

3D Engine Design And Implementation

To Live And Die For Real-Time

Anssi Gröhn

June 4th, 2007

University of Joensuu

Department of Computer Science and Statistics

Master's thesis

Abstract

In the beginning, two research questions are formulated; which set of features are needed in generic 3D graphics engine library to provide basis for further development, and how the minimal set of features must be implemented in order to provide maximal reusability in different projects. The history and usage of accelerated real-time 3D graphics are presented briefly. The concept of 3D engine is defined to be a two-edged sword: it abstracts and accelerates. Three existing 3D engines, namely Crystal Space, Irrlicht, and OGRE, are compared to each other and it is shown that their features are rather similar. The scale of things that a 3D engine must handle is presented by explaining underlying mathematical concepts, operations and 3D rendering pipeline in a general level. The story is continued by introduction of 3D engine design, which is based on requirements of a 3D application, where core features were clearly visible. From these requirements, the functionality of most crucial elements is described using class diagrams and descriptions, covering also three used optimization methods, and finally giving a short description of required external libraries. The research questions are answered by describing the eight necessary features. After concluding that the implemented 3D engine library is quite satisfactory, further research directions pondered, which include a study of optimization techniques and their effects in a 3D engine. Also few future projects are mentioned, which could be done to add new features to the designed engine.

ACM classes (ACM Computing Classification System, 1998 version): I.3.5, I.3.6

Keywords: Three-dimensional graphics, 3D, OpenGL, 3D Engine, Real-time graphics

Foreword

Tremble, ye petrified monoliths of Science;
Let new dawn shine!
I question, forge new,
In a flow - great and sublime!

I fall, and stand again,
Taller than those who stood before,
Let us ascend those giants
That have become the lore.

Keepers of the old,
Wise of the past,
Be warned -
New reign. . . has come to last.

“May the fate hold the becoming of a storm.” ¹

In Joensuu 2007,
Anssi Gröhn / eNtity

¹-Quote is from the intro of PC game Return to Castle Wolfenstein, by Id Software (2001).

Contents

1	Introduction	1
1.1	The Motivation: An Oasis, A Mirage and The Cruel Hand of Economy	1
1.2	Summary of Chapters	2
2	Soft Real-time 3D Graphics	4
2.1	Where real-time 3D graphics is used and whence it came?	4
2.2	3D Engine: What is this abomination of nature?	7
2.3	Daddy, Are 3D Engines Real?	10
3	Hard Real-time 3D Graphics	19
3.1	The Mathematical Concepts	19
3.2	Coordinate Systems and Rendering Pipeline	26
3.3	Virtual Cameras	29
3.4	Defining 3D Model Data	31
3.5	Materials and Textures	32
3.6	Lighting and Shading	34
3.7	Visibility Culling Revisited	37
4	The Grand Design	43
4.1	Requirements and Analysis	43
4.2	Language Selection and Applied Design Patterns	47
4.3	Math Classes and Data Structures	48
4.4	OpenGL Renderer Class	50
4.5	Particle System	52
4.6	Animation System	54
4.7	Scene Handling and Visibility Determination	55
4.8	Materials, Textures and Lighting	55
4.9	Shaders	60
4.10	Optimizing the performance	62
4.11	Libraries utilized	66
5	I Have Created a Monster! Time to Flee, But Where?	68
	References	74
	Appendix 1: Source code of GSE	79

Glossary

3D	Three-dimensional
API	Application Programming Interface.
CAD	Computer Aided Design
CVE	Collaborative Virtual Environment
Direct3D	A 3D Graphics API for Windows platform.
FPS	Frames Per Second; a unit to measure rendering speed.
GPL	GNU Public License; a free software license endorsed by the Free Software Federation.
Hack	A clever programming trick.
LGPL	Lesser GNU Public License; a non-viral free software license endorsed by the Free Software Federation.
SDK	Software Development Kit; collection of tools and libraries that assist in developing an application.
VR	Virtual reality; artificial worlds inside a computer.

1 Introduction

In this chapter, I present the things that have lead to this point, and give a short reasoning why I am writing this thesis. I also show where my thesis stands on the field of computer science by formulating research questions, and explaining how they are answered. In the last section of this chapter, I give a brief summary of all chapters in this thesis.

1.1 The Motivation: An Oasis, A Mirage and The Cruel Hand of Economy

The driving force for the thesis and the 3D Engine has been the initial concept of an Avatar Mirror for the netWork Oasis project, as I described in my B.Sc thesis (Gröhn, 2006). The idea was to create a large wall-sized screen which would display users' virtual self in different forms when they pass by. Also the Internet connections were to be visualized using a planetary view with arcs connecting the connection origins to Oasis.

The netWork Oasis project unfortunately went through a series of budget cuts, which caused the Avatar Mirror to be canceled. However, a properly designed 3D rendering engine can be used in a variety of applications. The alternative visualization purpose in this case was a marketing demo representing the Oasis facilities, where the user was taken on a tour in the rooms.

The required graphics rendering software provided me a topic for the thesis. Various free and open-source 3D engines exist already today, so writing one from the scratch might sound laborous. A brief survey of the features on a set of different engines is performed in Section 2.3. Developing everything from scratch will gave me a total control of the engine. The major drawback of this approach was, obviously, the required time, but the advantages are worth the effort - adding a new feature will be easier than reading the source of another engine and writing a hack.

The process of developing a 3D engine is interests me also from a more practical point of view, since I plan to create a game that utilizes 3D graphics. And since the engine in this thesis provides a rendering component to be used in that game, I am hitting

two birds with one stone - my Master's degree and a future game project. But, I argue that there exists no master solution, an all-purpose 3D engine, which would perform well in every possible game. This is dictated by the fact, that there are differences in the implementation, for instance, for a real-time space simulator, where rendering of the star systems is important (Gamasutra, 2002) and a top-female model simulator, where realistic hair animation and rendering might play a great role (Byoungwon et.al, 2005; Bertails, et.al, 2005). Some features needed in one are not needed in other, and some features are necessary in order to produce 3D graphics in general, regardless the game type. By finding out common features, I can create a set of core functionality, which can be used as a basis for various other engines that are more specialized to specific situations. Therefore, as my first research question I need to ask **which set of features are needed in generic 3D graphics engine library to provide basis for further development?**

Once the minimal set of features is found, one must ponder how they must be implemented in order to provide maximal reusability in various engines; which leads to research question number two - **how the minimal set of features must be implemented in order to provide maximal reusability in different projects?**

The answer to first question is determined by performing a literature review on the subject, and evaluating popular open-source 3D graphics engines by cross-comparing their feature lists. The second question is answered by designing a 3D graphics library base for generic 3D applications.

1.2 Summary of Chapters

Chapter 2 explains very generally the concepts of 3D graphics and 3D engine. Three existing engines are presented and their features compared against each other.

Chapter 3 goes through the core complex mathematical and computer graphics concepts related to 3D graphics and 3D engines, accompanied by explanations of hardware rendering pipeline and visibility determination techniques.

Chapter 4 describes the requirements for the 3D engine to be developed, converts them to features and represents implementation details using class diagrams and descrip-

tions. Also three optimization techniques are introduced, which accelerate the operation of a 3D engine.

Chapter 5 goes through what has been done; the good, the bad and the ugly, and presents a directions for further research.

2 Soft Real-time 3D Graphics

This chapter gives a soft introduction to real-time 3D graphics and 3D engines. Their usage, and where they came from are covered. A couple examples of existing 3D engines are also be inspected, concentrating mainly to the features they provide.

2.1 Where real-time 3D graphics is used and whence it came?

Three-dimensional (3D) graphics have been used in Virtual Reality (VR) applications for quite a time now. Military training simulations and collaborative virtual environments (CVEs) use it in order to provide more immersive experience - one example of this is the training game Tactical Iraq, which helps American soldiers to speed up the acquisition of spoken Arabic (Losh, 2006).

The 3D graphics have also been used in many medical applications, such as visualizing the measured tomography or magnetic resonance data as a 3D surface (Lorensen & Cline, 1987) and demonstrating pre- and post-surgical appearance of the patient's face in order to lessen the anxiety and fear of the operation (Gross, 1998).

A newer idea is the 3D desktop, which, for instance, can "turn" inactive windows sideways that only some of their content is visible, leaving more space to the other windows. This way the inactive windows are somewhat visible and the changes in their content is still visible to the user (Sun Microsystems, 2007). This method has been adopted also into the recently released Windows Vista operating system from Microsoft (Microsoft.com, 2007).

Another way to use 3D on desktop was invented by Apple, which has integrated 3D acceleration into the desktop manager and windows are drawn using 3D primitives. This enables the use of impressive effects such as the "genie" shown in Figure 1. Lately, this has been implemented also in the open-source window manager Beryl (Beryl-project.org, 2007).

At the moment, the game industry is probably the most active user of 3D real-time graphics, and the focus seems to be in providing visually appealing games. This is accomplished by including more detailed game characters, environments, effects and weapons into games. An example of this can be seen from the screenshots of pop-



Figure 1: The genie effect in action in MacOS X (Apple, 2007).

ular first-person shooter PC-games in Figure 2. The basic idea in those games has remained much the same, but graphics have received more detail. The quest for better (looking) games seems to be endless and new algorithms which would produce a so-called “quantum leap”² forward in performance would probably be more than desired in the game industry.

It can be said that the technology behind the modern real-time 3D graphics was pioneered by company called Silicon Graphics (SGI), which produced the world’s first 3D graphics workstation, IRIS 1400. In his article, Baum (1998) classifies hardware into three generations, where generation means the targeted set of features for which the system runs with full performance. In the case of first workstation, the feature was flat-shaded polygons³. The second generation implemented new accelerated Gouraud shading method and used Phong-lighting². The third generation in 1992 brought texture mapping and full-scene antialiasing. Baum also stated, that the texture mapping was considered to be a technological hype and was believed to have no market. The success of game industry has proven this otherwise.⁴ The consumer-grade systems fall into the third-generation category - they render textured and antialiased polygons with high speed. The SGI also developed an API for the handling the underlying hardware in more abstract manner. The product was called OpenGL and it was released under

²The term expresses a massive technological advancement or innovation.

³The concepts of polygon, shading and lighting are examined in Section 3.6

⁴The quote in the article of Baum (1998) “It’s a cool feature, but there’s no market” should be framed and put next to the infamous quote of Bill Gates regarding the amount of required memory in computers.



Figure 2: Screenshots from from 3D games from three different generations. First row: Quake 1 and Doom, Second row: Quake II and Unreal Tournament, Third row: Unreal Tournament 2007 and Doom III.

open-source license the very same year 1992 when the third-generation systems were released (SGI,2007).

2.2 3D Engine: What is this abomination of nature?

Modern applications use the underlying hardware with *3D graphics API*, which is a source code interface for computer program to instruct hardware to execute commands related to 3D drawing. In practice, it means that it abstracts all the technical details into more programmer-friendly form. At the moment, two dominating 3D APIs exist: The OpenGL created by SGI and Direct3D created by Microsoft. The most significant difference is that OpenGL is platform-independent and Direct3D is only available for Microsoft operating systems.

So, 3D graphics can be produced using 3D graphics APIs. The APIs operate on commands like “put this point here”, “put that point there”, “set drawing color to white”, “draw a line between this and that”, etc. Using the combinations of those commands, a programmer is able to generate quite complex scenes and images. This, however, is no different from static 3D graphics. 3D animation movies are done by setting points, defining the colors, drawing lines and surfaces and finally storing the ready image into a file, repositioning the points and going through a predefined sequence of movements until we have the final animation. But real-time 3D graphics are just this, defining points and animation, right? Not quite. The usual animation movie is as static and predictable as a single image - there exists no freedom of choice over what will happen next. The real-time 3D graphics are needed when user does not know (exactly) what will happen next, for instance in terms of position and shape change, and there exists a need to constantly provide a visual representation of the situation. So the production of real-time graphics is, in extremely simplified form, this:

```
While( True )  
    Get current position of points;  
    Draw things using points, lines, triangles and colors;  
Repeat
```

But the humans tend to think in larger terms than points, lines and colors - so it would be quite aggravating to work through a view of several hundreds of thousands of points and hundreds of colors. This can be solved by abstracting the underlying 3D graphics

API even further, into a level where programmer is able to handle greater wholes, objects which represent cars, houses, cities - instead of points, lines and colors, which create a visual representation of those objects. This abstraction is referred to as the *3D engine*, and in this context, objects are called as *3D models*. The 3D image drawing process, is known as *3D rendering*. As a stand-alone software, the 3D engine is useless since it needs something to draw, and it will always be used in conjunction with other software to visualize something.

Probably the most common software which has a 3D engine as its component is a *game engine*, which is a collection of code used to build a gaming application (Sherrod, 2007: p.4). That being said, the *3D engine* can be defined as a reduced game engine, which only provides the real-time 3D rendering capabilities while excluding other functionality needed in games. In this light, more accurate term would be *3D rendering engine*, but the simpler form is used instead. It should also be noted, that 3D engines are not exclusively for games, they can be used in any software which requires three-dimensional visualization.

