mailing lists instead of in the bug tracker.

The social and political infrastructure looks at factors that not only addresses the projects successfulness by looking at the technical quality, but instead focuses more on the *operational health* and the *survivability* of the project. The operational health refers to the projects ongoing ability to add new code contributions and developers to the project, as well as its responsiveness to new bug reports. The survivability looks at the projects ability to exist independently of any individual contributor, either developer or sponsor. Would for example the project survive if a very charismatic person, or a set of developers left the project? This is a very important aspect of a OSS project,

[1] The Unix philosophy promotes portability.

because the success is not as much tied into the technical quality, as it is to a robust developer base and solid foundation.

Fogel describes two main ways that most OSS projects organize themselves, either with a *benevolent dictator (BD)* or a *consensus-based democracy (CBD)*. The BD organization means that the final decision-making rests with one person. This gives the person a great deal of power in theory, however in practice the dictator does not have any more power than the other contributors. If the dictator runs the project like a true dictatorship, most contributors would leave the project, and by the nature of OSS they could take the source code with them and just start a fork of the project. Because of this the dictator does not gain anything by misusing his powers, and in reality most decisions are done through discussions between the developers. In a CBD organization no-one person have any absolute power, and most questions are solved by discussions between the developers. Most OSS projects that have a BD organization, move to a CBD organization in time. It is a much more healthy organization structure, because it does not rely on any specific individuals, but rather the group as a whole is responsible for making the decisions.

2.1.2 Performance analysis

This section presents some previous work that investigates the performance of various 3D Scene Graph APIs, and graphic libraries in Java.

OpenGL performance in C++, Java and Java 3D

Jacob Marner did an evaluation of Java for game development in his masters thesis [89], comparing OpenGL performance, He used the graphic libraries directly, and compared the performance in Java and C++. He also added the 3D Scene Graph API Java 3D to the comparison. The thesis showed that the performance in Java tends to be slower than that of C++. Optimizations were able to improve the performance of Java, but at the cost of readability of the code. He suggests that the optimizations would remove the productivity benefit grained when using Java, which is one of the main *selling point* to using Java in the first place. It was also shown that using Java 3D were much slower than using the graphic libraries directly. This thesis was written in 2002, which is 10 years ago (2012 as of this writing). Computer hardware have advanced a great deal since then, as well as changes and improvements to the graphic libraries and APIs. Therefore the findings in this thesis may be outdated today, however the methods used in this thesis is still relevant today.

Quake III Benchmark

This benchmark compares the performance between Java 3D, jMonkeyEngine2 [1], and Xith3D, using a port of the Quake III [2] viewer. The benchmarking was conducted in collaboration between members of the JGO-forum [3] as well as some of the developers of the APIs in 2005/2006 [55].

The Quake III viewer was first ported to Xith3D, and then later ported to both Java 3D and jME2. This viewer contains only the rendering part of the Quake III engine, and therefore all other parts regarding for example game logic or networking was not present in the benchmarks. A level for the Quake engine was loaded into the viewer, and used for the benchmarking. The level consisted of a total of 27 311 triangles, which were divided into 3704 different objects, with a total of 118 unique materials. The camera in the viewer is placed at the top of the map looking downwards, so that the whole map is visible at the same time. This ensures that all the geometry is drawn, stressing the engines the most.

This benchmark tests various aspects of the APIs. A Quake III level consists of much geometry separated into different objects, that use various materials and textures. Because of this, the APIs have to carefully determine how to batch the geometry together when creating Display Lists or VBOs. Also because of the various materials, it is necessary with renderstate sorting, so that as few OpenGL calls as possible are made. It is also necessary to carefully handle the loading of textures, as well as the level data. The final results were posted in 2006 [56], and the results can be seen in Table 2.1.

Quake III Viewer - Flight Benchmark			
	Java 3D	**jME2**	**Xith3D**
Average FPS (over 5 runs) - higher is better	191.154	285.882	161.62
Average Start Time (ms, 5 starts) - lower is better	5090.2	3906.4	4184.4
Average Used Memory (MB, heap) - lower is better	35.6	18.5	17.1
Average Used Memory (MB, non-heap) - lower is better	11	8.2	11.6

Table 2.1: Benchmarking of the Quake III Viewer ported to Java 3D, jME2 and Xith3D.

The results show that jME2 has the fastest rendering speed, with the highest FPS of the three APIs. The other two have a fairly similar FPS, with Java 3D being the second best and Xith3D being the slowest. When it comes to the loading time, jME2 and Xith3D have the fastest loading

[1] The reader should be aware that jMonkeyEngine2 is does not have the same code base as jMonkeyEngine3, which is part of the evaluation in this thesis. The difference between jME2 and jME3 is explained in greater detail in Chapter 3

[2] Quake III is a first-person multiplayer shooting game developed by id Software, released in 1999.

[3] Java-Gaming.org is found here: `http://www.java-gaming.org/`

times, with jME2 having the edge. Java 3D were a second slower than the other two. In regards to the memory usage when using the heap, Java 3D has by far the highest usage. This is due to a large overhead in Java 3D, which gives it almost double the memory usage of the other two. Xith3D has the lowest usage, followed closely by jMe2. When disabling the heap (native memory usage), Java 3D is on the same level as the other two, and the results show that Xith3D has the highest memory usage. jME2 uses the least amount of memory with the heap disabled.

This benchmark tests the performance of the APIs when several sub-systems operates on a complex scene together. This makes it a good representation of a real world scenario, and the results are descriptive of real world applications. The benchmarks showed that jME had the best performance overall - especially in terms of FPS, followed by Xith3D and Java 3D with the worst performance overall.

High-poly Benchmark

This benchmark compares the performance between several 3D graphics APIs for Java, and was conducted in 2009. This benchmark was done in collaboration between the main developers of the APIs [19]. The APIs in question were: Java 3D, Xith3D, jME2, jPCT, and 3DzzD.

The benchmark is simple, and just compares the FPS between the APIs when they are rendering two different 3D models separately. The models used in the benchmark are a car composed of approximately 1 million faces, and a tank composed of approximately 80 thousand faces. The results from the benchmark is shown in Table 2.2.

High-poly Benchmark					
	Java 3D	**Xith3D**	**jME2**	**jPCT**	**3DzzD**
Average FPS, Car - higher is better	120	510	525-530	580	N/A
Average FPS, Tank - higher is better	N/A	1950	2000	3500	2100

Table 2.2: High-poly benchmarking of a Tank and a Car model with jPCT, 3DzzD, jME2, Xith3D and Java 3D.

The benchmarks show that jPCT performed best in both tests, with 3DzzD, jME2, and Xith3D following with relatively similar results. Java 3D performed absolutely worst, with less than half the performance of the other APIs. It should be noted that Java 3D was not able to run the tank model, due to some problems with the loader. 3DzzD was not able to run the car model, because of some problems with the model converter between wavefront .obj and .3ds (which is the only format supported).

While the tests in this benchmark are atomic and isolated, they are not a good representation of the real performance between the various APIs. This is because essentially all the work done in these benchmarks are performed on the GPU. The benchmarks boils down to compiling the geometry to either Display Lists or VBOs, which are sent to the GPU (in addition to some possible overhead). Therefore this is more of a test of sending data to the GPU, rather than the performance of the APIs. Because of this, the results for all the API are relatively close to each other, except Java 3D which performs horribly bad in this benchmark.

While this is a valid benchmarking of the various APIs, the direct applicability to a real world scenario is arguable, due to the points mentioned before. However the tests show that the performance in terms of fps between the APIs is relatively similar, with jPCT having the best performance.

2.2 Real-time 3D Graphic Libraries

Real-time 3D computer graphics are focused on producing and rendering images (frames) in real time. Computer graphic libraries abstracts the communication between the graphics application, and the graphics hardware drivers. This enables the programmer to utilize the functions of a dedicated graphics processing unit (GPU) for producing 3D graphics. These functions include transformations, lighting, materials, depth buffering and much more. Graphic libraries differentiate in how they define the rendering pipeline, which is the process of converting vertices, textures, buffers and state to the framebuffer through a series of operations. This results in the image on the screen. They also differ in which of the features offered by the GPU, that they support.

There are mainly two libraries used in the industry for producing real-time 3D graphics today, one is open and cross-platform, while the other is proprietary and only available for the companies own solutions. These two graphic libraries are OpenGL and Direct3D.

OpenGL

OpenGL is a cross-platform graphics library maintained by the Khronos Group. The library first started out as a proprietary graphics library called Iris GL, and was developed by Silicon Graphics. They later reworked the library, and released it as an open standard, called OpenGL. OpenGL is maintained by the Kronos Group, which is a non-profit organization focused on the creation of open standards. OpenGL is a cross-platform library, supporting most platforms including the most commonly used operating systems Windows, Mac OS X and Linux. It also supports various mobile platforms (through the OpenGL ES version) like the Android and iPhone, as well as video game

consoles like Sony Playstation 3, Nintendo Wii, and Nintendo DS. The first release of OpenGL was in January 1992, which was the 1.0 release. The current release of OpenGL is the 4.2 version, which was released in August 2011. In addition to the new features that are introduced with each new version, the library can be extended through various *extensions*. These extensions provide additional functionality as new technologies are created, and are provided by various vendors. Extensions may be added as standard extensions to OpenGL if enough vendors agrees to implement it, and eventually it might be added to the core of the OpenGL library. An example of this is shader support (see Section 2.5 for more information), which was first available as an extension to OpenGL, but was later added to the core.

Direct3D

Direct3D is a proprietary graphics library developed by Microsoft. The library is a part of Microsofts DirectX API, which is a collection of libraries for handling tasks related to multimedia. Microsoft was previously part of the OpenGL Architecture Review Board (ARB), who maintains OpenGL at a lower level. They left this board in favor of their own proprietary solution. Direct3D is only available for Microsoft platforms, such as the Windows operating systems, and their video game consoles Xbox and Xbox 360. The first version of Direct3D was released in 1995, with DirectX 2.0. The most recent version of Direct3D is 11.1. It is not possible to extend the capabilities of Direct3D in any way like you can with extensions in OpenGL, so new features are only added through major official releases from Microsoft.

2.3 Immediate- and Retained-Mode

There are two different approaches when it comes to rendering computer graphics, and they are immediate- and retained-mode rendering.

Immediate mode gives the programmer absolute control over the rendering. This mode gives direct access to the rendering system, and the programming is done in a procedural fashion. The programmer explicitly defines what will be drawn during each render-cycle. This means that the application dictates directly what will be drawn for every frame. The programmer describes the scene each time a new frame is required, which means issuing all the low level commands for various functions, such as drawing primitives, performing various techniques for visual effects, and more. This means that it is up to the programmer to handle the data structure for scene management as well. See Figure 2.1 for an illustration.

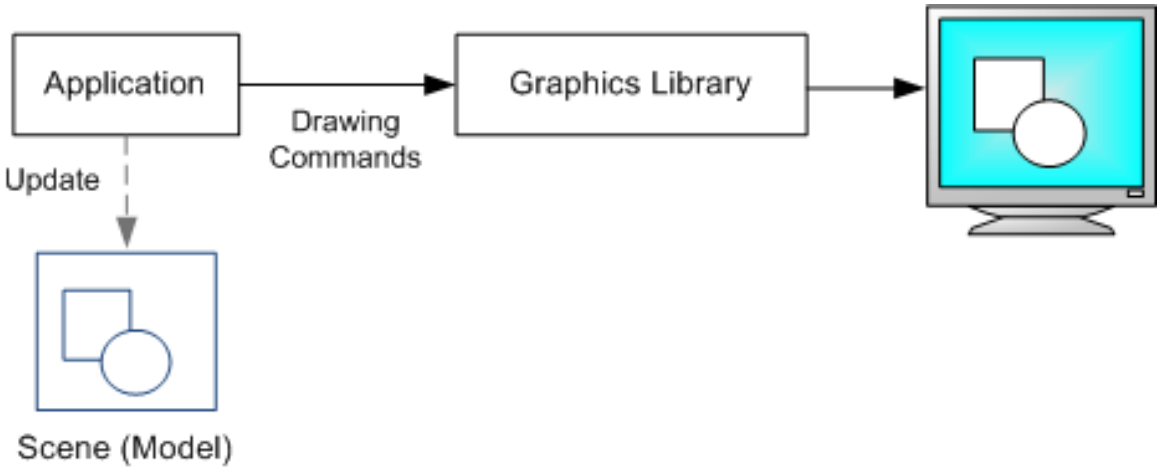


Figure 2.1: Illustration of Immediate mode rendering. Picture taken from [59]

Retained mode is more abstracted above the low-level handling of render-cycles. With retained mode the focus is on defining the various parts that compose the scene, and what effects to add to it. This is done in a declarative fashion, where the programmer do not need to have any particular knowledge of what goes on behind, but can instead focus on the composition of the scene. The scene management is handled by the retained mode API, and it is generally much simpler to use than an immediate mode API. See Figure 2.2 for more an illustration. 3D Scene Graph APIs are usually designed as retained mode APIs.

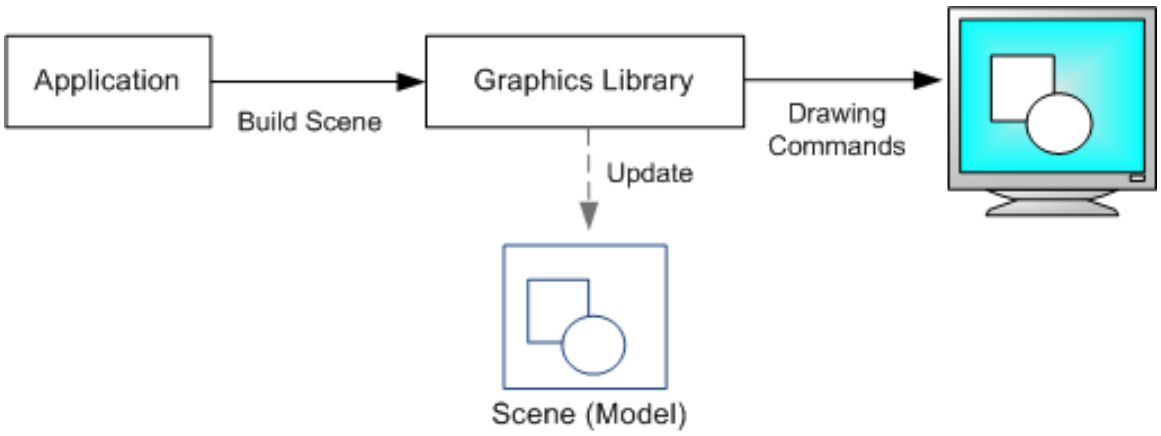


Figure 2.2: Illustration of Retained mode rendering. Picture taken from [59]

There are both advantages and disadvantages with each mode. The retained-mode offers handling of the scene management, as well as abstraction above the low level code. However this comes

at a price, both in regards to flexibility and efficiency. With immediate mode you have to implement the scene management yourselves, giving you the freedom to implement it however might suit the project best, and optimize it accordingly. This is often not possible when using retained mode, because you are restricted to its design. Retained mode APIs often has a larger overhead in order to maintain the flexibility, which increases the memory usage of the application.

With immediate mode you have to continuously define what to be drawn during each rendering-cycle. This gives a great deal of control over what is sent to the GPU. This also means that the data will be sent over and over again to the GPU for each frame. For large complex scenes this can affect performance. Retained mode APIs usually pre-load the data onto the graphics card, and then when a render-cycle takes place on the instructions, and not the data is necessary to send to the GPU. This heavily reduces the transfer overhead costs compared to the immediate mode.

2.4 Scene Graphs and 3D Scene Graph APIs

A scene graph is a data structure for describing a graphical scene. They are commonly used by vector-based graphics applications and in 3D applications, such as games and simulations.

The data structure arranges the logical and often (but not necessarily) the spatial representation of a graphical scene. Scene graphs are an abstraction over the low level graphics, and considers objects as boxes and spheres, instead of lines and vertices. Henry Sowizral formulated this in a very good way; A good scene graph design should allow programmers to focus more on scene contents like objects and their arrangement within the scene, thinking of the best way to present them, and forget about the complexities of controlling the rendering piplines [98]. The exact definition of a scene graph is partly fuzzy because implementations often just take the basic principles of a scene graph and adapt them to suit the needs of the particular application, in particular this is done within the games industry. Because of this, there is not always a clear consensus to what a scene graph should be.

When programming with low level graphic libraries such as OpenGL, the programmer needs to break down the objects into a series of instruction calls that are sent to the GPU for rendering. The programmer also has to keep in mind the order in which the instructions are sent to the system, as well as culling objects that are not visible. Therefore writing interactive 3D graphics applications has traditionally been a tedious and time-consuming task, which requires a high-level of expertise by the programmer. Because of this, people usually created their own abstraction above the low-level graphics commands.

OpenInventor introduced the classical concept of a scene graph API, and was an object-oriented toolkit for developing interactive 3D graphic applications [99]. OpenInventor bypasses shortcomings of previous graphic packages. This includes the *duplicate database* problem. In short, this means that applications had to store objects in a form suited to their needs, but had then to be converted to a format required by the graphics package. The foundation of this 3D toolkit is the scene database, which stores dynamic representations of 3D scenes as graphs (typically acyclic graphs), with objects called *nodes*. Various classes of nodes implement different parts. Each node in a scene database performs some specific function. There are *shape nodes* that represent geometric or physical objects, *property nodes* that describe various attributes of objects, there are *group nodes* which connect other nodes into graphs and subgraphs. In addition there are also other nodes, such as nodes for *camera* and *lighting*. A scene is represented by a hierarchical grouping of these nodes. An example of such a hierarchical grouping can be seen in Figure 2.3, and the rendered result in Figure 2.4. In this figure the *Bug* car is rendered in two different versions, one coloured with the rainbows colours, and the other rendered in wireframe. The same geometry is used for the rendering of both versions of the car, with the use of a *separator node*, which enables the sharing of properties among various shapes.

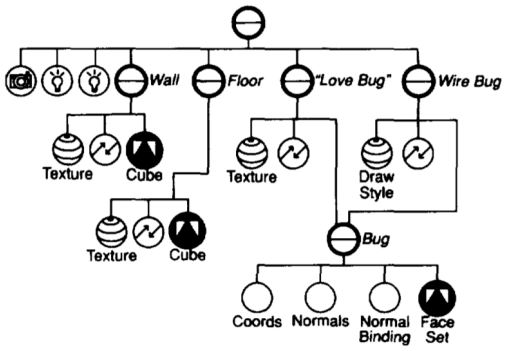


Figure 2.3: Simple scene graph describing a scene with a ”Bug” rendered with colours and another with wireframe [99].

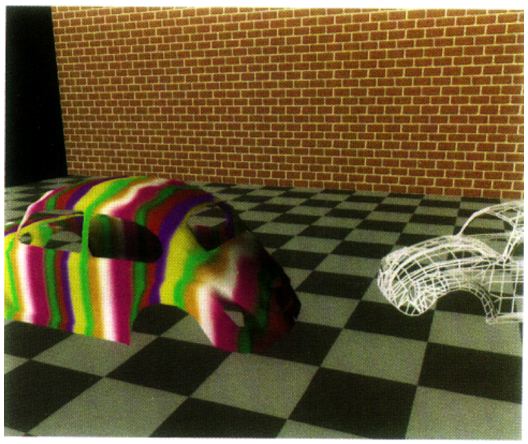


Figure 2.4: The rendered scene from the scene graph described in Figure 2.3 [99].

The OpenInventor toolkit also features *path objects*, which point to nodes in a chain from some node in the graph down to the node in question. This can for example be used for a pick operation. There are also *action objects*, which traverse the scene graph in order to perform specific operations, such as rendering, computing a bounding box or searching. It also supports simple animation,

through *sensor objects*, which calls user-defined callback functions when triggered. It uses a 3D event model to distribute user events to manipulators and other smart nodes in the system. This means that nodes can process events that are distributed to the nodes in the scene.

Döllner and Hinrichs described a *Generalized Scene Graph* [83], which emphasizes a generic and abstract specification of scene contents. They try to overcome various limitations of other scene graph architectures, such as support for different 3D rendering systems, multi-pass rendering, and declarative modeling of scenes. It is based on an object model with three main object categories; Rendering objects, Scene graph nodes, and Rendering engines. *Rendering objects* are 3D and 2D shapes, appearance and transformation attributes. *Scene graph nodes* hierarchically organize rendering objects and may generate or constrain rendering objects. *Rendering engines* traverses and interprets the contents of the scene graph.

This generalized scene graph support declarative scene modeling, by analysing the scene contents in a pre-evaluation phase. This improves the compactness and usability of the scene specification, and can be optimized so that the overhead is reduced to a reasonable amount. OpenInventor evaluate scene nodes in a depth-first order, which means that global information about light sources and cameras, is not available. Therefore you are required to arrange the contents in such a way that the depth-first order is preserved in OpenInventor. This is not a problem with the generalized scene graph because of the pre-evaluation, because rendering objects such as light sources and cameras are encountered during this pre-evaluation, and can thus be installed and enabled.

We mentioned at the start of this section that the definition of a scene graph often is a bit fuzzy in regards to the actual data structure. This is because of the way that programmers implement it. It often varies depending on the case of the application. Jason Gregory [86] mentions that the data structure used for representing the scene graph does not need to be a graph, and in fact the choice is usually some form of a tree structure. The basic idea is to partition three-dimensional space so that it is easy to discard those regions that are not within the frustrum, without having to cull all of the individual objects within them. Examples of suitable data structures are quadtrees and octrees, BSP trees, *k*d-trees, and spatial hashing techniques.

To summarize, a 3D Scene Graph API is a graphics API that uses, and is built around the scene graph data structure. OpenInventor who was previously mentioned, is an example of such an API. Many of these APIs also include other functions in addition to the data structure. What functions depends on what is the primary focus of the API. This can include various special effects, such as water, shadows or various processing effects. It can also includes other aspects, such as terrain

generation, water or particle systems. Some APIs also include other systems, such as networking and sound. The bottom line however, is that what is common to these APIs is the data structure used for representing the scenes.

2.5 Shaders

Traditional (old) graphics programs written for the CPU let the programmer take control of all steps in the rendering process. However with the rise of dedicated GPU chips the pipeline could not be programmed in the same way, and programmers where restricted to the Fixed-Function Pipeline (FFP). This meant that the functions and algorithms was fixed in the hardware and could only be configured, but not programmed by the programmer himself. This meant that the programs could not do anything that was not anticipated by the designers of the hardware. This is also known as *hardware transformation and lighting* (*hardware T&L*).

Shaders were designed to remove this limitation, and enables programming of certain stages of the rendering pipeline. This gives programmers the freedom, and ability to create effects that look much better. This was not possible with the old pipeline.

There are two forms of shader programming, *Offline Rendering* and *Real-time Rendering*. The main difference between the two, is that while real-time rendering uses the pipeline defined in the GPU, offline rendering is usually done on the CPU. there they have the ability to completely rewrite the rendering pipeline.

Offline rendering uses shading languages that are precomputed, and is most often used for films. One of the most known shading languages for offline rendering is the RenderMan Shading Language [92], and have been used in films such as *Toy Story*, *Jurassic Park* and *Star Wars: Episode I - III* [58]. Other offline rendering languages are Houdini VEX Shading Language and Gelato Shading Language.

When it comes to real-time rendering, it was first possible to write shader programs for various stages in the rendering pipeline in 2001. This was with the introduction of DirectX 8 and with some new extensions in OpenGL (ARB_vertex_program). The support was limited to low-level assembly code, which made it difficult for most developers [86]. Initially it only supported vertex and pixel shaders. There have been developed several Shading Languages that allows high level programming of shaders. The three main languages are NVIDIA Cg, HLSL (High-Level Shading Language) and GLSL (OpenGL Shading Language). These languages are based on the programming language C, and supports all the flow-control constructs of a modern language, such as loops, conditionals, and

function calls [68][93][43]. Shaders written in NVIDIA Cg works with both OpenGL and Direct3D, whilst GLSL only works with OpenGL, and HLSL only works with Direct3D. Figure 2.5 shows the different geometry processing and rasterization stages in the rendering pipeline. The stages that are usually referred to in regards to shader programming, are *Vertex shader*, *Geometry shader* and *Pixel shader* (also known as Fragment shader). These are marked in white.

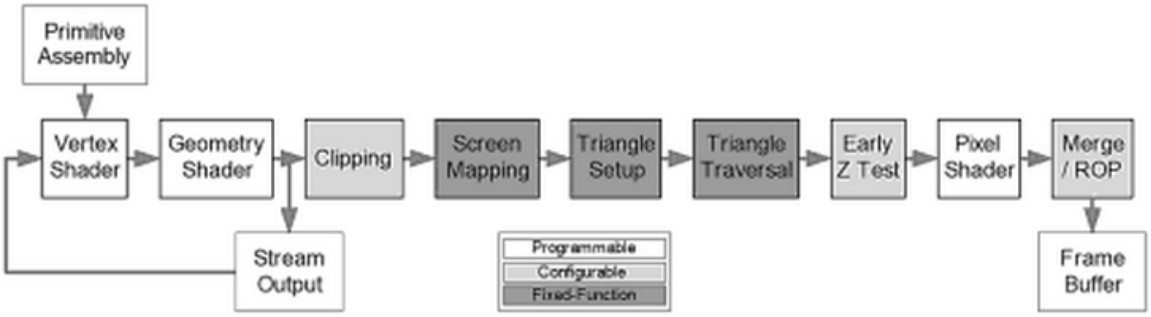


Figure 2.5: The different stages in the rendering pipeline, as implemented in a typical GPU. The white stages are programmable, the grey stages are configurable and the dark grey stages are fixed function. Picture is taken from Jason Gregory [86].

The Vertex Shader [86][61] handles transformation and shading/lighting of individual vertices. The input is a single vertex, although in practice many vertices are processed in parallel. Positions and normals are usually represented in model space or world space. Perspective projection is also applied, as well as per-vertex lightning and texturing calculations, and skinning for animated characters. It can also perform procedural animation by modifying the position of the vertex. Examples are foliage that sways in the breeze or an undulating water surface. Output is a fully transformed and lit vertex, where position and normal are expressed in homogeneous clip space. On modern GPUs, vertex shader has full access to texture data, a capability that used to be restricted to the pixel shader. This is useful when textures are used as stand-alone data structures, like height maps or look-up tables. Vertex shaders can manipulate properties such as position, color and texture coordinate. They cannot however, create new vertices.

The Geometry Shader [86][61], operates on a full primitive, consisting of one or more vertices. It is capable of culling or modifying primitives, as well as generate new primitives. It was first introduced in DirectX 10 and OpenGL 3.2 (however possible with OpenGL 2.1 using an extension). It is the newest addition to the shading language.

The Pixel Shader [86][61], also known as the fragment shader, operates per fragment. The fragments are generated from the rasterization that takes place on the vertices of a primitive. The

fragment has an associated pixel location, a depth value and a set of interpolated parameters such as color, a secondary (specular) color, and one or more texture coordinate sets. It is used to shade (light and otherwise process) each fragment. Can address one or more texture maps, run per-pixel lighting and do whatever else is necessary to determine the fragments color.

These shading languages differ from conventional programming languages, in that they are based on a *data-flow computational model*. This means that that computations in the model occurs in response to the data that flows through a sequence of processing steps [68].

Because of shaders it is possible to produce most of the effects seen in real-time graphics today. Figure 2.6 and Figure 2.7 shows a comparison between a game that uses no shaders, and one that uses shaders. With the increase in computational power the graphics we can create with real-time shading languages are starting to look more and more like the standard in offline rendering shading languages.

Figure 2.6: Screenshot from Half Life 2 (2004), using DirectX7, with no shader support.

Figure 2.7: Screenshot from Crysis 2 (2011), using DirectX11, with shader support.

2.6 Stereoscopic Rendering

There are several different techniques for producing stereoscopic images (3D-pictures). These techniques differ in visual quality, required equipment and exhaustion. Some of these techniques are Quad-Buffering, Anaglyphic, Checkerboard and side-by-side or cross-eyed. Another difference are whether they are active or passive, where the former requires electronics that interact with the display. The two most common techniques are Quad-Buffering and Anaglyphic, and they will be explained in further detail here.

Anaglyphic is a passive technology. This technique uses two color layers that are superimposed,

with an offset to each other, to the original picture. This produces a depth effect. When viewing this picture the user needs glasses, with different color on each eye. This is because one eye sees one of the color layers and the other in turn sees the other layer. The colors usually used are red and cyan or mixed blue and green. Drawbacks with this technique is that there is some color loss, due to the ”encoding” of the colors to produce the stereoscopic effect. The result is also not so good. This technique works on any graphics card. Anachrome is another technique that improves upon anaglyph, with less loss in color.

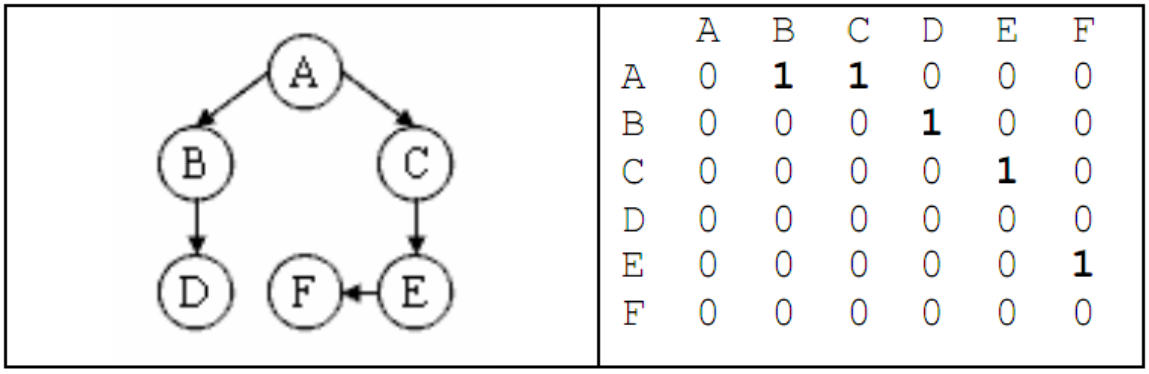
Quad-Buffering is an active technology. The technique uses four buffers for producing the images that are displayed. One back and front buffer for each eye. This means that different images are rendered for each eye, with correct perspective adjustments. This gives quad-buffering the best stereoscopic effect. This only works on special graphics card, like nVidia Quadro and AMD HD3D. Special liquid crystal glasses are needed. These communicate with the display, and blacks out the left/right eye in accordance with the refresh rate. A refresh rate of 120hz is required for displaying 60hz for each eye. The display uses alternate-frame sequencing in order to achieve the desired effect.

2.7 Design Structure Matrix

A Design Structure Matrix (DSM) (also called a Dependency Matrix) is constructed from the dependencies between elements in a system. It is a square matrix that shows all the dependencies between every element.

Figure 2.8 illustrates a DSM, and the system it was built from. On the left is a graphical representation of the elements in the system. A dependency is shown with an arrow between two elements, direction of the arrow illustrates that element X depends upon element Y. On the right is a DSM constructed from the same system. The rows and columns in the DSM represents all the elements in the system. The rows and columns can be switched with each other without any change to the DSM. Dependencies between the elements are shown with either a marking or a numbering in the DSM. The number corresponds to the number of direct and indirect dependencies there are between the two elements. In the figure, only dependencies with a depth of 1 are shown (dependencies an element has with itself, is not shown). This means that only *direct dependencies* between the elements are shown. For example element *A* depends upon element *B*, which is visible in the DSM, marked with a 1.

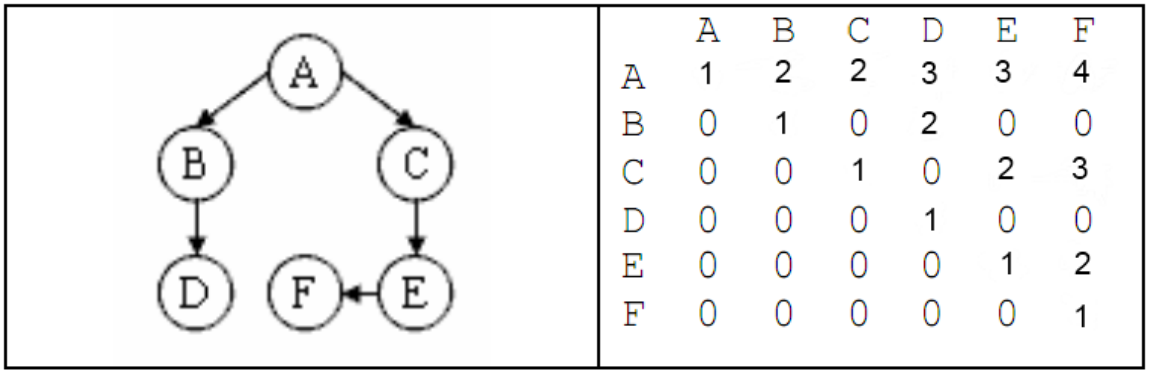
Figure 2.9 shows the same sy1stem as Figure 2.8, however here *indirect dependencies* are also



	A	B	C	D	E	F
A	0	**1**	**1**	0	0	0
B	0	0	0	**1**	0	0
C	0	0	0	0	**1**	0
D	0	0	0	0	0	0
E	0	0	0	0	0	**1**
F	0	0	0	0	0	0

Figure 2.8: To the left is a graphical representation of a series of elements, with dependencies between them shown with arrows. To the right is a Design Structure Matrix, showing the direct dependencies between the same elements (marked with a 1). Image taken from MacCormack et al. [88].

shown. In addition, the dependency that an element has with itself is also visualized, here with a depth of 1. Direct dependencies is shown with a depth of 2. Indirect dependencies is shown with a higher number, which reflects the number of elements that indirectly affects a dependency between two elements. For example element *A* indirectly depends upon element *F*, because it depends upon *C*. *C* in turn depends upon *E*, who in then depends upon *F*. This gives a depth of 4, which is also shown in the DSM.



	A	B	C	D	E	F
A	1	2	2	3	3	4
B	0	1	0	2	0	0
C	0	0	1	0	2	3
D	0	0	0	1	0	0
E	0	0	0	0	1	2
F	0	0	0	0	0	1

Figure 2.9: To the left is a graphical representation of a series of elements, with dependencies between them shown with arrows. To the right is a Design Structure Matrix, showing the dependencies every element has with itself (1's), direct dependencies (2's), and indirect dependencies (3's and higher). Image is a modified version, taken from MacCormack et al. [88].

When constructing a DSM it is important to choose which *level of analysis* to base it on. Level of analysis means how far up or down to go in the architecture when building the DSM. At a higher

level one would focus more on groups of source files, that are related to specific parts of the design. This would give a DSM that looks more at the dependencies between these groups. Alternatively one could go much lower, and focus on the function or class level.

One of the strengths of a DSM is that it is able to represent the relationships between a large number of elements in a system. A DSM is often much easier to look at, and helps with detecting various patters in the data. This can for example be to identify feedback loops, or to determine how to organize code in modules. Figure 2.10 shows a DSM with an *idealized modular form*. There is a diagonal line that goes from the top left, to the bottom right (marked in red). This line shows dependencies that the elements have with themselves. The square boxes along this diagonal shows suggestions for possible modules in the system. This is based off the logical grouping of the dependencies between the elements. Ideally dependencies should be kept within a module, which is the case with this example. This is however not necessarily the case with real world systems. Figure 2.11 shows a modified version of the same DSM, however here we can see that the structure is no longer ideal. There are dependencies that goes outside of the squared boxes, which means that the dependencies are not kept within the modules. Long horizontal and vertical lines with dependencies indicate that the element either depend upon many, or many are depending upon it. These are generally indications of a bad design, and it increases the chance that any changes to the code affects other parts of the system. The red circle in the Figure highlights such an example.

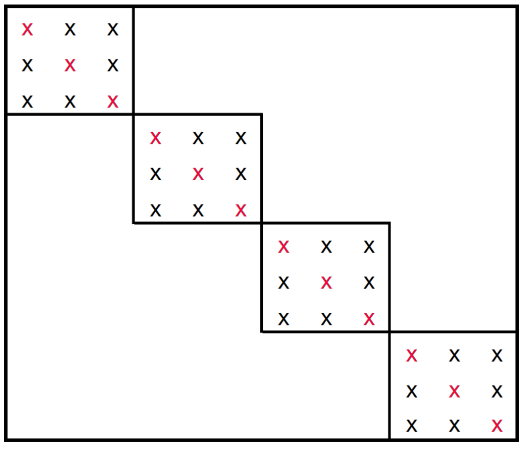


Figure 2.10: The picture shows a Design Structure Matrix with an idealized modular form. Image is a modified version, taken from MacCormack et al. [88].

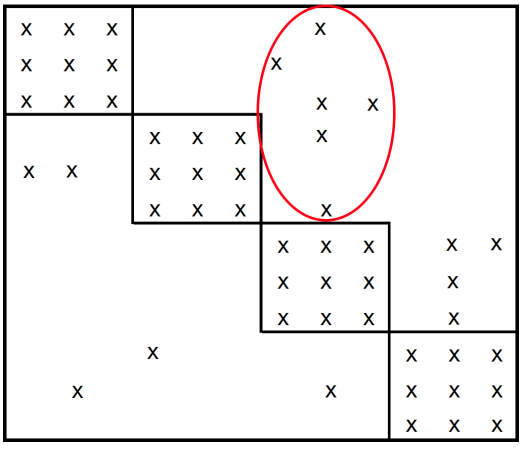


Figure 2.11: The picture shows a Design Structure Matrix that does not have an idealized modular form. Image is a modified version, taken from MacCormack et al. [88].

2.8 Software Licensing

A software license governs the usage or redistribution of software, granting rights and imposing restrictions on the use of the software. They typically allocate liability and responsibility between the parties that are entering into the license agreement. This includes limitation of liability, warranties and warranty disclaimers, and indemnity if the software infringes intellectual property rights of others.

Software licenses can be categorised in two different groups, namely *proprietary licenses*, and *free and open source licenses*. The following sections will give a brief explanation of each of them.

Proprietary software licenses grants the users access to the software if they accept a end-user license agreement (EULA). The ownership of the copies that the user gets remains with the software publisher. The EULA usually includes terms such as usage of the software, number of installations allowed, or the terms of distribution. A significant effect of this licensing is that if the ownership of the software remains with the software publisher, the end-user may be required to accept the license in order to use it. The user is not allowed to use the software at all unless accepting the EULA.

Free and open source software licenses leaves the users free to accept the license, or not, when used to study or privately modifying the source. In order to redistribute the software the user must accept the software license. These licenses usually fall into one of two categories: *copyleft licenses* or *permissive licenses*. To qualify as a free software license, the license must grant the rights described by The Free Software Definition [1] or one of the similar definitions based on this.

The copyleft license aims to give unlimited freedom to use, study, and privately modify the software. As long as the changes adhere to the terms of conditions, the user is also free to redistribute the software or any modifications to it. With copyleft licenses it is a rule that redistributions of the software cannot add restrictions to deny other peoples freedom. In other words; it cannot add any more restrictions to the license, than those that are already defined in the license. This fact prevents the software from ever being made proprietary. The *noncopylefted license* is another free license, which is similar to copyleft. However this license also makes it possible to add additional restrictions to it, which means that it is possible to make noncopylefted software proprietary.

The Permissive software licenses is similar to the copyleft in terms of modification to the software, however it only includes minimal requirements to the redistribution of the software. This gives the user the permission to use the code as part of a closed source software, or open source

[1] This definition was published by the Free Software Foundation (FSF), and defines *free software*. *Free* is here used in the sense of ”free as in freedom of speech, not as in free beer”. More information about this is available from the definition given at the GNU Projects website: `http://www.gnu.org/philosophy/free-sw.html`

software released under a proprietary software license. It should be noted that this type of license is as free as the copyleft license type, the only difference lies in how the software can be redistributed. The term *copyfree* have also been used to describe permissive licenses.

A *viral license* is a type of license that only lets you redistribute the software if, and only if they are licensed identically to the original. An example of a viral license is the GNU General Public License, which forces any derivative work to be licensed with the same license. This means that if you should choose to use some code that is licensed with GPL, then any code that uses the GPL licensed code would have to be released using the GPL license. Because of this, many find viral licenses undesirable, since it forces you to release all your code. Viral licenses have also been dubbed *General Public Virus* or *GNU Public Virus (GPV)*. This is due to the nature of the license, which is very similar to that of a virus [1].

A *dual license* offers two licenses with the software, one that for example is a viral license, that is offered for free. Then in addition it is common to offer another license, which is less restrictive, a permissive license, which you have to buy, but in turn lets you redistribute the software with your own licensing.

[1] A virus has the ability to replicate or convert objects that are exposed to them into copies of themselves.

Chapter 3

Investigation of 3D Scene Graph APIs

This chapter investigates the various 3D scene graph APIs that are available in Java. The APIs investigated are Java 3D, jMonkeyEngine3, Ardor3D, jReality, Aviatrix3D, Xith3D and jPCT. The first three APIs are investigated in more detailed than the others, the reason for this is given in Section 3.1.

3.1 Note about the investigation

This chapter features an investigation of the different 3D scene graph APIs available in Java. The evaluation of 3D scene graph APIs in this thesis will only focus on three APIs, Java 3D, Ardor3D and jMonkeyEngine3. The reasons for this is explained in greater detail in Section 1.4. Because of this, only these three APIs will be investigated in great detail. The chapter will also give an introduction to the other 3D scene graph APIs that are available in Java, because this will give a broader picture, and a better understanding of what alternatives are available in Java.

3.2 Java 3D

3.2.1 History

The Java 3D [1] project was started in 1997, as a collaboration between Intel, Silicon Graphics, Apple and Sun. The specification for the first 1.0 version were completed, but because a few critical features were missing, it was never released. The first official release of the API was version 1.1,

[1] Website: `http://java3d.java.net/`

which was released in 1998. During the time period from mid-2003 through summer 2004 the project was discontinued. After this, the project was released as a community source project in the summer of 2004. Sun and other volunteers continued the development after this. The latest stable version of Java 3D was released in 2008, which was 1.5.2. The entire source code of the API was released under the GNU GPL version 2, with the CLASSPATH exception in 2008. There have only been applied bug fixes to the API after this, and it is a common agreement amongst the community that the development of the project is dead.

3.2.2 Releases

The first official release of the project was in 1998, with the 1.1 version. There was a steady development in the project after this, up until the last version was released in 2008. This was the 1.5.2 version, and it is the latest version that was released, who offered any new functionality. There is a 1.6.0 version that was last updated in 2010, but it only contains bug fixes, and has not been officially released. The active development on the API has stopped, and the project is considered dead.

- 1.1.0 released in December 1998.
- 1.1.1 released in March 1999.
- 1.1.2 released in June 1999.
- 1.1.3 released in December 1999.
- 1.2.0 released in May 2000.
- 1.2.1 released in March 2001.
- 1.3.0 released in July 2002.
- 1.3.1 released in May 2003.
- 1.3.2 released in March 2005.
- 1.4.0 released in February 2006.
- 1.5.0 released in December 2006.
- 1.5.1 released in June 2007.
- 1.5.2 released in June 2008.

3.2.3 Community

The active development on Java 3D have stopped, but it still being used in the industry, especially within various CAD-fields. Because the active development have stopped, it have had an impact on the community surrounding the project, and therefore it is not as large as it once was. There is still activity on the forums however, and the community assists users that have questions or problems.

3.2.4 Showcase

Java 3D has been used for both visual simulation in scientific and industry applications, as well as for games. The following listing presents some of the projects done in Java 3D.

- Sweet Home 3D [66] is a free interior design application developed in Java 3D. It lets the user design the interior of houses in a 2D plan, and view it in a 3D preview.
- SINTEF [63] developed Virtual Globe [74], which is a client-server application for displaying very large (global scale) terrain models.
- World Wind [52] is a virtual globe application developed by NASA. World Wind uses satellite data to create globes of different planets, including Earth, the moon, Mars and Venus. Users can interact with the planet, by rotating and tiling the view, and zooming in and out.
- HVRC CREATE [21] is developed by IFE [23] in collaboration with EDF [69], and is a suite of tools for designing and testing 3D room and environment layouts.
- FlyingGuns [14] is both a flight- and battlefield simulator and game. The game lets the player operate WW1 fighter planes, and battle against the computer or other players online.

3.2.5 Licensing

Java 3D is licensed under the open source GNU GPL version 2, with the CLASSPATH exception. This is a copyleft license, which means that all redistributions must use the same licensing terms. The linking exception means that additional libraries that does not use a GPL-compatible license can be used in conjunction with it.

3.2.6 Packaging

Java 3D is divided into three different packages; *j3d-core*, *j3d-core-utils*, and *vechmath*. The j3d-core package contains all the core classes for Java 3D. The j3d-core-utils package includes many convenient and powerful classes that are additions to the core. These utility classes can be divided

into four main categories: content loaders, scene graph construction aids, geometry classes, and convenience utilities. The vechmath package contains many mathematical classes, such as vectors, matrices, and quaternions. In addition to this every Java3D program uses classes from the *java.awt* package. This package defines the Abstract Windowing Toolkit (AWT) in Java, which is used to create a window in which to display the rendering.

3.2.7 Scene Graph Structure

The scene graph in Java 3D is a tree, and is composed of nodes and arcs. A node is a data element, and an arc is a relationship between such data elements. There are two types of relationships between nodes, *parent-child-relationship* and *reference-relationship*. The first type is where one node may have one or more children. The other type of relationship is where a component may be associated with a scene graph node. These components cannot have children. Such components may be either geometry or other rendering attributes.

The organizing of the scene in Java 3D is done by adding one or more *virtual universes*. A virtual universe is a separate entity, with its own world space. It is not possible for objects in the scene graph to exist in multiple universes at the same time, nor interact between the universes. A *locale* can be attached to the virtual universe, which is a high resolution representation of a part of the virtual universe. The location of objects attached to the locale is relative to the locales location in the virtual universe. Figure 3.1 shows the typical layout of the scene graph in a Java 3D application. Beneath the locale, the subgraph is divided into two parts, the *content branch* and the *view branch*. The view branch contains the various parameters associated with the camera, and the view into the virtual world. The content branch contains all the content for the scene, this includes geometry, appearance, functionality components and lights. In other words, this is the content of the scene.

Nodes in Java 3D are either *group-* or *leaf-components*. A group may have children, whereas a leaf may not. Leaf nodes are the actual content that will be added to the scene, like geometry and light. In order to add this to the scene in Java 3D, it must be added to a group node. there are two main groups in Java 3D, *BranchGroup* and *TransformGroup* [1]. The transform group has a transform component, that determines its position in the scene. The branch group does not have any transform component, so its only purpose is to have children. The difference between the two come with Java 3Ds compiled and live scene graphs. When the application is in a running state, the scene graph in Java 3D is optimized, and considered *live*. When a scene graph is live, you can add groups to it,

[1] Note that there are other groups available in Java 3D, like *OrderedGroup*, *ShaderGroup* and *ViewSpecificGroup*. These are however irrelevant to the explanation above.

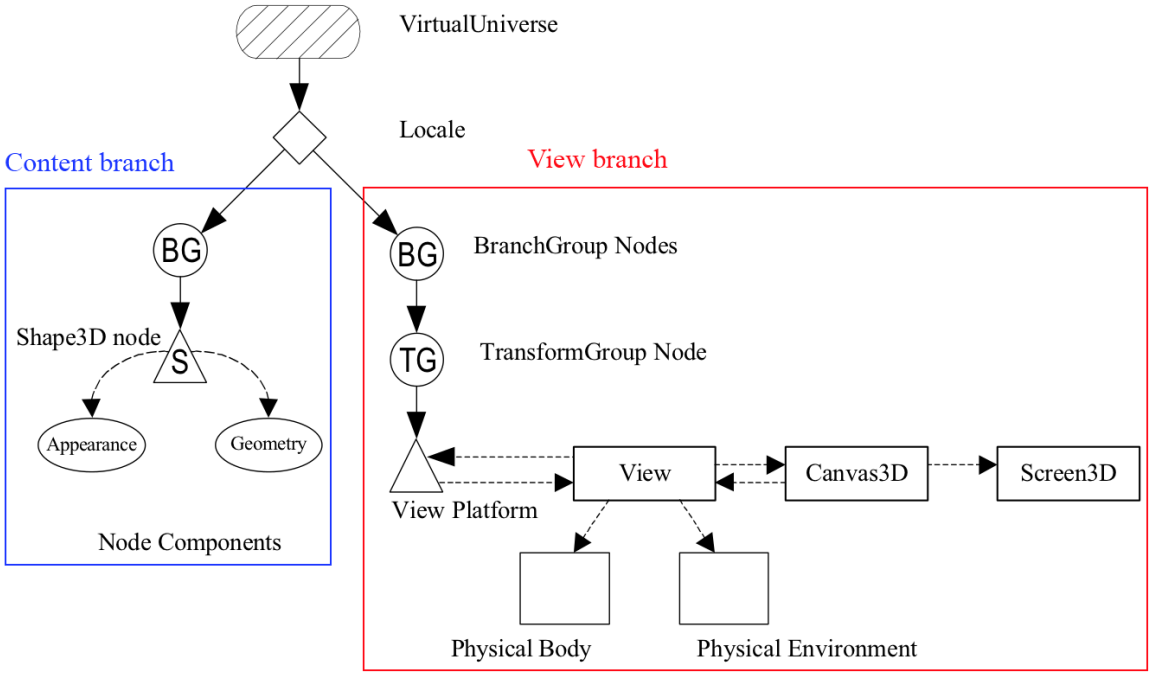


Figure 3.1: Scene graph structure in Java 3D. Picture is taken from a Java 3D tutorial [27].

both transform- and branch-groups, however only branch groups can be removed. The result of this is that for every part of the graph that might be removed after the scene is live, it is necessary to add a branch group above it. In addition to this, every group, and most of the other elements in Java 3D (both groups and leafs) have various capabilities that can be set. These capabilities determines various parameters, such as whether a branch group is allowed to be removed from a live scene graph or not. This adds a lot of verbosity to code written in Java 3D, and clutters much of the code.

Geometry in the form of leaf nodes can have different rendering capabilities set by attaching *appearances* to them. These describe various rendering aspects, such as materials, lighting, texturing, culling and shader programs. In order to position an object in the scene in Java 3D, it has to be added below a transform group. It is not possible to attach a transform component to a leaf node.

Light is a leaf node in Java 3D, and may be added to the scene graph as such. The light can be either a *point light*, *ambient light* or *directional light*. Appropriate parameters can be set for each light type.

3.3 jMonkeyEngine3

3.3.1 History

The jMonkeyEngine [1] project was started in 2003 by Mark Powell (also known as *mojomonk*). The development of the API continued up to its 2.0 version, who never got any official release. The reason for this is because the core developers left the project in 2008 [6]. Some of these went on and started their own project, Ardor3D. After the core developers left the project, there were some turmoil and uncertainty about the future of the project. The community continued the development, and officially released a 2.0.1 version in 2009 [29]. One of the community members, Kirill Vainer (also known as *Momoko_Fan*) started development of a test version for jME3 [30]. This was a complete rewrite of the engine from scratch. Vainers rewrite of jME was accepted by the community, and was accepted as the official code base for the new 3.0 version of jME. This marked a new era and focus for the project, and management and server rights was also passed on from the old developers, to the new [31]. One of the persons who took an active role in the transfer, and the role of new management were Erlend Sogge Heggen (also known as *erlend_sh*).

3.3.2 Releases

After the jME project officially supported the new 3.0 version, there has been a very active development in the project. There has been frequent alpha releases, from the first official alpha release in May 2010 [34], up until the most recent beta release for the project in October 2011 [35]. There hase been no official releases after this, but there are constant additions and bug fixes applied to the latest sources in repository. A list of the releases after the shift in management and developers for jME is given below.

- 2.0.1 released in September 2009.
- 3.0 alpha 1 released in May 2010.
- 3.0 alpha 2 released in August 2010.
- 3.0 alpha 3 released in November 2010.
- 2.1 released in March 2011.
- 3.0 alpha 4 released in March 2011.
- 3.0 beta released in October 2011.

[1] Website: `http://jmonkeyengine.com/`

3.3.3 Community

The community surrounding this project seems very alive, and vibrant. There is very much activity on the forum, and questions are usually answered within hours. Users are also publishing much of the work they have done with the API on the forums, that range from final applications/games, to new features and extensions to the API. The management for the jME project is also very actively monitoring the forums, and when people show off something noteworthy, it is very often published on their website and twitter. This further encourages, and engages the community, and serves as a big motivator for contributors. The project is also hosting events and challenges to engage the community. An example of this is the game contest they arranged during the winter 2011 to 2012 [32] [33].

It should also be noted that the community were able to continue the development of the project, even though the main developers left the project at the same time. This shows that there are many people who believe in it, and shows the strength and competence of its community.

3.3.4 Showcase

jME3 has been used mostly for games, but also to some degree in more scientifically oriented games/applications. The following listing presents some of the projects done in jME3.

- Climate Game 3.0 [11]: This is developed by Tygron [72], which is a company located in the Netherlands, who focus on creating what they call *serious games*. These games are not designed primarily for entertainment, but focus on other aspects relevant to society. The game they are developing in jME3 is about the management of climate changes, and urban development in Delft[1].
- 3079 [1]: This game is a futuristic, open-world, first-person action and role playing game.
- HostileSector [20]: A tactical turn-based game, were you command troops of soldiers in combat against other players online. The game mixes tactical depth with rpg elements and soldier-level customization options.
- Mythruna [48]: A rpg game under development, that features a randomly generated sandbox world and open world gameplay.
- mTheoryGame [47]: This is a single and multiplayer game, that focuses on weapon-based combat in changing environments.

[1] Delft is a city in the Netherlands

In addition to the games mentioned above, it should be mentioned that the community surrounding jME3 is constantly working on different systems, and implementing features in the API. The *Hex Engine* [18] is an engine built on top of jME by a user, which features HDR tone mapping, screen space ambient occlusion, AI and path-finding and much more. Voxel terrain have been created [75]. There is a vegetation system under development, called The Forester [70], and a realistic sky system [57]. In addition to this there is much more, and frequently new projects are announced on the jME forum [37].

3.3.5 Licensing

jMonkeyEngine3 is licensed under the New BDS License [53] [36]. This is a permissive free software license. The software is provided *as-is*, and the authors of the jMonkeyEngine3 API cannot be held responsible for any use of the software. The software may be redistributed freely in any way.

3.3.6 Packaging

jMonkeyEngine3 consists of several packages that offer different parts of functionality [1]. The core of the API is in the *jME3-core* package. This contains the core parts of the API, such as the scene graph data structure, the renderer, bounding volumes, math, controllers and more. In addition to these core functionalities it contains many additional features, not traditionally bundled with the core. This includes special effects such as water, shadows, and functionality such as animation and networking. The packages *jME3-lwjgl* and *jME3-lwjgl-natives* contains code specific to the renderer it supports, which is LWJGL [44]. The package *jME3-jbullet* contains code specific to the integrated physics library, which is jBullet [28]. The package *jME3-blender* contains a blender model import plugin. *jME3-desktop* contain desktop computer libraries for jME3. *jME3-jogg* contains code specific to the J-OGG audio plugin [26]. The package *jME3-niftygui* contains code specific to the GUI library NiftyGUI [54]. *jME3-terrain* contains the terrain system, and *jME3-plugins* contains basic import plugins for models such as OgreXML.

3.3.7 Scene Graph Structure

The organizing of the scene in jME3 is done by adding elements directly below the root in the scene graph. All the content is added directly below the root node, this includes geometry, materials,

[1] Please note that this examination of the packages in jME3 is based on the nightly build, downloaded on the 11.02.2012. This discussion reflects the state of the API at this time, and any changes after this have not been taken into account.

functionality components and lights. Views into the virtual world is added by attaching it to a *view port*. A view port has a location in the scene. *Scene processors* can be added to the view port, which control additional effects that can be added before, and after the normal rendering of the scene.

All elements that are added to the scene graph in jME3 are *spatials*. A spatial is an abstract structure, and holds information about the transformation of the element, as well as its parent. Spatials also have different render states associated with them. These are in the form of *render buckets*, which enforce different rendering behaviours for different rendering methods. This include shadows, transparency, translucency, opaque and more. This can be inherited, or overridden by potential children. Spatials are subdivided into two different groups, *geometry* and *node*. A node can only have children. The geometry element is used to represent geometry, and is a leaf node. Geometry elements can be added as children to node elements. The implication of storing the transformation in the spatial is that every element can have a transform component associated with it. This may reduce the verbosity, opposed to for example Java 3D. This is because it is not necessary to add additional elements to the scene graph just to express the transform of a geometry element. Figure 3.2 shows a typical jME3 scene graph.

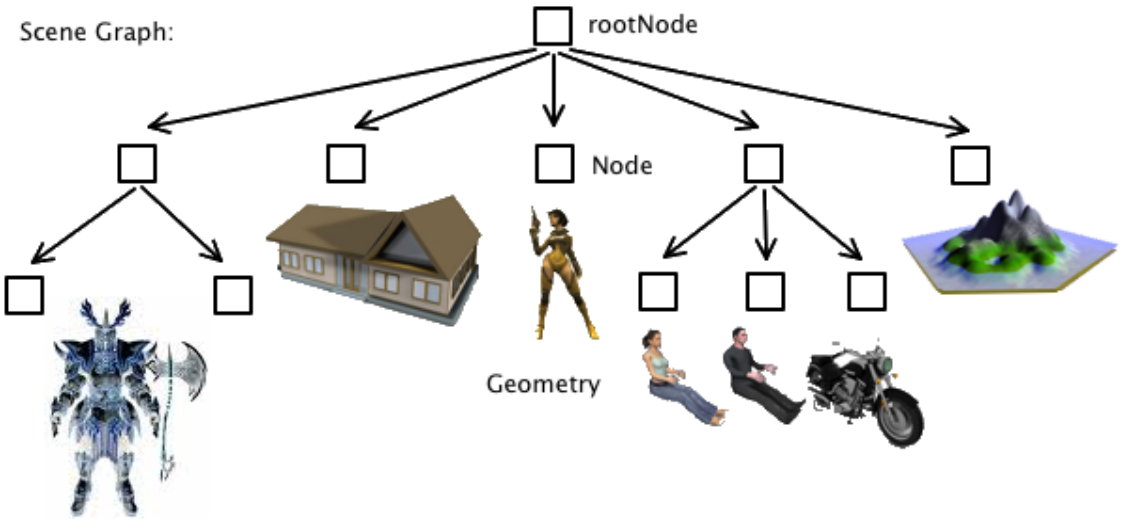


Figure 3.2: Scene graph structure in jME3. Picture taken from jME3 website [38].

Rendering aspects can be set for geometry by attaching *materials* to them. jME3 uses a fully shader based architecture, which means that it does not utilize the old fixed function pipeline. Therefore materials in jME3 are in reality shader programs. A material added to a geometry includes a material definition, which is a layer between the material and the shader programs. This is a layer

built in jME3 to simplify and make the process more manageable.

Light can be added to any spatial in the scene graph, by adding *light* component. This can be either a *spotlight*, *point light*, *ambient light* or *directional light*. Appropriate parameters can be set for each light type.

3.4 Ardor3D

3.4.1 History

The Ardor3D [1] project was started in September 2008, by Joshua Slack (also known as *renanse*) and Rikard Herlitz (also known as *MrCoder*). They were former core developers of the jMonkeyEngine project, but they left the project in order to start this one. The reason for starting a new project was because of a number of reasons, including issues with the naming, provenance, licensing and the community structure in the engine [2]. They also wanted to back a powerful open source Java engine, with a more organized corporate support behind it. Therefore they also founded the company Ardor Labs [2]. The first public release of Ardor3D came in January 2009, with the 0.2.1 version. There were frequent releases up until the most recent version, which was released in November 2010. This was the 0.7 version.

3.4.2 Releases

After the project was started in 2008, there have been a steady development in the project, leading up until the most recent release in 2010. There have not been released any official versions since the 0.7 version was released in 2010. There have however been additions and bug fixes to the latest source in the repositories. According to the milestones set by the project there are just some features and bug fixes that prevents them from releasing the 0.8 version. A list of the releases are given below:

- 0.2.1 released in January 2009.
- 0.3 feature complete on January 2009.
- 0.4 released in March 2009.
- 0.5 released in May 2009.
- 0.6 released in November 2009.
- 0.7 released in November 2010.

[1] Website: `http://ardor3d.com/`
[2] Website: `http://ardorlabs.com/index.php`

3.4.3 Community

There seems to be a good, helpful and active community surrounding the project. There is much activity on the forum, and questions are usually answered within hours. The core developers are very active on the forum, and are helpful when it comes to addressing questions or issues and problems. There is a showcase section on the forum, but the community does not published so much there.

3.4.4 Showcase

Ardor3D have been used for both visual simulations in scientific and industry applications, as well as for games. The following listing presents some of the projects done in Ardor3D.

- Rolls-Royce Marine has used Ardor3D for creating a GUI based platform for their next generation DP product [60].
- NASA Jet Propulsion Lab [51] and NASA Ames Research Center [49] has used Ardor3D. This have been as the 3D engine in their rover simulation tools [50]. It has also been used to create terrain and shadowing techniques to use in their rover simulation tools. Ardor3D has also been used with their CAVE environment [3].
- Danish artists used Ardor3D in a museum exhibit called *Adventures In Immediate Unreality* [4]. It was used for creating a demo with planets that featured vibrant, animated atmospheres.
- Caromble [9]: A game with breakout style gameplay [1], with unique comic-style graphics.
- Spaced [65]: A massively multiplayer online role-playing game set in space, developed by Fearless games [13].
- ArdorCraft API [5]: A framework for building blockworld games.

3.4.5 Licensing

Ardor3D is licensed under the open source zlib/libpng license [80]. This is a permissive free software license. The software is provided *as-is*, and the authors of the Ardor3D API cannot be not held responsible for any use of the software. The software may be redistributed freely in any way.

[1] Breakout is an arcade game developed by Atari in 1976. More information here: `http://en.wikipedia.org/wiki/Breakout_(video_game)`

3.4.6 Packaging

Ardor3D consists of several packages that offer different parts of functionality. The core of the API is in the *ardor3d-core* package. This contains the core parts of the API, such as the scene graph data structure, the renderer, bounding volumes, math, controllers and more. The packages *ardor3d-jogl* and *ardor3d-lwjgl* contains code specific to the two renderers it supports, which are JOGL [39] and LWJGL [44]. *ardor3d-awt* and *ardor3d-swt* contains code specific to the two different toolkits. The packages *ardor3d-animation*, *ardor3d-collada*, *ardor3d-effects*, *ardor3d-extras*, *ardor3d-terrain* and *ardor3d-ui* contains various extra features offered by the API. The package *ardor3d-examples* contains example code that demonstrates various features and capabilities of the API.

3.4.7 Scene Graph Structure

The organizing of the scene in Ardor3D is done by adding elements directly below the root in the scene graph. All the content is added directly below the root node, this includes geometry, materials, functionality components and lights. Views into the virtual world is added by *canvas renderers*, which provide control over the *render context* and *camera*.

All elements that are added to the scene graph in Ardor3D are *spatials*. A spatial is an abstract structure, and holds information about the transformation of the element, as well as its parent. Spatials also have different render states associated with them. These are in the form of *render buckets*, which enforce different rendering behaviours for different rendering methods. This include shadows, transparency, translucency, opaque and more. This can be inherited, or overridden by potential children. Spatials are subdivided into two different groups, *node* and *mesh*. A node can only have children. The mesh element is used to represent geometry, and is a leaf node. Mesh elements can be added as children to node elements. The implication of storing the transformation in the spatial is that every element can have a transform component associated with it. This may reduce the verbosity, opposed to for example Java 3D. This is because it is not necessary to add additional elements to the scene graph just to express the transform of a geometry element.

Rendering aspects can be set for meshes by attaching *material states* to them. These describe various rendering aspects, such as materials, lighting, texturing and shader programs. Light can be added to any spatial in the scene graph, by adding a *light state* to the spatials render state. *Lights* components can be added to this light state, and can be either *directional light* or *pointlight*. Appropriate parameters can be set for each light type.

3.5 jReality

jReality [42] is a 3D scene graph API designed for 3D visualization, and is specialized in mathematical visualization. jReality supports various rendering backends, including real-time environments such as OpenGL, but also exportable file formats such as RenderMan, PostScript, PDF and SVG. The API also supports spatialized audio, with an accurate audio rendering pipeline that includes physically accurate simulation of sound propagation that implicitly creates Doppler effects. The API is also built to allow seamless transition of applications from regular desktop use, to immersive virtual environments without modification to the source code. It supports various tracking devices. The project is developed, and maintained by members from various universities, including the Technical University of Berlin [67], University of Munich [73], and City College of New York [10] amongst others. The latest version of jReality was released in April 2012. jReality is licensed under the BSD license [8]. jReality have been used mostly for scientific applications, including The Portal [71] (Parallel Optical Research Testbed and Laboratory), which is a virtual reality CAVE installation, for mathematical visualization and research. Other applications include images created for the Imaginary exhibition [22], an animation of a flyby of Mars in collaboration with the Institute for Geodesy [24] and various other virtual environment applications.

3.6 Aviatrix3D

Aviatrix3D [7] is a pure retained-mode 3D scene graph API, and is specially focused towards the needs of the data visualisation and simulation markets. The API have been built to be as expandable as possible, in order to accommodate for the diversity in hardware that the target audience may be using. This extensibility extends to all levels of the API, from the scene graph data structure components to the various stages in the rendering pipeline. There is also a focus on multi-threaded functionality. Aviatrix3D supports integration with both AWT and SWT, and uses only OpenGL through JOGL. The most recent version of Aviatrix3D is 2.2, which was released in September 2011. The project is open source, and licensed under the GNU LGPL version 2.1 [16]. There is not so much information available, regarding which projects have used this API. However it has been used by Shapeways [62], who creates real world objects from 3D models. It has also been used in the Xj3D project [79], as the rendering engine. Xj3D is a toolkit/browser that displays 3D modelling formats such as VRML and X3D. This was created by the Web3D Consortium [77].

3.7 Xith3D

Xith3D [78] is a 3D scene graph API that is designed from the same basic architecture as Java 3D. Because of this, it is fairly easy to port code from Java 3D to Xith3D. Xith3D was created to be a fast scene graph and renderer, while at the same time allowing for extensions and enhancements to all stages of the rendering pipeline. The latest version of Xith3D is 0.9.6-beta1, and was released in January 2008. Xith3D is open source, and is licensed under the BSD license [8]. The project was initially started by David Yazel to create the game Magicosm [45]. Xith3D have also been used for other projects, such as HELI-X [17], which is a professional R/C helicopter flight simulator.

3.8 jPCT

jPCT [40] does not use a scene graph for representing the 3D scenes, so it is not really a 3D scene graph API. It is nevertheless included in this investigation, in order to give a complete picture of the 3D API alternatives available in Java. Instead 3D objects and functionality is simply added to the *world*, rather than the scene graph. jPCT aims to provide a small, fast and easy to use API for rendering 3D graphics. It features both a hardware and a software renderer for OpenGL, and supports integration with both Swing and AWT. It also supports Android. jPCT is an old project, and while it is unclear exactly when it was started, the 0.31 version was released in August 2000. The most recent release was version 1.24 in October 2011. The jPCT project is not open source, and the license is available here [41].

jPCT has been used for many games, including *Max the flyer 3D* [46], which is a game for Android where you control a space ship using the accelerometer. *Forgotten Elements* [15] is a online hack and slash multiplayer game. *SkyFrontier 3D* [64] is a racing game where you steer a space ship to its destiny, and have to avoid blocks by jumping and using the accelerometer.

Chapter 4

Evaluation Methodology

This chapter describes the methodology used for the evaluation of the 3D scene graph APIs. The evaluation will use an experimental approach that looks at the APIs at four different layers. These four layers cover different aspects of the API, and range from low level aspects such as performance, to system architecture and project management. Some of the methods are based on already defined frameworks, but some of them have been defined explicitly for this evaluation. Section 4.1 describes the idea behind this methodology. Section 4.2 describes the method used for evaluating the project management aspects, and the technical infrastructure of the open source projects. Section 4.3 describes the methodology used for evaluating the architecture of the APIs. Section 4.4 describes the method used for evaluating the features and capabilities of the different APIs. Finally Section 4.5 describes the methodology used for evaluating the performance of the APIs.

4.1 Comparing APIs

The evaluation of the APIs in this thesis will be done by investigating different aspects of the APIs, and then comparing them against each other. This comparison will use a four layered approach, which will be defined in this chapter. This approach builds on already defined frameworks for the more general parts of the comparison. For the more specific parts of the comparison, specific methods have been defined in cooperation with the employees at IFE. This is done due to a lack of any existing methods that were satisfactory.

We have defined a comparison method that divides the comparison into four layers. These four layers cover different aspects of the APIs, and are important to do in order to make this evaluation as comprehensive as possible. When evaluating the APIs it was important that it covered both

the high level aspects, such as project management and technical infrastructure, as well as the low level aspects. The low level aspects include what features the APIs offer, as well as how well they perform. It was also important to take a look at how the code behind the APIs were structured, therefore an intermediate layer between the two edges (top, bottom) were necessary to investigate. Therefore the architecture behind the APIs were also decided to be subject for the comparison. The results regarding each of the different aspects are used together when comparing the APIs against each other. This layered approach gives the evaluation both a *top-down*, and *bottom-up* approach. This means that the evaluation is able to give insight, and has the ability to highlight strengths and weaknesses more accurately and all-encompassing.

I was unable to find any already existing frameworks that were as all-encompassing as required, therefore it was evident that I had to define my own method for conducting this comparison. For the parts of this evaluation that are more general, I was able to find existing frameworks that were satisfactory. This applies to the more abstract high level concepts, such as project management, and system architecture. However the more low level aspects, such as the performance between the 3D scene graphs APIs, and the features offered by them, were too specific to find any sufficient frameworks. Therefore I have in cooperation with employees at IFE defined some methods that captures the most important aspects. The four layers are shown in Figure 4.1.

In essence this comparison method is composed of four individual comparison methods. The results from each of these methods can by themselves be used as an atomic evaluation (and in reality they are), and the results are not necessarily dependent on each other. However by taking the results from each of these methods and comparing them as a whole, we are able to investigate, and get a more complete picture of the APIs. This strengthens the results, and give a more accurate and comprehensive evaluation of the APIs.