In addition to drawing commands targeted at larger object structures, 3D engines must deliver very large object sets on screen fast, in order to maintain the illusion of continuous movement. Performance-wise speaking, there are four goals: higher resolution, more frames per second, more objects and more detail on screen (Akenine-Möller & Haines, 2002: p.345). The limits of the graphics hardware are met very fast by increasing the detail of the objects. For example, if a space shuttle is modeled into detail where every nut and bolt has its own 3D model, the amount of data grows to great in order to render it in real-time. Therefore, the rendering must be accelerated by reducing the amount of renderable data, which means that the images can be rendered faster and the illusion of movement prevails even in the case of highly detailed and numerous objects on screen. This makes 3D engine as a two-edged sword; it abstracts and accelerates. Some acceleration methods for doing this are listed next.

Visibility culling

The *visibility culling*, or *visibility determination* is a method which determines the visibility of renderable objects and reduces invisible objects from the herd of renderable objects, leaving only those that contribute to the final image (Lengyel, 2004: p.217). The implementation details of this method are examined in Section 3.3. There are dif-

ferent variations of this method, which according to Akenine-Möller & Haines (2002) are backface and clustered backface culling, hierarchical view frustum culling, portal culling, detail culling and occlusion culling. The most common of these are backface and hierarchical view frustum culling, which can be applied to any set of renderable objects. The rest of them are more useful in special situations. The details of these methods are covered in Chapter 3.

Levels of detail

The *levels of detail* is a method which reduces the number of drawn items as the size of the drawn object on screen diminishes. Since the underlying 3D rendering mechanism is forced to process all of the data which is sent to it, the computer must do (nearly) the same amount of work when the object covers only a pixel of the screen or all of the pixels. In this situation, using a simplified version of the renderable data lessens the burden of drawing dramatically and speedup is gained without sacrificing visual quality (Eberly, 2001: pp.359-360; King, 2000).

Bounding volumes

Bounding volumes are volumes (geometrical objects, such as spheres, boxes and ellipsoids) which encapsulate renderable geometry (Akenine-Möller & Haines, 2002: p.347). The process of checking each triangle for visibility culling is laborious and not suitable for real-time rendering in larger scale. The bounding volumes, however, are much lighter in that sense. For example, if an object consists of n triangles, the number of comparisons needed to determine visibility of an object requires at least $O(n)$ steps. But comparing the visibility of a sphere takes only $O(1)$ steps. Testing a sphere visibility might take more time if the n is very small, but this is very unlikely scenario.

Scene Handling

The models which are to be displayed might have some relations with each other, such as a wheel, which belongs to a car, bolts belonging to the wheel, etc. Keeping track of these relations and changing them is laborious without a uniform approach. A *scene graph* is a data structure designed for this purpose. It arranges the logical and spatial

relationships of the graphical scene in some manner (Sherrod, 2007). The scene graph can be, but is not restricted to be, implemented as an n -tree, and is more of a general data structure. According to Sherrod (2007), scene graph is usually defined by the application for which it is used, since the scene graph may contain bounding volumes or positional data, or simply be a highly abstract structure representing only the logical relationships of objects.

The previous paragraphs have given an overview what 3D engines need - an underlying 3D graphics API, and acceleration methods for reducing the amount of renderable data. The following section brings more tangible matters into a focus, taking a look into existing 3D engines.

2.3 Daddy, Are 3D Engines Real?

The feature list of the existing 3D engines at large is intimidating at least from the perspective of a person writing his/her own engine. But, in most cases the engines which have been around longest, have turned into game engines with support for physics simulation, sound and scripting required in games. Keeping this diversification in mind, I present three 3D engines with their features and cross-compare them.

The selected engines are available in the SourceForge, which is a free service, dedicated to hosting different open-source projects. At the time of writing this document, the SourceForge offers somewhat hefty bunch of 570 different 3D engines. Most of them are probably put into web in the hope someone else might also be interested in them. The three engines listed here have been selected because of their maturity and widely adopted user-base - which clearly indicates that they are doing something right. There exists, of course, other widely adopted and highly efficient engines which would have been as good a choice than these ones. However, the point of this paper is not to provide a complete feature-review of all possible engines out there, but a generic list which can be used in comparison when a new engine is designed.

Performance of 3D engines is usually measured in *frames per second* (FPS), which stands for the number of images displayed in a second and is also known as the *frame rate* (Sweet & Wright, 2000: p.607). Akenine-Möller & Haines (2002: p.1) also state that the interactivity (and real-time-ness) starts from 6 FPS, and after 72 FPS the changes in frame rate are almost indetectable. It should be noted, that the perfor-

mance of presented engines is not subject to inspection - it would be viable only in case of predefined set of features which would have to perform well in a specific scenario.



Crystal Space is a project started by Jorrit Tyberghein around 1997 (Linux.com Article DB, 2007). The spark which gave birth to it came from the era of Intel 486s and popular PC games Doom and Quake. The Linux.com interviewed Mr. Tyberghein about the development process of Crystal Space, and according to the interview, he was inspired by the efficient graphics drawing of the games, which triggered a six-month research session towards 3D graphics and finally led to comp.graphics.algorithms newsgroup, where portal culling algorithm was discussed. He found this algorithm to be easy enough for him to implement it, and began developing his own engine with portal culling. (Linux.com Article DB, 2007)

After two months of development, the Crystal Space was born; a working 3D engine with six degrees of freedom, lighting, mipmapping, moving objects, scripting and other features. When he found that he lacked time to work with his project, he decided to open it to the public instead of "letting it rot on hard disk". The community existed at first on mailing lists. During the last 10 years Crystal Space has become much more than a 3D engine - a cross-platform software development kit (SDK) for real-time 3D graphics. It is distributed under LGPL, which enables it to be used also in commercial games as long as the changes to the original code are released if the product is distributed. (Linux.com Article DB, 2007; GNU Project, 2007)

The development of Crystal Space project is split into four sections (Crystal Space, 2007):

1. The Crystal Space SDK: The main rendering engine.
2. Crystal Entity Layer(CEL) : The scene entity management library built on top of Crystal Space SDK. It completes the framework for creating games, for instance.

3. CEL Start : An environment for self-contained game packages which supports scripting to aid in the game development.
4. Crystal Core : A demo game utilizing the Crystal Space SDK.

The Crystal Space SDK is used in various games and modeling tools, and it supports Linux, Mac OS X and Windows platforms. Its renderer system is built using OpenGL graphics API.



Irrlicht is a german word, meaning “will-o’-the-wisp”. It means some kind of fairy-like creature living in the vicinity of swamps. The project is an open source high performance real-time 3D engine written for C++ and .NET languages. It supports four different renderer APIs; Direct3D, OpenGL, its native software renderer and the Apfelbaum software renderer. The engine runs on Windows, Linux, Mac OS X and Sun Solaris. (Irrlicht Engine, 2007)

The Irrlicht is used in over 30 different projects and it has an active community. It has several language bindings, allowing it to be used with Java, Python, Ruby, Basic, and so forth - a feature which will certainly increase the adoption rate.

The project is licensed under zlib license, which allows it to be used in commercial projects, even without mentioning the Irrlicht itself. Also the code itself can be used as a base of a commercial product, as long as no claims are made that the original software was written by the party using the code (ZLib, 2004).



The project was started by Steve Streeting in England, somewhere around 1999. He was developing an object-oriented library for Direct3D, called DIMClass at the time.

He realized that the library itself had become abstracted enough from the underlying graphics API and began an ambitious planning of OGRE, new object-oriented API and platform independent 3D engine. After registering the project in the SourceForge at 25th of February 2000, the passed years have brought various features and improvements to the OGRE. The latest version at the time of writing this paper is 1.4.0RC1, which was released in 11th of February 2007.

OGRE stands for Object-oriented Graphics Rendering Engine. Like the name implies, the mission of the project is to create a cohesively designed framework to be used in top-notch graphic solutions. It is not designed to be used in games in particular, but it has been used to create games as well.

The OGRE project stresses the flexible design and documentation over long feature list. It does not provide physics modeling or collision detection, but relies more on the framework where such features can be integrated using third-party modules specifically designed to be used in those tasks. This might make some features a bit harder to implement because of the need to connect an external library to OGRE, but on the other hand, it allows programmers to use their own, possibly more efficient algorithms in special cases. OGRE is released under the LGPL and can be used in commercial projects as long as the changes to the original code are released if the product is distributed. The OGRE supports Linux, Mac OS X and Windows platforms. (OGRE, 2007; GNU Project, 2007)



Table 1 shows a set of features which are implemented in these three projects and how the features relate to each other. The license under which the engines are released has effect to the projects themselves. Too restrictive license might disallow the usage of the library in commercial projects, for instance.

Table 1 lists supported platforms because of application portability. If libraries have already been ported to the desired platform, very little effort is usually required in order to port the whole 3D project for that platform. Projects can, of course, contain other code which is platform-specific and therefore portability is not only restricted by the 3D engine.

Table 1 also includes a shader support section, which is important if special graphics effects are to be included in the project. In games of high visual quality, this is almost a necessity. The *Image formats for textures* section is included, because the image support is necessary for including specific materials to 3D models. The textures types are necessary for special effects and it is assumed that each of the engines supports at least the usual texture format, rectangular color image with some constraints on width and height. The animation support is essential for 3D engine, since without movement the real-time rendering serves little purpose, which was explained in Section 2.2.

The acceleration algorithms are also covered in Table 1, since they are needed for complex scenes to be rendered sufficiently fast, and the spatial data structures speed up the rendering of very large models. One of the most important things is the support for different 3D model files. There exists a plethora of software which produce models and save it in different formats. In order to support the tool that was selected, the engine must have support for the file format it uses, which are covered in the *3D File formats supported* section. The special effects can be used to create visually stunning sceneries. This feature is also very important to gaming applications. The “other” section of the Table 1 includes features which were hard to classify.

Feature	Supported in engine		
	Crystal Space	Irrlicht	OGRE
License			
LGPL	X	-	X
ZLIB	-	X	-
Platforms			
Windows	X	X	X
MacOS X	X	X	X
Linux	X	X	X
Sun Solaris/SPARC	-	X	-
Graphics API			
OpenGL	X	X	X

Feature	Supported in engine		
	Crystal Space	Irrlicht	OGRE
Direct3D	-	X	X
Software Renderer	X	X	-
Vertex and fragment shader support			
low-level assembler	X	X	X
Nvidia Cg	X	-	X
Direct3D HLSL	-	X	X
OpenGL GLSL	-	X	X
Image formats for textures			
Adobe Photoshop (.psd)	-	X	-
JPEG File Interchange Format (.jpg)	X	X	X
Portable Network Graphics (.png)	X	X	X
Truevision Targa (.tga)	X	X	X
Windows Bitmap (.bmp)	X	X	X
Graphics Interchange Format (.gif)	X	-	-
Zsoft Paintbrush (.pcx)	-	X	-
DirectDrawSurface (.DDS)	X	-	X
Textures			
1D	-	-	X
2D	X	X	X
Compressed textures	X	-	X
Video textures	X	X	X
Animation			
Skeletal animation	X	X	X
Keyframe animation	X	X	X
Acceleration algorithms			
Mesh LOD	X	X	X
Material LOD	-	-	X
Occlusion culling	X	X	X
Frustum culling	X	X	X
Portal culling	X	-	X
Spatial data structures			
Octree	X	X	X
BSP	X	X	X

Feature	Supported in engine		
	Crystal Space	Irrlicht	OGRE
3D File formats supported internally			
3D Studio meshes (.3Ds)	X	X	-
B3D files (.b3D)	-	X	-
Alias Wavefront Maya (.obj)	X	X	-
Cartography shop 4 (.csm)	-	X	-
COLLADA (.xml, .dae)	-	X	-
DeleD (.dmf)	-	X	-
FSRad oct (.oct)	-	X	-
Microsoft DirectX (.x)	-	X	-
Milkshape (.ms3D)	-	X	-
Cal3D (.cal3D)	-	-	-
My3DTools 3 (.my3D)	-	X	-
Pulsar LMTools (.lmts)	-	X	-
Quake 3 levels (.bsp)	-	X	-
Quake 2 models (.md2)	X	X	-
MDL models (.mdl)	X	-	-
ASE format (.ase)	X	-	-
PovRay format (.pov)	X	-	-
OGRE format	-	X	X
Crystal Space format	X	-	-
Converters for popular formats	X	-	X
Special effects			
Particle Systems	X	X	X
Skyboxes	X	X	X
Billboarding	X	X	X
Fog	X	X	X
Bump mapping	X	X	X
Parallax mapping	X	X	X
Cube mapping	X	X	X
Sphere mapping	X	X	X
Dynamic shadows	X	X	X
Light maps	X	X	X
Transparency	X	X	X

Feature	Supported in engine		
	Crystal Space	Irrlicht	OGRE
Dynamic lights	X	X	X
Other			
Scene management	X	X	X
GUI system	X	X	X
Light virtualization	X	X	X

Table 1: The cross-comparison of features in Crystal Space, Irrlicht and OGRE.