The following sections will go into more detail regarding the various aspects that the evaluation will focus on. Section 4.2 covers the first level, *Project Management*. Section 4.3 covers the second level, *System Architecture*. Section 4.4 covers the third level, *System Features and Capabilities*. Section 4.5 will cover the fourth and final level, *System Performance*.

4.2 Open Source Project Management Aspects

When comparing the APIs it is important to consider the projects, and the people that are behind them. This is especially true in the case of open source projects, where there is a great risk involved, because it is unknown how the development of the API might continue in the future. Therefore it

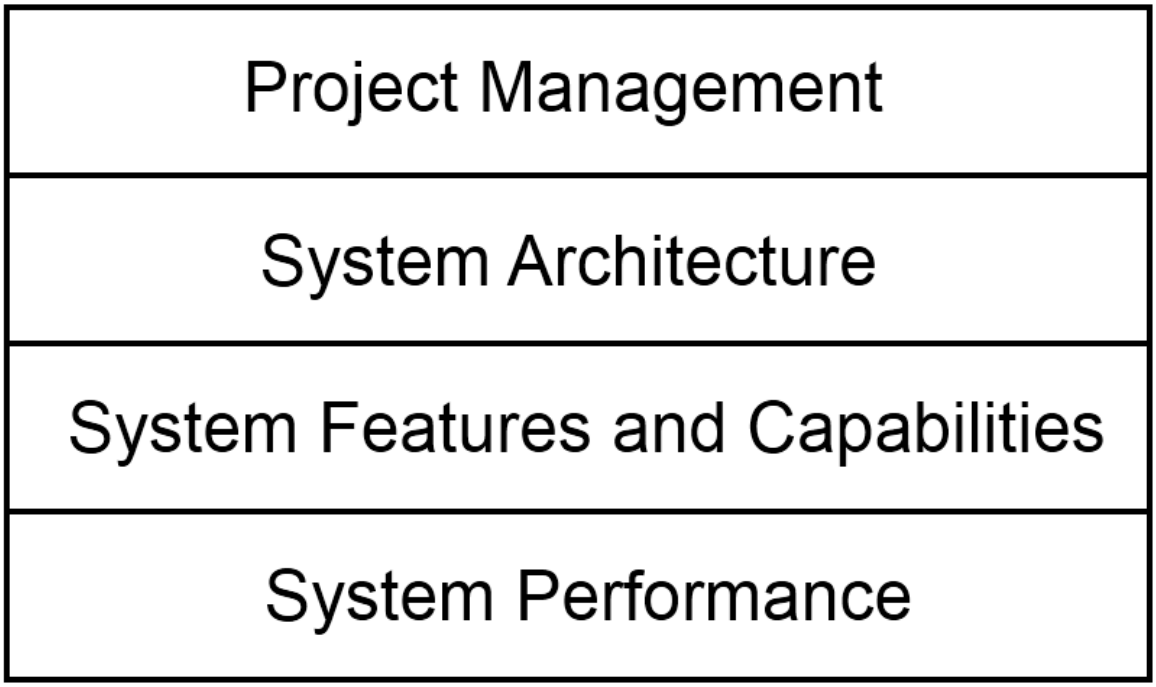


Figure 4.1: The four layers that are investigated during the evaluation of the APIs. The top level looks at the project management and technical infrastructure of the projects. The second layer looks at the architecture behind the APIs. Third layer investigates the different features and capabilities of the APIs. The fourth layer looks at the performance between the APIs.

is important to investigate the healthiness of the projects, in order to give an indication of how well they are performing, and how they might continue to perform in the future. Because of this, we will investigate the project management aspects, and the technical infrastructure of the APIs. This will give an indication to the healthiness of the APIs.

There are several important aspects to consider when measuring the *healthiness* of an OSS project. One of the most characterizing aspects of OSS development, is that much of the development is carried out by individuals on a voluntarily basis. These contributors are often located in diverse geographical locations, and may never actually meet in person. Because of this most communication between contributors are done over the internet, or through other communication channels. This is quite a contradiction to traditional software development, where all the team members are located at the same geographical location, where they have the ability to convey information by talking face-to-face with one another. With OSS projects you loose this because of its *scattered-nature*, and therefore it is important that they have the necessary tools to enable the required communication. Other aspects include the presence of systems for testing code, which is important to ensure that the code works, and helps identify errors and bugs in code builds. How easy is it to communicate bugs and issues, for both users and developers. These aspects, and more will be used for measuring the projects healthiness, or rather the degree of a projects *maturity*.

For the comparison of the project management aspects, I will use an already defined framework. The framework measures the maturity of the different OSS projects, and is described in greater detail in the following section.

4.2.1 Comparing Project Maturity

In order to measure the maturity of the OSS projects I will use a framework that focuses on five different categories related to the management of an OSS project. The method was described by Michlmayr [90], and is used to determine the successfulness of OSS projects. The assessment method looks at the presence of various channels and tools for communication and coordination within the project, and to what degree these are open or closed. This is divided into five categories, featuring a set of simple questions, that can be answered with a simple *yes or no*. A grading is applied to the answers, so that it is easy to compare the maturity between the various projects.

The reason I have chosen this framework for the comparison, is because it captures many important and critical aspects when it comes to project management, and the technical infrastructure in OSS projects. The assessment questions are shown below.

1. Version Control

1. Does the project use version control?
   - No, the project does not use any version control (0 points)
   - Yes, it is used but the repository is private (1 point)
   - Yes, it is used and the repository is available to the public (2 points)

2. Mailing Lists

1. Are mailing lists available?
   - No, the project has no mailing list at all (0 points)
   - Yes, there is one mailing list (1 point)
   - Yes, there are multiple, specialized mailing lists (2 points)
2. Are mailing list archives available?

3. Documentation

1. Does the project have documentation for users?
2. Does the project have documentation for developers?

4. Systematic Testing

1. Are release candidates available?
2. Does the source code contain an automatic suite?
3. Does the project use a defect tracking system?

5. Portability

1. Is the software portable?
   - No, it has been written specifically for a single platform (0 points)
   - Yes, it supports a limited number of platforms (1 point)
   - Yes, it automatically determines the properties of a platform (2 points)

The *version control* category focuses on the presence of tools for version control. This is an essential part of any project, and is absolutely required for developers working on the same code. It is an easy and effective way to coordinate the work to be done, and to integrate the changes

that are continually made to the code. The questions look at the presence of such version control systems within the projects, as well as whether they are made private or public. I define private version control as the presence of such a system, but read and write access is limited only to the core developers. A public version control on other hand I define as the presence of such a system, and read access is available to everyone. Write access does not necessarily need to be public, but there should be ways to submit patches or similar. This is important because it directly affects how easy it is for users and more importantly potential developers to browse the source code. This let them follow the development, as well as possibly contributing to the project. The most common tools that are used for version control are CVS [1] and SVN [2].

The *mailing lists* category refers to mailing lists as a tool for coordination in a project, as well as a place to look for documentation (through archives). While mailing lists were very popular a few years back, the use of discussion boards (also known as internet forums) are much more common these days. The only difference between the two, are that whereas with a mailing list, the people involved are notified directly by receiving a new email when someone replies. With discussion boards the participants have to visit the discussion board themselves to see the reply. However most discussion boards have the ability to notify participants with email when new replies have been made. Discussion boards usually archives discussions as well, which means that there are no need to put extra effort into archiving systems. Because of this I feel that it is not fair to evaluate the projects based on a criteria that is to some degree outdated. Therefore I have chosen to modify point *2. Mailing Lists* in the evaluation described by Michlmayr, to accommodate more to what is common in the industry today (the paper by Michlmayr was published in 2005). More specifically I have modified it, so that it also incorporates *discussion boards*. The modified category is shown below.

2. Mailing Lists and/or Discussion Boards

1. Are mailing lists and/or discussion boards available?
   - No, the project has no mailing list/discussion board at all (0 points)
   - Yes, there are mailing lists/discussion boards (1 point)
2. Are mailing list and/or discussion board archives available?

The *documentation* category looks at the presence of documentation for the project. Documentation is not only important for learning how the various parts of the system works, but also for

[1] More information about CVS can be found here: `http://cvs.nongnu.org/`
[2] More information about Subversion can be found here: `http://subversion.apache.org/`

learning the various guidelines for the project, such as the coding style. This category addresses both the need for user documentation, as well as developer documentation. User documentation focuses more on how to get the various parts of the system to work, and is on a much more entry-level to the project. The developer documentation is geared more towards the developers of the project, and is much more in-depth and of a technical nature. Of the two, Michlmayr showed that the user documentation is the most important for a project, as it will be used by both the users and the developers at the entry-level to the project. Documentation is created specifically for the purpose of educating users and developers, and discussions and questions asked in mailing lists or discussion boards do not count as documentation.

The *systematic testing* category looks at the release strategy for the project, as well as the presence of various systems for handling important processes for testing code and filing bug reports. The presence of release candidates preceding official releases indicates a well defined release plan. The presence of automatic testing suites for regression or unit tests are important in order to assure the quality of new code. It also looks at the presence of tools for handling bug reports, which are important to improve the quality of the software.

The *portability* category looks at whether the code has been ported to a variety of platforms, or if it is just available to that one platform. Code that is only available for one platform, is in many cases hard-coded to suit that one platform. While this might be good in some cases, it often means that many characteristics of the system has not been highlighted yet. Defects are often found during the porting process of a system, which would not otherwise have been highlighted. Because of this, if the project is available on various platforms, it is most likely to have a more robust code base than if not.

The technical infrastructure of the projects are very important due to the nature of OSS projects. It is a necessity that the technical infrastructure is aimed towards collaboration between the individuals involved in the project, which are in many cases separated by large geographical distances. Developers need these tools to convey information, and in order to be able to coordinate their work. It is also important for attracting new possible developers, as well as users. Karl Foger focuses on many of the same points as the described framework, in his book ”Producing Open Source Software” [85]. The framework for measuring the project maturity is summarized in Table 4.1.

Framework for measuring Project Maturity			
1. Version Control			
Does the project use version control?	No (0 points)	Yes, private (1 point)	Yes, public (2 points)
2. Mailing Lists and/or Discussion Boards			
Are mailing lists and/or discussion boards available?	No (0 points)	Yes (1 point)	
(In case of mailing lists present) Are mailing list archives available?	No (0 points)	Yes (1 point)	
3. Documentation			
Does the project have documentation for users?	No (0 points)	Yes (1 point)	
Does the project have documentation for developers?	No (0 points)	Yes (1 point)	
4. Systematic Testing			
Are release candidates available?	No (0 points)	Yes (1 point)	
Does the source code contain an automatic suite?	No (0 points)	Yes (1 point)	
Does the project use a defect tracking system?	No (0 points)	Yes (1 point)	
6. Portability			
Is the software portable?	No (0 points)	Yes, limited number of platforms (1 point)	Yes, automatically determines platform (2 points)

Table 4.1: Framework for measuring the Project Maturity.

4.3 System Architecture Analysis

As part of this evaluation it is very important to consider the data and code structure of the systems. The structure often reflects the quality, and thought that has been put into the design of the systems architecture. It is common that the design of the systems, often mirror the organizations that develop them [87]. When investigating the quality of the system architecture for the APIs, I will focus on the modularity of the systems.

The modularity refers to a systems logical partitioning of the design into appropriate modules. The contents of a module should ideally perform related tasks. The concept of modularity describes at heart the interdependency within modules, and the independence between different modules [81]. With functionality grouped in appropriate modules, it makes it much easier to change different parts of the system. This increases the flexibility of the system, both in terms of the scope and the scale of the system. The opposite of a modular system, is one where the code is tightly integrated with the other code that is not only located within the same module, but also between the code inside different modules. This results in code that is designed to work specifically with some modules, and creates dependencies between that code and the modules. These dependencies increases the risks when making changes to the system. The other parts of the system that depends on it, will be impacted by the change. In the worst case it may break parts of the system. The risk for this increases with the number of dependencies between different parts of the system, both direct and indirect. Code that is modular, and clearly separated into different modules may also be easier to manage and maintain. This is because the different parts are more separated by functionality and focus. This makes it easier to maintain and also comprehend. It might also lower the ”entry-level” for volunteers that may want to contribute to the project as well, because the more isolated parts are not so intricate.

While it is hard to measure the modularity of a API directly, there are various concepts one could choose to focus on, in order to measure different degrees of modularity in a API. These concepts focus on different aspects, and uses a variety of different metrics. For example one could look at the coupling and cohesion between the elements in the system, which gives indications of the design structure of the software [94] [82]. Another concept is to look at how many files that have to be be changed in order to apply a change to a file, which have been used in order to measure how the software *decay* over time [84]. In order to investigate the modularity of the APIs, I have chosen to use Design Structure Matrices in order to extract, and calculate much of the necessary information. DSMs are described in greater detail in Chapter 2, Section 2.7. The framework, and metrics I will

use for comparing the system architecture of the APIs is described in greater detail in the following section.

4.3.1 Comparing System Architectures

When comparing the modularity of the different APIs I will create *Design Structure Matrices*. These show the dependencies between the elements in the APIs, and will be used as a basis for the analysis of the systems modularity. DSMs are described in greater detail in Chapter 2, Section 2.7.

I have chosen to construct the DSM from the classes in the systems. From the DSM I will calculate some metrics. The first metric is called the *System Stability*. This value describes the likeliness that when applying a change to any part of the system, it will not affect other parts of the system. The second metric is the *Cyclicality* within the systems. This metric captures dependency loops, and have been shown to directly relate to the number of bugs in the code.

System Stability

The system stability is a metric that indicates how easy it is to modify, and maintain an API. It measures how likely a change to a part of the system is to affect other parts of the system. When many parts are affected, it makes it harder to maintain it, because the affected parts might also need to be modified. As more parts require to be modified, it also increases the likeliness that some of the parts may be broken. In order to calculate the system stability it is necessary to also calculate another metric, called the *propagation cost*.

The propagation cost is a metric that gives a cost value to the degree of *coupling* within the system. This cost measures how likely a change to one element in the system is to affect another element, on the average. This is present with both direct dependencies between elements, and indirect dependencies through a series of chains of dependencies. This is based on the research by MacCormack et al. [88].

The dependencies that are used when calculating the propagation cost are extracted from a *visibility matrix*. A visibility matrix shows all the direct and indirect dependencies between the elements in the system. The visibility matrix holds all the dependencies between the elements, for a given path length. The path length is the number of indirect dependencies that should be shown in the matrix. The visibility matrix is given by multiplying the matrices for each path length together, resulting in the visibility matrix. Figure 4.2 shows the matrices for the different path lengths, who are multiplied into the final visibility matrix. This visibility matrix is created from the system shown

in Figure 2.8 found in Chapter 2. The first matrix features a path length of 0, and therefore only captures each elements dependency with itself. Following the path length for 1 shows the direct dependencies between the elements in the system. The path lengths of 2 and 3 the captures indirect dependencies. The path length of 4 shows no dependencies, and therefore marks that there is no need to further investigating higher path lengths. The final matrix shows the visibility matrix, for path lengths 0 to 4. From this visibility matrix several metrics can be extracted, including the propagation cost.

M^0

	A	B	C	D	E	F
A	**1**	0	0	0	0	0
B	0	**1**	0	0	0	0
C	0	0	**1**	0	0	0
D	0	0	0	**1**	0	0
E	0	0	0	0	**1**	0
F	0	0	0	0	0	**1**

M^1

	A	B	C	D	E	F
A	0	**1**	**1**	0	0	0
B	0	0	0	**1**	0	0
C	0	0	0	0	**1**	0
D	0	0	0	0	0	0
E	0	0	0	0	0	**1**
F	0	0	0	0	0	0

M^2

	A	B	C	D	E	F
A	0	0	0	**1**	**1**	0
B	0	0	0	0	0	0
C	0	0	0	0	0	**1**
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

M^3

	A	B	C	D	E	F
A	0	0	0	0	0	**1**
B	0	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

M^4

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	0	0	0	0
C	0	0	0	0	0	0
D	0	0	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

$V = \Sigma M^n ; n = [0,4]$

	A	B	C	D	E	F
A	**1**	**1**	**1**	**1**	**1**	**1**
B	0	**1**	0	**1**	0	0
C	0	0	**1**	0	**1**	**1**
D	0	0	0	**1**	0	0
E	0	0	0	0	**1**	**1**
F	0	0	0	0	0	**1**

Figure 4.2: The picture shows the visibility matrix for path lengths from 0 to 4. The final visibility matrix shows all the dependencies for the elements. Image taken from MacCormack et al. [88].

The propagation cost can be calculated in two ways, either by looking at the *Fan-Out Visibility* or the *Fan-In Visibility*. The Fan-Out is calculated by summing along the rows of the visibility matrix, and then dividing the result with the total number of elements in the visibility matrix. Elements with a high Fan-Out Visibility depends upon many other elements in the system. The Fan-In Visibility is calculated by summing along the columns instead of the rows, and then dividing by the total number of elements. Elements with a high Fan-In Visibility has many other elements that depends on it. The dependency relationships between the elements have a symmetric nature, because if element A depends upon element B, then element B naturally has element A depending upon it. Because of this, the calculation of the Fan-Out and the Fan-In Visibility will give the same result. The propagation cost is given by computing the average of either the Fan-Out or the Fan-In Visibility. Equation 4.1 shows how to calculate the propagation cost.

$$\text{Propagation Cost} = \frac{\sum_{n=0}^{\infty} M^n}{n * n} \times 100 \quad \text{where} \quad n = \text{path length} \tag{4.1}$$

Equation 4.2 shows the calculation of the propagation cost based on the example in Figure 4.2, using the Fan-Out Visibility.

$$\text{Propagation Cost} = \frac{6+2+3+1+2+1}{6*6} \times 100 \quad = \quad 42\% \tag{4.2}$$

The propagation cost summarizes the visibility at the system level, and it shows on average how many elements that could be affected by a change to any element in the system.

The system stability is then calculated by subtracting the propagation cost from 100, giving us the likeliness (in percentage), that a change to one element is to **not** affect other elements in the system. In other words, the system stability indicates how likely a change to the code are not to break other parts of the system. Therefore a higher percentage for the system stability is better. The equation for the system stability is shown in Equation 4.3.

$$\text{System Stability} = 100 - \text{Propagation Cost} \tag{4.3}$$

Cyclicality

The cyclicality metric looks at the number of dependency cycles within the system, also known as component loops. This metric has been shown to have a direct impact on the number of bugs in systems [97]. This metric is based on data from the system analysis software tool Lattix [1].

A component loop is where the dependencies between components form a loop. Lets say we have components A, B, and C. Component A depends upon B, B depends upon C, and C depends upon A. This forms a loop between their dependencies. Such loops are problematic, because it complicates the design process. That is because it must be coordinated with the changes that are done to the other components as well, in an iterative fashion [91] [96] [95]. This is because all components are all either directly, or indirectly affected by the changes. The names component loops and cycles will be used interchangeably throughout this section.

[1] Lattix is a system analysis tool, that enables analysis of the systems using design structure matrices. More information about Lattix can be found here: `http://www.lattix.com/`

Cycles in the software can be identified by using DSMs. This can be done by looking at the dependencies in the DSM. Dependencies that are located below the *superdiagonal* [1] in the DSM, indicates a *feed-forward* dependency. Such a dependency indicates that given the mark *(i, j)* in the DSM, component *i* provides data to component *j*. These dependencies are ok, because they do not spawn any loops. The dependencies that are located above the superdiagonal in the DSM are the ones of concern. These are called *feedback* dependencies, and they are the ones that spawn loops. This is because with a feedback dependency, component *j* provides data to *i*. Because of this, feedback dependencies may be the cause of undesirable loops in the architecture. In Figure 4.3 you can see the feed-forward dependencies below the superdiagonal, the feedback dependencies above, and a cycle is shown with a square box. There are two types of problematic cycles that I will focus on in this evaluation, which are *intrinsic cyclicality* and *hierarchical cyclicality*.

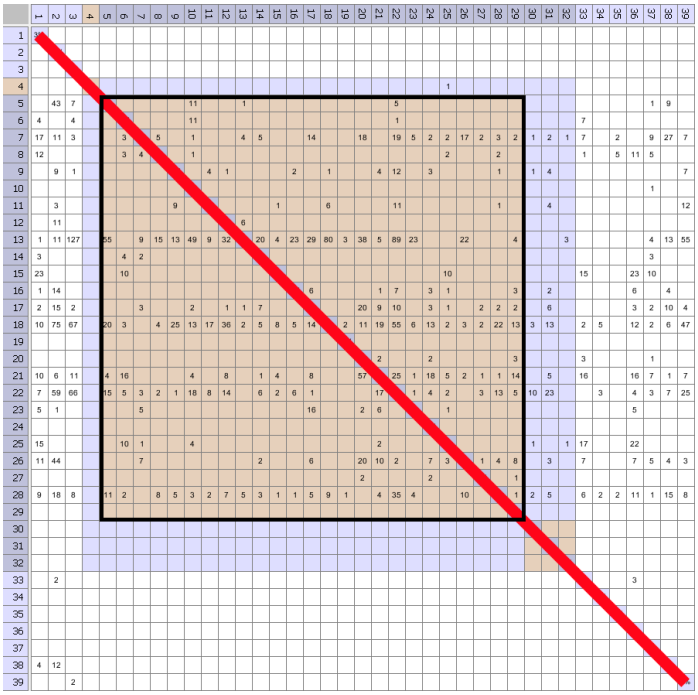


Figure 4.3: A flat DSM. The superdiagonal is illustrated by the red line. A component loop is shown with the black rectangle.

Intrinsic (system) cyclicality are loops that are formed when only considering the flat DSM. A flat DSM means that the hierarchical ordering of the different parts of the system are disregarded, and will not affect the result. The main implication of this, is that the results will not be affected

[1] The superdiagonal is shown in Figure 4.3 as the red line.

by dependencies that goes between the different parts of the system (packages). This metric has been shown to have the strongest correlation to the number of bugs in the software. The intrinsic cyclicality indicates how likely there are to be problematic loops in the code, and a lower percentage is better. Equation 4.4 shows the intrinsic cyclicality ($P_{I,is}$), which is the probability that a randomly chosen component of version s of application i belongs to such a loop. This is given by the ratio of components involved in loops in the flat DSM ($C_{I,is}$), to the total number of components (N_{is}). Figure 4.4 shows intrinsic loops in a flat DSM.

$$P_{I,is} = C_{I,is}/N_{is} \tag{4.4}$$

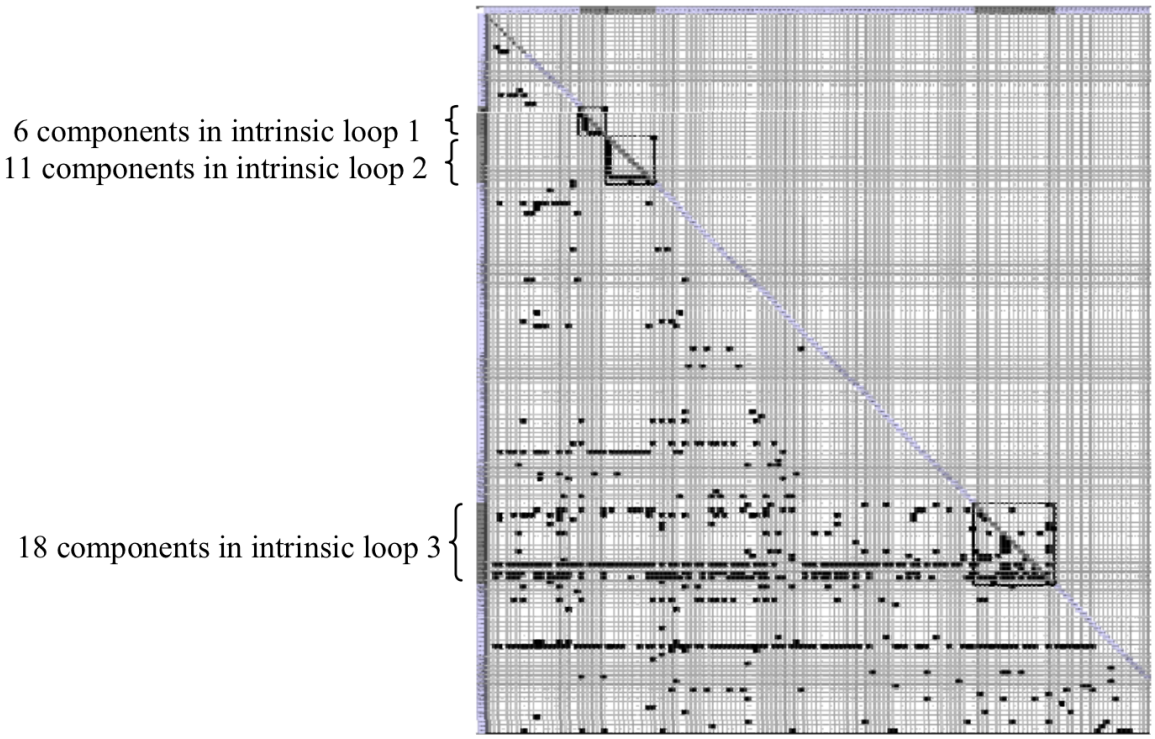


Figure 4.4: Flat DSM structure showing the intrinsic (system) loops. Image is taken from Sosa et al. [97].

Hierarchical cyclicality loops differ from intrinsic loops by taking the hierarchical structure of the DSM into consideration. This means that the results are affected by dependencies that goes between different packages. This value will generally be larger than the intrinsic cyclicality, this is because it takes the hierarchical structure into the calculation. Ideally the hierarchical cyclicality

should be as close to the intrinsic cyclicality as possible. The hierarchical cyclicality indicates how likely there are to be problematic loops in the code, taking into consideration the hierarchical structure of the code. A lower percentage for the hierarchical cyclicality is better. Equation 4.5 shows the hierarchical cyclicality ($P_{H,is}$), which is the probability that a randomly chosen component of version *s* of application *i* belongs to such a loop. This is given by a function of the number of components that are involved in loops, taking into account the constraints of the subsystems and layers used by the programmers to organize their code ($C_{H,is}$). To identify $C_{H,is}$, the number of components in loops in the sequenced hierarchical DSM are counted. The value is then divided by the total number of components (N_{is}). Figure 4.5 shows hierarchical loops in a DSM.

$$P_{H,is} = C_{H,is}/N_{is} \tag{4.5}$$

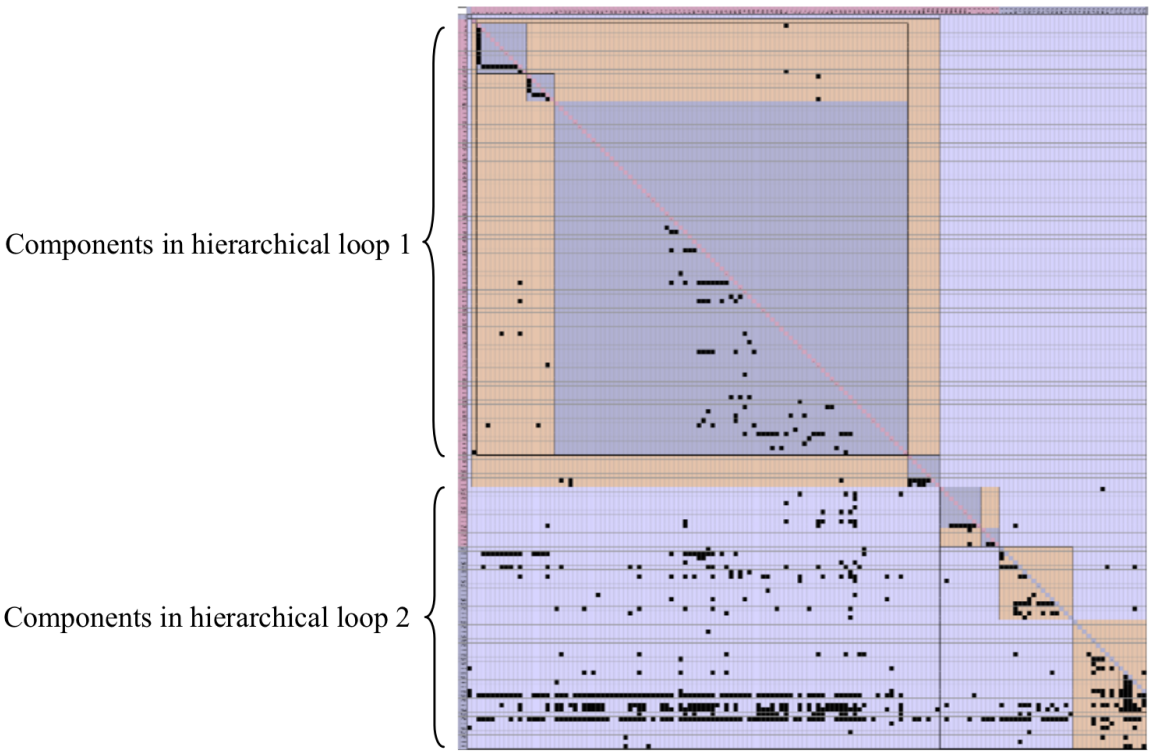


Figure 4.5: Hierarchical DSM structure showing the hierarchical loops. Image is taken from Sosa et al. [97].

Capturing dependency data

There are two ways to capture the dependencies between the source files, either by extracting the calls dynamically or statically. The former requires the system to be in a running state in order to capture the dependencies. With the latter it is possible to extract the dependencies from just examining the source code. I will only examine the static dependencies for this evaluation. This is also because it captures the design from the programmers perspective.

4.4 System Features and Capabilities

The evaluation will also investigate what the APIs offer in terms of features and their capabilities. This is very important to take into consideration, because it might highlight some crucial features that they might lack, or some clever function that they have implemented. While some of the aspects we will look at are trivial and easy to implement, there are others that are much more important. Some design choices might prevent the APIs from offering certain capabilities. The focus for this comparison will range from techniques and algorithms, to what platforms and technologies they support. The following section will describe this in much greater detail.

4.4.1 Comparing Features and Capabilities

For this comparison it was chosen to use an experimental approach. The reason for this is that no satisfying frameworks for comparing the features and capabilities between 3D scene graph APIs were found. Therefore we have defined our own method for the comparison of the different features and capabilities offered by the APIs. This was done in cooperation with employees at IFE, who have experience with which features and criteria that are important and useful when using a 3D scene graph API. Please note however that the criteria that are covered in this comparison does not cover every aspect of the APIs. This is because covering every aspect is not feasible, so therefore focus was chosen on those that were identified as the most important. It is also worth noting that they may have been influenced to some degree by what are IFEs needs during development [1].

The comparison consists of a series of *yes/no* questions, that are grouped into different tables based on which category they belong to. The covered categories include supported platforms, rendering aspects, scene graph specifics, model loaders, additional features, utilities, and Java specific features.

[1] IFE works with visualization within different fields, such as nuclear power, oil & gas, architecture, electricity production, and transportation.

There will be no scoring to the answers from the questions in this comparison. The reason for this is because it is hard to give values to the different features and capabilities offered by the APIs. It is our impression that it will be better to just list the results, and give an overall picture of what the APIs offer. By doing this we can highlight the offered features, and what they might lack.

The first category looks at what platforms are supported by the APIs. This covers the main desktop platforms, such as Windows, Linux, and Mac OS X. In addition the mobile Android platform is included. Apples iOS were not included, because the Java language is not supported on it. The first category can be seen in Table 4.2.

Supported platforms	
Does it work on Windows?	Yes / No
Does it work on Linux?	Yes / No
Does it work on Mac OS X?	Yes / No
Does it work on Android devices?	Yes / No

Table 4.2: System Features and Capabilities: Supported Platforms.

Comparison of rendering aspects are shown in Table 4.3. Here we investigate which graphics libraries are supported. The comparison also looks at the support for shader programs, and in case which shader languages. Another of the questions also look at whether the APIs support multiple bindings to the graphic libraries or not [1]. This question is hard to rate as good or bad, because there can be different ideologies and ways of thinking behind this. Supporting multiple bindings are good for modularity. Also in the case that one project should be discontinued, it is easier to switch, or to add a new one to the API. On the other hand, it might be hard to maintain support for multiple bindings. This can lead to reducing the overall quality for all the maintained bindings, or that only one of them is always maintained, leaving the other outdated. The other questions look at whether they have support for very low-level, but very important rendering functions. These cover the support for multi-pass rendering, frame buffer objects, and pre/post processing effects. These are all functions that are required to produce most of the visual effects that are used in modern days. It also looks at support for stereoscopic rendering, and here we are interested in *true* stereoscopic rendering, which involves the use of quad buffering. In addition it looks at the support for multiple canvas, which have many different uses, including virtual environments such as CAVEs.

Table 4.4 shows the questions related to the scene graph data structure. They look at whether

[1] To clarify; by binding we are referring to software libraries that act as wrapper libraries between the low level graphics APIs, such as OpenGL, and the Java programming language.

Rendering	
Does it support OpenGL?	Yes / No
Does it support Direct3D?	Yes / No
Does it support multiple bindings to the supported graphics libraries?	Yes / No
Does it support shaders?	Yes / No
Does it support GLSL?	Yes / No
Does it support Cg?	Yes / No
Does it support HLSL?	Yes / No
Does it support multi-pass rendering?	Yes / No
Does it support frame buffer objects (FBO)?	Yes / No
Does it support full screen pre/post-processing effects?	Yes / No
Does it support stereoscopic rendering?	Yes / No
Does it support multiple canvas (example: CAVE)?	Yes / No

Table 4.3: System Features and Capabilities: Rendering

the APIs support exporting/importing. It also looks at whether attributes are inherited hierarchically between the nodes in the scene graph or not. Here we are not considering transforms, but rather other attributes such as rendering states, culling states and others. It also checks whether the APIs support picking.

Scene graph	
Does the scene graph support export/import?	Yes / No
Are attributes inherited hierarchically (minus transforms)?	Yes / No
Does it support picking?	Yes / No

Table 4.4: System Features and Capabilities: Scene graph.

The category for model loaders covers which modelling formats are supported, and what can be loaded through them. There are many modelling formats that could have been added here, but it was chosen that it should be limited to only two, VRML [76] and COLLADA [12]. VRML is a ISO-standard [25], and both VRML and X3D (its successor) is used much in the industry. VRML is used much within different CAD-fields, and it is supported by many CAD-tools. The MPEG4-standard also contains parts of VRML. COLLADA is used as a interchange file format, and is used more in the entertainment industry. By checking for support for these two formats, we are able to see how well they support formats within different fields of 3D visualization. One format geared towards the CAD industry (VRML), and the other against the entertainment industry (COLLADA). The

questions also look at what types of animations can be loaded through the model loaders, namely key frame, and skeletal animation. The questions for this category can be seen in Table 4.5.

Model loaders	
Does it have a model loader for VRML?	Yes / No
Does it have a model loader for Collada?	Yes / No
Can you load key frame animation through one of the model loaders?	Yes / No
Can you load skeletal animation through one of the model loaders?	Yes / No

Table 4.5: System Features and Capabilities: Model loaders.

The next category looks at additional features that the APIs might offer. These can be seen in Table 4.6. The first question looks at whether the APIs offer fixed function functionality. Whether this is supported directly, or is emulated through shader programs is considered the same. This concept is described in more detail in the Chapter 2, Section 2.5. Furthermore this category looks at whether the APIs support features such as compression of textures, optimizing static geometry. It also looks at whether they have built in standard geometry, this ranges from low level primitives such as triangles, lines, to triangle strips, indexed arrays, and even cubes and spheres. Another of the questions identifies if there are any form of callback functionality that can be added to nodes. By this we mean code that can be added to nodes, which can be run at certain triggers, such as time, collision or others. In addition to this, the questions look at whether there are built in support for tracking devices, such as for example head trackers. Lastly are there any implemented navigation styles in the APIs, like first person controls, or orbit camera.

Additional features	
Does it have fixed function functionality (can be emulated)?	Yes / No
Does it support compression of textures?	Yes / No
Does it optimize static geometry?	Yes / No
Does it have built in standard geometry?	Yes / No
Does nodes have some kind of callback functionality?	Yes / No
Does it support tracking devices?	Yes / No
Does it have implemented different navigation styles?	Yes / No

Table 4.6: System Features and Capabilities: Additional features.

Table 4.7 shows the questions regarding additional utilities and systems offered by the APIs. This covers some general utilities such as optimization of geometry, by for example merging meshes, or creating optimized triangle strips of models. It also looks at whether it is possible to generate normals automatically, and if they have texture loaders. Then there are some questions that looks at whether there are official integration of auxiliary (or internal) systems in the APIs, such as physics, networking, and audio systems. We only consider the system as integrated with the API if it have been officially added into the API, making it trivial to use for the end user. Unofficial integration of systems with the API will not be considered as valid, however it can be mentioned if deemed significant. Then the questions look at integration of other systems in the APIs, such as animation, 2D user interfaces, particle systems, terrain systems and more.

Utilities	
Can you optimize the geometry?	Yes / No
Can you generate normals?	Yes / No
Does it have a texture loader?	Yes / No
Does it have integration with a physics system?	Yes / No
Does it have integration with a networking system?	Yes / No
Does it have integration with a audio system?	Yes / No
Does it have an animation system?	Yes / No
Does it have a system for 2D user interfaces?	Yes / No
Does it have a particle system?	Yes / No
Does it have a shadow system?	Yes / No
Does it have a terrain generation system?	Yes / No
Does it have a water system?	Yes / No

Table 4.7: System Features and Capabilities: Utilities.

The final category covers which Java specific features that the APIs offer. Table 4.8 shows the questions for this category. These questions investigates whether it supports Swing and SWT, which are two Java graphical widget toolkits. This relates primarily to whether you can have a canvas in either of the two. Then it looks at whether they have a standalone component, and if they support exclusive fullscreen mode.

Java specific features	
Does it support Swing?	Yes / No
Does it support SWT?	Yes / No
Does it have a standalone component?	Yes / No
Does it support fullscreen exclusive mode?	Yes / No

Table 4.8: System Features and Capabilities: Java specific features.

4.5 System Performance

As part of this evaluation we will investigate the performance of each of the APIs. This will be done by developing several performance benchmarks. This is an important aspect to consider, because it directly affects the speed, how much can be rendered, and the visual quality of the applications that can be developed in the APIs. Investigating the performance also gives information as to the quality and how optimized the code base for each of them are.

Existing solutions for benchmarking performance exists, such as Unigine [1] and 3DMark [2]. These are examples of complete software solutions that tests the computers performance regarding various aspects, such as the CPU speed or the rendering of various 3D graphical effects and techniques. These solutions are geared towards benchmarking hardware and low level libraries. Unigine and 3DMark tests the performance of OpenGL and DirectX (among other things), whereas we want to benchmark the performance of the APIs built on top of these low level libraries. To my knowledge, it does not exist any good well defined frameworks for benchmarking 3D graphical APIs in Java.

Since there do not exist any satisfactory frameworks for benchmarking the performance, I have chosen to develop my own benchmarks for measuring the performance of the APIs. These benchmarks have been designed to be atomic, and test specific parts of the APIs in isolation. Such aspects include state sorting, frustrum culling, and dynamic geometry. The benchmarks have been designed in cooperation with the employees at IFE. They have given their input as to which aspects of a 3D scene graph API are important to investigate, when measuring the performance.

More detailed information on the design and implementation of the performance benchmarks (testbeds) are given in Chapter 5.3.

[1] Unigine can be found at: `http://unigine.com/`
[2] 3DMark can be found at: `http://www.3dmark.com/`

4.6 Summary

The evaluation methodology proposed in this chapter investigates the APIs at four different layers. Each layer covers an important aspect of the APIs. The layers are *Project Management and Technical Infrastructure*, *System Architecture*, *System Features and Capabilities*, and *System Performance*.

The project management and technical infrastructure layer measures the projects maturity, which is a metric that classifies how healthy the open source project is.

The system architecture layer uses design structure matrices (DSM) to investigate the architecture of the APIs. Several metrics are calculated from the DSM. These look at the connectedness and cyclicality in the architecture. In essence this measures the stability of the APIs, considering how likely a change to the code is to affect other parts of the API. This affects both maintainability and the likeliness of bugs.

The technical features and capabilities are investigated by a series of yes/no questions, that looks at the presence of various important features. This ranges from high level questions that looks at what platforms are supported, to more technical aspects, such as support for multi-pass rendering.

The performance is investigated by a series of benchmarks (testbeds) that tests the performance of various aspects of the APIs.

Chapter 5

Testbed Design and Implementation

This chapter will give information about, and give a closer look at both the design and the implementation of the performance benchmarks. Section 5.1.1 gives general information about the benchmarks. Section 5.2 gives an overview at the design of the different benchmarks. Section 5.3 gives a closer look at the actual implementations of the various benchmarks.

5.1 General Information

There were several important aspects to keep in mind when designing and implementing these benchmarks.

First it was important that they should be kept as atomic as possible. This means that one benchmark should only attempt to test one specific aspect of the APIs. This is to keep the results clear, without any additional noise interfering with the results. If several aspects were mixed into one benchmark, it would become harder to analyse the results, and also there might be unwanted side effects of having different aspects operating at the same time. One could argue however that this makes the benchmarks non-representative of actual applications, and real-world use of the APIs. This is true, however these benchmarks are designed specifically to be just that, micro-benchmarks measuring the performance, and not actual applications.

Another important aspect where that the APIs should not be tweaked in any way, in order to improve their performance. This means that the benchmarks should be set up as they would have been done in a regular application. This way any optimizations that are done automatically behind the scenes would be applied, but none that the user must defined explicitly. There is a downside to this, which is that there might be some settings that could be turned on, which could have greatly

improved the performance to some degree for some of the APIs. However once you start enabling such tweaks there is no end to how much can be configured, and in many cases the end result would end up being so far from an actual application. Secondly such a tweak should have been enabled by default, unless it is only usable for special cases, in which case the original reasoning still stands.

5.1.1 Identical Implementations

A very important aspect to consider when designing, and implementing the benchmarks, was the fact that each implementation of the benchmarks in the various APIs had to be identical to the others. The reason for this is that if the settings were to differ, it might result in different workloads, or operations done in the various implementations. This would affect the results from the benchmarks, and would alter the results in an unintended way. How much difference it would make, depends on what the setting would affect. This could be a huge difference, or it could be barely noticeable. Nevertheless, it was deemed of utmost importance that the implementations of the benchmarks should be made identical between the APIs. This means that special care have been taken to ensure that all the implementations are identical. This includes how the APIs are set up and initialized, how the collection of data is handled, the creation of geometry, and the order of random calls.

5.1.2 Metrics

In order to compare the performance between the APIs, I have defined some metrics that will be used for the comparison. These metrics covers both the speeds at which the APIs operate, as well as stuttering, and memory and garbage creation.

The *time per frame (tpf)* captures the time it took to render each frame. This is an important metric because it tells us how much time the APIs spends rendering, and by investigating this metric, it is possible to find out how much time is spent doing the various tasks.

From the tpf many other metrics can be calculated. The *total time* used rendering the benchmark, which is achieved by just adding all the tpf values registered for the benchmark. The *average frames per second (avg fps)* can be calculated. This is probably one of the most commonly used metrics used when comparing the performance of 3D graphical applications.

Stuttering in applications occurs when the rendering speed suddenly drops to a much lower or higher value. This is often perceived by users, and frequent stuttering can often break normal use of an application. Because of this, the *number of spikes* during the execution of the benchmarks, are used as a metric for the evaluation. A spike is considered to be a difference in the tpf greater or

equal to a forty percent increase in the tpf.

The last metric we look at is the *base memory usage*, and the *memory garbage created* by each of the APIs. The first metric is the base memory usage for each of the APIs, and the garbage is the memory it creates during the execution on top of the base usage.

The metrics mentioned above are the ones that will be used for the main evaluation of the performance between each of the APIs. However in addition to these metrics, several other metrics are captured in the benchmarks. The reason these was omitted from the evaluation is because some of these are just metrics calculated based of the aforementioned metrics, or the results are too similar between them to hold any real significance.

5.1.3 Environmental configuration

The same computer was used for all the benchmarks. No additional applications, except the operating system was running on the computer during the benchmarks. The specification on the computer used is as follows:

- CPU: Intel Core i7 2600K Quad Prosessor 3.4GHz
- Graphics: ASUS GeForce GTX 580 1536MB
- Graphics Driver: GeForce 295.73,
  Default settings (except force vsync off)
- Memory: Kingston HyperX 8 GB 1600MHz DDR3
- Motherboard: ASUS P8Z68-V PRO, Socket-1155 ATX
- Operating System: Windows 7 Professional 64-bit, Service Pack 1
- Java version: Java(TM) SE Runtime Environment (build 1.7.0-b147),
  Java HotSpot(TM) 64-Bit Server VM (build 21.0-b17, mixed mode)

- LWJGL [1] used as OpenGL binding for Ardor3D and jMonkeyEngine3
- JOGL [2] used as OpenGL binding for Java 3D

[1] Lightweight Java Game Library can be found here: `http://lwjgl.org/`
[2] JOGL can be found here: `http://jogamp.org/jogl/www/`

5.2 Testbed design

The benchmarks are designed to be as atomic as possible, testing only a specific feature. The following subsections give an overview over the design behind each of the various benchmarks.

5.2.1 Benchmark: Dynamic Geometry

When rendering there are two modes to use, immediate or retained. The first gives absolute control over the rendering cycle, processing all the data on the CPU, and then sending it to the graphics card. As the amount of data to send increases it might impact performance, because all the data is sent every frame to the graphics card. The alternative is to use retained mode, where the data is stored in the graphics card memory. This eliminates the performance hit of transferring it to the graphics card. This is mostly used for static geometry, so in most cases there is a performance benefit to this. Most commonly used is either Display Lists or Vertex Buffer Objects (VBOs).

In the case of geometry that changes dynamically during the rendering it might be counter-effective to do this. The benefit from generating static geometry on the graphics card really depends on how often the geometry changes, or more specifically how many frames there are between the changes. This affects whether it is prudent to generate lists or not. If there is time for it, there will be a benefit, but if the geometry changes too rapidly it will only become counter-effective.

Therefore this benchmark will attempt to identify whether any of the APIs are generating static lists out of the dynamically changing geometry. If they are, it might be a problem, and will most likely affect the performance in a negative way.

5.2.2 Benchmark: Frustrum

When rendering 3D graphics it is important to keep the performance in mind. We are dealing with limited resources, and the time spent rendering should be used as effectively as possible. Time spent on something that is not visible to the user, are in most cases time and resources wasted. An example of this is geometry that are located outside of the cameras view frustrum, that is never visible to the user.

In most cases geometry that are located outside of the frustrum are culled away, meaning that they are removed from the rendering, and possibly from any other operations done on the geometry by the API. It is interesting to investigate how effective the APIs are at performing the frustrum culling.

5.2.3 Benchmark: Node stress add and removal

In order to add objects to the scene when using a scene graph structure the objects must be attached to the graph structure. This means that the objects must be added somewhere beneath the root node in the graph. As objects are added to, or removed from the graph, the APIs need to maintain internal structures that represents the relationships of the objects in the graph. How this is handled greatly affects the performance.

Because of this, it is very important that the APIs handles this in a good, and optimized way. Therefore this benchmark will investigate how the APIs performs when objects are rapidly added, and removed to, and from the scene graph. Depending on what operations they do during the process of add and removals, the performance might greatly differ between them. This could also lead to unstable performance, resulting in noticeable stuttering in the rendering. It is also very interesting to see how well their algorithms scale, as the number of objects are increased.

5.2.4 Benchmark: Picking

Picking of objects in a 3D scene is a very common operation. Therefore it is interesting to investigate how well the different APIs performs when doing such picking operations. In most implementations picking operations are solved by mathematically casting a ray into the scene, and then checking for collisions with the potential objects. Performance may vary based on how good their algorithms are, which might vary greatly based on how the APIs are built up internally, and how well they are utilizing their graph structure in the pick process. For example by using bounding volumes, the performance can be greatly increased. Therefore this benchmark will look at how well the APIs performs when doing picking operations, and especially how well it scales when the workload increases. The pick results must be on the primitive level.

5.2.5 Benchmark: State sort

OpenGL calls are very expensive, and excessive changes between OpenGL states might drastically decrease the performance during rendering. Because of this, it is important to make as few of them as possible. In cases where there are many OpenGL calls, a common approach is to sort the objects based on what OpenGL calls they require. This is commonly known as state sorting. By sorting the objects by their states, the number of OpenGL calls can be kept to a minimum.

How well the APIs handles rendering many different OpenGL states will be investigated in this benchmark. This will be tested by adding many objects in randomized order, that could possibly be

sorted during rendering for better performance.

5.2.6 Benchmark: Transparency sort

Transparent objects are surfaces that it is possible to see through, either fully or partially. The problem with transparent surfaces in OpenGL is the Z buffer, which is used for regular rendering. When using the Z buffer for rendering, pixels that are behind other pixels are not being drawn. This gives problems with transparent surfaces, since they can be seen through and the objects behind should be partially visible. When writing to the Z buffer this is not possible, because the objects behind (occluded) are not rendered.

The general solution to this is to draw all the opaque surfaces first, using the Z buffer. Then afterwards writing to the Z buffer is disabled, and then all transparent surfaces are rendered. Disabling writing to the Z buffer however means that the objects must be sorted manually, in order to be rendered correctly. The blending of the translucent surfaces is usually done by sorting the objects *back to front* based on their difference from the camera.

This means that transparent objects must be manually sorted by the APIs. Therefore it is interesting to measure the performance between them, in order to see how well it is implemented in each.

5.3 Testbed implementation

This Section will give an overview over the implementation of the various benchmarks. First some information about the geometry used in the benchmarks are given. Then a closer look at the base benchmark class is given, which is built upon by all the specific benchmarks. This is followed by some information about the output (results) given from a benchmark. Then the following subsections give information about the implementation of each of the specific benchmarks. Then lastly some information is given about a GUI that was developed to allow for easier execution of the benchmarks.

5.3.1 Geometry

In order to keep the benchmarks identical between the APIs, it was necessary to use a common data structure for the geometry that is used in the benchmarks. The geometry consist of either cubes or spheres, which were created based on these common data structures.

The cubes were created using indexed triangle arrays, and the spheres were using triangle strips. The sphere was based on the implementation of spheres (gluSphere) found in the OpenGL Utility Toolkit (GLUT) [1]. The data structures for both geometry structures contains arrays with vertex data, and optionally normals and texture coordinates.

In each of the APIs there is an implementation, that uses the underlying data structuers to create the appropriate geometry. Figure 5.1 shows a screenshot of the geometry used.

5.3.2 Base benchmark class

This class is used as the base for all the benchmarks. It takes care of all the essential operations that are necessary for all the benchmarks. This includes setting up and initializing the APIs, and the benchmark logic such as the data collection. Through inheritance each specific benchmark can extend and override necessary functions.

The idea behind this class is to keep the implementation of each specific benchmark as clean as possible. This is achieved by hiding all the benchmark logic, and the setup and initializing of the APIs in this class. Then each specific benchmark can override, and add only the specifics for that benchmark, keeping the code clean and improving readability. Note that this is one way to achieve this, there are other and arguably equally good ways to do this.

[1] The source code for GLUT can be found here: `http://www.opengl.org/resources/libraries/glut/`

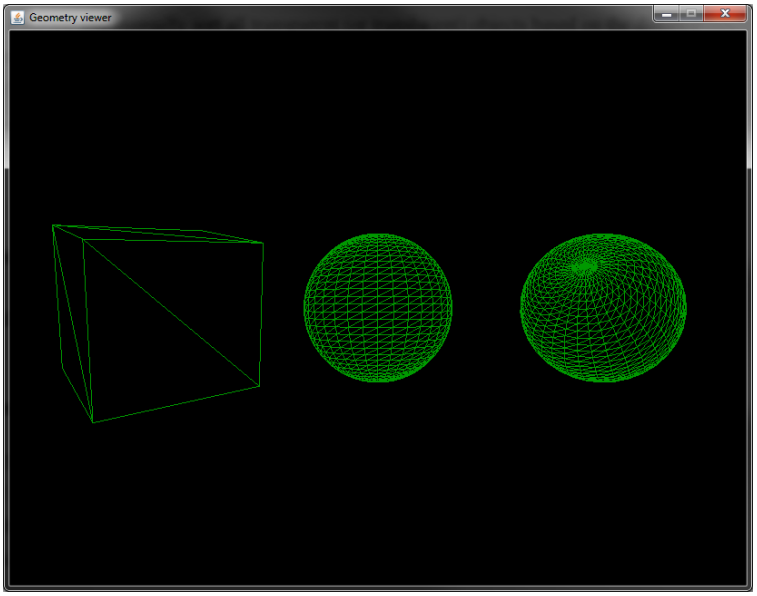


Figure 5.1: Ported version of gluSphere using triangle strips, and cubes using indexed triangles. Layers are built on top of these, with implementations in each of the APIs.

This class takes care of setting up the API for the benchmark. This includes initializing the low level rendering context, initializing the scene graph, and setting up various data used for capturing the different metrics.

The execution of the benchmark is divided into two different periods. At the start of each period there is a warmup period. During this period no data is collected. The reason for this warmup period is to eliminate any initial stuttering that could be caused by initializing the benchmark. This warmup period is defined both in terms of time (seconds) and number of frames. The warmup period is followed by a data collection period. This period is the actual benchmark period, and data for the metrics are logged during this period. At the end of each data collection period the data is written to csv files.

The calculation of metrics during the data collection period is kept to a minimum in order to minimally affect the results. For each frame the time per frame is stored in an array. The current memory usage is calculated by subtracting the total heap memory in the JVM by the current free memory. This is also stored in an array. Additionally the tpf is checked to see whether it is higher or equal to the spike threshold. Should it be higher, a counter is increased. There is also a simple check to see if the current tpf is greater than the current maximum tpf. This is used to find the highest tpf registered throughout the execution of the benchmark. Apart from these, no additional

data is collected throughout the data collection period. At the end of the data collection period, before data is written to csv files, additional metric data are calculated based on the ones gathered during the data collection period.

Every benchmark starts with a object count (number of geometry) of 32. The benchmark will run the warmup and data collection for this object count, and then increase the object count by the power of two.

It should be noted that it is hard to measure the exact memory usage in Java. This is because the Java Virtual Machine (JVM) features an automatic memory management system, where the memory is dynamically allocated and deallocated by the JVM. The memory heap in the JVM can be monitored, but is limited to getting the free amount of memory, in addition to the full possible memory space. So we can monitor the total used memory, but not the exact memory used. The automatic removal of memory from the JVM heap is also problematic, because there is no exact control over when this happens. This causes issues when trying to identify the base memory usage of the APIs, because we do not know if the JVM have removed potential memory garbage or not. It is however possible to overcome this limitation to some degree. In the benchmarks the memory is captured for every frame. Creating graphs from this later, enables us to spot correct values easier. Ideally at the start of the benchmark, we should see the base memory usage of the API. Should the graph rise throughout the execution of the benchmark, it indicates garbage created. Sudden drops in the graph would show times when the JVM has performed garbage collection. The memory usage after a garbage collection shows the base usage of the API. There are limitations to this however. Firstly it is not given that all the garbage was removed in the garbage collection process, so what is thought of as the base usage could also contain some garbage. Secondly there is the case where no garbage collection have taken place before increasing the object count as part of the benchmark execution. If this was the case, then the initial memory usage would not be correct. Should a garbage collection take place during the execution, we would get the correct memory usage. In the event it does not, we would not be able to get the correct base memory usage for the API. This is a limitation with the design in Java, and there is really no way to work around this. It is also worth noting that this memory overview only looks at the heap memory in the JVM. Additionally the APIs might use native buffers, which exist outside of the JVM heap. It is not possible to monitor these through the Java code. Therefore we note these limitations, and as consequence the memory data collected should be considered approximate values, and not absolute representations of the actual usage.

5.3.3 Output from benchmarks

The benchmarks produces a series of output files. The output from a benchmark consists of one text file that is intended to be read directly. This contains summarized data from the benchmark. In addition to this, several csv files are created. These contain detailed data from the benchmark, and are intended to be parsed by external applications, in order to better analyse the results.

5.3.4 Benchmark: Dynamic Geometry

This benchmark tests how well the APIs handles geometry that is dynamic, which means that the vertices are constantly changing. This might cause problems for the APIs if they try to generate static lists of the geometry.

The implementation consists of a cluster of spheres, where they have dynamically changing radii. By shrinking and expanding the radii of the spheres, their vertices changes position every frame, effectively making them dynamic. The triangle counts for each sphere vary between 58 and 218 triangles. Each sphere is rotated individually around a center point. This is to prevent the APIs from doing any optimizations on them, because they could be considered static due to non-changing positions.

If any of the APIs are trying to create static lists from the geometry, this should be evident from the results. If this happens at regular intervals, it will most likely show up as spikes in the performance. If they however try to generate static lists at every frame, it would be evident from a much lower framerate throughout the whole benchmark.

Figure 5.2 shows a screenshot from the benchmark.

5.3.5 Benchmark: Frustrum

The Frustrum benchmark tests how effective the APIs are at culling away objects that are outside of the cameras view frustrum.

The implementation consists of a cluster of cubes, where each cube rotates individually around a center point. The rotation is to prevent the APIs from doing any optimizations on them, because they could be considered static due to non-changing positions. Cubes are used for this benchmark because it is the culling that is of importance here, and not the triangle count being rendered. Therefore cubes are used, which have a low triangle count. The cluster is moved along an elliptic trajectory, which moves it inside and outside of the frustrum. This trajectory takes the cluster both

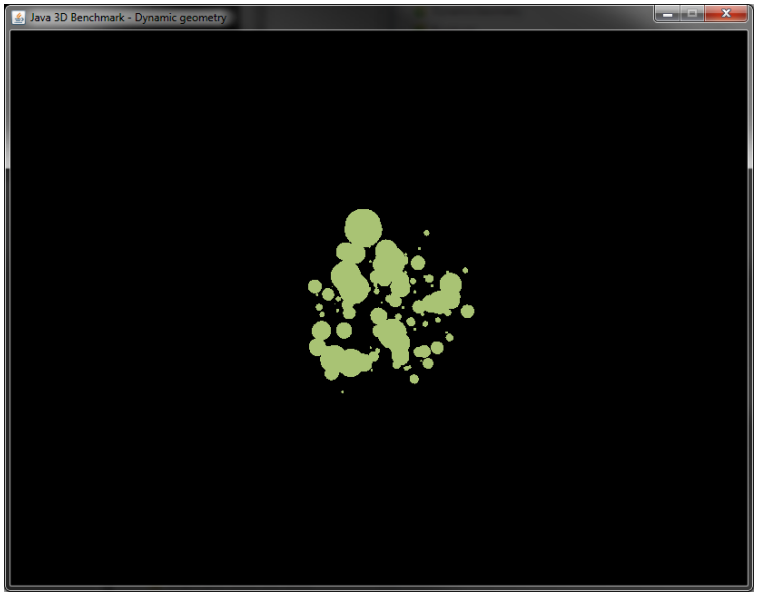


Figure 5.2: Screenshot from the benchmark *dynamic geometry*, rendering 128 spheres with randomly shrinking and expanding radii.

beyond the *far plane*, and behind the *near plane* of the frustrum. Figure 5.3 shows an illustration of the elliptic trajectory that the cluster follows.

The results should clearly indicate a difference in the performance when the whole cluster is visible, and when the cluster is partially or completely hidden outside of the view frustrum. Figure 5.4 shows several screenshots of the cluster as it moves along the elliptic trajectory.

5.3.6 Benchmark: Node stress add and removal

This benchmark tests how well the APIs handles rapid addition and removal of objects to and from the scene graph.

The implementation consists of a series of cubes that are created, and kept in memory. During the execution of the benchmark, a certain amount of these cubes are added to, or removed from the scene graph every frame. Cubes are used for this benchmark because it is the process of adding or removing that is of importance here, and not the triangle count being rendered. Therefore cubes are used, which have a low triangle count (a cube consists of 12 triangles).

The cubes that are used in the benchmark are generated at the start of each period. The same cubes are reused throughout the benchmark, meaning that cubes that have been added and later

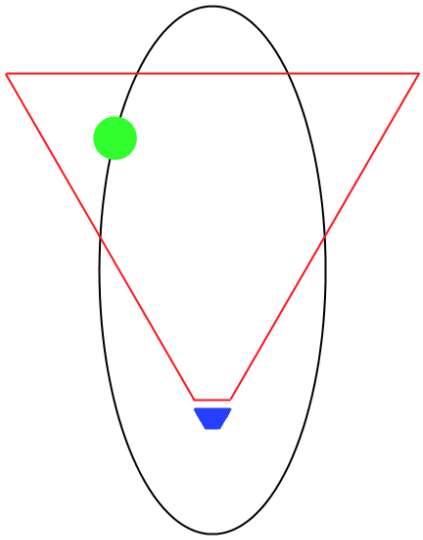


Figure 5.3: Illustration of the elliptic trajectory that moves the cluster of cubes around the cameras view frustrum. The camera is depicted in blue, the cameras view frustrum in red, the cluster of cubes in green, and the elliptic trajectory in black.

removed, might be added back to the scene graph again. This could be problematic depending on how the different APIs handle the references to the objects internally. The problem is that while the benchmark code has removed the object from the scene graph, internally the APIs might still contain references to the objects. When the object is then later added back to the scene graph, it might be at an advantage (or disadvantage) if it is still kept internally in memory. Should this be the case, the benchmark is still able to measure the performance of the add and removal operations. If the mentioned situation should arise, and the code internally in the APIs should be able to handle such a situation, it is to be viewed as a feature.

There are ways to work around the problem highlighted above, such as keeping multiple series of cubes in memory, and only adding and removing a cube once per benchmark period. This would remove the problem, however it would increase the memory usage of the benchmark, especially with higher object counts. Cubes could also be cloned during runtime, however this would only add noise to the results.

The results should indicate how well the benchmarks performs, when rapidly adding and removing nodes to and from the scene graph. As the number of cubes increases, the results will also show how well the algorithms scale. Possible spikes and stuttering should also be evident in the

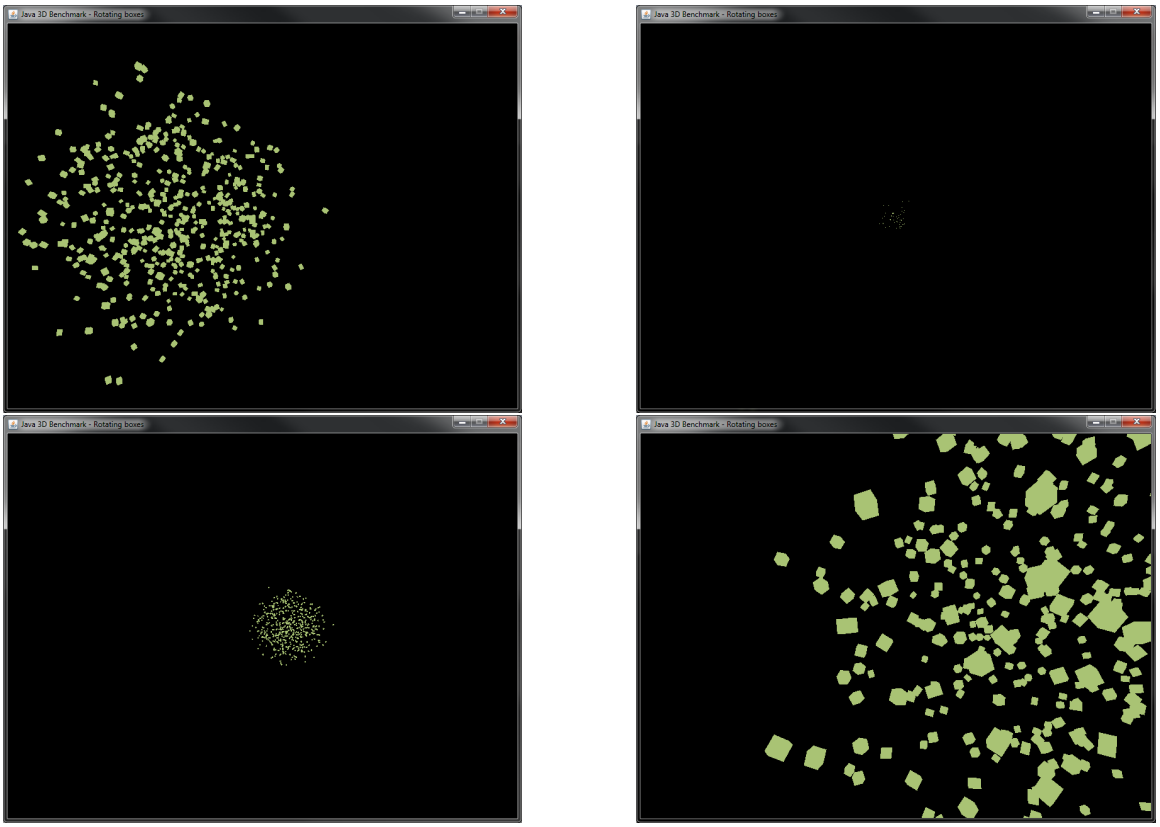


Figure 5.4: Screenshot from the benchmark *frustrum*, rendering 512 cubes. The four figures shows some positions of the cluster of rotating cubes, as it moves along an elliptic trajectory around the cameras view frustrum. In the upper left, the cluster moves into the scene, having passed the camera on the left side. In the upper right, the cluster have moved partially past the far plane, hiding almost all of the cubes outside of the frustrum. In the lower left, the cluster have re-emerged on the right side, moving towards the camera. In the lower right, the cluster is moving past the camera on the right, going past the near plane of the frustrum.

results, and are not unlikely to happen. Figure 5.5 shows a screenshot from the benchmark.

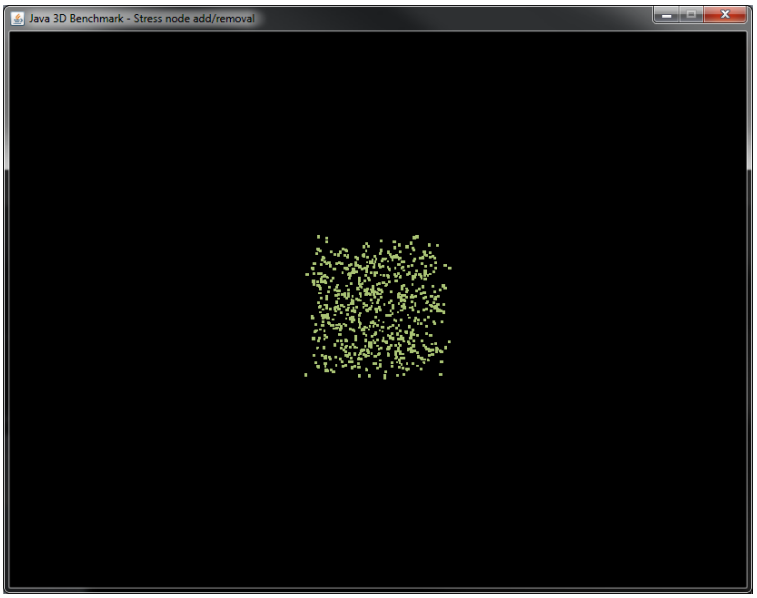


Figure 5.5: Screenshot from the benchmark *node stress add and removal*, rendering 2048 cubes that are randomly being added or removed from the scene graph.

5.3.7 Benchmark: Picking

The Picking benchmark looks at the performance in the different APIs when performing rapid picking operations.

The implementation consists of a cluster of spheres, that are individually rotating around a center point. The rotation is to prevent the APIs from doing any optimizations on them, because they could be considered static due to non-changing positions. Each sphere has a triangle count between 218 and 478. The triangle count here is higher than in the other benchmarks which also use spheres. The reason for this is that we want there to be more triangles to check for collisions with.

The picking is done by casting a series of rays out from the cameras location and into the cluster of rotating spheres. A total of 81 rays are picked into the cluster every frame, and checked for collisions. The picking operation is set to return results on the primitive level, which means that the pick results contains which triangles were picked. The results from the picking operations are iterated over, although no additional calculations are done on them. The reason for iterating over them is to prevent the APIs from doing any hidden optimizations, like only calculating the triangle

results upon the request for the results.

The results should show any limitations in the picking implementation in the APIs. By forcing the APIs to present the picking result at the primitive level, we would be able to expose any bad or naive implementations. For example if they are doing the primitive picking without taking advantage of bounding volumes. This should show up clearly in the results, due to the high triangle count and the sheer number of pick rays. Figure 5.6 shows a screenshot from the benchmark.

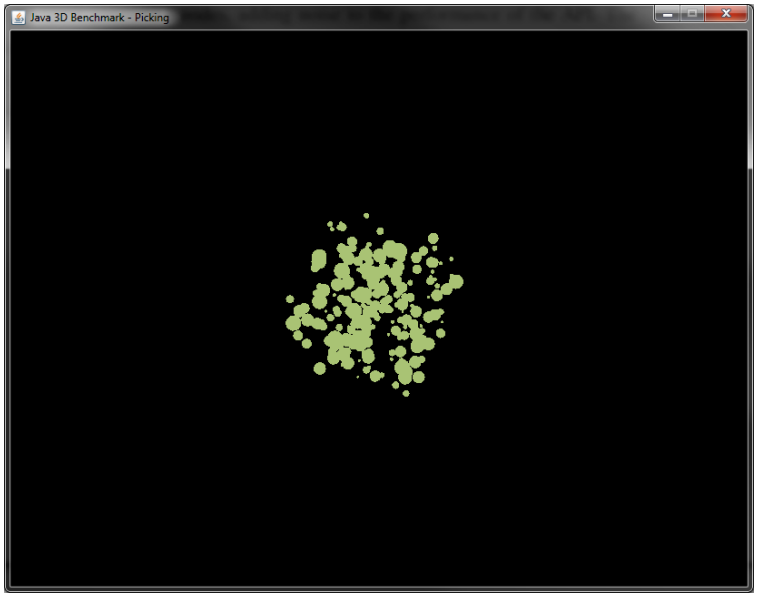


Figure 5.6: Screenshot from the benchmark *picking*, rendering 256 spheres that are randomly being rotated. At the same time 81 pick rays are being shot out from the cameras location, and into the scene every frame.

5.3.8 Benchmark: State sort

The state sort benchmark looks at how the APIs handles the rendering of many objects, which could potentially be sorted with regard to various OpenGL states.