In Table 1, the license section shows that the licenses allow each of the engines to be used in any commercial or non-commercial project. The OGRE and Crystal Space, however, require that the source code of the engine is distributed with the project. The Irrlicht bears no such restriction.

It can be seen in the platforms section of Table 1, that the most common platforms (Windows, MacOS X, Linux) are supported by all of them. The graphics API section shows that the Crystal Space does not support Direct3D library, but it can be used on Windows platform since the OpenGL library is ported also to it. The software renderer might be useful in some situations, but usually 3D acceleration is available for any relatively modern graphics card.

Shaders allow very versatile lighting and shading effects to be created. There are very little differences in the shader support. The *Vertex and fragment shader support* section in Table 1 show that low-level assembler shaders are supported in all of them. Nvidia Cg library, which can generate low-level assembler code for shaders, is selected by Crystal Space and OGRE. Only OGRE supports all four shader formats (assembler, Nvidia Cg, HLSL and GLSL).

Image formats support is more than adequate, the most popular formats (png, jpeg, tga, bmp) are supported by every engine, which can be seen in section *Image formats for textures*. Texture support is also quite similar, the standard texture format and video textures are supported in each engine. Crystal Space and Irrlicht, do not seem to support 1D textures, but since 1D textures are only useful in special cases, the lack of it causes negligible effects. There is a lack of support for compressed textures in Irrlicht.

Or if the support exists, I was unable to find any reference to it. The section *Animation* in Table 1 shows that animation is supported by skeletal animation and the older keyframe interpolation in all three engines. *Skeletal animation* means that certain parts of a 3D model are attached to a bone, which can be animated. When bone is moved, the attached part of a 3D model is moved also. *Keyframe interpolation* means that there exists various different 3D models, which represent same object in different postures, and transitions between postures is interpolated in some manner. Acceleration algorithms exist in each engine, where the most distinctive feature is Material LOD, which apparently is only implemented in OGRE. Spatial data structures are also supported in every engine.

It might seem that in the section of supported 3D file formats the Irrlicht would be the sovereign ruler, but the number of supported formats does not necessarily mean superiority. OGRE and Crystal Space have chosen a different approach to file formats by using exporters to convert needed formats into their own format, although Crystal Space supports some formats also directly. Irrlicht pursues compatibility by including support for all formats in direct manner.

The special effects which were selected for inspection, were implemented in all three engines, no surprises there. All engines contain some kind of a GUI rendering system, an arbitrary number of lights can be added to the scene and all have scene management for more complex visualization needs.

The comparison can be concluded by saying that all engines are virtually alike in the perspective of rendering engine. No drastic differences exist and the basic functionality is there. The presented table could also be used to compare base features of other engines, if such need should arise.

So far, the real-time 3D graphics concepts have been handled in a very general level. But in order to know how 3D engines are implemented, more detailed information about underlying mathematics, and 3D computer graphics is required. In the following chapter, focus is shifted to more in-depth workings of 3D engines.

3 Hard Real-time 3D Graphics

Following paragraphs introduce detailed information, which is required to understand the nature of techniques related to real-time rendering of 3D models. The included topics cover mathematical concepts, different coordinate systems, rendering pipeline functionality and virtual cameras. Also defining 3D model data, materials, textures, and workings of lighting and shading are explained. Finally, visibility determination is explained by using bounding sphere as an example. Although this is also the “dreaded math chapter”, mathematical concepts are not discussed very extensively to avoid too detailed description of the subject - more detailed information and proofs can be found on any linear algebra textbook.

3.1 The Mathematical Concepts

The most basic tool can be considered to be a *vector*, which in our case is a n -tuple of real numbers and can be written as

$$\mathbf{v} = (v_1, v_2, \dots, v_n)$$

where the v_i are the *components* of the vector. A vector with n components can be used to represent a coordinate, a direction and velocity in n -space. In terms of 3D computer graphics, the n is usually 2, 3 or 4 (Lengyel, 2004: p.12). The *3D space*, where our vectors exist, is a vector space with linearly independent components forming an orthonormal basis.

The most used operations on vectors are sum, difference, scalar multiplication, magnitude, component-wise product, dot product and cross product. The *sum* ($\mathbf{p} + \mathbf{q}$) and the *difference* ($\mathbf{p} - \mathbf{q}$) are component-wise operations of two vectors. The *scalar multiplication* ($\mathbf{p} * \alpha$) is an operation between a vector and a real number, where each component of a vector is multiplied by a real number. The *magnitude* $\|\mathbf{p}\|$ is defined as the square root of the sum of the square of each component. It is also referred as the *norm* or the *length* of a vector (Lengyel, 2004: p.13)

The *component-wise product* ($\mathbf{p} \circ \mathbf{q}$) produces a new vector, of which components are retrieved by multiplying each component of \mathbf{p} with corresponding component of \mathbf{q} .

The *dot product* ($\mathbf{p} \cdot \mathbf{q}$), which is also called as *scalar product* or *inner product*, gives a measure of the difference between the directions of two vectors. It is calculated by sum of the products of each component, or by the product of the magnitudes and cosine of the angle between the vectors. The *cross product*, also known as the *vector product*, produces a vector perpendicular to both of the given vectors. The cross product follows the *right-hand rule*, which can be interpreted as follows: When the fingers of the right hand point along vector \mathbf{p} and palm of the hand points along vector \mathbf{q} , then the thumb points to the direction of the cross product $\mathbf{p} \times \mathbf{q}$. (Lengyel, 2004: pp.14-26)

A *matrix* is an array of numbers with specific number of columns and rows. A matrix is referred as $m \times n$ matrix if it has n rows and m columns. If the number of columns and rows are the same, the matrix is said to be a *square matrix*. A vector can be represented as a $1 \times m$ or $n \times 1$ matrix. The most used operations on matrices are sum, difference, multiplication, transposition, determinant and inverse. The determinant and inverse exist only for square matrices. Like with vectors, the *sum* and *difference* of matrices are element-wise operations. The *multiplication* can be defined so that the entry on i th row and j th column of \mathbf{AB} is the dot product of the i th row of \mathbf{A} and the j th column of \mathbf{B} . The *transposition* is the mirroring of the elements in respect to the matrix diagonal. The value in the i th row and j th column swaps places with value in j th row and i th column. The *determinant* of matrix \mathbf{M} ($\det \mathbf{M}$) can be said to be a scalar value derived from the elements in the matrix. The elementary definition exists for $n = 2$, and is recursively applied for any $n \times n$ matrices. (Lengyel, 2004: pp.33-65)

The *inverse* (\mathbf{M}^{-1}) of a matrix \mathbf{M} , is such matrix that $\mathbf{M}^{-1}\mathbf{M} = \mathbf{M}\mathbf{M}^{-1} = \mathbf{I}$. The inverse does not exist for every matrix, and those matrices without inverse are called *singular matrices*.

A *point* in 3D space is an arbitrary location represented by a vector with 3 components.

$$\mathbf{p} = (p_x, p_y, p_z)$$

A *line segment* is an one-dimensional finite entity which connects two points.

$$\mathbf{L} = (\mathbf{p}_{start}, \mathbf{p}_{end})$$

A *line* or *ray* is a infinite entity which travels through a specific point and points into specific direction. The ray is specified with a point and a direction vector.

$$\mathbf{R} = (\mathbf{p}_{pos}, \mathbf{v}_{dir})$$

A *plane* is defined by a point \mathbf{p} and a direction vector \mathbf{n} , where point lies in the plane and direction vector is perpendicular to the plane. The vector \mathbf{n} is referred as the *normal vector*. The plane equation is

$$Ax + By + Cz + D = 0,$$

where A, B and C represents the component values of the normal vector and condition $D = -\mathbf{n} \cdot \mathbf{p}$ holds. In the case of 3D real-time graphics, the normal vector is usually normalized to unit length. In this case the equation

$$d = \mathbf{n} \cdot \mathbf{q} + D$$

gives a signed distance d from plane to point \mathbf{q} and can be used in variety of situations, including visibility culling. (Lengyel, 2004: pp.105,107)

A *polygon* is a closed figure with n sides. It is defined by n points, which in this context are also called as *vertices* (Wolfram Mathworld, 2003). The order, in which vertices of the polygon are defined, is called as the *winding order*. The winding order defines the direction the polygon is facing. There exists two ways to define it, clockwise meaning backside or clockwise meaning frontside - both conventions are used.

Basic operations for vertices can be considered to be translation, rotation and scaling, which are defined below (Eberly, 2001: pp.8-9)⁵. The combination of these is called as a *transform*. All of these operations are performed by multiplying the matrix representation of vectors with the matrix representation of the operation(s).

A *translation* \mathbf{T} represents a move of a spatial location. The operation is inversed simply by negating the translation values. Translation can be performed for vertices and other coordinates, and is done with following matrix:

$$\mathbf{T}(t) = \mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{ with inverse } \mathbf{T}(t)^{-1} = \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

⁵There exists also shearing, but it has little usage in most 3D engines.

A direction vector, however, should not be affected by a translation. Vectors in 3D space usually consist of 3 components. The method of separating a position vector (a_1, a_2, a_3) from a direction vector pointing from origo towards point (a_1, a_2, a_3) , is to use a 4-component representation. The fourth component of a position vector is 1 and in the case of a direction vector it is 0. In this manner, the translation operation works correctly for both cases. When 3-vectors are converted into this 4-component form, they are said to be in *homogeneous coordinates*. A graphical illustration of translation operation is shown in Figure 3. By multiplying the point p and direction vector d by translation matrix \mathbf{T} , following results are produced.

$$\mathbf{p}' = \mathbf{T}(\mathbf{t}) * \mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \dots = \begin{pmatrix} p_x + t_x * 1 \\ p_y + t_y * 1 \\ p_z + t_z * 1 \\ 1 \end{pmatrix}.$$

$$\mathbf{d}' = \mathbf{T}(\mathbf{t}) * \mathbf{d} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} d_x \\ d_y \\ d_z \\ 0 \end{pmatrix} = \dots = \begin{pmatrix} d_x + t_x * 0 \\ d_y + t_y * 0 \\ d_z + t_z * 0 \\ 0 \end{pmatrix} = \mathbf{d}.$$

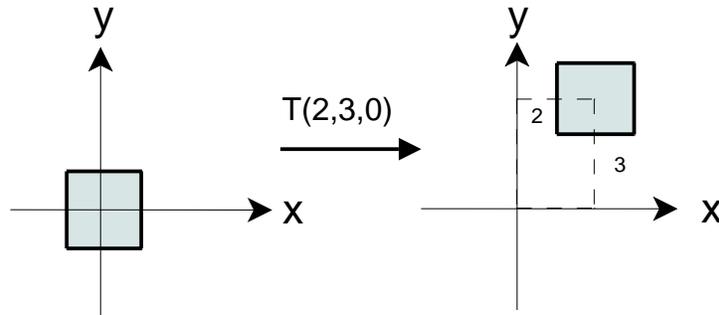


Figure 3: An example of translation operation where quad is moved from origin to 2 units along X-axis and 3 units along y-axis

Rotations are always performed left-handed around the origin, $(0, 0, 0)$. If rotation is desired to be made around arbitrary point P , the vertices must be first translated by $-P$, rotated, and translated by P . An illustration of rotation operation can be seen in Figure 4, where an object is rotated 90 degrees over z -axis pointing up from paper surface. The inverse of rotation matrix is the transpose of it. Rotations \mathbf{R}_x , \mathbf{R}_y , \mathbf{R}_z along axes X , Y and Z by angle ϕ are performed with following matrices:

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) & 0 \\ 0 & \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y(\phi) = \begin{pmatrix} \cos(\phi) & 0 & \sin(\phi) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos(\phi) & -\sin(\phi) & 0 & 0 \\ \sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

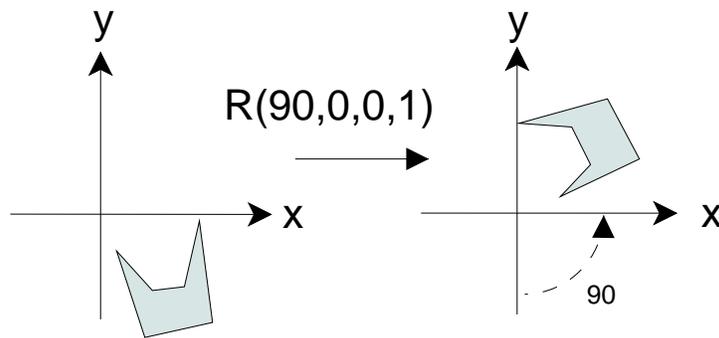


Figure 4: An example of rotation operation.

The scaling operation is used to enlarge or shrink an entity along x , y and z axis. If the scaling factor for each axis is the same, the operation is said to be *uniform* and *non-uniform* otherwise. If one or three of the scaling factors are negative, the operation yields a *mirror matrix*, which is also called as *reflection matrix*. The mirror matrix can reverse the winding order of the vertices, which can lead to incorrect results⁶. The mirror matrix must be treated as a special case in order to preserve the winding order. (Akenine-Möller & Haines, 2002: p.30).