The implementation consists of a cluster of cubes, that are individually rotating around a center point. The rotation is to prevent the APIs from doing any optimizations on them, because they could be considered static due to non-changing positions. Cubes are used for this benchmark because it is the process of sorting based on OpenGL states that is of importance here, and not the triangle count being rendered. Therefore cubes are used, which have a low triangle count (a cube consists of 12 triangles). Different states are used for the cubes, varying between the three main states in

OpenGL. These are lighting, texturing, and shaders. The total number of cubes being rendered are split between the three main states. For example with 32 cubes this would be divided by three, giving 11 lit, 11 textured and 10 shaded cubes. The square root of the number for each state decides how many permutations within each state should be created. This ensures that some cubes can be sorted together, because they have the same state. The textures used are generated at runtime, using rgb colors, with no mipmapping. The shader program consists of a simple, minimal vertex and fragment program, that changes color based on a uniform passed to it.

The results should show if the APIs are doing any efficient sorting of the objects to be rendered. If they are taking into account the states of the objects, there should be visible performance gains. Figure 5.7 shows a screenshot of the benchmark.

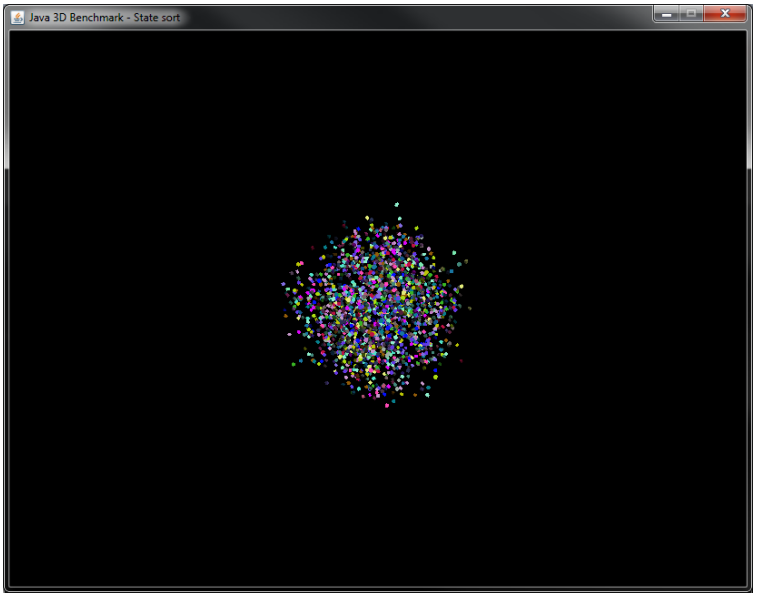


Figure 5.7: Screenshot from the benchmark *state sort*, rendering 2048 cubes that are randomly being rotated. Different states are used for the cubes, varying between variances of light, texture, and shader states.

5.3.9 Benchmark: Transparency sort

This benchmark investigates how well the different APIs are handling sorting of transparent objects.

The implementation consists of a cluster of spheres that are individually rotating around a center point. The rotation is to prevent the APIs from doing any optimizations on them, because they could be considered static due to non-changing positions. Each sphere has a triangle count between 58 and

110. Each sphere is transparent, with identical material and blending settings. Transparent objects are set to be sorted *back to front*.

The results show how efficient the APIs are at sorting transparent objects. Figure 5.8 shows a screenshot from the benchmark.

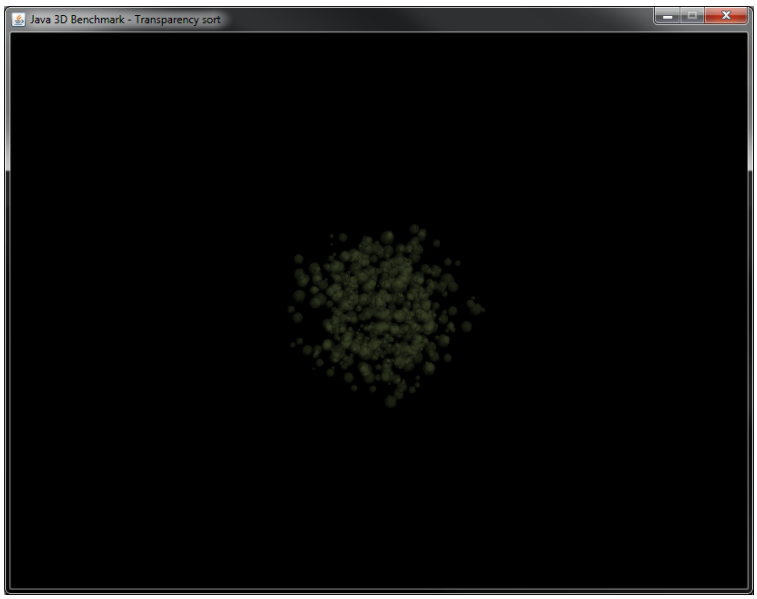


Figure 5.8: Screenshot from the benchmark *transparency sort*, rendering 512 transparent spheres that are randomly being rotated.

5.3.10 Benchmark starter

A simple GUI application was developed to make the benchmarks more accessible. It lets the user choose which benchmark to start, in addition to specifying various parameters for the benchmark. Such as duration of the warmup period, number of frames to collect data for, and which object count to stop after. Figure 5.9 shows a screenshot of this.

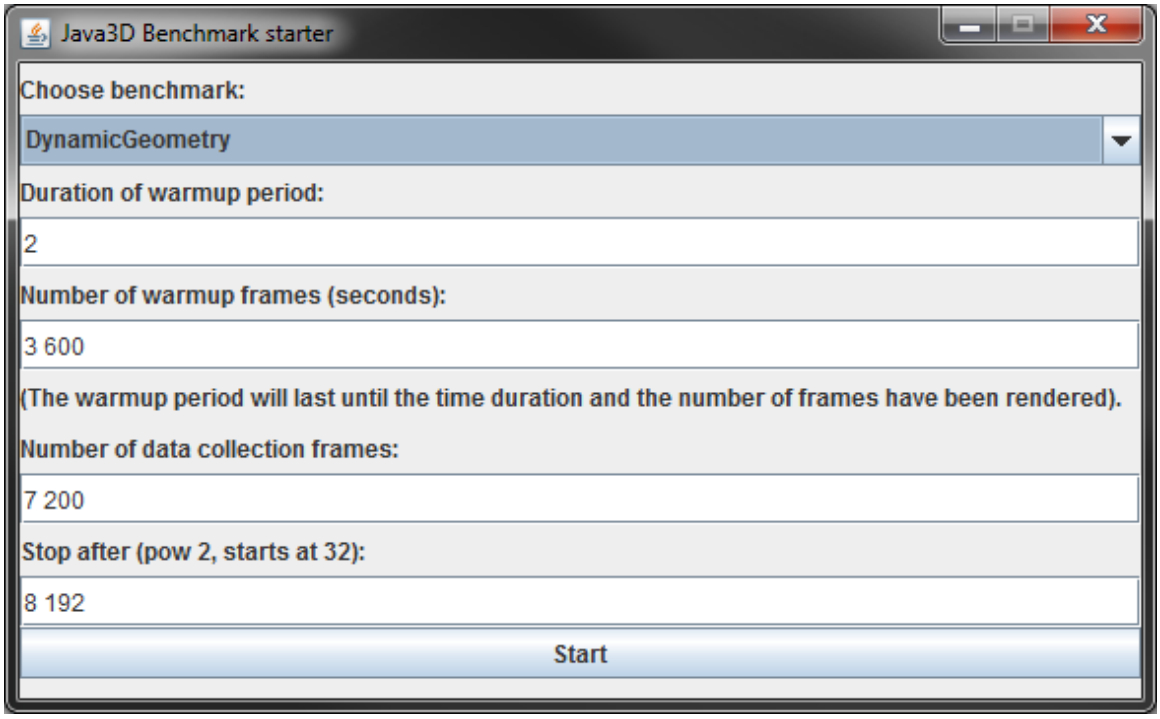


Figure 5.9: GUI for starting a benchmark, with the ability to specify various arguments.

Chapter 6

3D Scene Graph APIs Evaluation Results

This chapter shows the results from the comparison of the different APIs. The comparison is divided into to four different layers, covering different aspects of the APIs. Section 6.1 shows the results from comparing the maturity of the projects. This measures both the project management aspects and the technical infrastructure of the APIs. Section 6.2 compares the system architecture of the APIs. Section 6.3 compares the features and capabilities offered by the different projects. Section 6.4 covers the last part of the comparison, which is the results from a series of performance benchmarks. The methodology behind these comparison methods is described in greater detail in Chapter 4.

6.1 Comparing Project Maturity

The results were collected by investigating the websites for the APIs, and related information that could be found browsing the internet. Table 6.1 shows the results from investigating all the APIs. It shows the project maturity for the APIs.

We see that the results for the first question, which relates to the whether the projects use version control or not [1], show that all the projects have publicly available version control. They offer easy and convenient ways of submitting patches.

[1] Note that privately available version control were defined to be one where only the core developers have both read and write privileges to the project. A publicly available version control system should allow read access to everyone, however write access can be limited to only the core developers. There should be easy, and convenient methods to apply patches however.

Project Maturity for all APIs			
	Java 3D	**Ardor3D**	**jME3**
1. Version Control			
Does the project use version control?	Yes, public (2)	Yes, public (2)	Yes, public (2)
2. Mailing Lists and/or Discussion Boards			
Are mailing lists and/or discussion boards available?	Yes (1)	Yes (1)	Yes (1)
(In case of mailing lists present) Are mailing list archives available?	Yes (1)	Yes (1)	Yes (1)
3. Documentation			
Does the project have documentation for users?	Yes (1)	Yes (1)	Yes (1)
Does the project have documentation for developers?	Yes (1)	No (0)	Yes (1)
4. Systematic Testing			
Are release candidates available?	Yes (1)	Yes (1)	Yes (1)
Does the source code contain an automatic suite?	No (0)	No (0)	No (0)
Does the project use a defect tracking system?	Yes (1)	Yes (1)	Yes (1)
6. Portability			
Is the software portable?	Yes, limited number of platforms (1)	Yes, limited number of platforms (1)	Yes, limited number of platforms (1)
Project Maturity	**9**	**8**	**9**

Table 6.1: The Project Maturity for All APIs.

The second category looks at the presence of mailing lists and/or discussion boards. These are useful for many aspects, such as coordination within projects, discussions regarding design decisions or asking other people for help. We see here that all the projects provide this. Mailing lists have been replaced to a large degree by discussion boards in the recent years, therefore it is not surprising that only Java 3D offers mailing lists. Neither Ardor3D, nor jME3 have mailing lists, but they do have discussion boards. This is not considered a problem, because either is sufficient. Regarding the second question in this category, relating to archives, they are available in all the projects. This mainly relates to mailing lists, where an archive is something that needs to be set up. In the case of discussion boards, they rarely delete old discussions, so an archive in that sense exists in most cases. jME3 is a new start for the jME project, because of the fork of the original developers to Ardor3D [1]. As a part of this, jME3 switched to a new forum. The old forum for the previous jME3 versions were kept in an archived forum, so they still exist.

The results from the documentation category shows that all the projects have documentation for getting the users started with the APIs. However there are huge differences here regarding the amount of information available. Java 3D has very good documentation, a large amount of information exists on how to set it up, how to use it, and tutorials that teaches new users how everything works. jME3 is similar to Java 3D, and has excellent availability of information to new users. Ardor3D on the other hand has information for new users, but it pales in comparison to the other two. It has information on how to get the API working, however the information is so outdated that by just following the information it is not possible to get it to work. The information on how the API is built up, and how it works is also scarce. It is there however, but it is so superficial that it hurts the overall presentation of the project. The main source of information for code usage in Ardor3D comes from a series of code examples that demonstrate various usages of the API. While the examples in themselves are good, they severely lack presentation and documentation. There are hardly any comments in the code, so it requires the users to read and understand the code. While this can be expected to some degree in developer code, it is not good for user code. This is also a problem when it comes to documentation for developers with Ardor3D. I have chosen to give Ardor3D zero points here to illustrate that there is a problem with the documentation. Arguably this could have been done for user documentation. The only source of developer documentation in Ardor3D is the previously mentioned code examples. Apart from these there are no additional documentation about the API. I have chosen to be this harsh with Ardor3D because documentation is very important, and what they currently offer is simply not good enough. That being said, they

[1] This is described in greater detail in Chapter 3.

have the ground work done, in the form of all the code examples. If they put the time and effort into it, this could have been converted into a series of very good tutorials. When it comes to Java 3D and jME3 they have very good documentation for developers, and it is nothing in particular that is necessary to highlight.

When it comes to the systematic testing category the results are similar between the projects. All the projects have clear plans for the features that should be included in each version. They all plan, and set dates for when code should be freezed in order to keep up with planned releases. They all seem to keep an open dialogue between developers and the users, when deciding upon which features to aim for with the different releases of the software. When it comes to automatic testing suites the answer were no for all the projects. In general it seems to be a common understanding for all the APIs that it is hard to test many parts of the systems when it comes to highly visual aspects, as is with 3D scene graph APIs. The general approach is that core developers, and users of nightly builds are always testing the new features added independently, to check for errors. So it is worth noting that there is a quality assurance of the code, however there is no formal well defined framework behind it. All of the projects offer similar systems for tracking bugs and defects in the code. These systems lets the users submit bugs encountered, and optionally patches to fix them.

Regarding the portability category, the results are also here similar between the projects. All the projects support the main desktop platforms, such as Windows, Mac OS X, Linux, and also Solaris. In addition to this, both Ardor3D and jME3 supports mobile devices that are running the Android operating system.

6.1.1 Summary

The results from measuring the project maturity for the open source projects reveals that there are fairly small changes between them. The final maturity score for the projects were 9 for Java 3D, 8 for Ardor3D, and 9 for jME3. Overall the projects fulfilled most of the requirements in the framework. The maximum value is 11 points. The one thing all APIs did not have, were formal and well defined frameworks for testing code. This is in large part because it is hard to write automatic testing code for code that in many cases must be visually inspected in order to determine if there is anything wrong, or not. Instead they use a form of peer review process, where other contributors to the projects look over the new code that is added, and tests individually for defects.

There is one other aspect that should be highlighted from this comparison, which is the documentation. Documentation for Java 3D and jME3 were excellent, for both new users and developers. Ardor3D on the other hand is severely lacking documentation. The available documentation is very

scarce, and the presentation is not good. This hurts the project severely, and creates a very high entry-level to the project. On the other hand, Ardor3D has the ground work done, and if they direct attention to it, many of the poorly documented code examples could be transformed into very informative and adequate documentation.

In summary this shows that Java 3D and jME3 have the best project maturity, with identical scores. Ardor3D follows closely, with just a little lower score. While in terms of maturity values the projects are relatively close, the state of the documentation in Ardor3D is so bad that in reality the difference is much larger. The current state of the documentation in Ardor3D gives it a very high entry-level, which might discourage potential users and contributors.

6.2 Comparing System Architecture

The comparison of the system architecture of the APIs is done on the core package for each of the APIs. The reason for this was to keep the comparison equal between the APIs, because the core should only contain the core functions required to use the libraries. Adding external libraries might skew the results, which is undesirable.

The results from comparing the system architecture of the APIs were collected by using the system analysis tool Lattix [1]. This tool is able to construct a design structure matrix from analysing the static binaries from the libraries. The metrics used for the comparison is also calculated by Lattix.

6.2.1 Java 3D

Here are the results from investigating the system architecture of Java 3D. The binaries investigated are the 1.5.2 version, and were extracted from *j3dcore.jar*.

The core of Java3D features a completely flat structure. This means that the are no hierarchical division of the classes based on their functionality, but instead all the classes in the core are located directly beneath the *javax.media.j3d* package. This has complicated the analysis of its architecture.

Table 6.2 contains the results from investigating Java 3D, and Figure 6.1 shows the DSM for the core. Because of the flat structure in Java 3D, the DSM contains all the 407 classes in Java 3Ds core. This greatly reduces the ”readability” of the Figure, and while it is possible to see some of the dependencies in it, it was mainly included for completeness sake.

Results for Java 3Ds core	
System Stability	26%
System Cyclicality	74%
Hierarchical Cyclicality	74%

Table 6.2: Results from investigating the system architecture of Java 3D. The calculated metric are extracted from a DSM constructed for the core of Java 3D. The results are rounded.

The system stability for the core of Java 3D is 26%, which is a very weak value. The implication of this is that it may be hard to maintain the API, because the connectedness in the API is very deep. Because of the flat structure of Java 3D, it is not feasible to analyse the stability any further. This is because the results are unmanageable when going deeper than the top level (*javax.media.j3d*),

[1] Lattix can be found here `http://www.lattix.com/`

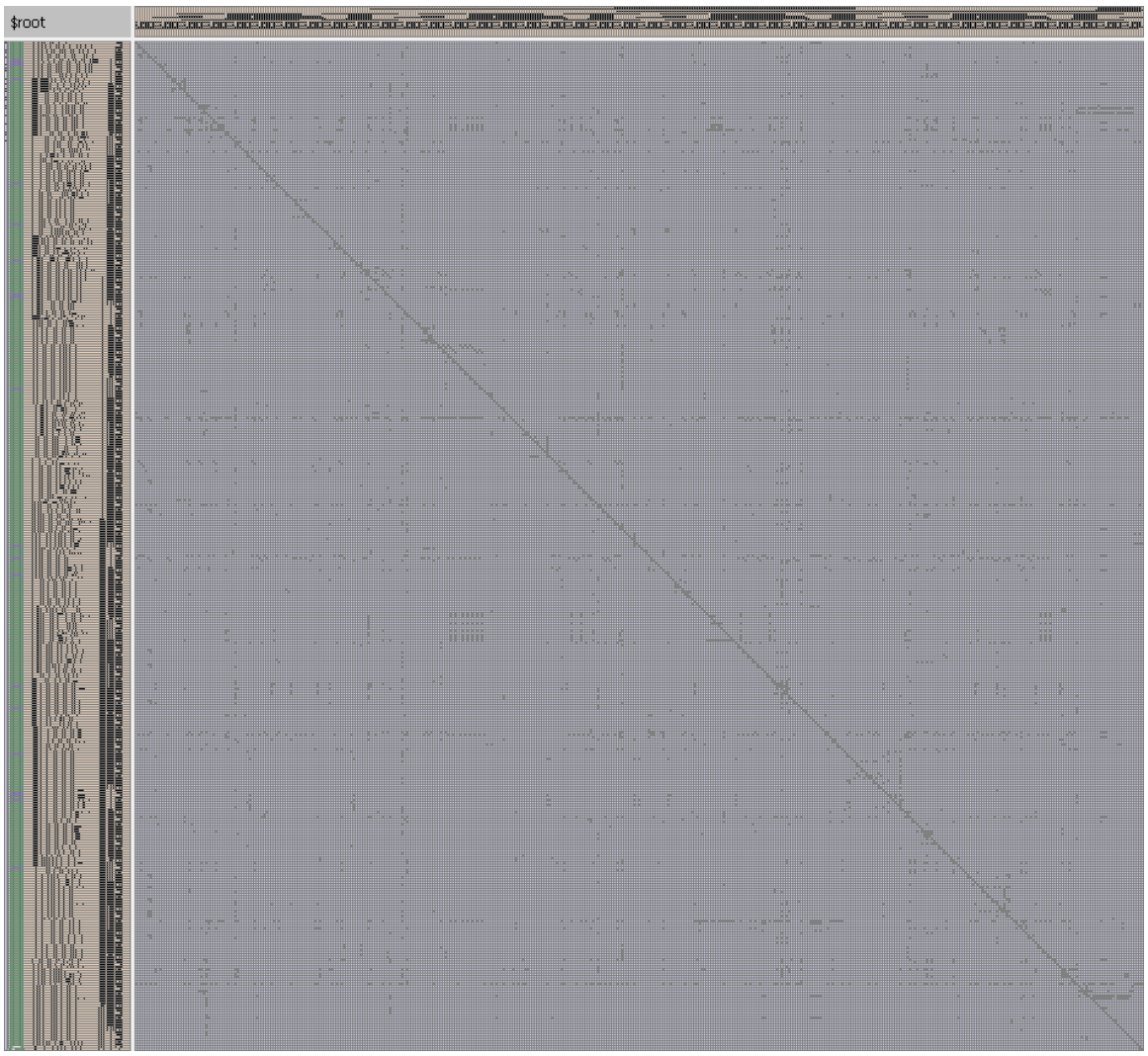


Figure 6.1: Design structure matrix created from the core of Java 3D (j3dcore.jar). Java 3D has a flat structure, therefore the DSM shows all 407 classes in the core.

due to the inability to aggregate the classes into groups. Therefore it is not possible to give any indications to which parts of the core are the most problematic.

The system cyclicality is 74%. This is bad, and it means that there is a large number of cycles formed in the core. This value implies that more than two thirds of the core is involved in problematic cycles. This means that much iterative work, and coordination is required in order to maintain the code. It might also increases the likeliness of bugs in the code. Because of the problems with the flat structure mentioned before I am unable to investigate this any further.

We note that the hierarchical cyclicality for Java 3D is the same as the system cyclicality. This is because of Java 3Ds flat structure, which means that there are no hierarchical score to add to the calculation.

The architecture of Java 3D is flat, with all the classes located at the same place hierarchically. This impacted the analysis of its architecture, and prevented a more detailed analysis of the architecture. The results show that Java 3D have a very low stability, indicating very much connectivity in the core. The system cyclicality shows that more than two thirds of the core is involved in such cycles. The hierarchical cyclicality is identical to the system cyclicality, because there is no hierarchical structure in the core.

6.2.2 jMonkeyEngine 3

Here are the results from investigating the system architecture of jMonkeyEngine3. The binaries investigated have been downloaded from the jMonkeyEngine3 website [1], and were extracted from *jME3-core.jar*.

The core of jME3 consists of three main packages, *checkers.quals*, *com.jme3* and *jme3tools*. The *com.jme3* contains the core parts of the API, while the other two just provide some annotations and tools. Within the core you find core packages such as the scene graph data structure, renderer and bounding volumes. In addition to this, the core of jME3 contains many other packages, such as water, network, and shadow. These are packages that does not offer core functionality, and is only required in projects that have special need of them. Including such packages in the core can be problematic, and is generally undesirable. This is because it might add additional coupling and connectedness within the core, and it overall clutters the core. It is also makes the maintainability harder, because any changes to utilities such as water, requires a complete rebuild of the whole core. This is a bad design choice of the core, and additional utilities and features should be included in

[1] Downloaded the latest nightly build, on the 11.01.2012. Available here: `http://jmonkeyengine.com/nightly/`

an additional library, instead of in the core itself. The DSM in Figure 6.2 shows that some of these packages could be extracted from the core with little effort. This is because they have few other packages that actually use, and depend upon them. For example the *network*, *shadow*, and *water* packages are not required by any of the other packages in the core.

Table 6.3 contains the results from investigating jME3. Table 6.4 shows the system stability for each of the sub-packages within the core, and Figure 6.2 shows the DSM for the core.

Results for jME3s core	
System Stability	78%
System Cyclicality	43%
Hierarchical Cyclicality	89%

Table 6.3: Results from investigating the system architecture of jME3. The calculated metrics are extracted from a DSM constructed for the core of jME3. The results are rounded.

We see that the system stability for the core were 78%. This is a very good value when considering the core as a whole. The metric shows that when looking at the core in a broad sense, it has a good stability. When investigating the sub-packages in more detail, we see that there are varying stability for the various packages. We see that many of the packages that were previously highlighted as additional utilities and features have a very good stability. This is good because these packages should not affect the critical parts of the core. For example we have the *shadow* package, who have a stability of 99%. It is of more concern that many of the critical parts of the core have a much lower stability. For example the *renderer* have a stability of 42%, and the *bounding* have a stability of 29%. The high stability of the packages considered additional utilities have skewed the total system stability score for jME3. If these packages were removed from the results, the overall system stability for the core would be much lower.

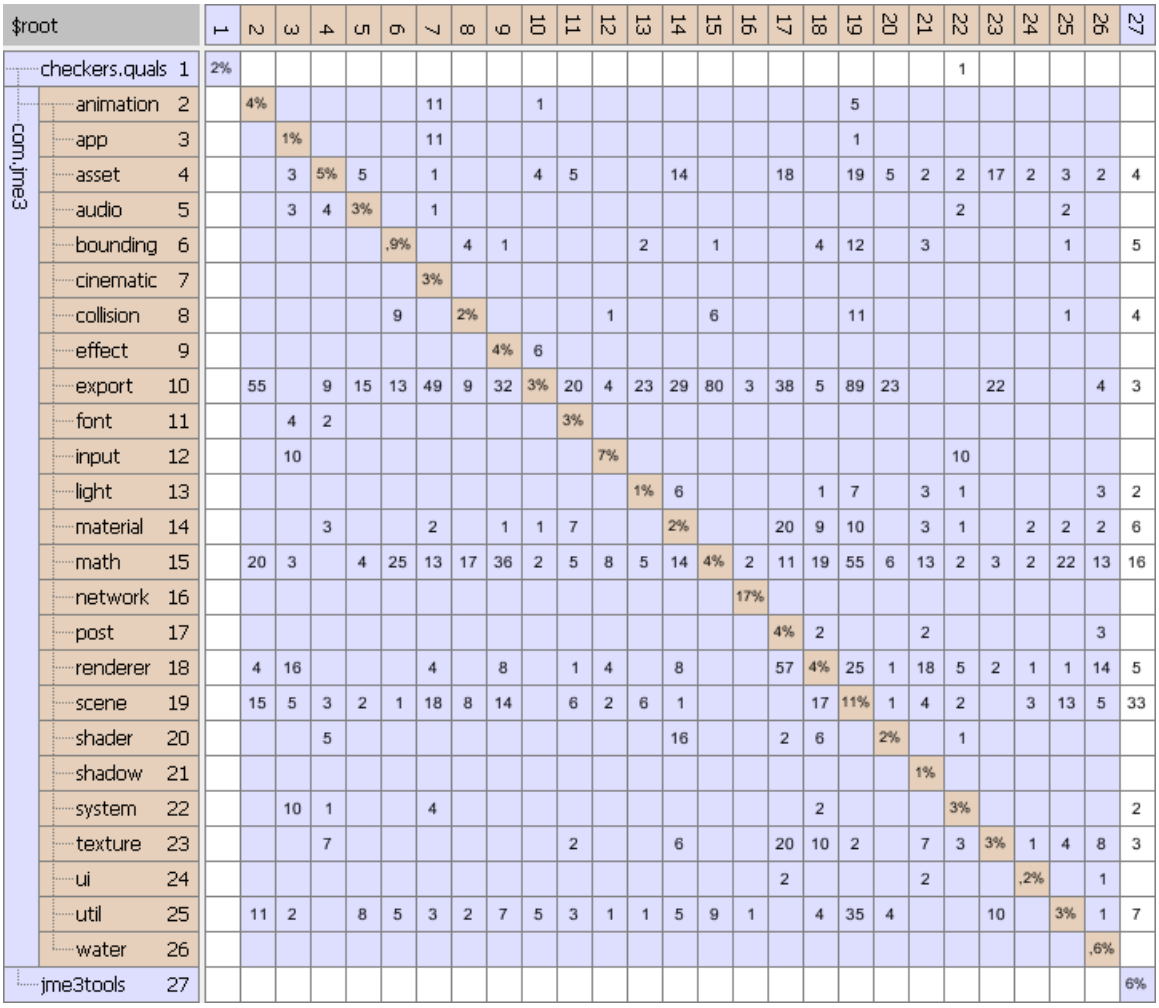
The system cyclicality for the core is 43%. This means that almost half of the core is involved in problematic cycles, which is not good. Looking at the sub-packages in more detail, we see that most packages are involved in cycles to some degree. We see that many of the packages that are critical to the core functionality are involved, such as *scene* with 3.9% and *renderer* with 3.0%.

The hierarchical cyclicality for the core is approximately 89%. This value is generally expected to be higher than the system cyclicality, but in this case it is more than twice as large. This is a very high value, and it means that most of the core is involved in cycles. Investigating the packages highlighted when looking at the system cyclicality, shows that the values were increased.

The architecture of jME3s core seems cluttered, and contains much functionality that does not

Detailed results for sub-packages beneath com.jme3 in jME3s core			
Sub-packages	**System Stability**	**System Cyclicality**	**Hierarchical Cyclicality**
com.jme3.animation	97,5%	1,3%	1,1%
com.jme3.app	97,7%	0,6%	0,2%
com.jme3.asset	69,4%	1,9%	4,9%
com.jme3.audio	39,2%	1,7%	3,2%
com.jme3.bounding	29,9%	0,9%	0,9%
com.jme3.cinematic	98,6%	1,9%	3,2%
com.jme3.collision	43,9%	1,5%	2,1%
com.jme3.effect	98,5%	0,9%	2,1%
com.jme3.export	77,4%	2,1%	1,9%
com.jme3.font	40,7%	1,9%	2,8%
com.jme3.input	93,7%	0,6%	6,8%
com.jme3.light	29,9%5	1,3%	1,3%
com.jme3.material	43,9%	1,5%	2,1%
com.jme3.math	50,8%	3,4%	4,3%
com.jme3.network	96,6%	8,6%	16,7%
com.jme3.post	96,0%	0,6%	4,1%
com.jme3.renderer	42,3%	3,0%	3,6%
com.jme3.scene	74,3%	3,9%	10,7%
com.jme3.shader	43,9%	1,3%	2,1%
com.jme3.shadow	99,8%	0,0%	1,1%
com.jme3.system	86,3%	1,1%	2,8%
com.jme3.texture	67,6%	1,3%	2,8%
com.jme3.ui	95,5%	0,0%	0,2%
com.jme3.util	64,8%	0,6%	3,4%
com.jme3.water	99,9%	0,0%	0,6%

Table 6.4: Detailed results from investigating the sub-packages of com.jme3 in the core of jME3. The calculated metrics are extracted from a DSM constructed from the core of jME3. The results are rounded.



$root	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
checkers.quals 1	2%																					1					
com.jme3																											
animation 2		4%					11			1									5								
app 3			1%				11												1								
asset 4			3	5%	5		1			4	5			14			18		19	5	2	2	17	2	3	2	4
audio 5			3	4	3%		1															2			2		
bounding 6						,9%		4	1				2		1			4	12		3				1		5
cinematic 7							3%																				
collision 8						9		2%				1			6				11						1		4
effect 9									4%	6																	
export 10		55		9	15	13	49	9	32	3%	20	4	23	29	80	3	38	5	89	23			22			4	3
font 11			4	2							3%																
input 12			10									7%										10					
light 13													1%	6				1	7		3	1				3	2
material 14				3			2		1	1	7			2%			20	9	10		3	1		2	2	2	6
math 15		20	3		4	25	13	17	36	2	5	8	5	14	4%	2	11	19	55	6	13	2	3	2	22	13	16
network 16																17%											
post 17																	4%	2			2					3	
renderer 18		4	16				4		8		1	4		8			57	4%	25	1	18	5	2	1	1	14	5
scene 19		15	5	3	2	1	18	8	14		6	2	6	1				17	11%	1	4	2		3	13	5	33
shader 20				5										16			2	6		2%		1					
shadow 21																					1%						
system 22			10	1			4											2				3%					2
texture 23				7							2			6			20	10	2		7	3	3%	1	4	8	3
ui 24																	2				2			,2%		1	
util 25		11	2		8	5	3	2	7	5	3	1	1	5	9	1		4	35	4			10		3%	1	7
water 26																										,6%	
jme3tools 27																											6%

Figure 6.2: Design structure matrix created from the core of jME3 (jME3-core.jar). The main classes with functionality in the core is located beneath the com.jme3 package. The two other packages contains annotations and tools.

belong in the core of a 3D scene graph API. The system stability showed initially a good value, that indicated that the core was relatively stable. Upon further investigation, the results for the stability were shown to be skewed by some of the sub-packages in the core. These packages does not offer core functionality, and should be extracted into an additional libraries with utilities and additional functionalities. These packages had a much better stability than the more critical core packages. By ignoring the packages that does not belong in the core, the stability value were lowered significantly. This indicates that the stability is not so good. The high percentages of both the system and the hierarchical cyclicality, also show that there is not much separation, nor abstraction layers between the different parts of the core. This greatly increases the likelihood of bugs in the code, and complicates the maintainability of the system.

6.2.3 Ardor3D

Here are the results from investigating the system architecture of Ardor3D. The binaries investigated have been created from the latest sources found in the SVN [1], and were extracted from *ardor3d-core.jar*.

The core of Ardor3D features a hierarchical structure, where the core is hierarchically separated into sub-packages based on functionality. The topmost package is *com.ardor3d*. Table 6.5 shows the calculated metrics for the core of the API. The calculated metrics for all the sub-packages that constitute the core of Ardor3D are shown in Table 6.6. The DSM for the core is shown in Figure 6.3.

Results for Ardor3Ds core	
System Stability	67%
System Cyclicality	38%
Hierarchical Cyclicality	93%

Table 6.5: Results from investigating the system architecture of Ardor3D. The calculated metrics are calculated from a DSM constructed for the core of Ardor3D. The results are rounded.

The system stability for the core of Ardor3D is 67%. This is a good value, considering it describes the stability of the whole core. When looking at the stability for each sub-package in more detail, it is clear that most parts of the core are very stable. The packages with the lowest stability are the *bounding*, *image*, *light* and *renderer* packages. The package with the lowest stability in

[1] Ardor3D was built from the latest sources found in their SVN repositories. This was done in January 2012, using Revision 1786.

$root		1	2	3	4	5	6	7	8	9	10	11	12	13
com.ardor3d	annotation 1	1%		5		2					1		3	4
	bounding 2		2%				3		3	3	10			9
	framework 3			2%		9				7				
	image 4			1	6%	3				22	4		2	20
	input 5					11%								
	intersection 6		7	1			3%		1		3			
	light 7							,9%		4				
	math 8		30	1	14	8	12	9	14%	72	100	10	9	43
	renderer 9		3	4	2	2				19%	61	1	11	74
	scenegraph 10		15			1	10			55	17%	3	11	91
	spline 11										3	,9%		
	ui.text 12												,7%	1
	util 13		12	3	29	2	1	10	64	83	134		9	24%

Figure 6.3: Design structure matrix created from the core of Ardor3D (ardor3d-core.jar). The hierarchically grouped sub-packages beneath com.ardor3d offer various core functionality.

Detailed results for sub-packages beneath com.ardor3d in Ardor3Ds core			
Sub-packages	**System Stability**	**System Cyclicality**	**Hierarchical Cyclicality**
com.ardor3d.annotation	73,4%	0,0%	0,2%
com.ardor3d.bounding	40,6%	2,1%	2,3%
com.ardor3d.framework	88,0%	0,7%	1,6%
com.ardor3d.image	41,9%	2,1%	2,1%
com.ardor3d.input	98,1%	0,0%	10,6%
com.ardor3d.intersection	69,0%	1,4%	3,4%
com.ardor3d.light	50,5%	0,7%	0,9%
com.ardor3d.math	66,2%	8,3%	13,5%
com.ardor3d.renderer	50,7%	12,6%	18,6%
com.ardor3d.scenegraph	76,7%	3,9%	16,5%
com.ardor3d.spline	99,4%	0,0%	0,9%
com.ardor3d.ui.text	99,3%	0,0%	0,7%
com.ardor3d.util	64,8%	6,0%	21,8%

Table 6.6: Detailed results from investigating the sub-packages of com.ardor3d in the core of Ardor3D. The calculated metrics are extracted from a DSM constructed from the core of Ardor3D. The results are rounded.

the core, were the *bounding* package, with a stability of 40%. These are important packages, that should be improved. Apart from these packages, most of the other packages have a relatively good stability.