⁶This is true when lighting and/or face culling are enabled, more on the subject in Sections 3.6 and 3.7.

The scaling matrix is $\mathbf{S}(\mathbf{s})$:

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \text{ with inverse } \mathbf{S}^{-1} = \begin{pmatrix} \frac{1}{s_1} & 0 & 0 & 0 \\ 0 & \frac{1}{s_2} & 0 & 0 \\ 0 & 0 & \frac{1}{s_3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

By means of linear algebra, it can be proven that the matrix multiplication is not always commutative (Lang, 1986). From this can be concluded, that translation, rotation and scaling operations are not always commutative either (Figure 5).

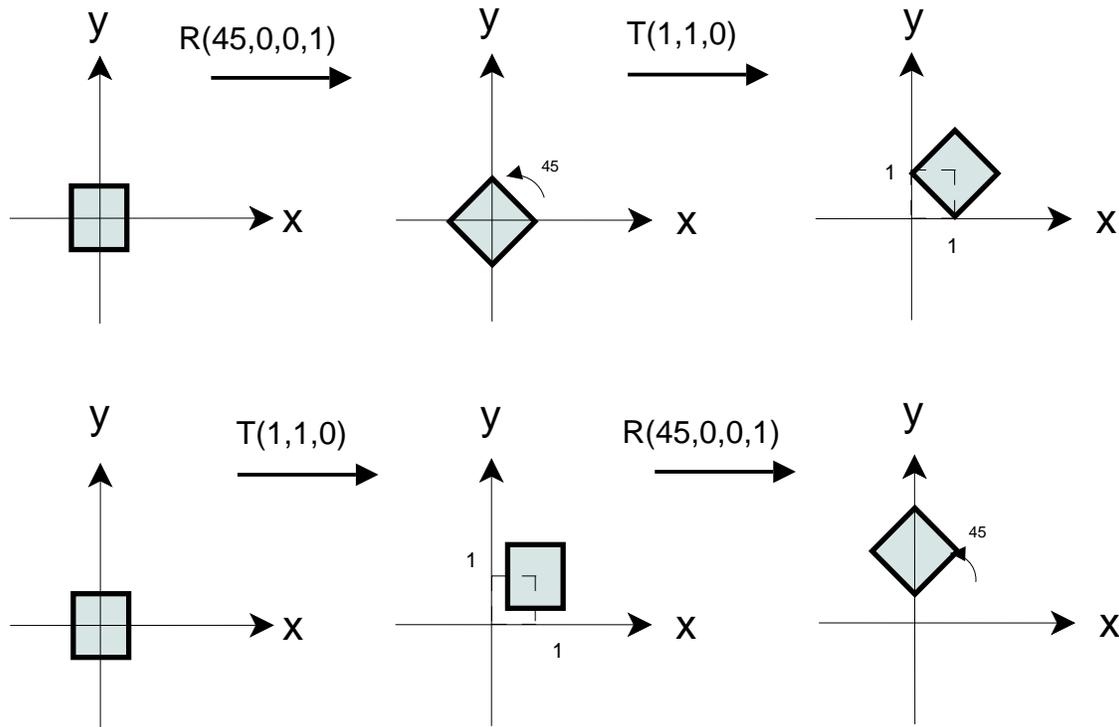


Figure 5: An example how order of operations has effect on object.

Models are transformed by transforming the vertices in the model. Since the vertices of a model are coordinates in three-dimensional space, they can be transformed by multiplying with 3 x 3 or 4 x 4 matrices. Using a 3 x 3 matrix to transform vertices is sufficient until we change our coordinate system in 3D space using offset vector without affecting the rotation or the scale of an object. This transform cannot be expressed using a 3 x 3 matrix, but can be done by extending coordinate vectors to four-dimensional homogeneous coordinates and applying transformations using 4 x 4 matrices. (Lengyel, 2004: p.81)

Euler transform is used to orientate an object into desired direction. It is built on idea that there exists an initial direction where object is facing. According to Akenine-Möller & Haines (2002: p.37), the negative z-axis is usually selected as the direction where object is facing, and head direction is positive y-axis. Euler transform \mathbf{E} is calculated by multiplying three matrices in presented order:

$$\mathbf{E}(h, p, r) = \mathbf{R}_z(r)\mathbf{R}_x(p)\mathbf{R}_y(h)$$

where h stands for head, p for pitch and r for roll and their values represent the clockwise rotation angles around in their respective axes. The rotations are illustrated in Figure 6. Euler transform suffers from *gimbal lock*, which is a situation where one degree of freedom is lost in result of concatenated rotations. This occurs, because axis rotations in Euler transform are evaluated independently in global scope. (Akenine-Möller & Haines, 2002: pp.37-38)

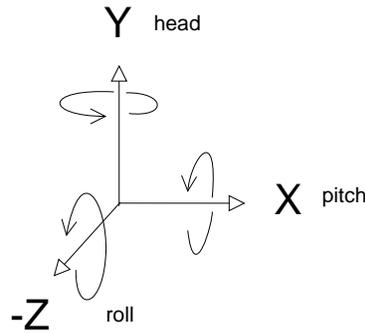


Figure 6: Euler transform axis and rotation directions.

Quaternions are an extension to complex numbers created by William Rowan Hamilton (Wolfram Mathworld, 2004). Later it was shown that quaternions can be used to represent rotations and do not suffer from the gimbal lock, which affects Euler transform (Svarovsky, 2000). A quaternion is represented as a 4-vector, and is marked as

$$\hat{\mathbf{q}} = (q_i, q_j, q_k, q_w) = iq_x + jq_y + kq_z + q_w, \text{ where following holds}$$

$$i^2 = j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k$$

Quaternion multiplication differs from regular vector multiplication and is produced by using cross and dot product, which yields an quaternion representing the concatenation of rotation operations. Quaternions can also be used to interpolate smoothly between two rotations. (Lengyel, 2004: p.86)

3.2 Coordinate Systems and Rendering Pipeline

Models and vertices can be expressed using different coordinate systems. Usually they are expressed in *model coordinates*, where coordinates are relative to model center point. This coordinate system is also known as *model space*. When all desired transforms are applied to a model, meaning the model is in correct location and oriented properly, it is said to be in *world coordinates* or in *world space*. This means that model has received the final position and orientation in a virtual world. From world coordinate system, models can be transformed into *view coordinates*, or *view space*, where their position is relative to the viewer. View coordinates are further projected to *window coordinates*, which are two-dimensional coordinates on screen. These coordinate system conversions are essential in order to produce pixels on screen from given 3D data, and are done in graphics rendering pipeline. (Akenine-Möller & Haines, 2002: p.14; Eberly, 2001: p.80).

When looking at the big picture, the main function of *graphics rendering pipeline* is to draw, or *render*, a two-dimensional image using specific view to a virtual world. The virtual world is constructed from geometry data, and a view to the world is usually expressed with camera analogy (defined in Section 3.3). When discussing about rendering virtual worlds, the virtual worlds are often called as *scenes*. Scenes can be enhanced using lights, materials, textures and lighting models, but these topics are more carefully examined in later sections. The rest of this section aims to explain the workings of rendering pipeline - how the image of a scene is produced.

The geometry data in a scene is defined by *drawing primitives*, which are also called as *rendering primitives*, and consist of points, lines and polygons. Those points, lines and polygons are constructed from vertices, and are positioned and oriented by transform matrices.

In order to draw specific primitives, the rendering pipeline must be instructed to draw the desired primitive, and vertices forming the primitive must be sent to the pipeline right after (Woo *et.al*, 1999: p.43). Rendering pipeline usually converts more complex drawing primitives (polygons) into more traceable primitives, such as triangles. This process is referred to as *triangulation*, and it ensures that the data is displayable and shown correctly - and more efficient, since current 3D hardware is optimized for triangle drawing. The rendering pipeline can be divided into three different stages,

which are known as the application, geometry and rasterizer stage. (Akenine-Möller & Haines, 2002: pp.22,437-438)

Application stage consists of defining and transforming 3D models, setting view parameters, and setting material properties and additional rendering settings. Programmer has the most control over program in this stage. (Akenine-Möller & Haines, 2002: p.22).

Geometry stage consists of model and view transform, lighting, projection, clipping and screen mapping (Lengyel, 2004: pp.5-6). The process starts by computing transformations of models. Next phase is to calculate view volume representing visible 3D space, followed by conversion of model coordinates into view space. The process is continued by computing the lighting of vertices according to defined material and light source parameters. After lighting, view volume is transformed into *unit cube* or *canonical view volume*, which is used to determine the set of vertices that lie entirely inside the virtual view. The unit cube has its minimum point at (-1,-1,-1) and maximum point at (1,1,1). The unit cube transform operation is known as the *projection*. After projection phase, 3D models are said to be in *normalized device coordinates*. After this, drawing primitives lying partially outside the unit cube are *clipped*, which means that new vertices are created at the intersection point of drawing primitive and view volume. Those new vertices replace the vertices which lie outside (See Figure 7). The new set of vertices is used to create a new drawing primitive, which lies completely inside the view volume. Those drawing primitives, that lie completely outside the viewing volume, are discarded and those originally completely inside, are left intact. Screen mapping phase converts 3D coordinates into *window coordinates*, which consists of 2D screen coordinates (x and y) and depth coordinate z . The depth coordinate represents the position on z -axis of the unit cube. (Akenine-Möller & Haines, 2002: pp.13-19,23;Lengyel, 2004: pp.5-6)

The *rasterizer stage* converts drawing primitives and pixel data into *fragments*, which are basically pixels with relative depth value, and perform tests to discard some fragments and keep others, and finally convert fragments into pixels contributing to the image (Lengyel, 2004: p.7). The color and depth values for a fragment can be obtained in three different ways; interpolating the values of drawing primitive vertices, using a constant value obtained from a single vertex, or using some other source⁷. Ac-

⁷See flat and Gouraud shading in Section 3.6 for details.

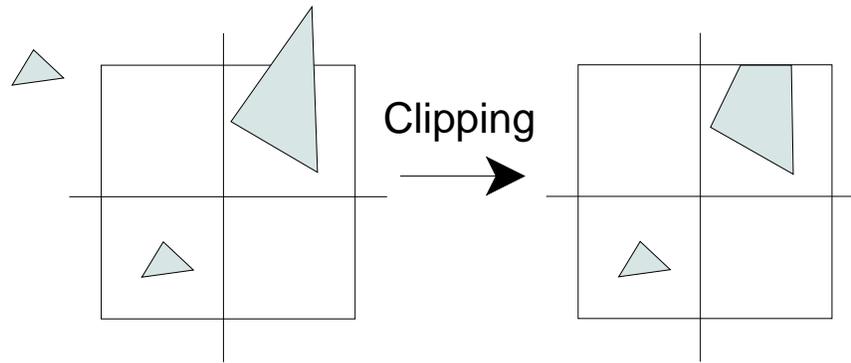


Figure 7: Illustration of the clipping operation.

According to Lengyel, (2004: pp.8-9), series of operations can be performed on fragments in order to select which ones are finally drawn. These operations are scissor testing, alpha testing, depth buffer testing and stencil buffer testing. *Scissor testing* allows only fragments inside a specific rectangular part of the screen to be rendered. *Alpha testing* allows only fragments with alpha value greater, equal to, or less than a reference value to be rendered. *Depth testing* performs a test similar to alpha testing, but by using depth values of fragments. *Stencil testing* is performed with values in stencil buffer. The test can be used, for instance, in the conjunction of dynamic shadows (Akenine-Möller & Haines, 2002: p.261). After these tests, blending (mixing fragment values with existing values in the image), color dithering (using different colors with adjacent pixels to achieve intermediate color) and logical operations (boolean operations between new and existing fragment color values) are performed, and finally a pixel is inserted into image (Lengyel, 2004: p.9).

In older hardware the rendering pipeline was implemented as *fixed-function pipeline*, which meant that the process of calculating vertex positions and parameters or pixel color could not be changed. In modern hardware, the rendering pipeline is programmable via vertex and fragment shaders. A *vertex shader* can do arbitrary set of predefined operations to vertices, and *fragment shader*, also known as *pixel shader*, can change pixel color as it sees fit. Vertex shaders may send data to fragment shaders to instruct their operation. Through programmable pipelines, more complex lighting and graphical operations can be performed. (Akenine-Möller & Haines, 2002: p.182)

3.3 Virtual Cameras

3D rendering uses *camera analogy*, which is a human-friendly way to handle setting up objects and views to them. And as the name implies, the process is similar to setting up a normal camera for taking a photograph. Table 2 lists the corresponding concepts between photographing and rendering an image (Woo *et al.*, 1999: pp.96-97).

Phase in taking a photograph	How 3D applications act
Setting up camera and pointing it to scene	Viewing transformation
Position models into a scene	Modeling transformation
Choose lens / adjust camera zoom	Projection transformation
Choose size for photograph	Viewport transformation
Snap photograph	Render scene

Table 2: Camera analogy in 3D application.

The virtual camera is set up by specifying position, horizontal field of view angle and aspect ratio of the screen. The visible space in a scene is limited to a specific *view frustum*, which is formed by six planes and is a (deformed) box containing every visible object in a 3D space. View frustum is used in geometry stage of the rendering pipeline to determine which vertices are visible before sending them to rasterizer⁸. The frustum is defined in view space, where origin is the eye (and camera) location and x -axis points to right, y -axis up and z -axis either opposite to the viewing direction or towards it, depending the implementation of the 3D graphics library (Lengyel, 2004: pp.111,112).

As Lengyel (2004: pp.113-116) points out, frustum planes can be calculated from the parameters used in the camera analogy. The *focal length*, expressed as e in the diagrams, is distance from camera position to the projection plane and it depends on *horizontal field of view angle* α . It is defined by following equation

$$e = \frac{1}{\tan(\alpha/2)}$$

Vertical field of view angle depends on the aspect ratio of screen and horizontal field of view:

$$\beta = 2 \tan^{-1}(\alpha/e)$$

⁸For details, see Section 3.2

Larger fields of view can be acquired by smaller focal lengths and vice versa. The zooming in-effect of virtual camera is equivalent to the decrease of field of view angle. The view frustum is also used in conjunction of visibility culling, where by comparing bounding volume to frustum planes yields information about 3D model visibility in application stage. Illustrations of view frustum and its construction can be seen in Figures 8 and 9.

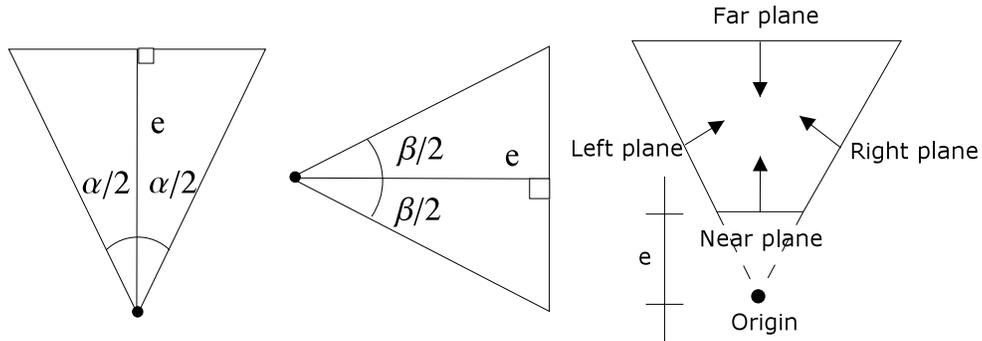


Figure 8: The frustum constructed from virtual camera properties. Left: Horizontal field of view. Center: vertical field of view. Right: Frustum planes, with arrows representing plane normals.

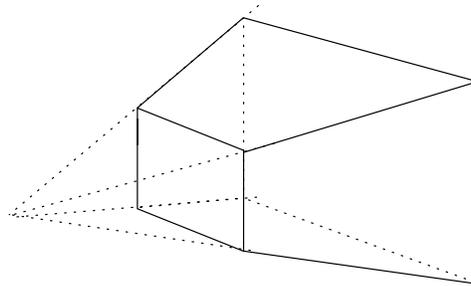


Figure 9: The viewing volume defined by of view frustum.

After camera has been set up, image is constructed by using either perspective or orthographic projection. The used projection maps the x - and y -coordinates to proper places in the projection plane. The projection is concretized into 4×4 matrix doing this. *Perspective projection* produces perspective distortion into the image, which means that objects further in distance appear to be smaller, while *orthographic projection* or *parallel projection* does not affect the relative sizes of objects (Woo et al., 1999: p.103).

3.4 Defining 3D Model Data

Polygons and lines can be combined to construct more detailed and complex three-dimensional entities. A set of polygons is referred as a *mesh*. *3D models*, such as cars, are usually stored as meshes. An example of this can be seen in Figure 10.

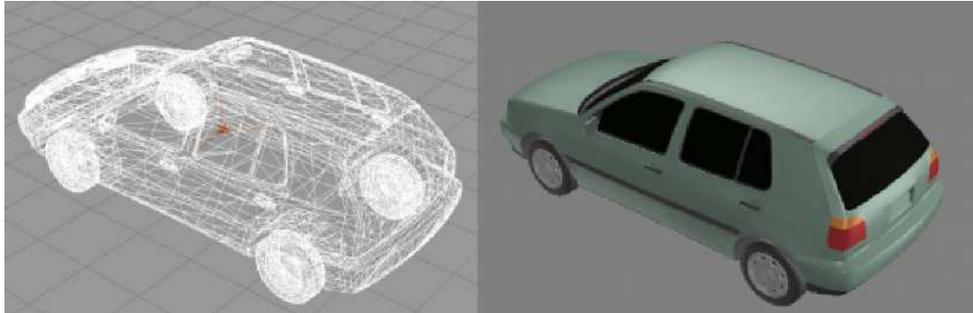


Figure 10: The left side shows a 3D model of a car presented without surfaces. The right side shows the same model with surfaces and materials added.

In some occasions it is more feasible to present the 3D model with parametric curves and construct surfaces with them. Storing an equation of a curve takes much less space than full set of vertices and their connectivity data. In this case, parametric curves must be converted into vertices and faces on the fly in order to get image on screen. This procedure is called *approximation of surfaces* and is done by using short line segments or small polygonal regions (Sweet & Wright, 2001: pp.92, 437).

More complex 3D models are not constructed by explicitly programming the polygonal data into an application, but by using dedicated software to create a data file which uses specific format to represent drawing primitives. These programs are referred to as *3D modeling programs* or *3D modelers* (Akenine-Möller & Haines, 2002: p.438).

Modeling programs have two main categories, namely solid-based and surface-based. *Solid-based modelers* are used mostly in CAD applications. In CAD applications the 3D models are molded with tools that correspond to machining processes (such as drilling or cutting). An object is considered to be a lump of material, which is processed to resemble something - very much in the same manner as sculptures are made. *Surface-based modelers* focus on handling the surface of objects and do not consider objects solid or non solid. They usually allow direct editing of vertices and polygons of models. Surface-based modelers may also contain objects which appear solid, such as spheres or boxes, and contain an internal representation for those objects. Both of

these modelers contain a *faceter*, which converts internal model representation into displayable polygons. (Akenine-Möller & Haines, 2002: p.439). In addition to polygonal data editing, 3D modeler programs may have an option to apply different colors, materials and textures to objects and even animate the models to some degree.

In real-time 3D rendering less detail means faster operation. Polygons can be *subdivided* into any degree, where number of triangles representing a surface are increased. For example, approximating a sphere with icosahedron requires more subdivisions when then viewing distance diminishes (Woo et.al, 1999: p.89), and more detail is needed to maintain credible approximation of a sphere. This means that model detail must be as low as possible, while maintaining reasonable visual quality. This must be evaluated on a case-by-case basis when 3D model is intended to be rendered in real-time.

3.5 Materials and Textures

Vertices, lines and polygons can be enhanced by adding color, materials or images to them in order to obtain more impressive results. *Materials* are defined by diffuse, ambient, emission and specular color, and shininess parameters. These parameters, combined with lighting, eventually yield the final color of a fragment or a pixel (Eberly, 2001: p.101). The effect of material properties is more carefully explained in Section 3.6, since material properties are used with lighting, but logically they should be grouped with the textures. This is because they affect the surface color of a drawing primitive. To put it briefly, *ambient color* corresponds to the color of material when it is lit by indirect lighting, *diffuse color* corresponds to the color in direct lighting and *emission color* means the color emitted by material regardless the lighting. *Specular color* and *shininess* control the color and brightness of highlights.

By using images, more details can be added to 3D model without increasing the number of drawn faces, which can be seen in Figure 11. An image, which is to be displayed over faces, is referred to as a *texture* and each pixel on screen, which is calculated from a texture, is referred to as a *texel*, a texture pixel. The process of applying an image over a surface in a 3D modeler program is called *image-based texturing* or simply *texturing*. The process of actually coloring the fragments in the rasterizer stage with correct values is referred to as *texture mapping*. Using texturing can cause great sav-

ings in modeling (less detail needs to be modeled), memory (less data needs to be sent to pipeline) and speed (less data means faster rendering) (Akenine-Möller & Haines, 2002: p.117). Texturing brings also more complexity to 3D modeling, and it is worthwhile to remember that realism can suffer from a texturing if it is poorly done. The texture in Figure 11 is rotated 90 degrees, and although it represents a stone wall, it does not give the desired effect.

Texturing is performed by using *texture coordinates* in vertices, and each vertex of a polygon usually has its unique 2D coordinates⁹. The coordinate defines which part of the image is selected into this vertex. In the fragmentation phase of rendering pipeline, surface of a polygon is converted into pixels and texture coordinate values between vertices are interpolated among them. Color value for each pixel is sampled from the texture map.

In addition to image-based texturing, there exists multiple other texturing methods, such as gloss and bump mapping. Each of these methods contribute more realism to the final image, if done properly. The details of these special effects are out of the scope of this document, but it can be said that *gloss mapping* defines the shininess value and *bump mapping* the direction of a surface normal of a pixel.

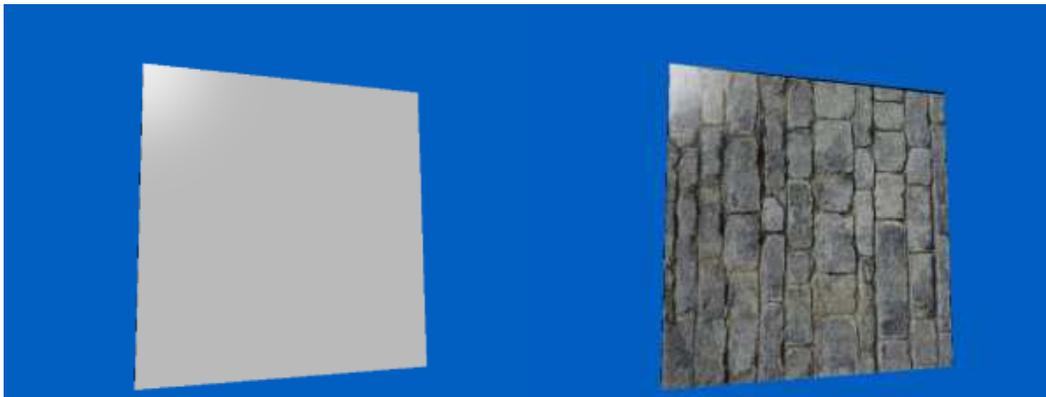
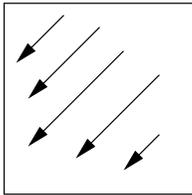


Figure 11: By textures much detail can be added to 3D models. Above is an example of how texturing brings about more realistic view of a stone wall.

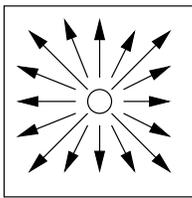
⁹The exists also 1D and 3D texture coordinates, but are needed only in special cases (Woo et.al, 1999: pp.370,372).

3.6 Lighting and Shading

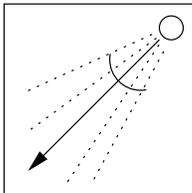
Lighting is the interaction between materials and light sources, and is implemented using a model, which imitates lighting in real world. There are *light sources*, which emit light (photons), that can bounce off from surfaces of objects. Those photons reach to eye of the observer, and are interpreted as colors. *Shading* is the process of doing lighting computations and determining fragments' (or pixels') color (Eberly, 2001: pp.101-102). Light sources can be classified as follows: directional lights, point lights, spotlights and ambient lights (Sweet & Wright, 2000: p.186, Eberly, 2001: p.100).



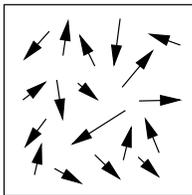
A *directional light* is positioned (virtually) infinitely far away and all the rays casted by light are parallel to each other. An example of such a light is the sun. Although in real life the sun is not infinitely far away and rays are not casted exactly in parallel, this is the most suitable way to simulate daylight in a 3D application.



Point lights are considered to be positional, since they have a location in space in contrast to directional light. Point light emits light equally in every direction. The light produced by a light bulb is quite similar to point light.



A *spotlight* is a restricted form of point light - it has a cone which restricts the emitted light inside. The size of the cone is usually defined by a cutoff angle. In addition to parameters of point light, a spotlight must have also a direction where it is pointing to. A spotlight can be used to simulate a flashlight, for instance.



Ambient light does not have any specific direction where it is coming from. It can be considered to have entered the scene and bounced around so that it has become directionless. Every object illuminated by ambient light is evenly lit on every surface in all directions .