The system cyclicality for the core is 38%. When investigating the sub-packages in more detail, it is clear that most packages are involved in cycles to some degree. Some of the most important packages that offer core functionality are involved, such as the *renderer* with 12.6%, and the *scenegraph* with 3.9%. Other packages with high degrees of cyclicality are *util* with 6%, *math* with 8.3%, and *bounding* with 2.1%.

The hierarchical cyclicality is 93%, which is more than twice as large as the system cyclicality. This is a very high value, that indicates that most parts of the core is involved in problematic hierarchical loops. The same packages that were highlighted when examining the system cyclicality, have also increased greatly here. The *scenegraph* with a system cyclicality of 3.9%, had a hierarchical cyclicality of 16.5%. Other parts, such as the *bounding* package did not increase as much, with only an increase from 2.1% to 2.3%. This shows that the cyclicality problems are not affecting the whole core, but only parts of it. They are of concern however, since they are concentrated around some of the more important core functionalities, such as the renderer and the scene graph data structure.

The architecture of Ardor3D appears very clean, and the core contains the required functionality. The system stability showed a very good value for the core. There were relatively high percentages

for both the system and the hierarchical cyclicality. This shows that there is not so much separation, nor abstraction between the different parts of the core. Especially the hierarchical cyclicality were high, and greatly affected important packages such as the renderer and the scene graph data structure. In summary, the structure is clean, and the metrics show that the stability of the system is good. There are high degrees of cyclicality within the core however, and especially the cycles that goes across the hierarchical structure can be problematic.

6.2.4 Summary

Here is a summary comparison of the results from investigating the system architecture of the different APIs. Table 6.7 shows a comparison of the metrics collected for the cores of all APIs.

Comparison of results from system architecture analysis			
	Java 3D	jMonkeyEngine 3	Ardor3D
System Stability	26%	* 78%	67%
System Cyclicality	74%	* 43%	38%
Hierarchical Cyclicality	74%	* 89%	93%

Table 6.7: Comparison of the results from investigating the system architecture of the different APIs. The metrics are calculated from design structure matrices, constructed from the core of each API. The results are rounded.

The core of Java 3D features a completely flat structure, with all the classes located at the same place hierarchically. This severely complicated the analysis of Java 3D, and prevented further investigation of many of the identified problems. jMonkeyEngine3s core contained many sub-packages with different functionality. Many of these sub-packages offered functionality that extends well beyond what should be put directly in the core. Some of these functionalities include effects such as water and shadows, and other functionalities such as networking or animation. This clutters the core, and should be removed. This also skewed some of the results for jME3. The core of Ardor3D were well organised hierarchically, and it contained functionality that is required to be in the core.

Looking at the system stability of the APIs, the results show that jME3 have the highest stability rating. Upon further investigation, it was identified that the results for jME3 were being skewed by some of the sub-packages that does not belong in the core. The sub-packages that contain the core functionality had much lower values, and therefore in reality the stability of jME3 is much lower. The result for jME3 is therefore marked by a asterix (”*”) in the table to indicate this. Ardor3D had a very good stability value, although when investigating the sub-packages, some of the more

important parts of the core were shown to have a little lower values than the others. The stability for Java 3D is alarmingly low. The reason for the low value is unclear, as it was not possible to investigate the results for Java 3D any further.

The system cyclicality shows that Ardor3D has the least amount of system cycles. jME3 has a little higher value, with a difference of approximately 6%. Java 3D on the other hand had a much higher value, where almost two thirds of the core is involved in such problematic cycles.

The hierarchical cyclicality for Java 3D is identical to the system cyclicality. This is because there is no hierarchical structure in its core, and therefore there is no change in the value. As a result of this, Java 3D has the lowest value. The value for Java 3D however is meaningless, because it tells us nothing. Looking at the other two APIs, we see that both have an alarmingly high degree of hierarchical cyclicality. This means that most of their cores are involved in such hierarchical cycles.

In summary, the core of Java 3D has a completely flat structure, with all the classes located within the same hierarchical location. jME3 has a cluttered core, with many classes that offer functionality far outside of what is required in the core of 3D scene graph APIs. Ardor3D had the cleanest and most structured core, with a well organized hierarchical structure, that offers only the required functionality. The calculated metrics show that Ardor3D have the best stability, and the least amount of problematic cycles of the three. However most of Ardor3Ds core is involved in hierarchical cycles. The results for jME3 were skewed, because the packages that does not belong in the core gave it a better value, that did not necessarily reflect the real state of the core. This was evident when investigating the sub-packages of the core in greater detail. This does not mean that the values are necessarily very bad, but rather the values are lower than what is shown. Especially the system stability is much lower than what is shown. The worst of the three APIs were Java 3D, who have a alarmingly low stability. The system cyclicality is also very high, including more than two thirds of the API. Due to the flat structure in Java 3D, it was not possible to investigate the cause of the problems with its architecture.

6.3 Comparison of Features and Capabilities

This section contains the results from comparing the features and capabilities. The information in the results were collected by inspecting both the documentation for the various APIs, the source code, API web site as well as browsing their forums and mailing lists where necessary.

In the following the tables, a *yes* to the question is marked by an *X* in the table, this means that the feature is supported. Similarly a *no* is marked by a -, and indicates that the feature is not supported. In the event that more information is necessary to further elaborate on the results, additional information is given.

Supported platforms			
	Java 3D	Ardor3D	jME3
Does it work on Windows?	X	X	X
Does it work on Linux?	X	X	X
Does it work on Mac OS X?	X	X	X
Does it work on Android devices?	-	X	X

Table 6.8: Supported Platforms.

The results regarding which platforms the APIs support is given in Table 6.8. We see here that all APIs support the majory platforms. Both Ardor3D and jME3 supports Android devices. Java 3D does not support Android, which is not surprising, considering that it was released around 2008, and the last stable version of Java 3D was released in 2008.

The results regarding rendering aspects in the different APIs is given in Table 6.9. We see that all APIs support OpenGL, however only Java 3D supports Direct3D. Both Java 3D and Ardor3D supports multiple bindings to the supported graphic libraries, whereas jME3 only supports one. The reason jME3 chose to only support one is because they would rather focus solely on supporting one binding well, instead of support multiple, but not as effective. This was based on previous experience, where supporting multiple bindings were problematic due to one being updated and the other becoming out of date.

We see that all APIs support shaders, and the language GLSL. Only Java 3D however supports Nvidias Cg shader langauge. Microsofts HLSL only works with Direct3D, so not surprisingly neither Ardor3D nor jME3 suppots it. Java 3D does not support HLSL, even though it supports Direct3D. We see that both Ardor3D and jME3 supports multi-pass rendering, full screen pre/post-processing effects, as well as the use of FBOs. These are techniques that are necessary for producing

Rendering			
	Java 3D	Ardor3D	jME3
Does it support OpenGL?	X	X	X
Does it support Direct3D?	X	-	-
Does it support multiple bindings to the supported graphics libraries?	X	X	-
Does it support shaders?	X	X	X
Does it support GLSL?	X	X	X
Does it support Cg?	X	-	-
Does it support HLSL?	-	-	-
Does it support multi-pass rendering?	-	X	X
Does it support frame buffer objects (FBO)?	-	X	X
Does it support full screen pre/post-processing effects?	-	X	X
Does it support stereoscopic rendering?	X	X	X
Does it support multiple canvas (example: CAVE)?	X	X	X

Table 6.9: Rendering

many visual effects. The fact that Java 3D does not support this, completely removes the possibility to produce any such effects.

True stereoscopic rendering (quad buffering) are supported in all the APIs. They also support rendering to multiple canvas, which can be used for creating virtual environments such as CAVEs.

Scene graph			
	Java 3D	Ardor3D	jME3
Does the scene graph support export/import?	X	X	X
Are attributes inherited hierarchically (minus transforms)?	-	X	X
Does it support picking?	X	X	X

Table 6.10: Scene graph.

Table 6.10 shows the results for features regarding the scene graph. We see that all the APIs support exporting and importing of the scene graph. In Ardor3D and jME3 attributes are inherited from parents, Java 3D on the other hand does not inherit attributes hierarchically. Note that transforms are not considered as part of the attributes, however transforms are inherited hierarchically in all APIs. They all support picking operations.

The results regarding model loaders are shown in Table 6.11. We see here that only Java 3D supports VRML. Both Java 3D and Ardor3D support COLLADA. jME3 does not officially support

Model loaders			
	Java 3D	Ardor3D	jME3
Does it have a model loader for VRML?	X	-	-
Does it have a model loader for COLLADA?	X	X	-
Can you load key frame animation through one of the model loaders?	X	X	-
Can you load skeletal animation through one of the model loaders?	-	X	X

Table 6.11: Model loaders.

COLLADA, but there was a COLLADA loader under development by some contributors. The implementation works to some degree, but the contributors stopped the development, and the project seems to be dead. It is worth noting that jME3 has its own binary model format for storing models (.j3o). The model format when loading models in jME3 are preferred to be in Ogre XML.

Both Java 3D and Ardor3D supports loading key frame animation, however this is not possible in jME3. Skeletal animations are possible to load from model loaders in Ardor3D and jME3, but not in Java 3D.

Additional features			
	Java 3D	Ardor3D	jME3
Does it have fixed function functionality (can be emulated)?	X	X	X
Does it support compression of textures?	-	X	X
Does nodes have some kind of callback functionality?	X	X	X
Does it have built in standard geometry?	X	X	X
Does it optimize static geometry?	X	X	X
Does it support tracking devices?	X	-	-
Does it have implemented different navigation styles?	-	X	X

Table 6.12: Additional features.

Table 6.12 shows results for additional features. We see that all APIs support fixed function functionality. jME3 which is completely shader based emulates the FFP through shaders with similar behavior. Both Ardor3D and jME3 supports texture compression, through formats like DDS (DirectDraw Sufrace). Java 3D does not support texture compression. All the APIs offer ways to add callback functionality to nodes in the scene graph. Standard geometry is supported in all APIs.

Optimizations for static geometry is done in all APIs. Only Java 3D supports tracking devices, although we note that adding such support for the other APIs is not a huge task. Java 3D does not have an built in navigation styles, but the other two offer some different navigation styles.

Utilities			
	Java 3D	Ardor3D	jME3
Can you optimize the geometry?	X	X	X
Can you generate normals?	X	X	-
Does it have a texture loader?	-	X	X
Does it have integration with a physics system?	-	-	X
Does it have integration with a networking system?	-	-	X
Does it have integration with a audio system?	X	-	X
Does it have an animation system?	-	X	X
Does it have a system for 2D user interfaces?	-	X	X
Does it have a particle system?	-	X	X
Does it have a shadow system?	-	X	X
Does it have a terrain generation system?	-	X	X
Does it have a water system?	-	X	X

Table 6.13: Utilities.

The results for additional utilities offered by the APIs are shown in Table 6.13. We see here that all APIs offer utilities for optimizing the geometry, such as triangle stripifiers. Both Java 3D and Ardor3D offer tools for generating normals, and Ardor3D also offer tools for generating normal maps. jME3 does not have any utility for this. Java 3D does not have any texture loader, whereas the other two have.

When it comes to integration with a physics system, only jME3 has built in integration. This is by using either the native Bullet [1] physics library (written in C), or the Java port, jBullet [2]. Neither Java 3D nor Ardor3D have integration with a physics library. Unofficially however, physic libraries have been integrated in applications that use the two other APIs. So while it is not officially integrated, it is absolutely plausible to implement physics libraries in the other two APIs with relative easy.

Only jME3 offers a networking system, through the SpiderMonkey utility. Both Java 3D and jME3 have integration with audio systems, whereas Ardor3D have only planned to do it in a future release (no specifics). Java 3D does not have any system for 2D user interfaces, however both

[1] Bullet can be found here: `http://bulletphysics.org/wordpress/`
[2] jBullet can be found here: `http://jbullet.advel.cz/`

Ardor3D and jME3 has good systems for 2D user interfaces. Ardor3D uses their own user interface system, whereas jME3 uses nifty-gui [1]. Regarding particle, shadow, terrain, and water systems, Java 3D offers neither, while both Ardor3D and jME3 do.

Java specific features			
	Java 3D	Ardor3D	jME3
Does it support Swing?	X	X	X
Does it support SWT?	-	X	-
Does it have a standalone component?	X	X	X
Does it support fullscreen exclusive mode?	X	X	X

Table 6.14: Java specific features.

Table 6.14 shows results that are specific to Java features. Here we see that most features are supported by all the APIs. We see however that SWT is only supported by Ardor3D.

6.3.1 Summary

The results from the comparison of features show that most of the features investigated are offered by jME3 and Ardor3D. Many are also offered by Java 3D, however some of the more advanced rendering features are lacking.

The results show that all the major platforms are supported by all APIs, however Java 3D does not support Android. Regarding rendering capabilities all APIs support the most important languages, which are OpenGL and GLSL. Java 3D additionally supports Microsofts Direct3D, and Nvidias shader language Cg. Java 3D does not support advanced rendering features, such as multi-pass rendering, FBOs, nor pre/post-processing effects. These are features which are necessary in order to produce many visual effects. These are supported in both Ardor3D and jME3. All APIs support true stereoscopic rendering (quad buffering), as well as multiple canvas. The model format COLLADA is not supportd by jME3, but is by Java 3D and Ardor3D. VRML is only supported by Java 3D. The results for additional features are fairly similar, however Java 3D does not support texture compression. Neither Ardor3D nor jME3 supports tracking devices. Looking at additional utilities the main differences are, that jME3 does not support utilities for generating normals, and Ardor3D does not have any integration with audio systems. Apart from this, utilities between Ardor3D and jME3 are fairly similar. Additionally Java 3D does not have support for many of the

[1] nifty-gui can be found here: `http://sourceforge.net/projects/nifty-gui/`

various systems, such as terrain generation, water, and particles. Regarding the Java specific features in this comparison, the main difference was that SWT was only supported by Ardor3D. Aparat from that they offered the same features.

6.4 Comparison of System Performance

The environmental configuration used for these performance benchmarks are described in Chapter 5.3, Section 5.1.3. The csv files produced by the benchmarks were imported into Microsoft Excel [1], where they were formatted and plotted into graphs.

The benchmarks were posted on the JGO-forum [2], where they were discussed and some improvements to the code were made.

6.4.1 Dynamic Geometry

The *dynamic geometry* benchmark consists of a set of rotating spheres, which have their radius shrunk and expanded during the test. This results in their vertices constantly changing, which makes the geometry dynamic. This should give trouble if the APIs should try to create static lists of the geometry, before sending it to the GPU. The results indicate that there were some problems with one of the APIs, which seem to perform some time intensive operations at regular time intervals. This resulted in large spikes. The execution of this benchmark were run from 32 spheres to 8192 spheres.

The total execution time the APIs used when running this benchmark is shown in Figure 6.4. We see that of the three APIs, jME3 were much faster than the other two. jME3 were 9 minutes faster than Java 3D at 8192 spheres. Java 3D was a little faster than Ardor3D, who used 3.5 minutes longer time than Java 3D. Figure 6.5 shows the average fps for each of the APIs. We can here see the same differences, with jME3 being the fastest. The difference in average fps between Ardor3D and Java 3D is of only 0.4 fps.

When studying the time used rendering each frame, we can see that the APIs have relative stable tpfs. This can be seen in Figure 6.6. What is of interest here however is that the tpf for Ardor3D spikes to much higher values than its average at specific time intervals. At the most the tpf spikes from an average of 66 milliseconds, and up to 320 milliseconds. Apart from the spikes, Ardor3D has a tpf that is a little higher than Java 3D. We see that the tpf for jME3 is the fastest, with a difference in the average tpf with Ardor3D of 26 milliseconds at 2048 spheres. Figure 6.7 shows the total number of spikes throughout the benchmark. We see here that there are a few spikes for both Java 3D and jME3 in the start, when the benchmark has low object counts. At higher object counts there are not registered any spikes for these APIs. Therefore they are of no real concern,

[1] More information here: `http://office.microsoft.com/en-us/`

[2] `http://www.java-gaming.org/topics/benchmarks-for-3d-scene-graph-apis/26147/view.html`

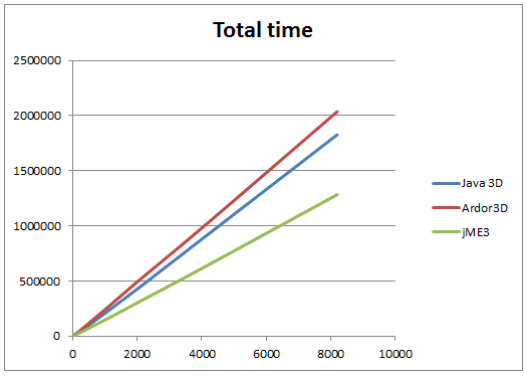


Figure 6.4: The total execution time for the *dynamic geometry* benchmark. Showing results for Java 3D, Ardor3D and jME3. The time is shown in milliseconds.

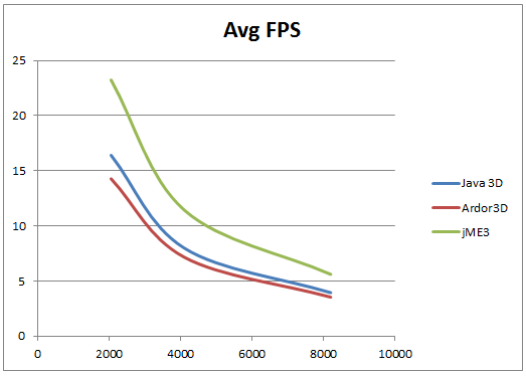


Figure 6.5: The average fps for the *dynamic geometry* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows the average fps from the range of 2048 to 8192.

since it is the values at higher object counts that are of the most importance. Ardor3D on the other hand shows that it has some spikes throughout the whole benchmark. The amount of spikes seems to only increase with higher object counts, showing an almost linear increase for object counts higher than 2048.

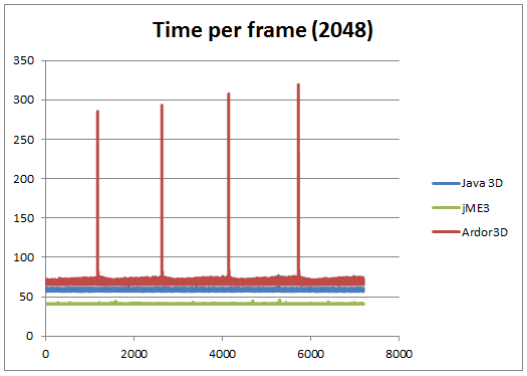


Figure 6.6: The time per frame for the *dynamic geometry* benchmark. Showing results for jME3. The graph only shows results when rendering 2048 objects. The time is shown in milliseconds.

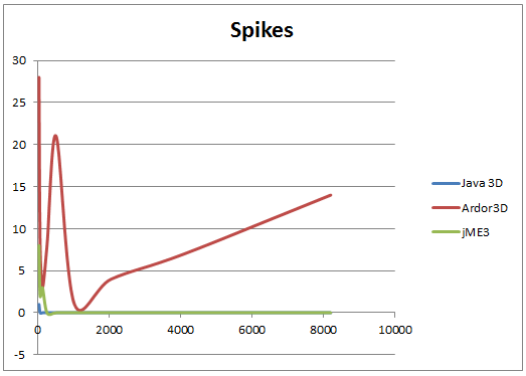


Figure 6.7: The number of spikes in the time per frame for the *dynamic geometry* benchmark. Showing results for Java 3D, Ardor3D and jME3. A spike is given with an increase in tpf equal to, or greater than 40 percent.

Figure 6.8 shows a memory overview for the APIs, and Figure 6.9 shows a overview just for Java 3D. Looking at the memory overview for Ardor3D, we see that the base memory usage seems to rise throughout the benchmark. At certain intervals the JVM performs garbage collection, which leads to a drop in the base usage. These drops in memory seem to correspond to the spikes found

in the tpf for Ardor3D. This might indicate that the spikes are somehow related to the removal of garbage, perhaps in the process of cleaning up static lists created in OpenGL. Java 3D seem to have a constant amount of garbage created throughout the whole benchmark, but always drops down to approximately the same level. jME3 creates much garbage at the begging of the benchmark, but throughout the execution it shrinks to a much lower value. jME3 has the lowest base usage of the three. Ardor3D comes second, with Java 3D not far behind. It is worth highlighting though, that the base memory usage in Ardor3D rises to much higher values than the base usage in Java 3D, as more garbage is created. Therefore it is not so easy to declare either one of them better than the other.

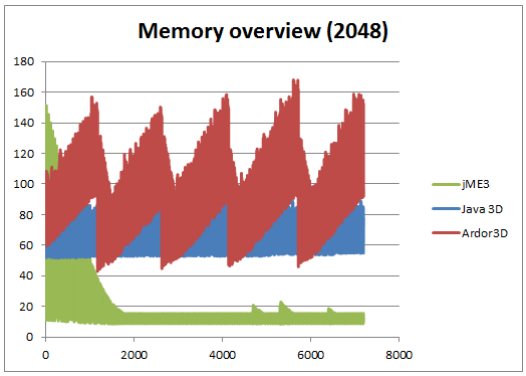


Figure 6.8: Memory overview for the *dynamic geometry* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 2048 objects. The memory is shown in megabyte.

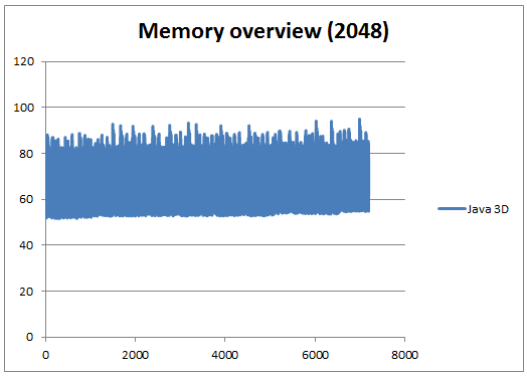


Figure 6.9: Memory overview for the *dynamic geometry* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 2048 objects. The memory is shown in megabyte.

To summarize, the benchmark showed that jME3 was the fastest API, followed by Java 3D. Ardor3D were a little slower than Java 3D. There were registered large spikes in the tpf for Ardor3D during the benchmark. The exact reason for this is unclear, however a correspondence between the spikes in the tpf and when garbage collection was done were found. There might be some connection here. Perhaps Ardor3D is creating, and maintaining static lists throughout the whole benchmark, which are constantly changing due to the dynamic geometry. Apart from the spikes in Ardor3D, the performance was otherwise stable throughout the benchmark.

6.4.2 Frustrum

The *frustrum* benchmark consists of a series of rotating cubes, that together forms a cluster. This cluster is moved around the scene along an elliptic trajectory, and along this path it passes partially outside the far plane of the cameras view frustrum, and fully past the near plane. At these points

along the path, the APIs should cull away any objects that are not inside of the view frustrum.

The execution of this benchmark were run from 32 to 16384 cubes in the cluster. In Figure 6.10 the total execution time when running this benchmark is shown. We see here that jME3 was the fastest of the three, with a total time of approximately 190 seconds at 16384 nodes. Ardor3D were slower than jME3, with an approximately total time of 300 seconds. Java 3D was the slowest, with a time more than three and a half times slower than jME3, of approximately 690 seconds. Looking at the average frames per second for the APIs in Figure 6.11, we also see the same relationship between them. jME3 has an average frame rate of approximately 38, whereas Ardor3D and Java 3D is at 23 and 10 frames per second.

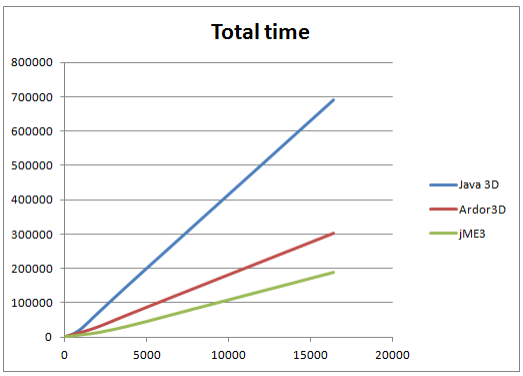


Figure 6.10: The total execution time for the *frustrum* benchmark. Showing results for Java 3D, Ardor3D and jME3. The time is shown in milliseconds.

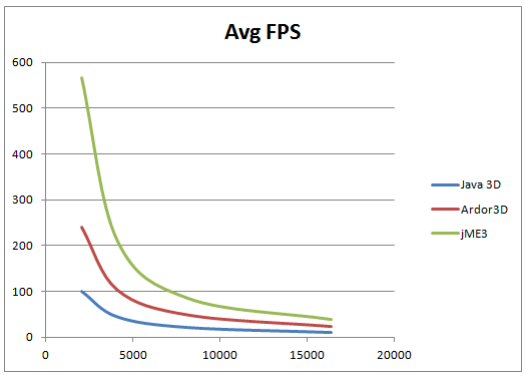


Figure 6.11: The average fps for the *frustrum* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows the average fps from the range of 2048 to 16384.

It is interesting to investigate this difference further by looking at the time used rendering each frame. This is shown in Figure 6.12. In this benchmark the cluster of nodes are being moved between and outside the near and far plane of the view frustrum. If the APIs are doing any culling of objects that are outside of the view frustrum, we should see a change in the time used rendering each frame. As less nodes should be rendered when they are outside of the frustrum, we should see a decrease in the time. In the Figure we can see that all the APIs have distinct jumps in the time used rendering each frame, especially for jME3 and Ardor3D. All APIs have a small stable tpf in the start. Then the tpf jumps, and stays stable for a longer time. This is followed by a short drop, then a longer period with a higher tpf, and then a final drop again. This coincides with the elliptic trajectory the cluster follows, with the high and low tpf values corresponding to when the cluster is outside or inside of the view frustrum. There is a greater difference between the tpf for jME3 and Ardor3D when the cluster is inside or outside of the view frustrum, compared to that of Java 3D.

The tpf when the cluster is outside of the frustrum shows the object overhead for all the APIs. This is because none of the cubes are being drawn. When the cluster is inside the frustrum we see the difference in rendering speed between the APIs. It is interesting to note that the tpf for Java 3D is almost the same regardless of whether the cluster is inside or outside of the frustrum.

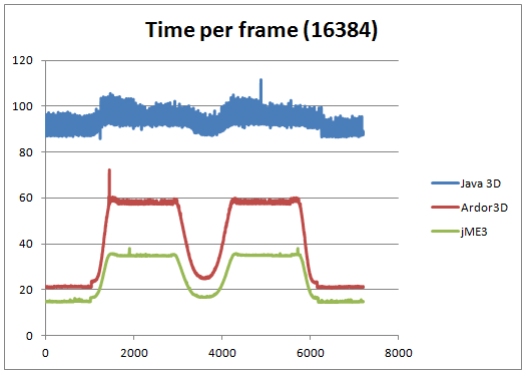


Figure 6.12: The time per frame for the *frustrum* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 16384 objects. The time is shown in milliseconds.

Looking at the memory overview in Figure 6.13, we see that both Ardor3D and jME3 has a stable memory usage, with no garbage creation. The usage for the APIs lies at approximately 55 megabyte for jME3, and 70 megabyte for Ardor3D. Java 3D on the other hand shows a distinct zigzag pattern, with seemingly a constant amount of garbage being generated throughout the duration of the benchmark. The usage of Java 3D lies at approximately 207 megabyte. Both the peaks, and the base usage for Java 3D increases throughout the benchmark, and at the end, the base usage for Java 3D lies at approximately 234 megabyte.

Figure 6.14 shows the spikes in the time per frame for all the APIs. There are very few spikes for all of the APIs in this benchmark. Most of the spikes that occurred, were with relatively small number of cubes in the cluster, between 32 and 256. The Figure shows that Ardor3D had 12 spikes at 32 cubes, and then one spike at 64 and 128 cubes. Java 3D had some spikes for the first periods (32 cubes with 7 spikes, 128 cubes with 5 spikes). jME3 also had some spikes for the first periods (32 cubes with 13 spikes, 64 cubes with 8 spikes, and 128 cubes with 14 spikes). It is unclear why these spikes occurred, and it is more surprisingly considering that they only happened when rendering few cubes. However since the values stabilize, and the number of spikes actually are at zero for the higher object counts, it should not be placed as much emphasis on this as the other metrics captured in this benchmark.

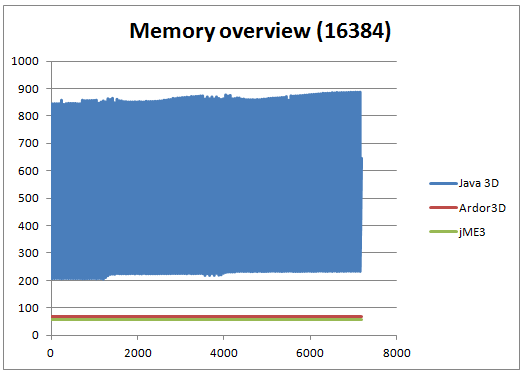


Figure 6.13: Memory overview for the *frustrum* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 16384 objects. The memory is shown in megabyte.

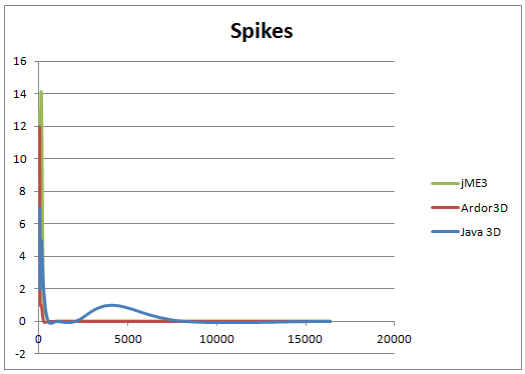


Figure 6.14: The number of spikes in the time per frame for the *frustrum* benchmark. Showing results for Java 3D, Ardor3D and jME3. A spike is given with an increase in tpf equal to, or greater than 40 percent.

To summarize, all the APIs seem to perform frustrum culling. There are visible differences in the time per frame for all APIs, as the cluster of cubes in the benchmark moves in and outside of the cameras view frustrum. The performance of the three APIs shows that jME3 was the fastest, being a little faster than Ardor3D, and about three and a half times faster than Java 3D.

6.4.3 Node stress add and removal

The *node stress add and removal* benchmark consists of a number of cubes that are being added to, and removed from the scene each frame. The performance of the APIs when doing such rapid add and removal operations to and from the scene graph might impact the performance, especially if any internal calculations, or optimizations are done. If these operations are costly, it might lead to unstable performance, and many spikes in the frame rate.

This benchmark was run from 32 to 8192 cubes. The total time used for the execution of the benchmark is shown in Figure 6.15. Java 3D used the longest time of the three APIs, and the time duration also rises at a much higher rate than the two others. Ardor3D was the fastest, with a total time of approximately 147 seconds at 8192 cubes. jME3 were three times slower, using approximately 460 seconds. Java 3D were more than seventeen times slower than Ardor3D, with an approximately time of 42 minutes (2547 seconds). This indicates that much more work is done internally in Java 3D as nodes are added and removed. This is also reflected in Figure 6.16, which show the average frames per seconds for each of the APIs.

Looking at the time per frame (when rendering 2048 objects) for each of the APIs in Figure

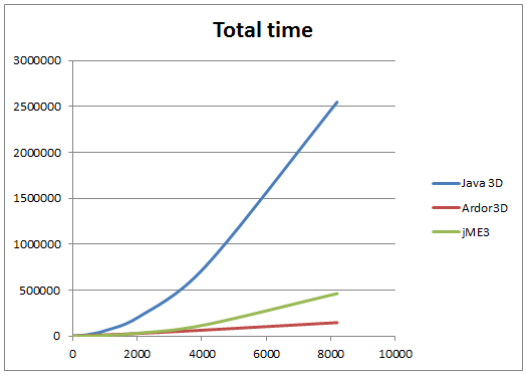


Figure 6.15: The total execution time for the *node stress add and removal* benchmark. Showing results for Java 3D, Ardor3D and jME3. The time is shown in milliseconds.

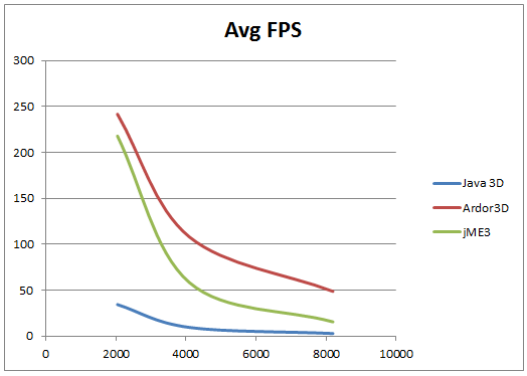


Figure 6.16: The average fps for the *node stress add and removal* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows the average fps from the range of 2048 to 8192.

6.17, we see the same relationship. The time used rendering each frame spikes for all of the APIs. This is very apparent for Java 3D in the Figure, where it spikes from a couple of milliseconds, to 60-80 milliseconds at average, and tops at values as high as 115 milliseconds. The scale makes it hard to read the values for jME3 and Ardor3D, therefore Figure 6.18 shows the results for only these two. Here we see that these two APIs also have continuous spikes throughout the whole duration of the benchmark. Ardor3D spikes from around 1 millisecond, and up to about 6.5 milliseconds. jME3 spikes a little higher, going from about the same, around 1 millisecond, and up to about 8 to 11 milliseconds.

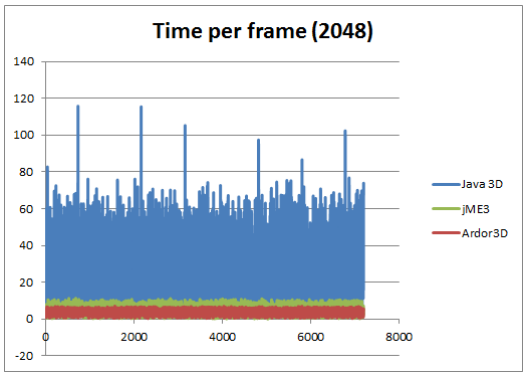


Figure 6.17: The time per frame for the *node stress add and removal* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 2048 objects. The time is shown in milliseconds.

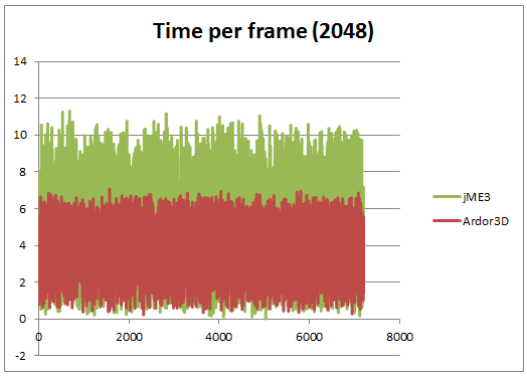


Figure 6.18: The time per frame for the *node stress add and removal* benchmark. Showing results for Ardor3D and jME3. The graph only shows results when rendering 2048 objects. The time is shown in milliseconds.