Additionally to type-specific parameters, lights are defined by intensity and color. These parameters are used in similar manner regardless of the type of light. The intensity is divided into three different parts, namely ambient (a color of indirect lighting), diffuse (a color of direct lighting) and specular intensity (a color of highlights). Although this division is not realistic, it is done in order to give graphics application

more control over the scene appearance (Akenine-Möller & Haines, 2002: p.68). Table 3 gives a brief summary of each light type and the parameters related to them.

Directional light	Specular intensity Diffuse intensity Ambient intensity Direction vector
Pointlight	Specular intensity Diffuse intensity Ambient intensity Position vector
Spotlight	Specular intensity Diffuse intensity Ambient intensity Position vector Direction vector Cutoff angle
Ambient light	Ambient intensity

Table 3: Lights and their parameters in computer graphics.

Shading is done with different *shading models*, of which the most commonly used are flat (per polygon-shading), Gouraud (per vertex-shading), and Phong (per pixel-shading). In *flat shading*, an entire polygon (usually a triangle) is filled with one color. In *Gouraud shading*, lighting is determined for each vertex and values are interpolated across the polygon surface. *Phong shading* does not interpolate the colors of vertices, but the normal vectors of vertices among fragments, and performs lighting calculations for each pixel separately (Eberly, 2001: p.102). Flat shading and Gouraud shading are the fastest in the sense of rendering speed, and are usually hardware-accelerated. Phong shading is the slowest, since each calculations are done for each fragment in a polygon. It can be said, that flat shading is the most coarse of the three and Phong shading is the smoothest one. The smoothness of Gouraud shading can be increased (or decreased) by changing the subdivision rate of the polygonal surface. By using vertex and fragment shaders, the shading model can be specified by the programmer.

The lighting model used in most 3D real-time graphics libraries is a *local lighting model*, where lighting does not take into account light reflected from other surfaces,

but only light coming directly from light sources (Akenine-Möller & Haines, 2002: p.81). The intensity values of fragments are determined by values calculated from parameters presented in Table 3. The calculation of individual components and total intensity is done by using equations in Figure 12, where I denotes the intensity, M denotes the material, and L denotes the light and suffix marks the property which is used. The factor d stands for attenuation, and is defined by constant attenuation factor $A_{constant}$, linear attenuation factor A_{linear} and quadratic attenuation factor $A_{quadratic}$. The term \mathbf{p} is the coordinate where lighting is evaluated. Term \mathbf{n} stands for the normal vector of the point on a surface, \mathbf{l} means the direction vector from point to light source and \mathbf{v} means the view vector, in other words the direction where observer (for instance, a camera) is facing. Function $\text{clamp}\{V, [0,1]\}$ forces the component values of vector V into range between 0 and 1. (Akenine-Möller & Haines, 2002: pp.74-75,78,80-81; Eberly, 2001: pp.102-104):

$$\mathbf{l} = \frac{L_{position} - \mathbf{p}}{\|L_{position} - \mathbf{p}\|}$$

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

$$I_{ambient} = M_{ambient} \circ L_{ambient}$$

$$I_{diffuse} = \max\{(\mathbf{n} \cdot \mathbf{l}), 0\} M_{diffuse} \circ L_{diffuse}$$

$$I_{specular} = \max\{(\mathbf{n} \cdot \mathbf{h}), 0\}^{M_{shininess}} M_{specular} \circ L_{specular}$$

$$d = \begin{cases} \frac{1}{A_{constant} + A_{linear}\|\mathbf{l}\| + A_{quadratic}\|\mathbf{l}\|^2} & , \text{ if L is positional light} \\ 1 & , \text{ otherwise} \end{cases}$$

$$c_{spot} = \begin{cases} \max\{-\mathbf{l} \cdot L_{direction}, 0\}^{L_{exponent}} & , \text{ if L is a spotlight} \\ 1 & , \text{ otherwise} \end{cases}$$

$$I_{total} = \text{clamp}\{G_{ambient} \circ M_{ambient} + M_{emission} + c_{spot}(I_{ambient} + d(I_{diffuse} + I_{specular})), [0, 1]\}$$

Figure 12: The lighting equations in real-time 3D graphics.

The ambient intensity $I_{ambient}$ is determined by the component-wise product of ambient color of material and ambient intensity of light source. Eberly (2001: p.103) stated that diffuse intensity $I_{diffuse}$ is calculated using *Lambert's Law*, which means that surfaces are ideally matte (without shininess) and the reflected light is calculated by the cosine of \mathbf{n} and \mathbf{l} . The presented specular equation $I_{specular}$ is the *Blinn-Phong lighting*

equation (Akenine-Möller & Haines, 2002: pp.76,77), which is a faster variation of *Phong lighting equation*. The $I_{specular}$ describes the effect of photons bouncing off to the direction of view vector. Lengyel (2004: p.175) points out that $M_{shininess}$ controls the sharpness of highlight - small value produces a highlight which fades out over large area, whereas high value produces a sharp highlight which is visible only on a small area.

Attenuation d can be controlled in three ways, and natural lighting model is obtained by setting $A_{constant} = 0$, $A_{linear} = 0$, $A_{quadratic} = 1$. The c_{spot} is used in conjunction of spotlights to determine when evaluated point is outside the cone (Eberly, 2001: pp.101,103). *Global ambient* value ($G_{ambient} \circ M_{ambient}$), represents amount of ambient light in the whole scene, and does not depend on light sources (Woo et.al, 1999: p.212). According to Lengyel (2004: p.211), the total intensity I_{total} consists of sum of global ambient, material emission, and attenuated ambient, diffuse and specular intensities. Total intensity is clamped into suitable range, either by cutting or scaling, depending on speed and desired result (Akenine-Möller & Haines, 2002: p.84).

The total intensity equation holds for one light only. In the case of multiple lights, the intensity values of other lights is added to existing intensity and finally clamped. And as Akenine-Möller & Haines (2002: p.83) pointed out, each light source needs its own calculations, which takes time - hence the rendering of an image with multiple light sources takes more time than rendering with only a single light source. It should also be noted, that in some 3D graphics library implementations, the lighting can be disabled for back-facing polygons, which speeds up the lighting calculation process (Woo et.al,1999: p.199).

3.7 Visibility Culling Revisited

The visibility culling was briefly explained in Section 2.2, and it is essentially a way to speed up rendering process by drawing less. It was also mentioned, that two most common visibility culling techniques are backface and hierarchical view frustum culling, which are covered in detail below. The rest, which are suitable for more special situations, are left uncovered.

Let us consider drawing of a ball. A viewer looks at the ball from a certain angle. Despite the direction where ball is viewed, the other half is not visible. From this can

be concluded, that there is no need to render the side which is not visible - assuming that we do not handle cases where ball is made of translucent material. All faces on the non-visible side have their backs facing the viewer, and this information can be used to determine which faces can be discarded during the drawing process. Determining the faces which are back-facing the viewer and not rendering them, is called *backface culling* (Akenine-Möller & Haines, 2002: p.359).

The underlying graphics library can determine the face direction by examining the vertex winding order, and calculating a surface normal for a flat polygon. Because of the triangulation, that flat polygon is usually a triangle. If surface normal points towards the observer, a triangle is front-facing and if the surface normal points away from the observer, triangle is back-facing. Woo et.al (1999: p.668) outlines the normal calculation operation for a flat polygon, which can be used in case where three vertices form a triangle (\mathbf{v}_1 , \mathbf{v}_2 and \mathbf{v}_3). The surface normal calculation operation is defined as

$$\mathbf{n} = (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)$$

and the direction in respect to the view vector \mathbf{v} can be determined, for instance, by dot product

$$t = \mathbf{v} \cdot \mathbf{n}$$

where direction can be determined from signed scalar value t . Negative value denotes that surface normal \mathbf{n} points towards observer and triangle is front-facing, while positive value stands for back-facing polygon.

The hierarchical view frustum culling, mentioned in Section 2.2, is performed by using bounding volumes. There exist several bounding volumes, such as sphere, axis-aligned box, oriented box, lozenge, cylinder and ellipsoid. All of these have different intersection test algorithms, and some intersection tests take more time than others. However, in some situations the tightness of a bounding volume becomes more critical than the time consumed in intersection tests. Bounding volume selection is always a speed-accuracy tradeoff for an arbitrary 3D model.

For clarity and simplicity, the bounding sphere and its intersection test algorithms was selected to be inspected more carefully. Bounding sphere is useful when speed is required in intersection test, but construction of an absolutely optimal bounding sphere is quite demanding (Lengyel, 2004: p.223). In most cases, a relatively tight bounding sphere produces rather optimal results, and is not as time-consuming to solve than an optimal bounding sphere.

Listing 1: A pseudo-code algorithm for calculating a loose bounding sphere.

```
Sphere BoundingSphere( VectorXYZ vertices[] )
{
    VectorXYZ min, max;
    Sphere boundingSphere;
    min = max = vertices[0];
    for each vector v in vertices
    {
        if ( v[X] < min[X] )
            min[X] = v[X];
        else if ( v[X] > max[X] )
            max[X] = v[X];
        if ( v[Y] < min[Y] )
            min[Y] = v[Y];
        else if ( v[Y] > max[Y] )
            max[Y] = v[Y];
        if ( v[Z] < min[Z] )
            min[Z] = v[Z];
        else if ( v[Z] > max[Z] )
            max[Z] = v[Z];
    }
    boundingSphere.center = (min + max) / 2;
    boundingSphere.radius = ((max - min) / 2 )^2;
    return boundingSphere;
}
```

Calculating a loose bounding sphere is relatively simple, and is done by determining maximum and minimum extents of vertices (Eberly, 2001: p.26). An example of such operation is shown in Listing 1. A more precise bounding sphere (although not optimal) can be found by calculating average center of points, and a radius which encapsulates all the vertices (Eberly, 2001: p.27). A pseudo-code for this is in Listing 2.

The view frustum culling by bounding spheres assumes that the bounding spheres and frustum planes are in world coordinates. Technically, they could be in any coordinates, as long as they are in same coordinate system. Let there be a bounding sphere S and view frustum F . The bounding sphere is outside the view frustum if it is completely behind any of the frustum planes. The spatial relation between sphere S and plane P is determined by the distance of the sphere center to the plane. Following equations yields distance d between S_{center} to P , when plane is in normalized form.

$$d = P_{normal} \cdot S_{center} - P_D$$

Listing 2: A bounding sphere centered at average of points.

```
Sphere BoundingSphereTighter( VectorXYZ vertices [] )
{
    VectorXYZ sum = {0,0,0};
    for each vertex v in vertices
    {
        sum += v;
    }
    VectorXYZ center = sum / vertices.size();
    Real radius = 0;
    for each vertex v in vertices
    {
        VectorXYZ diff = v - center;
        if ( diff.length() > radius ) radius = diff.length();
    }
    return Sphere( center , radius );
}
```

Listing 3: A pseudo-code for view frustum culling with bounding spheres.

```
IntersectionType SphereIntersectsFrustum( Sphere sphere , Frustum frustum )
{
    for each Plane p in frustum
    {
        if ( DotProduct( p.normal , sphere.center ) - plane.d < -sphere.radius )
            return DOES_NOT_INTERSECT;
    }
    return INTERSECTS;
}
```

If d smaller than the negative radius of S , it is completely behind plane P and therefore outside the frustum. Using this knowledge, an algorithm (see Listing 3) can be constructed to test all frustum planes (Eberly, 2001: p.158). Unfortunately, this view frustum culling method does not always cull objects which are outside the view volume. This can be seen from Figure 13, where a sphere on the top-left corner of the view frustum is not culled although it is not visible. If such cases are to be eliminated, there should be another bounding volume surrounding the view frustum, against which the sphere would be checked.

Very sophisticated systems, which animate 3D models, will have to constantly update also the bounding volume transforms for the animated parts of models. When hierarchical view frustum is used, bounding volumes higher in the hierarchy must reflect the changes in lower bounding volumes. And since this hierarchy is usually implemented by using tree data structures, the changes can be reflected by merging the bounding

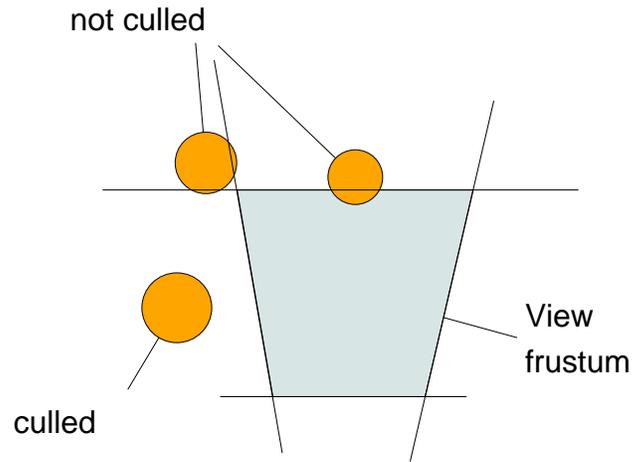


Figure 13: The error in view frustum culling.