Figure 6.19 shows the number of spikes for the different number of objects for all the APIs. We see here, what was confirmed by the Figures for the time per frame, that there are many spikes in the time per frame for all the APIs. It is interesting to note that jME3 spikes more than Java 3D, even though it has shown better performance results for the previously mentioned metrics. This might be because it is much faster at the rendering than the others, but when the nodes are added or removed there is a workload being enforced on it, that would naturally decrease the performance. This would in turn lead to the number of spikes reported. The Figure shows that the number of spikes for Ardor3D seems to have stabilized, between 550 and 700 spikes. Java 3D also seems to stabilize, between 1100 and 1350 spikes. jME3 on the other hand only seems to only increase in the number of spikes per period, with a total of 2074 spikes at 8192 objects. It is worth noting however, that the spikes in Java 3D are much more severe, than the ones in both Ardor3D and jME3. This is because the time per frame in Java 3D changes much more, than the other two.

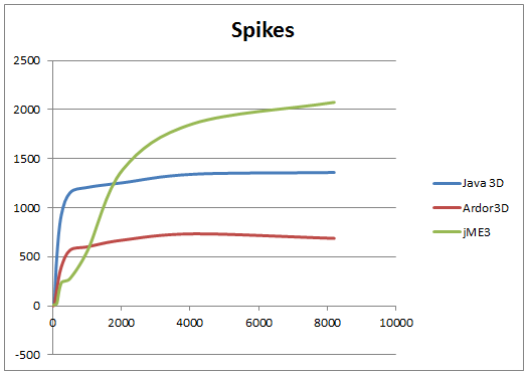


Figure 6.19: The number of spikes in the time per frame for the *node stress add and removal* benchmark. Showing results for Java 3D, Ardor3D and jME3. A spike is given with an increase in tpf equal to, or greater than 40 percent.

The memory overview for all the APIs is shown in Figure 6.20. We see that all APIs create a lot of garbage during the execution of the benchmark. Both jME3 and Java 3D shows zigzag patterns, that at times spikes very high. We see that jME3 has a base usage of approximately 38 megabyte, whereas Java 3D lies at about 40. Figure 6.21 shows the memory overview for just Java 3D. Ardor3D starts with a base usage of approximately 150 megabyte. The Figure shows that the garbage created in Ardor3D increases at fixed intervals, with a fixed amount throughout the whole benchmark. It would seems as though the base usage in Ardor3D only rises for the duration of the benchmark, however this is most likely not the fact. Considering the memory overview for all object counts (32 to 8192), we can see similar patterns, however the garbage collection kicks in

after a while, and bumps the garbage down to the base usage, as seen in the start of the benchmark. Therefore we suspect that the recorded memory overview for Ardor3D might not be over a long enough time period to capture this. So the reader is advised to keep this in mind. It needs to be pointed out however, that although the garbage collection rises throughout the benchmark, it never reaches the peak of garbage created by Java 3D.

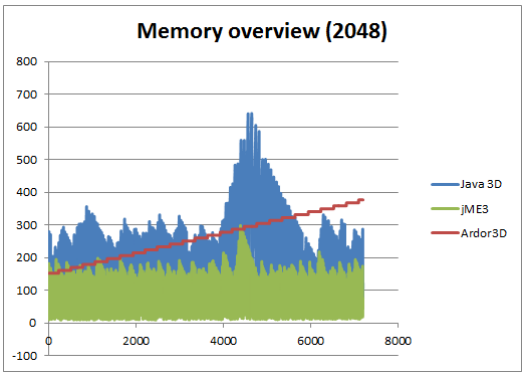


Figure 6.20: Memory overview for the *node stress add and removal* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 2048 objects. The memory is shown in megabyte.

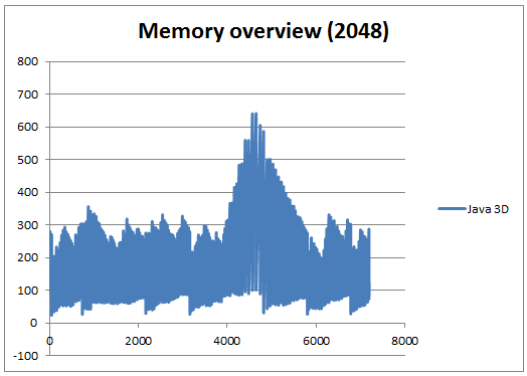


Figure 6.21: Memory overview for the *node stress add and removal* benchmark. Showing results for Java 3D and Ardor3D. The graph only shows results when rendering 2048 objects. The memory is shown in megabyte.

To summarize, Ardor3D and jME3 were both several orders of magnitude faster than Java 3D for this benchmark, with Ardor3D being more than seventeen times faster than Java 3D. Ardor3D was the fastest API in this benchmark. All APIs showed spikes in the time used rendering each frame, however the number of spikes that had an increase in time equal to or greater than 40 percent seemed to stabilize for Ardor3D and Java 3D as the number of objects increased, indicating that the APIs scales well when increasing the object count. jME3 on the other hand, showed a slight increase in the number of spikes as the number of objects increased. Although jME3 reported more spikes than Java 3D, it is important to note that the severity of spikes in Java 3D is several orders of magnitude more critical than those in jME3. This is because a spike in jME3 only involves an increase in time per frame of 30 milliseconds, whereas a spike in Java 3D could mean an increase in over 115 milliseconds.

6.4.4 Picking

The *picking* benchmark attempts to assess the performance of the APIs when performing rapid picking operations on geometry. The test picks 81 rays from the cameras location, and into the

scene. The scene consists of a cluster of rotating spheres. The result from the pick operations are required to contain geometry information for the intersection. This means that we should be able to obtain the vertices for the picked triangles.

The benchmark were run from 32 to 4096 spheres. In Figure 6.22 we can see the total execution time used for all the APIs. We see here that Java 3D was the fastest API. At 4096 spheres, Java 3D was over 7 minutes faster than jME3, and over 18 minutes faster than Ardor3D. This is also evident when looking at the average frames per second, shown in Figure 6.23. This Figure shows object counts between 512 and 4096. Here we can see the same difference between the APIs. Where at 4096 Java 3D has an approximate average fps of 30, jME3 of 10, and Ardor3D of 5.

We have to mention that on the first picking operation performed by jME3, after each increase in object counts, the operation results is a large spike in the performance. This might be caused by caching, because jME3 does some calculations regarding the objects world locations for later use. However this is not captured by the benchmark, because this occurs during the warmup period. We chose only to mention this here, instead of apply any form of penalty to jME3, because this was the chosen design of the benchmark. This could be noticeable in a real application though, so the reader should be aware of this, although this is something that could potentially be ”hacked” away during loading of an application.

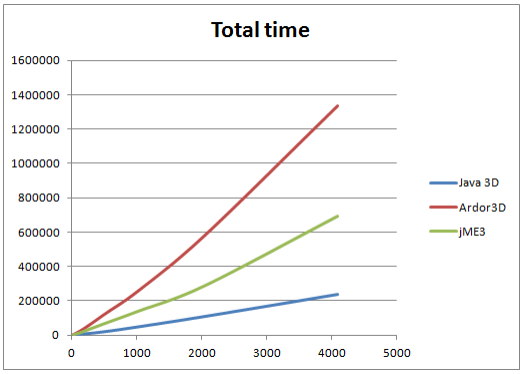


Figure 6.22: The total execution time for the *picking* benchmark. Showing results for Java 3D, Ardor3D and jME3. The time is shown in milliseconds.

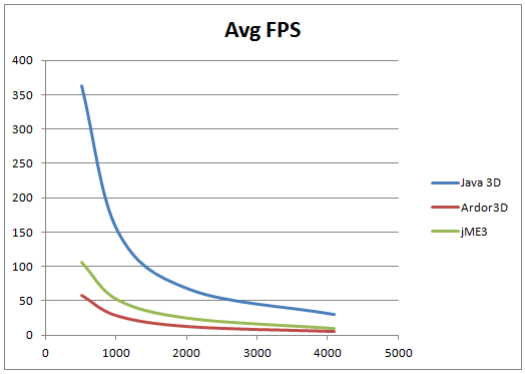


Figure 6.23: The average fps for the *picking* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows the average fps from the range of 512 to 4096.

Figure 6.24 shows the time used rendering each frame for the APIs. We see here that Java 3D is almost three times faster at rendering each frame than jME3, and almost six times faster than Ardor3D. Java 3D lies at an average tpf of 31, whereas jME3 lies at 91, and Ardor3D at 176. The Figure also shows that there are some spikes in the tpf for all APIs, however of the three, Ardor3D

and jME3 spikes the most. We note however that none of the spikes were so high that they are any cause any concern. This is also evident by looking at Figure 6.26, which show the total number of spikes for each object count. Here we see that there are some spikes for all APIs at the lowest object counts. There are however no spikes at higher object counts, so it is not considered a problem for any of the APIs.

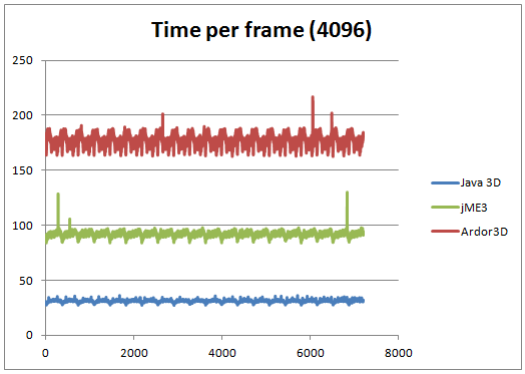


Figure 6.24: The time per frame for the *picking* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 4096 objects. The time is shown in milliseconds.

The memory overview for each API is given in Figure 6.25. Here we see that jME3 has a base memory usage of approximately 120 megabyte. It generates some garbage throughout the benchmark, compared to Java 3D though, it is of no significance. Java 3D on the other hand generates much more garbage than jME3 in this benchmark. We see a distinct zigzag pattern. The amount of garbage created in Java 3D is reduced during the benchmark. The base memory usage in Java 3D is lower than jME3, and lies at around 75 megabyte. Ardor3D has a base memory usage of 239 megabyte at the start of the benchmark. Throughout the benchmark Ardor3D creates garbage, which is removed by the garbage collector at intervals. There is also a zigzag pattern for Ardor3D, however it is so rapid, that it is hard to see it. It is of concern that the base usage of Ardor3D does not drop down to the initial value that it has at the beginning of the benchmark. At the end of the benchmark the base usage was at 302 megabyte, this means an increase in approximately 60 megabyte. It is unclear if the base usage would drop lower if the benchmark had been run for a longer time duration. At lower object counts there was also an increase in the base usage throughout the benchmark for Ardor3D. These results indicates that there might be a memory leak in Ardor3D when doing picking operations on the primitive level.

Java 3D was the API that performed best in this benchmark. It was shown to be much faster

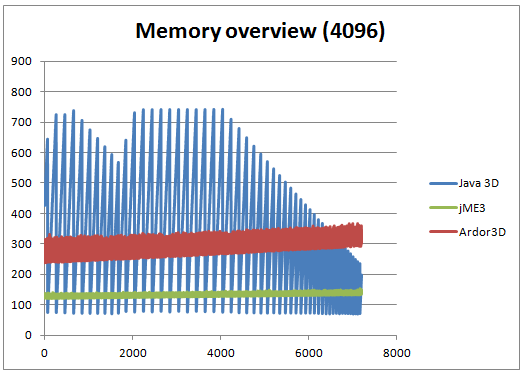


Figure 6.25: Memory overview for the *picking* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 4096 objects. The memory is shown in megabyte.

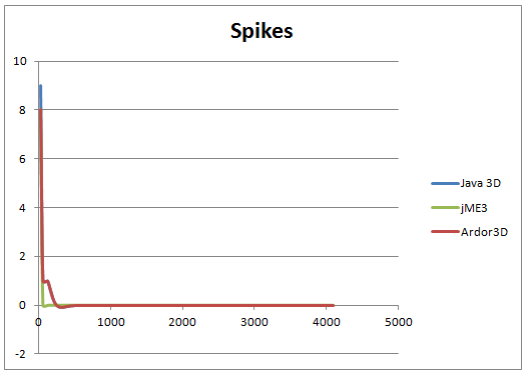


Figure 6.26: The number of spikes in the time per frame for the *picking* benchmark. Showing results for Java 3D, Ardor3D and jME3. A spike is given with an increase in tpf equal to, or greater than 40 percent.

than jME3 and Ardor3D, and it also had a much more stable time rendering each frame. Java 3D was three times faster than jME3, and almost six times faster than Ardor3D. There also appears to be a memory leak in Ardor3D when performing picking operations. We should also mention that there is a large spike in jME3 after adding more objects to the scene, during the first pick operation. This is related to some of the cached data calculated by jME3. This is not noticeable in the results, because it occurs during the warmup period of the benchmark, however it is a limitation that we want the reader to be familiar with.

6.4.5 State sort

When using the OpenGL pipeline different states needs to be set when rendering different types of materials. For example rendering lit materials requires various light states to be set, whereas different states needs to be set when rendering textures or shaders. Changing between these states is a costly operation, and therefore changes between states should be kept to a minimum by sorting geometry based on states. The results from this benchmark show that there is a difference in the performance between the APIs. This benchmark was run from 32 to 16384 cubes, which were rotated randomly in a cluster of cubes.

For this benchmark it is important to keep in mind that jME3 features a complete shader based approach, which means that they have dropped the fixed function pipeline in OpenGL. Effectively this means that every material used is a shader, and thus there will not really be any difference between setting a light material, and setting a shader material for a object. The benefit of sorting

objects with the same texture is still there though. Although they do not use the fixed function pipeline, they emulate all its functions, so the rendered result is the same as with the other APIs. This should be kept in mind by the reader, and regardless of the results it is a consciously design decision, and the validity of the results will be the same as for the other APIs.

Figure 6.27 shows the total execution time for the APIs when running this benchmark. We see here that jME3 was the fastest of all the APIs. Java 3D and Ardor3D had similar execution times at lower object counts, however at higher counts than 4096 Java 3D started using longer time than Ardor3D. The difference at 16384 objects were approximately 110 seconds. jME3 was almost twice as fast as Ardor3D, using only 350 seconds to execute the benchmark at 16384 nodes. Ardor3D and Java 3D used respectively 611 and 726 seconds at the same object count. The average frames per second is shown in Figure 6.28. This shows a similar correspondence between Ardor3D and Java 3D, where they have fairly similar results, however Ardor3D is slightly faster. These results also show that jME3 is faster than the other two.

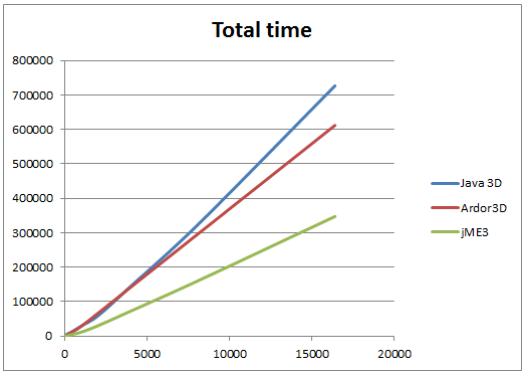


Figure 6.27: The total execution time for the *state sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The time is shown in milliseconds.

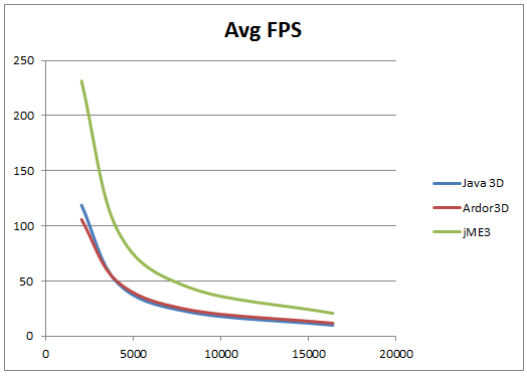


Figure 6.28: The average fps for the *state sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows the average fps from the range of 2048 to 16384.

Looking at the time used rendering each frame, which is seen in Figure 6.29, we see that both jME3 and Ardor3D have stable rendering times. The ordering, in terms of performance between the APIs is the same as before when considering the time used rendering each frame. The rendering time for jME3 lies between 45 milliseconds and 47 milliseconds, and Ardor3D lies between 80 milliseconds and 84 milliseconds. Java 3D on the other hand spikes more in the time used rendering each frame than the other two. Not only does it use more time than the other two, it also spikes higher, going from 91 milliseconds to 106 milliseconds. The spikes however are not alarmingly high, as they do not exceed the 40 percent increase in time that is required in order to be classified

as a spike. The number of such large spikes are shown in Figure 6.31. We see here that the number of large spikes are low for all the APIs. There are some spikes for Ardor3D and jME3 at low object counts, and there are a couple of spikes for Java 3D at 4096 objects. However no spikes are registered for higher objects counts for any of the APIs. This indicates that it is not a major problem in any of the APIs, since this is of most importance and interest at higher object counts.

Memory overview in Figure 6.30 shows that there is stable memory usage in Ardor3D, when rendering 16384 cubes. It does not generate any significant amount of garbage during the execution of the benchmark. The base usage lies at 80 megabyte. jME3 has a base memory usage of 244 megabyte at the start of the benchmark, however during the execution of the benchmark the usage rises with a constant amount of garbage being generated. The garbage created is not removed by the garbage collector in the Java virtual machine during the benchmark, so it is unclear whether the base usage rises temporarily and will drop after a fixed amount of time, or if it continues to rise if the duration were to be extended. Looking at the memory overview for jME3 for all object counts, shows that it is only with an object count of 16384 that the usage rises like this. With lower object counts the usage is constant, and similar to that of Ardor3D. We are therefore unable to make any direct assumptions about jME3, however the collected data would suggest a base memory usage of 244 megabyte at the start, with a linear increase throughout the benchmark. Java 3D creates a lot of garbage, which is clearly visible in the Figure. It shows a zigzag pattern, that spikes from a base usage of 244 megabyte, up to 900 megabyte.

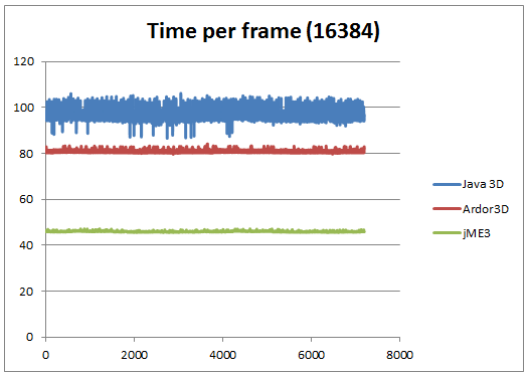


Figure 6.29: The time per frame for the *state sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 16384 objects. The time is shown in milliseconds.

The results from this benchmark shows that of the three, jME3 were much faster than the other

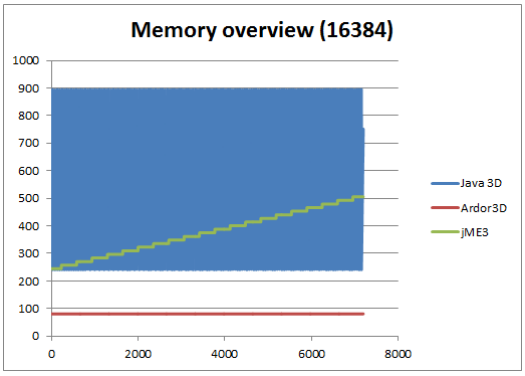


Figure 6.30: Memory overview for the *state sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 16384 objects. The memory is shown in megabyte.

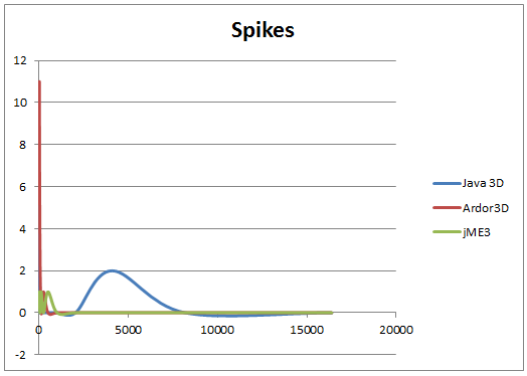


Figure 6.31: The number of spikes in the time per frame for the *state sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. A spike is given with an increase in tpf equal to, or greater than 40 percent.

two. Whether this is because it is generally faster to do materials with shaders, better sorting algorithms or less overhead in the API is unclear. Nevertheless, it is evident that jME3 performs the best. The performance of Java 3D and Ardor3D were fairly similar, however Ardor3D proved to be slightly faster than the other. There were registered some fps spikes when rendering low object counts for both Ardor3D and Java 3D, the reason for this is unclear, and since there were no spikes registered for higher objects for neither, it is not deemed to be of any significant importance.

6.4.6 Transparency sort

The *transparency sort* benchmark consists of a set of rotating spheres, that together form a cluster of spheres. These are all transparent, which means that they need to be sorted manually in order to display, and blend correctly. This benchmark tests how well each of the APIs handles this case.

The execution of this benchmark were run from 32 spheres to 16384 spheres. The total execution time the APIs used when running this benchmark is shown in Figure 6.32. We see here that jME3 is the fastest of the three, with a total execution time at 16384 objects of approximately 450 seconds. The results show that Ardor3D and Java 3D have fairly similar execution times. At higher object counts Ardor3D is a little faster, with a difference at 16384 objects of approximately 45 seconds. The average frames per second is shown in Figure 6.33. Here we see that Ardor3D and Java 3D have very similar frame rates in the benchmark, with Ardor3D being only a little faster. jMe3 has the best fps, with a difference to Ardor3D of 6 fps, at 16384 objects. The approximately fps at 16384 nodes for the APIs were 16 for jME3, 10 for Ardor3D, and 9 for Java 3D.

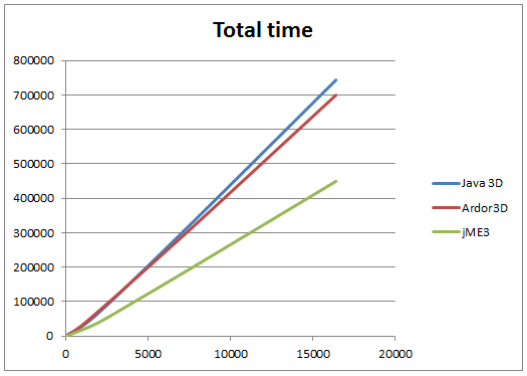


Figure 6.32: The total execution time for the *transparency sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The time is shown in milliseconds.

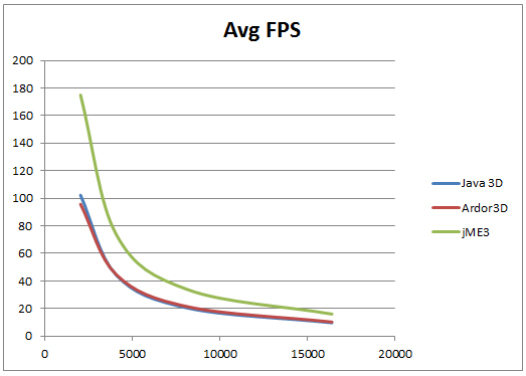


Figure 6.33: The average fps for the *transparency sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows the average fps from the range of 2048 to 16384.

Figure 6.34 shows the time used rendering each frame in the benchmark. We see here that jME3 has the lowest, and a very stable tpf throughout the whole duration of the benchmark. The tpf for Ardor3D and Java 3D is much higher, using almost twice as long. jME3 lies at an average tpf of 59 milliseconds at 16384 objects. Ardor3D lies at an average tpf of 92 milliseconds, and Java 3D lies at an average of 98 milliseconds. We see that Java 3D has a higher variation in the tpf than the other two APIs. Looking at Figure 6.36 we can see the number of spikes for each API. We see here that there were no spikes at all for Ardor3D. jME3 had a couple of spikes in the start, with an object count of 32 and 128, however there were just a couple of spikes. There were recorded no more spikes for jME3 at higher object counts, so it is not considered to be a problem. Java 3D spikes some in the start, with object counts of 32 and 64. Then we see that there are some spikes at higher object counts. There does not seem to be any apparent pattern to the spikes for Java 3D, and since they are not present at the highest object count, it does not seem to be any major issue for this benchmark.

The memory overview for each API is shown in Figure 6.35. We can see here that both jME3 and Ardor3D have a constant base memory usage throughout the whole benchmark, with no additional garbage being created. jME3 had the lowest base memory usage, of 55 megabyte. Ardor3D had a base usage of 73 megabyte. Java 3D on the other hand uses much more memory. The base usage lies at around 190 megabyte. We see a clear zigzag pattern for Java 3D, which shows that it generates a lot of garbage throughout the benchmark.

These results show that when it comes to sorting transparent objects, it was shown that jME3

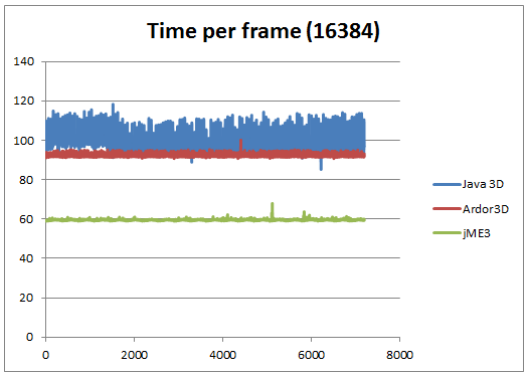


Figure 6.34: The time per frame for the *transparency sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 16384 objects. The time is shown in milliseconds.

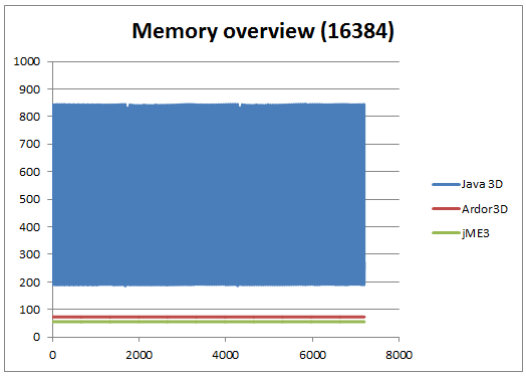


Figure 6.35: Memory overview for the *transparency sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. The graph only shows results when rendering 16384 objects. The memory is shown in megabyte.

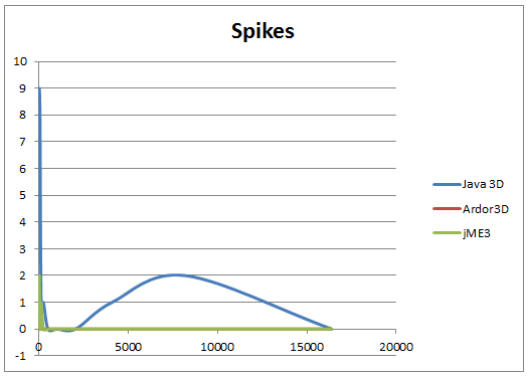


Figure 6.36: The number of spikes in the time per frame for the *transparency sort* benchmark. Showing results for Java 3D, Ardor3D and jME3. A spike is given with an increase in tpf equal to, or greater than 40 percent.

was much faster than the other two APIs. The results for Java 3D and Ardor3D were also shown to be very similar, however of the two, Ardor3D was the fastest. There were also noted a greater deviation in the time used rendering each frame in Java 3D than were noted for the other two APIs. So jME3 is the most efficient at sorting transparent objects, then Ardor3D, who is closely followed by Java 3D.

6.4.7 Summary

The *dynamic geometry* benchmark identified some issues with Ardor3D. There are spikes in the time used rendering each frame for the API. The exact cause of this is unclear, however it is suspected that it might be related to creation of static lists. It was shown to be a correlation between the intervals in memory garbage created and removed, and when the spikes occurred. Although there were performance spikes with Ardor3D, the performance were only a little slower than Java 3D. jME was the fastest API in this benchmark.

The *frustrum* benchmark showed that all APIs perform effective frustrum culling. There are visible changes in the time used rendering each frame when the cluster of cubes is inside, or outside of the cameras view frustrum. The performance results show that jME3 was the fastest of the three, being a little faster than Ardor3D, and much faster than Java 3D. There were also more noticeable differences in the tpf when the cluster where inside or outside of the frustrum in jME3 and Ardor3D, than in Java 3D. This indicates more effective culling algorithms in these APIs.

Results from the *node stress add and removal* benchmark showed that both Ardor3D and jME3 were several orders of magnitude faster than Java 3D. Ardor3D was the fastest of all three APIs. It was three times faster than jME3, and seventeen times faster than Java 3D. Running the benchmark at higher object counts showed that the number of spikes in rendering time for both Ardor3D and Java 3D seem to scale well. The spikes for jME3 showed a slight increase as the object counts increased. The reader should however keep in mind that a spike in Java 3D is much more severe than one in jME3, because of the duration of the spike. A spike in jME3 could last 30 milliseconds, whereas one in Java 3D would last more than 115 milliseconds.

The *picking* benchmark showed that Java 3D had the best performance, both in regards to rendering speeds, and stability regarding the number spikes in the rendering speed. jME3 were much slower than Java 3D, at 4096 there were a difference in total time of 7 minutes between the two. Ardor3D were even slower, with a difference over 18 minutes. There also appears to be a memory leak when doing picking operations in Ardor3D. It should also be noted that the first pick operation in jME3 after increasing the object count resulted in a long spike in the tpf. This is caused by some

internal operations, and caching of some of the calculated data. This occurs during the warmup period of the benchmark, therefore it is not recorded in the result. However I felt this should be mentioned, as it could be noticeable in a real application.

Results from the *state sort* benchmark show that jME3 performs much better than the other two, when presented with many different rendering states. It should be kept in mind that jME3 uses a completely shader based rendering approach, which means that it does not have to change as many of the states as when using the old fixed function pipeline. Regardless, the benchmark showed that jME3 performs much better than the other APIs. Ardor3D and Java 3D performed similar, with Ardor3D being slightly faster. Ardor3D used almost twice as much time rendering as jME3 did.

The *transparency sort* benchmark also showed that jME3 was faster than the other APIs. Java 3D and Ardor3D showed similar results, with Ardor3D being slightly faster. There were a greater deviation in the tpf for Java 3D than Ardor3D. This will give it a more unstable performance, although none of the spikes were too severe.

Looking at the results from the benchmark as a whole, it shows the general trend that jME3 is the fastest API of the three. Then there is Ardor3D, and Java 3D, which are closer together in terms of performance. However of the two, Ardor3D is the fastest. The object overhead is much smaller in jME3 than the other APIs as well. There is a noticeable gap in performance between jME3 and the two others. It is unclear whether the difference in speed between jME3 and the others are due to more optimized algorithms, or if it is a combination of the shader based architecture, and recent graphics drivers being more optimized towards shaders. Regarding memory usage, there is generally a much lower memory usage in both Ardor3D and jME3, than in Java 3D. In addition Java 3D creates much more garbage than the other two. However, as pointed out when talking about the results from each benchmark in greater detail above, there were exceptions to this general trend. The dynamic geometry benchmark had issues in both jME3 and Ardor3D, with high spikes in the tpf. The node stress add and removal benchmark showed that Ardor3D was fastest, and jME3 was slowest. The picking benchmark showed that Java 3D was much faster than the other two.

6.5 Summary

The results from each comparison shows that there are strengths and weaknesses to each API. This section gives a complete summary of every layer from the comparison, grouped by each comparison method.

6.5.1 Project maturity

The comparison of project maturity shows that there was little difference between the APIs. In terms of project maturity score, Java 3D and jmonkeyEngine3 scored a 9, while Ardor3D got a 8. The total maximum score was 11. The aspect all projects lacked were the presence of a well defined automated testing suite for the code. It seems a common agreement between the different projects, that it is hard to write automated tests for code that is very much dependent on being manually inspected for graphical glitches and the likes. Therefore it is not feasible to maintain test code. Another aspect that needs to be highlighted is the state of documentation in Ardor3D. The documentation in Ardor3D is very bad, and much of the information is superficial and outdated. For example, the explanation on how to get the project to work is so outdated, that it is not possible to get it up and running by following the guide. Ardor3D has many code examples that show the various features and capabilities of the API. The main limitation of these examples are that they just contain the code, without any comments or explanations. This is an important factor because it makes it harder for new users to get started with the API. Java 3D and jME3 on the other hand have excellent user, and developer documentation.

6.5.2 System architecture

The analysis of the architecture of the core of the APIs highlighted various aspects regarding their structure, and other aspects. The core of Java 3D features a completely flat structure, which means that there are no hierarchical separation of functionality into sub-packages. Instead all the classes that make up the core of the API are located at the same place hierarchically. jME3 has a good hierarchically structured architecture, with functionality divided into different sub-packages. The core of jME3 however offer way too much functionality outside of what is expected to be in the core of a a 3D scene graph API. For example the core of jME3 offers functionality such as networking and animation, and special effects such as water and shadows. This clutters the core, and is a peculiar design choice. Ardor3D has a very good hierarchical structure, with similar divisions into sub-packages such as jME3. Ardor3D includes only functionality that is expected to be in the core.

The metrics calculated from the design structured matrices shows different traits with the APIs. The results for jME3 were skewed by all the sub-packages that were included in the core, who only provide additional functionality outside of the core functionality required. These sub-packages had much better values than the critical parts of the core, and this greatly affected the results. Because of this the results for jME3 are much better than what is the reality, and does not necessarily reflect the state of the core ”one-to-one”. There were also some problems with Java 3D because of its flat structure. The flat structure prevented a more in depth investigation of the results, because it was not feasible nor possible to calculate much of the data required to investigate it in further detail. The metrics show that the stability for Ardor3D were the best. The real stability for jME3 is unclear, because the results were skewed. The initial value for jME3 showed it had the best stability of the three, but upon further investigation of its sub-packages, it was shown to have a much lower stability for the ”core parts” of the core. This does not necessarily mean that its actual stability is that bad, but it is most likely lower than that of Ardor3D. The stability for Java 3D is alarmingly low. The reason for this is unclear because of the previously mentioned problems. It indicates however that there are very much connectedness in the core. The system cyclicality for both Ardor3D and jME3 were relatively low, but upon further investigation it was identified that many of the most important parts of the cores are involved in such problematic cycles. The system cyclicality for Java 3D was very high, covering more than two thirds of the core, which is not good. The hierarchical cyclicality for Java 3D is the same as the system cyclicality. This is because there is no hierarchical structure, and therefore there is no change in the value. The hierarchical cyclicality for the two other APIs were more than doubled compared to the system cyclicality. This is natural, but it is of concern that the values were so high that it covers almost the whole core (approximately 90%). The high degrees of cyclicality in all APIs means that the maintainability may be much harder, and it might require more iterative work because changes must be applied and cascaded to many places. This also increases the risk of introducing bugs in the code.