Listing 4: A pseudo-code for view frustum culling with bounding spheres.

```

Sphere SphereMerge( Sphere sphereOne , Sphere sphereTwo )
{
    centerDifference = sphereTwo.center - sphereOne.center ;
    radiusDifference = sphereTwo.radius - sphereOne.radius ;
    radiusDifferenceSqr = radiusDifference * radiusDifference ;
    lengthSqr = centerDifference.Length()*centerDifference.Length();
    if ( radiusDifferenceSqr >= lengthSqr )
    {
        if ( radiusDifference >= 0.0f )
            return sphereTwo;
        else
            return sphereOne;
    }
    else
    {
        length = centerDiff.Length();
        t = ( length + sphereTwo.radius - sphereOne.radius ) / ( 2 * length );
        return Sphere(sphereOne.center + t * centerDifference , (length + sphereOne.radius + sphereTwo.radius)/2);
    }
}

```

volumes of children to the parent from bottom to up. Eberly (2001: pp.148-149) presented an algorithm for merging two spheres into smallest sphere containing the two. This algorithm is described in Listing 4.

The major difference between view frustum culling and backface culling is stage where they are performed; view frustum culling is usually done on application stage and backface culling on geometry stage. They also operate on different sets of data; view frustum culling culls entire objects, the backface culling works on polygon-level.

Any view frustum culling technique requires three methods; a method of constructing the bounding volume used in that technique, intersection test between view frustum and

bounding volume, and merging two bounding volumes into one. When using bounding volumes, the speed-accuracy tradeoff must be taken into account.

4 The Grand Design

This chapter goes through the requirements for a 3D visualization application and pulls together the necessary features needed in a 3D engine. Based on those features, the core design of a 3D engine is presented with diagrams. Also three optimization techniques, which are needed in the implementation, are presented. The diagrams and explanations are meant to describe the implementation in more readable and moreover, understandable, manner. If a compelling need for knowing the technical details should arise, they can be viewed from source code, which is available at `ftp://ftp.cs.joensuu.fi/pub/Theses/`¹⁰. Since it is prudent for each project to have a name, I gave the name GlowScape Engine (GSE) to this one.

Section 4.1 covers the requirements for 3D application, and connects them to corresponding features presented in Chapter 3. Section 4.2 handles higher-level design decisions, such as Object Oriented Programming (OOP) paradigm compliance. Sections 4.3 to 4.9 focus implementation details of core features. Section 4.10 describes the applied optimization techniques, and Section 4.11 lists external library dependencies.

4.1 Requirements and Analysis

The requirements set has been divided into two factions; *basic requirements*, which provide a rough outline for the functionality and *supplementary requirements*, which perform a more fine-grained dissection of basic requirements.

Basic requirements

The requirements have been compiled using the initial idea of Avatar Mirror for netWork Oasis project, which is described in my B.Sc Thesis (Gröhn, 2006). To put it briefly, netWork Oasis was designed to be a place, which would blend virtual and real environments seamlessly. The Avatar Mirror, located in netWork Oasis, was supposed to be a large wall-sized screen, which would display users' avatars when they pass it by. Additionally, the Internet connections were to be visualized using a planetary view, where arcs would show connections related to netWork Oasis.

¹⁰The source code is packed together with the electronic form of this thesis.

Although Avatar Mirror was never built, the requirements give a ground to stand on - there are no special never-seen-before effects, but things that can be done with almost any 3D engine. Visualization requirements for this application were categorized into three parts:

1. User visualization
2. User environment visualization
3. Data stream visualization

First of all, it was decided that there would be a support for animated avatars, which was a way to present users. Also more abstract presentations, such as sandstorms and amoeba-like creatures were desired. Additionally, there was a need for a scene where Internet connections originating from and leading to Oasis were pin-pointed on the globe. Visualization type needed to be changeable with different backdrops and avatar sets - namely an aquarium with fish and desert with whirlwinds tossing around sand, were mentioned. It was also decided that support for various different scenes would be included. One of these scenes was to be active at a time.

Supplementary Requirements

User visualization was decided to include indication of users' state, which slides from *available* to *do not disturb* and back again. This could be accomplished by using different colors, movement speed/patterns, emblems and combinations of the previous. An artist was to be able to change 3D models easily and design them with a 3D modeling tool. 3D models were to be exported into some format, which the application could read. The format was required to be widely adopted, so maximal compatibility could be achieved and modeling would not be tied into a specific tool. Moving 3D models were not to overlap each other in a scene in order preserve some degree of immersion.

User environment visualization had almost the same set of requirements as user visualization, but excluding the state indication. Scene background itself did not need to be animated.

In data stream visualization, user was to be able to rotate the globe in some manner to see where the data stream is coming from and where it is going. The ability of zooming

User visualization	Support for animated 3D models which can be created in several modeling applications. Support for more abstract visualizations, such as sandstorms and amoebas. Indicate users' state between "available" and "do not disturb". 3D model avatars must not overlap.
Avatar environment visualization	Support for widely adopted 3D format.
Data stream visualization	Provide an option for rotating the globe. Provide a way to display connections on the globe between two points (expressed as longitude and latitude and their direction).

Table 4: The compiled requirements for a visualization application.

the globe in and out to see the location more closely, was also considered. Locations were to be expressed with longitude and latitude so global position data (for instance, given by a GPS device) could be used. The users themselves were to provide their own location. All the previously listed requirements are compiled in Table 4.

Analysis of requirements

The requirements, which were defined for the visualization application, needed to be converted to suitable requirements for a 3D engine. They provided guidelines to engine design, but there existed other important factors, which had to be taken into consideration - requirements gathered for a single project most likely would not reflect the requirements for variety of other applications. Therefore, it was wiser to focus into basic functionality of the engine and aim to provide a working basis for further 3D rendering engine development. In order to produce a working solution, functionality had to be converted to programmable entities, which are covered next.

There existed an apparent need for a class, which contained 3D model vertex data. That class had to allow access to vertex data, so it could be sent to the rendering pipeline.

Vertex data contains only 3D coordinates, so some method of describing rendering primitives was also needed. I took care of this by designing a class that contains vertex indices, and drawing primitive type. These primitives are drawn from 3D model vertex data.

Two requirements listed in Table 4, **support for animated 3D models which can be created in several modeling applications** and **support for widely adopted 3D format**, were met by including support for two different formats; 3DS and Milkshape 3D. 3DS format is supported in many modeling applications, but it does not include skeletal animation support. Skeletal animation is quite effective way to animate human-like models. Milkshape 3D (MS3D) is the internal format of Milkshape 3D modeler, which has various plug-ins that are able to import other formats. MS3D supports also skeletal animation, and other formats can be converted into MS3D using Milkshape 3D modeler. GSE needed also an animation subsystem, which takes care of the correct transformations of vertices according to data provided by MS3D format.

Table 4 lists also an item **support for more abstract visualizations**, which was fulfilled by including a highly configurable particle system. Particle system was to be able to provide various particle movement patterns, which practically meant that the animation movement had to be controllable directly in the application which used the particle system - a predefined set of movement patterns might not have been suitable for every occasion. This required that design patterns had to be carefully examined in order to find proper solution for a particle system class.

The third requirement in Table 4, **indicate users' state between "available" and "do not disturb"**, indicates the need for an object management system. The object management system was to provide a logical way to access any object in the scene. This was accomplished by designing a scene graph class, which could be used to group avatar models logically and retrieve them for handling. Additionally, the next requirement **3D model avatars must not overlap**, was taken care of by designing a bounding volume scheme with bounding volume objects connected to a scene graph.

The last two requirements in Table 4, **provide an option for rotating the globe** and **provide a way to display connections on the globe between two points** was met by designing a proper transformation manipulation methods using operations presented in Section 3.1, and creating a method for approximating parametric curves from a set

of points, which further on could be rendered as a contiguous quad array, where two adjacent quads were connected by one edge.

3D models saved in 3DS or Milkshape 3D format are represented as triangles. Therefore, GSE had to support at least rendering of triangle primitives. The hierarchical frustum culling was required to accelerate rendering if several users were connected to the system. In order to provide frustum culling, camera and frustum classes were needed. Also texture/material support with lighting was a necessity for more visually appealing graphical output.

4.2 Language Selection and Applied Design Patterns

A large part of GSE is designed according to object-oriented programming (OOP) paradigm. Its benefits have been found to be modularization, code reuse, better modeling of the problem space and readability (Gwinn, 1992; Barry, 1990). Since the nature of this project also requires efficient code, I chose C++ as the implementation language. It allows to use OOP paradigm, and provides means to write very hardware-specific code in time-critical sections, if necessary.

Data management is necessary for any program, that handles information. One storage method is to use a global object, that contains other objects or primitive data types. However, OOP guidelines advise against such global object variables, and since only a one object is allowed, this imposes a problem. Enter *singleton*, an object, that has only one instance at any given time (Bilas, 2000). Singletons provide full control over creating and destroying virtually global objects. Singletons can be used in conjunction with customized *container classes*, which can store other objects with a specific identifier and retrieve them. They assist also in resource management - an example of this is destroying all textures by destroying the texture container object - provided that all objects are actually stored into the container. When this container scheme is applied properly, dreadful memory leaks can be reduced.

It is quite easy to determine, that less code equals less maintaining of code. The C++ standard contains a high-level feature called templates (Schildt, 1998: p.372). *Templates* provide a way to create generic functions and classes. This means that a programmer can use one function (or class) with several different data types, without having to write a specific implementation for each one explicitly - instead, it is done by the

compiler. Templates can lead to more reusable code, but there have been cases where they have negatively affected readability and compile times (Kees & Miller, 1999). Advantages of utilizing this technique in GSE are explained in Section 4.10.

4.3 Math Classes and Data Structures

The mathematic operations described in Section 3.1 are a necessity when dealing with 3D graphics. They are needed often and in various places. This also meant, that I had to implement them as efficiently as possible (some optimization techniques are covered in Section 4.10). Matrices are restricted to square matrices, and most common sizes are 3×3 and 4×4 . Rotations performed with Euler transforms are sufficient to the point when multiple rotations about each of the axes has to be made, and a gimbal lock is encountered. Hence, I implemented an option to perform rotations using quaternions, which effectively prevent gimbal lock from occurring. Additionally, I implemented various utility functions, such as conversions of rotation matrices to quaternions and back again, calculation of eigenvectors and eigenvalues for matrices (used in calculation of bounding volumes), and so forth. The diagram of the core classes of mathematics package is in Figure 14.

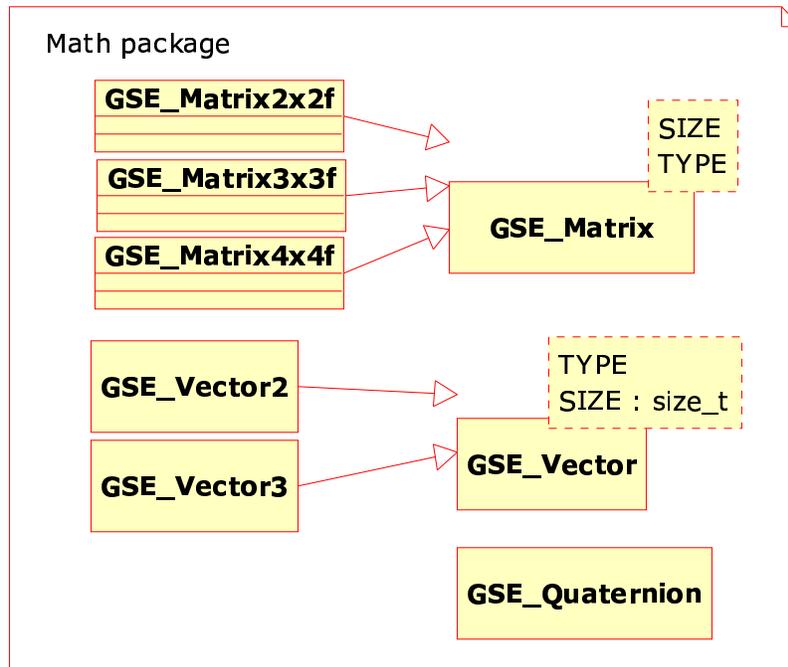


Figure 14: UML diagram of the core classes in the mathematics package.

I used directed graph data structure and graph renderers throughout the GSE. *Graphs* contain graph nodes and graph edges. *Graph nodes* represent various things, such as a renderer command, a 3D object or an animation joint. *Graph edges* connect different nodes together. *Graph renderers* are classes, which do something with the data in graph nodes. Graph renderers execute queues of commands, that are formed from connected graph nodes. The name “renderer” is perhaps misleading, since renderer classes do not necessarily produce a rendered image, but update the graph structure, change node values, and so on.

I implemented graph, node and edge classes using C++ templates. Each node class has a *Run-Time Type Identification* (RTTI) member variable. RTTI variable is used by graph renderer to determine which type of node is handled. RTTI is passed as a template parameter, and prevents situations where nodes from different types of graphs would be linked by edges (or stored in graph with different type) - and cause undetermined actions. Graphs provide a foundation for renderer operation, animating, and scene representation in GSE. Graph class diagrams are illustrated in Figure 15.