6.5.3 System features and capabilities

The comparison of features and capabilities shows that overall the APIs seem to support most of the features, although Java 3D is lacking some crucial features. All the major desktop platforms are supported by the APIs, and Ardor3D and jME3 additionally support the Android platform. In terms of graphic libraries, all the APIs support OpenGL, and the shader language GLSL. When it comes to rendering capabilities both Ardor3D and jME3 offers all the listed capabilities and features. Java 3D on the other hand lacks some of the most crucial features required for producing most visual

effects. Most noteworthy is the support for multi-pass rendering and frame buffer objects. All the APIs support true stereoscopic rendering (quad buffering). They also support multiple canvas, which means that they can be used for creating virtual environments, such as a CAVE. Only Java 3D however has built in support for tracking devices, such as head tracking device. When it comes to modelling formats, only Java 3D supports VRML. Ardor3D and Java 3D supports COLLADA. There was a community project for jME3 that were developing a COLLADA loader for it, but it was abandoned and has been marked as dead. All the APIs have built in tools and utilities for optimizing geometry in various ways. jME3 is the only one of the APIs who does not offer utilities for generating normals. Both Ardor3D and jME3 have built in many systems for producing various visual effects, such as systems for water, shadow, and terrain generation. jME3 also has a tight built in integration with a physics library, in addition to built in networking and audio systems. None of the other APIs have built in physics support in the API, however it has been successfully implemented by users of both APIs.

6.5.4 System performance

The comparison of the performance shows that overall jME3 was significantly faster than the other two APIs. The other two APIs had fairly similar performance, but Ardor3D was faster than Java 3D overall. The performance difference shows that there is less object overhead in jME3 opposed to the other two. This might be related to the shader based architecture in jME3, but not necessarily. The other two APIs rely on the fixed function pipeline, which might have more overhead. The graphic drivers might also be more optimized towards the use of shaders as well, giving jME3 an advantage. This does not discredit the results for jME3, on the contrary the fixed function pipeline is on its way out, being slowly deprecated in newer versions of OpenGL [1].

When it comes to the specific benchmarks, the dynamic geometry benchmark showed that jME3 was fastest, followed by Java 3D, and last Ardor3D. Ardor3D had several spikes during the benchmark, that occured at regular time intervals. The reason for these spikes are unclear, but there seems to be a correlation between the spikes, and the garbage collection performed by the Java virtual machine. The frustrum benchmark showed that all APIs perform effective frustrum culling. jME3 was also the fastest here, followed by Ardor3D, and last Java 3D. There seems to be much more overhead in Java 3D than the other APIs. This is because there is very little difference in the time used rendering each frame, comparing the times when the cluster of cubes are inside of the frustrum, and when they are outside. For Ardor3D and jME3 there are a significant difference in the

[1] http://www.opengl.org/wiki/History_of_OpenGL#Deprecation_Model

times comparing inside, or outside of the frustrum. The node stress add and removal benchmark rapidly adds, and removes nodes to and from the scene graph. The results show that Java 3D were several orders of magnitude slower than the other two. In this benchmark Ardor3D was the fastest of the three. The picking benchmark showed that Java 3D was the fastest API, and Ardor3D was the slowest. In jME3 there was a huge spike when performing the first pick operation after each increase in object counts, but this was not captured by the benchmarks, because it occurs during the warmup period. The benchmark also showed that there might be a memory leak in Ardor3D when performing many rapid picking operations. The state sorting benchmark showed that jME3 was significantly faster than the other two APIs, whereas the performance of the other two were fairly similar to each other. The transparency sort showed similar results to the state sort, where jME3 was also fastest, and the other two have similar results. The time used rendering each frame were more unstable in Java 3D than in Ardor3D however.

6.6 Final conclusion

The evaluation have shown to be a intricate and complex process, with results that highlight different aspects of the APIs. This section gives a final conclusion to the evaluation and comparison of the APIs.

The individual results for each comparison is shown in Table 6.15. This table shows which API that scored best for each individual comparison. For a more thorough explanation of the results, the reader is referred to the summary given in Section 6.5, and the sections regarding the results for each specific comparison method in this chapter.

Final Comparison Results			
	Java 3D	Ardor3D	jME3
Project management and technical infrastructure	1	2	1
System architecture	3	1	2
System features and capabilities	2	1	1
System performance	3	2	1

Table 6.15: The final results from the comparison done in this thesis. The scoring is in the range of 1 to 3, where 1 is the highest value (best), and 3 is the lowest (worst).

The results from the comparison are too intricate to give a definitive black and white answer to the evaluation. There is no definitive ”winner” amongst the APIs. This is because there are both

strengths and weaknesses associated with each API. So while the results are unable to directly name one API ”the best”, it is possible to give recommendations based on the comparison.

In essence the question of which API is ”best”, boils down to which of the highlighted aspects are most important. This is in many cases personal, and depends on the project and personal/organizational preferences. From the results we are able to give the following recommendations based on possible requirements.

If performance is very important, then jMonkeyEngine3 will be the most suited API. It offers excellent performance, which were proven to be several orders of magnitude faster than the two others.

If a good and well defined system architecture is desired, then Ardor3D is the best choice. It offers the best, cleanest, structured, and most organzied architecture of all the APIs.

If certain capabilities are important, then jME3 or Ardor3D are the best candidates. These APIs support most of the possible capabilities that are required by a 3D Scene Graph API. The only note is regarding model loaders, where there are limitations to what formats are supported.

If it is important that specific technical features and algorithms are implemented, either for the purpose of rapid development, or abstraction above the algorithms, then the answer is twofold. If the requirement lies with more visually complex graphical features (usually through shaders and frame buffer objects), then the best alternative is jME3. Ardor3D also offers a wide range of features, but not to the extend that jME3 does. Should the requirements not extend what is possible with FFP, then Java 3D is also a viable option. Keep in mind however, that whatever Java 3D offers, the same is most likely available in jME3 and Ardor3D as well.

If documentation is important, then either Java 3D or jME3 is best suited. They both have wast amounts of documentation and tutorials available. Most problems have also been tackled before, and is available through the forums. The presence of good documentation also makes the APIs easier to use, and promotes rapid development and easier transition to the APIs.

If the community surrounding the API is important, then arguably jME3 is the best choice. The reason for this is because it has the largest, and most active community. The community is also actively contributing to the project. Keep in mind, that quality does not necessarily reflect quality. Ardor3D has a smaller community, and Java3D is to a large degree abandoned.

Related to the community aspect is the question regarding the lifespan of the projects. Depending on whether you are planning on using the API long-term, or just for a couple of projects this may, or may not be so relevant. Giving a definitive answer to this is not easy, and there is no definitive answer. However the history have shown that the jME project were able to survive, even

after the core developers abandoned the project. The developers of Ardor3D were part of the core developers of the old jME project, which they left in favour of Ardor3D. This does not necessarily mean that they will abandon Ardor3D, but history have shown that they are capable of doing it. There is nothing that suggests that none of the projects will be abandoned however, but it is worth mentioning. Also keep in mind that all the projects are also open source, so if the developers should leave, the project may continue in the hands of others.

Chapter 7

Summary, Discussion, Future Work and Conclusions

This chapter discusses and concludes the work in this thesis.

Section 7.1 gives a summary of what have been done. Section 7.2 discusses both the proposed methodology, and the results from applying it to a selection of APIs. Section 7.3 suggest possible future work based on this work. Finally Section 7.4 concludes the thesis.

7.1 Summary

This thesis presents an evaluation of a set of 3D Scene Graph APIs available in Java. The APIs are Java 3D, Ardor3D and jMonkeyEngine3. The reasons for choosing these are explained in Chapter 1.

Two research objectives were defined for this thesis:

1. Identify or develop a suitable methodology for evaluating 3D Scene Graph APIs.
2. Evaluate the APIs using the chosen methodology.

No suitable method for conducting the evaluation was found, therefore a methodology has been developed. This methodology divides the evaluation into four different layers, that each compares different aspects of the APIs. By inspecting different aspects of the APIs, the evaluation is able to give a more thorough, representative and valid result. The aspects analysed in greater detail were:

1. Project management and technical infrastructure

2. System architecture
3. System features and capabilities
4. System performance

These layers were then analysed using the following methods:

1. Project management and technical infrastructure were investigated by using a metric called the *project maturity*. This metric is a rating of the projects healthiness based on a series of yes/no answers. The answers looks at the presence of versioning systems, documentation and bug tracking tools amongst others.
2. System architecture was investigated by creating design structure matrices from the static source files for each project. From these three different metrics were calculated: System stability, system cyclicality, and hierarchical cyclicality.
3. System features and capabilities were investigated by a series of yes/no questions. These identify what features and capabilities the APIs offer, or alternatively do not offer.
4. System performance was investigated by defining and implementing a series of performance benchmarks. These were designed to be atomic, and tests the performance when the APIs are performing different tasks.

Each API was investigated in detail using the defined methods for each layer. The results from each layer were then compared against the results for the other APIs. This investigation of different aspects with each API gives the evaluation both an in-depth comparison of each specific aspect, as well as the broader picture, by considering all the layers combined.

The results from the comparison showed that there were both strengths and weaknesses with all three APIs, and none could be declared a definitive ”winner”. Instead several recommendations were made, based on different criteria that may be important for the project or based on personal/organisational preferences.

7.2 Discussion

I think the evaluation methodology proposed in this thesis has worked well. It is no small task to evaluate something as comprehensive as a 3D Scene Graph API, and I think the layered approach is able to capture the most important aspects.

The final results from the work were not able to declare a definitive API as better than the others, however upon reflection I think this would probably not have been possible either way. The reason for this is because the APIs are so complex, and the needs of people/projects vary so much. Therefore a specific API might be perfect for some cases, while unsuitable for others. From the results, I have been able to draw conclusions and recommendations to what use-cases each API might be best suited for. I think it would have been difficult to get more definitive results than this.

I think the layers that were defined are appropriate and representative for what should be the focus in an evaluation of this magnitude. I do recognize however that some of the methods might not have been ideal, but this is largely because of the APIs in question, and not necessarily because of the methods.

There are some aspects that the proposed evaluation methodology does not capture. This includes a proper and structured investigation of how it is to work with the APIs in practice. Chapter 3 gives a brief insight into this, but it is mostly theoretical. During the work with this thesis, I have also gained much experience with, and impressions of each API that is not presented in the thesis. This could possibly have been added in some way to give the reader a more practical impression of the APIs. Presenting this in a non-subjective manner is not easy however, and is one of the reasons that it has not been included.

It might also be argued that this thesis covers too much ground, and that by focusing on only one of these layers, the evaluation could be done in greater detail. This is a true and valid point, however I felt it was necessary to cover each of these layers in order to make the evaluation as thorough and representative of the state of the APIs as possible. This is also evident from the results, because they show that each API has its own strengths and weaknesses.

The following discussions focus on each of the methods applied to the different layers, and identifies possible improvements or shortcomings.

I think the project maturity metric, which was used for measuring the state of each projects management and technical infrastructure was sufficient for this evaluation. There are shortcomings with it however, that if corrected could have made the results from this method clearer. The scoring given for each question in this method does not reflect the importance of the aspect it investigates. If the scoring were affected by the importance of the tool, then perhaps the results would show a larger diversity. Some of the questions are not easily answered with a simple yes/no either. An example of this is the case of Ardor3D, where the user documentation was present, but was in such a poor state that it did not really qualify as a yes (although it received a yes due to other reasons, see Chapter 6 for more information). This shows that if another method was used, perhaps it would

better highlight some of the differences between the management of the projects. On the other hand, the APIs are very similar, so another method would likely yield similar results over all.

The metrics used for analysing the system architectures were effective, but there were some problems with the results for some of the APIs due to other factors. With Java 3D there were problems due to a limitation with the method used. The method did not handle the flat hierarchical structure of Java 3D very well. To clarify, it was able to calculate the metrics, however it was impractical to investigate the cause of the results because the DSM grew so large. The problem encountered with jMonkeyEngine3 was entirely caused by the packages included in the core of the API. The developers had included many packages that are not sensible to put there (networking and special effects like water, shadows etc.). This added noise to the results for jME3, and gave it a much better result than what was really representative of the state of the core. The method worked well with Ardor3D however, because it had a ”normal” core structure. The problems encountered here might indicate that this method might not be so suited for this evaluation. Had I used another method for investigating the architecture of the systems, I might have been able to get more informative and descriptive results, but that is not certain.

I think the tables created for the technical features and capabilities worked well, and are able to capture the most important aspects of what a 3D Scene Graph API should offer. It is worth noting however that the tables defined only contain a fraction of the possible features and capabilities that could have been investigated. The aspects covered in the tables might also be biased to a degree by what is important for IFE when choosing an API, but I argue that this does not affect the credibility of it as IFEs requirements are typical for industrial users. An improvement to the method could be to take the investigation even further. As it is now, it only looks at the mere presence of these features. The improvement would be to look at how, and how well implemented the features and capabilities are. This would be a huge task however, and would require a proper framework to define what is the ”right” way to implement each feature.

I think the performance benchmarks were able to capture the differences between the APIs very well. The results were also consistent between the benchmarks. There could be done some additional changes to the benchmarks however. A possible change could be to isolate the data collection. The benchmarks collects data for each frame in the rendering loop. By isolating the data collection to only the periods that are specific to the benchmarks, the results could be more focused. This can be done by for example only capturing data for the actual transparency sorting operation. Another change related to the data management could be to use a baseline for each benchmark. This is better explained through an example: One would first run the transparency benchmark without

any transparent objects. Then the benchmark would be rerun, this time with transparent objects. These two results could then be compared against each other. This would show different aspects, such as the object overhead and the actual performance impact of applying the different effects.

Some might also argue that it is unfair to compare the performance between jME3 which uses a shader based architecture, to Ardor3D and Java 3D which use the fixed function pipeline (FFP). While this could affect the results to some degree, I do not think it is the source of the large difference in performance. A factor that may have impacted the performance to a degree might be the graphic drivers. Most likely the graphic drivers are more optimized towards shaders than FFP. This is because recent versions of OpenGL are aimed more towards the use of shaders, deprecating much of the FFP.

Regarding the results from each of the different layers, they could have been investigated in even greater detail. The results are discussed and investigated to a certain degree, however an even more in-depth study could have been done. This would have given a deeper understanding of what is the source of the results, and it would give a greater insight into each of the APIs. Unfortunately I could not go into such detail in this thesis, because of the scope and time frame. It was not feasible to investigate all the results in such detail, because it would simply take too much time. This is something that could be extended upon in future work.

7.3 Future Work

- Other APIs could be included in the evaluation, including those mentioned in Chapter 3.
- Investigate other methods to use for the different layers in the proposed methodology. Other methods might have different angles or views, which might yield different results.
- Investigate the results from this thesis in greater detail, to give an even greater understanding of the results.
- Each layer in the proposed methodology could be extracted into their own standalone methodologies. These could include more than one method for the comparison of that layer. This would give a more detailed investigation of that layer.
- The work regarding the technical features and capabilities of the APIs could be extended in some ways:

    1. The scope of the questions could be extended, to include many other aspects of the APIs.

2. The investigation of this could be more in-depth, also considering how it is implemented, looking at factors such as efficiency, extendibility and more.

- The performance benchmarks could be extended in various ways:

  1. Analyse what the APIs do behind the scenes in greater detail, by for example profiling or intercepting the OpenGL calls.
  2. Measure the performance differently. An example is to run each benchmark first without the ”benchmark specific” parts, and then with it. A practical example would be to first run the transparency sorting benchmark without transparency, and then rerun it with transparency. Comparing these results would show object overhead, and how much performance is lost when doing the transparency sorting.
  3. Isolate data collection to the parts specific to the benchmarking code. This means collect only data for the actual ”benchmark specific” part, like the transparency sort. The benchmarks in this thesis collects data for the whole rendering of the frame.
  4. More complex and compound benchmarks that test various aspects of the API at the same time, opposed to the atomic benchmarks that were done in this thesis. This makes the benchmarks more reminiscent of actual applications.
  5. Investigate the difference between the garbage created by the APIs, and how this affect the performance.
  6. Implement the benchmarks in another programming language to compare performance of 3D Scene Graph APIs in Java opposed to other languages.

7.4 Conclusion

When I started the work on this thesis, I knew that it would be a difficult and complex task. In order to make the evaluation as thorough and detailed as necessary it quickly grew in complexity. Because of this, the work has been both very exciting, interesting and educational, but also at times difficult and daunting.
The research objectives set for the thesis were:

1. Identify or develop a suitable methodology for evaluating 3D Scene Graph APIs.
2. Evaluate the APIs using the chosen methodology.

With regard to these research objectives, the thesis successfully achieves both. A methodology for the evaluation has been defined, and the APIs have been evaluated with the proposed evaluation methodology.

The process of finding a suitable methodology for the evaluation was a long and challenging process. During my study I was unable to find any existing methodology that was sufficient for the scope of this evaluation. Therefore it was necessary to define a new methodology. It was important that this methodology was complex enough, so that it could cover the most important aspects of the APIs. With respect to this, the proposed methodology is satisfactory.

The proposed methodology was applied to a selection of the APIs available in Java[1]. The results from the comparison showed that no API was entirely superior to the others, but that there were important differences between them. Each API has different strengths and weaknesses, that makes it more suited for different tasks. From this, the results give clear indications to different cases where each API is best suited.

My study of previous work within this field, showed that there has been very little work done when it comes to comparing different 3D Scene Graph APIs against each other. The fact that I had to define a methodology for this evaluation is a testimony to that. Therefore the work in this thesis has great research value, and may help future work within this field.

The results from this thesis identify different areas of each API that could be improved. This can be very helpful for the developers of the APIs in question. In addition, this thesis holds a huge informational value for developers choosing an API to use, or those curious about the state of 3D Scene Graph APIs for Java.

In closing, this thesis contributes by defining a methodology to compare 3D Scene Graph APIs. This method was demonstrated by evaluating a selection of APIs. Both the methodology and the results of the evaluation encapsulate valuable information and research of interest to both developers and researchers alike.

[1] Java 3D, Ardor3D and jMonkeyEngine3.

References

[1] 3079. https://sites.google.com/site/3079game/. [Online: accessed 7-May-2012].

[2] A new focus: Ardor3D. http://blog.renanse.com/2008/09/new-focus-ardor3d.html. [Online: accessed 7-May-2012].

[3] Ardor Labs clients. http://ardorlabs.com/clients/. [Online: accessed 7-May-2012].

[4] Ardor3D used in museum exhibit. http://anders-kristoffer.dk/wiki/index.php?n=AdventuresInImmediateUnreality.AdventuresInImmediateUnreality. [Online: accessed 7-May-2012].

[5] Ardorcraft API. https://code.google.com/p/ardorcraft-api-examples/. [Online: accessed 7-May-2012].

[6] At the end of the tour... - Blog of Joshua Slack. http://blog.renanse.com/2008/08/at-end-of-tour.html. [Online: accessed 7-May-2012].

[7] Aviatrix3D. http://aviatrix3d.j3d.org/. [Online: accessed 7-May-2012].

[8] BSD License. http://www.opensource.org/licenses/bsd-license.php. [Online: accessed 7-May-2012].

[9] Caromble. http://www.caromble.com/. [Online: accessed 7-May-2012].

[10] City College of New York. http://math.sci.ccny.cuny.edu/. [Online: accessed 7-May-2012].

[11] Climategame 3.0. http://climategame.nl/. [Online: accessed 7-May-2012].

[12] COLLADA. http://collada.org/. [Online: accessed 7-May-2012].

[13] Fearless games. http://fearlessgames.se/tiki-index.php. [Online: accessed 7-May-2012].

[14] FlyingGuns. http://www.flyingguns.com/. [Online: accessed 7-May-2012].

[15] Forgotten Elements. http://www.forgottenelements.com/. [Online: accessed 7-May-2012].

[16] GNU Lesser General Public License. http://www.gnu.org/copyleft/lesser.html. [Online: accessed 7-May-2012].

[17] HELI-X. http://www.heli-x.net/. [Online: accessed 7-May-2012].

[18] Hex Engine. http://mtheorygame.com/projects/hex-engine/. [Online: accessed 7-May-2012].

[19] High-poly Benchmark. http://www.java-gaming.org/index.php/topic,20248.0.html. [Online: accessed 7-May-2012].

[20] Hostile Sector. http://www.hostilesector.com/. [Online: accessed 7-May-2012].

[21] HVRC CREATE. http://www.ife.no/departments/visual_interface_technologies/products/create. [Online: accessed 7-May-2012].

[22] Imaginary exhibition. http://www.imaginary2008.de/. [Online: accessed 7-May-2012].

[23] Institute for Energy Technology. http://www.ife.no/. [Online: accessed 7-May-2012].

[24] Institute for Geodesy. http://www.igg.tu-berlin.de/. [Online: accessed 7-May-2012].

[25] ISO/IEC 14772-1:1997 and ISO/IEC 14772-2:2004 — Virtual Reality Modeling Language (VRML). http://www.web3d.org/x3d/specifications/vrml/. [Online: accessed 7-May-2012].

[26] J-Ogg. http://www.j-ogg.de/. [Online: accessed 7-May-2012].

[27] Java 3D API Tutorial. http://java.sun.com/developer/onlineTraining/java3d/. [Online: accessed 7-May-2012].

[28] jBullet - Java port of Bullet Physics Library. http://jbullet.advel.cz/. [Online: accessed 7-May-2012].

[29] jME Forum discussion - "2.0.1 stable - 2.1 and 3.0 ahead". http://jmonkeyengine.org/2009/09/07/2-0-1-stable-2-1-and-3-0-ahead/. [Online: accessed 7-May-2012].

[30] jME Forum discussion - "jME3 project". http://jmonkeyengine.org/groups/development-discussion-jme3/forum/topic/jme3-project. [Online: accessed 7-May-2012].

[31] jME Forum discussion - "New branch in code and management". http://jmonkeyengine.org/groups/site-project/forum/topic/approved-new-branch-in-code-and-management?topic_page=1&num=15. [Online: accessed 7-May-2012].

[32] jME3 Beta 1 Game Contest. http://jmonkeyengine.org/2011/11/06/beta-1-game-contest/. [Online: accessed 7-May-2012].

[33] jME3 Beta 1 Game Contest Results. http://jmonkeyengine.org/2012/01/15/beta-1-game-contest-results/. [Online: accessed 7-May-2012].

[34] jME3 SDK - First Alpha Release. http://jmonkeyengine.org/2010/05/17/jme3-sdk-first-alpha-release/. [Online: accessed 7-May-2012].

[35] jME3 SDK Beta released. http://jmonkeyengine.org/2011/10/22/jmonkeyengine3-sdk-beta-released/. [Online: accessed 7-May-2012].

[36] jME's BSD License. http://jmonkeyengine.org/wiki/doku.php/bsd_license. [Online: accessed 7-May-2012].

[37] jMonkeyEngine forum, Free announcements. http://jmonkeyengine.org/groups/free-announcements/forum. [Online: accessed 7-May-2012].

[38] jMonkeyEngine Scene Graph. http://jmonkeyengine.org/wiki/doku.php/jme3:the_scene_graph. [Online: accessed 7-May-2012].

[39] JOGL. http://jogamp.org/jogl/www/. [Online: accessed 7-May-2012].

[40] jPCT. http://www.jpct.net/. [Online: accessed 7-May-2012].

[41] jPCT License. http://www.jpct.net/license.html. [Online: accessed 7-May-2012].

[42] jReality. http://www3.math.tu-berlin.de/jreality/. [Online: accessed 7-May-2012].

[43] Language Syntax (DirectX HLSL). http://msdn.microsoft.com/en-us/library/windows/desktop/bb509615(v=VS.85).aspx. [Online: accessed 7-May-2012].

[44] Lightweight Java Game Library. http://lwjgl.org/. [Online: accessed 7-May-2012].

[45] Magicosm. http://magicosm.net/. [Online: accessed 7-May-2012].

[46] Max the flyer 3D. https://play.google.com/store/apps/details?id=mk.grami.max. [Online: accessed 7-May-2012].

[47] mTheoryGame. http://mtheorygame.com/category/m-theory-game/. [Online: accessed 7-May-2012].

[48] Mythruna. http://mythruna.com/. [Online: accessed 7-May-2012].

[49] NASA Ames Research Center. http://www.nasa.gov/centers/ames/home/index.html. [Online: accessed 7-May-2012].

[50] NASA Ames Research Center robotics visualization tool used to track and guide rovers. http://ti.arc.nasa.gov/publications/2934/download/. [Online: accessed 7-May-2012].

[51] NASA Jet Propulsion Lab. http://www.jpl.nasa.gov/index.cfm. [Online: accessed 7-May-2012].

[52] NASA World Wind. http://worldwind.arc.nasa.gov/java/. [Online: accessed 7-May-2012].

[53] New BSD License. http://www.opensource.org/licenses/BSD-3-Clause. [Online: accessed 7-May-2012].

[54] NiftyGUI. http://sourceforge.net/projects/nifty-gui/. [Online: accessed 7-May-2012].

[55] Quake III Benchmark. http://www.java-gaming.org/topics/porting-quake-iii-to-java3d/11748/view.html. [Online: accessed 7-May-2012].

[56] Quake III Benchmarking. http://blog.renanse.com/2006/10/and-now-benchmarks-you-can-run.html. [Online: accessed 7-May-2012].

[57] Realistic sky system. http://jmonkeyengine.org/groups/user-code-projects/forum/topic/a-new-sky-system. [Online: accessed 7-May-2012].

[58] RenderMan Movies. https://renderman.pixar.com/products/whats_renderman/movies.html. [Online: accessed 7-May-2012].

[59] Retained Mode Versus Immediate Mode. http://msdn.microsoft.com/en-us/library/ff684178(v=vs.85).aspx. [Online: accessed 7-May-2012].

[60] Rolls Royce Marine next generation DP product. http://www.rolls-royce.com/marine/products/automation_control/dynamic_positioning.jsp. [Online: accessed 7-May-2012].

[61] Shader Stages (Direct3D 10). http://msdn.microsoft.com/en-us/library/windows/desktop/bb205146(v=vs.85).aspx. [Online: accessed 7-May-2012].

[62] Shapeways. http://www.shapeways.com/. [Online: accessed 7-May-2012].

[63] SINTEF. http://www.sintef.no/home. [Online: accessed 7-May-2012].

[64] SkyFrontier 3D. https://play.google.com/store/apps/details?id=com.fsilva.marcelo.skyfrontier. [Online: accessed 7-May-2012].

[65] Spaced. http://fearlessgames.se/tiki-index.php?page=FearlessGames. [Online: accessed 7-May-2012].

[66] Sweet Home 3D. http://www.sweethome3d.com/index.jsp. [Online: accessed 7-May-2012].

[67] Technical University of Berlin. http://www.math.tu-berlin.de/. [Online: accessed 7-May-2012].

[68] The Cg Tutorial: Chapter 1. Introduction. http://developer.nvidia.com/object/cg_tutorial_home.html. [Online: accessed 7-May-2012].

[69] The EDF Group. http://www.edf.com. [Online: accessed 7-May-2012].

[70] The Forester. http://jmonkeyengine.org/groups/user-code-projects/forum/topic/the-forester/. [Online: accessed 7-May-2012].

[71] The Portal. http://www.tu-berlin.de/3dlabor/ausstattung/3d-visualisierung/portal/. [Online: accessed 7-May-2012].

[72] Tygron Serious Gaming. http://www.tygron.nl/. [Online: accessed 7-May-2012].

[73] University of Munich. http://www.mathematik.uni-muenchen.de/. [Online: accessed 7-May-2012].

[74] Virtual Globe. http://www.virtual-globe.info/. [Online: accessed 7-May-2012].

[75] Voxel terrain. http://jmonkeyengine.org/groups/user-code-projects/forum/topic/terrain-side-project/. [Online: accessed 7-May-2012].

[76] VRML97. http://www.web3d.org/x3d/vrml/. [Online: accessed 7-May-2012].

[77] Web3D Consortium. http://www.web3d.org/realtime-3d/. [Online: accessed 7-May-2012].

[78] Xith3D. http://xith.org/. [Online: accessed 7-May-2012].

[79] Xj3D. http://www.xj3d.org/. [Online: accessed 7-May-2012].

[80] zlib/libpng License. http://www.opensource.org/licenses/Zlib. [Online: accessed 7-May-2012].

[81] C.Y. Baldwin and K.B. Clark. *Design Rules: The Power of Modularity*, volume 1. The MIT Press, 2000.

[82] H. Dhama. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software*, 29(1):65–74, 1995.

[83] J. Döllner and K. Hinrichs. A generalized scene graph. *Vision, modeling, and visualization 2000: proceedings: November 22-24, 2000, Saarbrücken, Germany*, page 247, 2000.

[84] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, 2001.

[85] K. Fogel. *Producing Open Source Software: How To Run a Successful Free Software Project*. O'Reilly Media, Inc., 2005.

[86] J. Gregory, J. Lander, and M. Whiting. *Game Engine Architecture*. AK Peters, 2009.

[87] R.M. Henderson and K.B. Clark. Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms. *Administrative Science Quarterly*, pages 9–30, 1990.

[88] A.D. MacCormack, J. Rusnak, C.Y. Baldwin, and Harvard Business School. Division of Research. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015, 2006.

[89] J. Marner. Evaluating java for game development. *Dept. of Computer Science, Univ. of Copenhagen, Denmark, March, http://www. rolemaker. dk/articles/evaljava*, 2002.

[90] M. Michlmayr. Software process maturity and the success of free software projects. In *Proceeding of the 2005 conference on Software Engineering: Evolution and Emerging Technologies*, pages 3–14. IOS Press, 2005.

[91] J. Mihm, C. Loch, and A. Huchzermeier. Problem-solving oscillations in complex engineering projects. *Management Science*, pages 733–750, 2003.

[92] R.J. Rost and J.M. Kessenich. *OpenGL Shading Language, Third Edition*. Addison-Wesley Professional, 2006.

[93] R.J. Rost and J.M. Kessenich. *OpenGL Shading Language, Third Edition*. Addison-Wesley Professional, 2006.

[94] R.W. Selby and V.R. Basili. Analyzing error-prone system structure. *Software Engineering, IEEE Transactions on*, 17(2):141–152, 1991.

[95] R.P. Smith and S.D. Eppinger. Identifying controlling features of engineering design iteration. *Management Science*, 43:276–293, 1997.

[96] R.P. Smith and S.D. Eppinger. A predictive model of sequential iteration in engineering design. *Management Science*, pages 1104–1120, 1997.

[97] M. Sosa, J. Mihm, and T. Browning. Product architecture and quality: A study of open-source software development.

[98] H. Sowizral. Scene graphs in the new millennium. *Computer Graphics and Applications, IEEE*, 20(1):56–57, 2000.

[99] P.S. Strauss and R. Carey. An object-oriented 3d graphics toolkit. *ACM SIGGRAPH Computer Graphics*, 26(2):341–349, 1992.

List of Figures

List of Tables

Appendix A

Glossary of Terms

API core: The most important parts of an API. Provides the core functionality of the API.

API (Application Programming Interface): A specification intended to be used as an interface by software components to communicate with each other. An API may include specifications for routines, data structures, object classes, and variables.

AWT (Abstract Window Toolkit): Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is now part of the Java Foundation Classes (JFC) — the standard API for providing a graphical user interface (GUI) for a Java program.

AVG FPS (Average Frames Per Second): The average number of frames per second.

Base memory usage: The memory usage of the application when "idle".

CAD (Computer-Aided Design): the use of computer systems to assist in the creation, modification, analysis, or optimization of a design. Computer Aided Drafting describes the process of creating a technical drawing with the use of computer software.

CAVE (Cave Automatic Virtual Environment): An immersive virtual reality environment where projectors are directed to three, four, five or six of the walls of a room-sized cube.

COLLADA (COLLAborative Design Activity): An interchange file format for interactive 3D

applications.

CSV (Comma Separated Values): Data stored in plain text, usually separated by a comma or tabular.

Deprecated functionality: In the process of authoring computer software, its standards or documentation, or other technical standards, deprecation is a status applied to features, characteristics, or practices to indicate that they should be avoided, typically because they have been superseded.

DSM (Design Structure Matrix): A matrix representation of a system or project, often created from the dependencies within the system.

FBO (Frame-Buffer Object): An extension to OpenGL for doing flexible off-screen rendering, including rendering to a texture. By capturing images that would normally be drawn to the screen, it can be used to implement a large variety of image filters, and post-processing effects.

FFP (Fixed-Function Pipeline): Fixed hardware pipeline with set algorithms.

FPS (Frames Per Second): The frequency (rate) at which an imaging device produces unique consecutive images called frames.

Frustrum culling: The process of removing objects that lie completely outside the viewing frustum from the rendering process.

Full screen pre/post-processing effects: Effects applied before or after the rendering to produce visual effects to the whole image.

GPU (Graphics Processing Unit): Used to refer to the graphics card and its processing unit (hardware).

GUI (Graphical User Interface): A graphical user interface for interacting.

HDR (High-Dynamic Range): A set of methods used in image processing, computer graphics,

and photography, to allow a greater dynamic range between the lightest and darkest areas of an image than current standard digital imaging methods or photographic methods.

Hierarchical Cyclicality: Cycles formed between code in a system, hierarchical structure is taken into account.

IFE (Institute for Energy Technology): IFE is an international research foundation for energy and nuclear technology, located in Norway.

Indie game development: Video games created by individuals or small teams without video game publisher financial support. Indie games often focus on innovation and rely on digital distribution.

Intrinsic Cyclicality: Cycles formed between code in a system, hierarchical structure is ignored.

Java: Programming language. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.

JGO: Java-Gaming.org - A internet forum about Java Game Programming with all the subjects that include (game logic, graphics, sound, ++). Url: `http://www.java-gaming.org/`

jME: jMonkeyEngine

JVM (Java Virtual Machine): A virtual machine that can execute Java bytecode.

Memory garbage: Garbage created by the application, for example temporary objects.

Multi-pass rendering: Rendering of a frame multiple times, applying different effects for each frame producing complex visual effects.

Occlusion: An object hidden (occluded) fully or partially behind another object.

OpenGL: A standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics.

OSS (Open Source Software): Computer software that is available in source code form.

Picking: The process of picking objects in three-dimensional space.

Pick ray: A ray cast in three-dimensional space with a origin and a direction.

Project Maturity: A metric for measuring a OSS projects healthiness.

Software patch: A piece of software designed to fix problems.

State sorting: Sorting the objects that are rendered based on OpenGL states required by the materials.

Swing: The primary Java GUI widget toolkit. It is part of Oracle's Java Foundation Classes (JFC) — an API for providing a graphical user interface (GUI) for Java programs.

SWT (Standard Widget Toolkit): A graphical widget toolkit for use with the Java platform, maintained by the Eclipse Foundation in tandem with the Eclipse IDE. It is an alternative to the Abstract Window Toolkit (AWT) and Swing Java GUI toolkits provided by Sun Microsystems as part of the Java Platform, Standard Edition.

System Stability: A metric that measures how likely is a change to code in a system is to affect other parts of the system.

Testbed: A platform for experimentation of large development projects. Testbeds allow for rigorous, transparent, and replicable testing of scientific theories, computational tools, and new technologies.

TPF (Time Per Frame): The time used rendering each frame.

Transparency sorting: Sorting of transparent objects so that the the rendered order is correct.

True stereoscopic rendering: Stereoscopic images created using Quad Buffering.

VBO (Vertex Buffer Object): An OpenGL feature that provides methods for uploading data (vertex, normal vector, color, etc.) to the video device for non-immediate-mode rendering.

Viewing frustrum: The region of space in the modeled world that may appear on the screen; it is the field of view of the notional camera. The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid (hence the name).

VRML (Virtual Reality Modelling Language): A standard file format for representing 3-dimensional (3D) interactive vector graphics, designed particularly with the World Wide Web in mind.

Z-buffer: A buffer for the management of image depth coordinates in three-dimensional (3-D) graphics, usually done in hardware, sometimes in software.