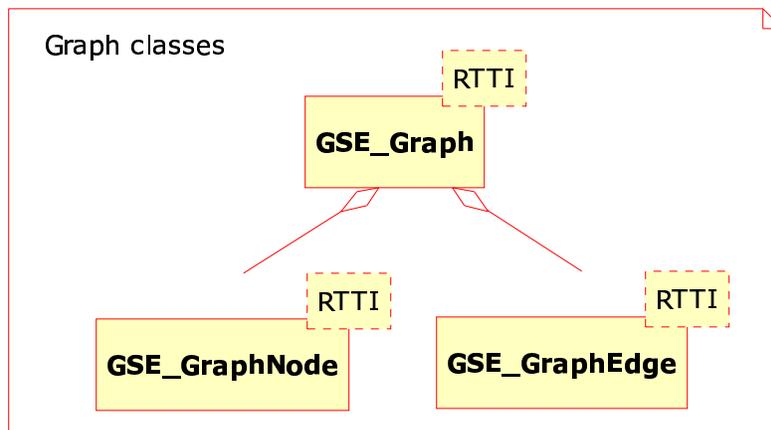


Figure 15: UML diagram of the graph classes.

I separated the actual data from graph nodes, and implemented an external storage with container classes. I found that name “container class” does not fully reflect the purpose of the class, and decided to use name “manager class” instead, which implies that all actions concerning the data objects, “employees”, must go through the “manager”. I also implemented a restriction, that forces a pointer to a managed object to be stored into appropriate manager automatically. This restriction handles the issue of memory leaks mentioned in Section 4.2.

When object pointers are stored into a manager, they can be retrieved either by index (since object pointers are stored into dynamic array) or by memory address (pointer stores the memory address of an object). Retrieving by index is not very reliable method (if index would be used throughout the program execution), since objects can be deleted from any point in the array. This means that the indices can change when objects are deleted. This leaves only a memory address, which (in human terms) is not very descriptive name for an object. Therefore, manager classes needed to provide a way to retrieve objects by a name. This name could be, for example, a string (character array) or a symbolic name (an integer).

I implemented this feature by using a mapper class template. A *Mapper class* creates type-to-pointer mapping for arbitrary types, where type is given as template parameter. This means that an arbitrary mappings can be created as needed. This feature is used, for instance, in Camera Manager where cameras can be retrieved and deleted by using their name (character array).

I wrote an octree data structure class to be used in spatial geometry partitioning. An *octree* is a tree data structure, where each node has eight child nodes at maximum (Ginsburg, 2000). Ginsburg also states, that it is an ideal structure for representing virtual 3D world, enclosed by cubes. I also wrote utility methods to construct an octree from geometry data (vertices). The octree class has been made as general as possible, so that different variations of the octree data structure can be implemented. One such variation is *loose octree* (Ulrich, 2000), which uses objects instead of their geometry, and can be used as spatial data structure in a scene graph. This is reserved for the future use, if ordinary scene graph should prove to be inefficient in some situation. Octrees can also be used in conjunction of rendering very large terrains. Octrees restrict the amount of renderable data in efficient manner when combined to hierarchial view frustum culling.

4.4 OpenGL Renderer Class

OpenGL was chosen as the graphics API, since it is robust, it has been available for a very long time and is available for several platforms. I designed GSE to be platform-independent, and also API-independent where possible. I wrote the initial implementation for graphics renderer using OpenGL, but it can be extended to include Direct3D,

or software renderer, if necessary. The renderer component uses its own internal command system to call OpenGL API function calls.

The initial requirement for the renderer to operate, is a working OpenGL context in a window. This is accomplished by creating a specific *screen entity*, which creates a new window according to given parameters. The renderer class does not contain method to create a window where images could be drawn. I did this deliberately to keep engine as modular as possible - this way OpenGL context to a window can be created by different means, and renderer system can still be used.

Image rendering is performed by traversing a render graph. A *render graph* is a directed graph with nodes representing rendering commands and their data. Renderer is a class which handles these nodes, and converts the graph representation into API function calls. The renderer and graph are paired together with a global traveller template function, which takes a graph and a traveller class as its parameters. In the function, nodes are passed to renderer's Enter()- and Leave()-methods, where actual logic resides. Class diagram for OpenGL renderer classes is shown in Figure 16.

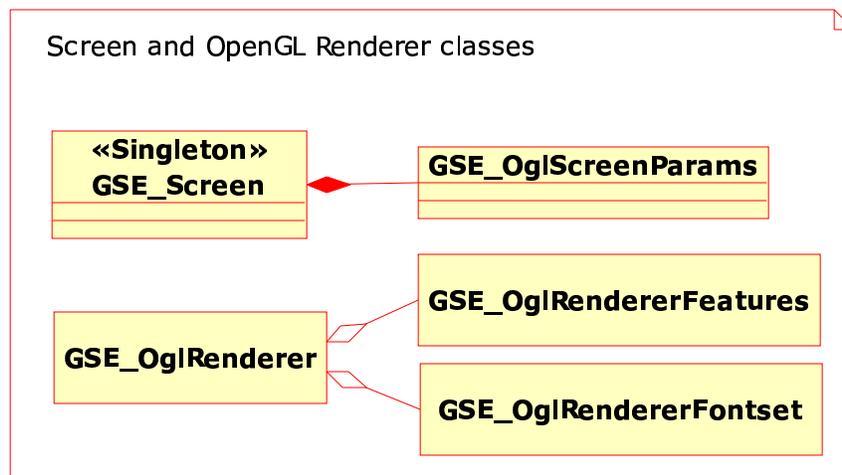


Figure 16: Diagram of the OpenGL renderer classes.

I designed the renderer to encapsulate as many of the OpenGL-related library calls as possible, and separate the actual rendering from the data representation classes. This is beneficial in many ways; for instance, a particle system does not know how particles are rendered - which allows different outputs to be generated from same data, if required. Encapsulating the OpenGL library calls also makes it easier to include support for another 3D graphics API libraries, such as Direct3D, by writing another renderer class using other API function calls.

View settings are modified using a camera class. A camera can be moved, rotated and its properties can be altered at any time. Each camera automatically creates a frustum, which can be used in frustum culling. Class diagram of these is in Figure 17.

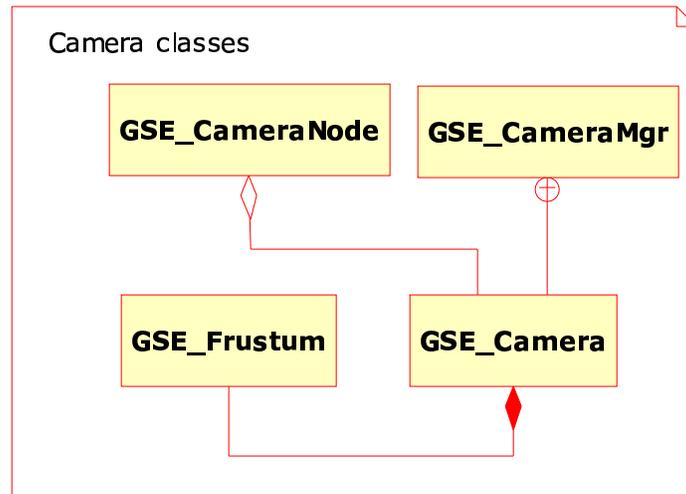


Figure 17: Camera and frustum classes diagram

4.5 Particle System

The particle system should be as flexible as possible in order to provide variety of different visualizations, and efficient enough to keep number of particles high. High number of particles is essential in simulating large entities consisting of very small pieces. An example of these entities is a sandstorm, which was mentioned in Section 4.1. I implemented a particle system using policy-based class design and templates, as described by Gamedev.net (2003), to meet these requirements.

Alexandrescu (2001: p.2) defines *policy-based class design* as constructing a class with complex behavior from many smaller classes, each of which takes care of a one structural or behavioral aspect. *Policies* are interfaces, which must be implemented when individual policies are created. In our case, policies implement specific actions, which modify the particle class attributes. Particle system itself does not know how the particles are modified as the time passes, but knows how to apply policies to particles.

The particle system hierarchy is based on design presented by Gamasutra (2002). Particle system consists of following classes: Particle, ParticleSystem and ParticleSystemMgr. Each *Particle* belongs to one *ParticleSystem*. *ParticleSystems* are managed by

ParticleSystemMgrs. Individual particles have position, size, velocity and energy parameters. These parameters are updated by ParticleSystem, using the policies. There are two types of policies, initializer policy and action policy. *Initializer policies* are used when new particles are created. An example of an initializer policy is a size policy, which defines the initial size of a particle at birth. *Action policies* are used when particles are updated, an example of this is a *move policy*. There exists also a CompletePolicy, which can combine size, energy, velocity, movement and other policies into a single policy. Some of the commonly needed policies are already implemented, and several more can be constructed when needed. The particle system class diagram is shown in Figure 18.

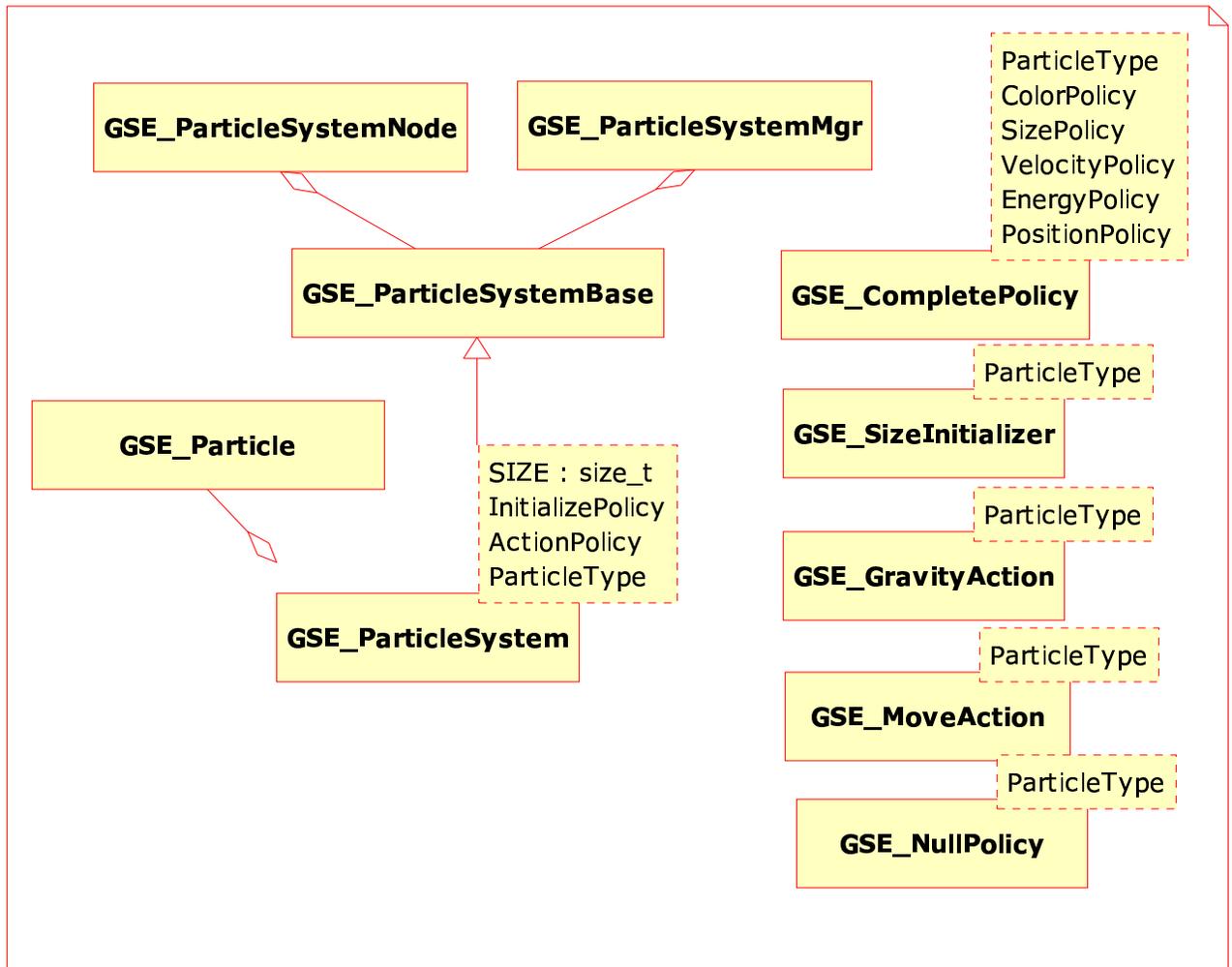


Figure 18: UML diagram of the particle system.

4.6 Animation System

I built the animation system also on a graph data structure. Animation graph contains animation nodes, which can be either joint nodes for skeletal animation, or particle system nodes for particle system animation (Figure 19). Animation renderer class handles updating of nodes and their data. The animation system handles the correct timing of animations, but it must be told explicitly how much time has passed since last update. This feature can be used to animate objects in slow-motion, for example. Animation system supports also looping, stopping and continuing the animation.

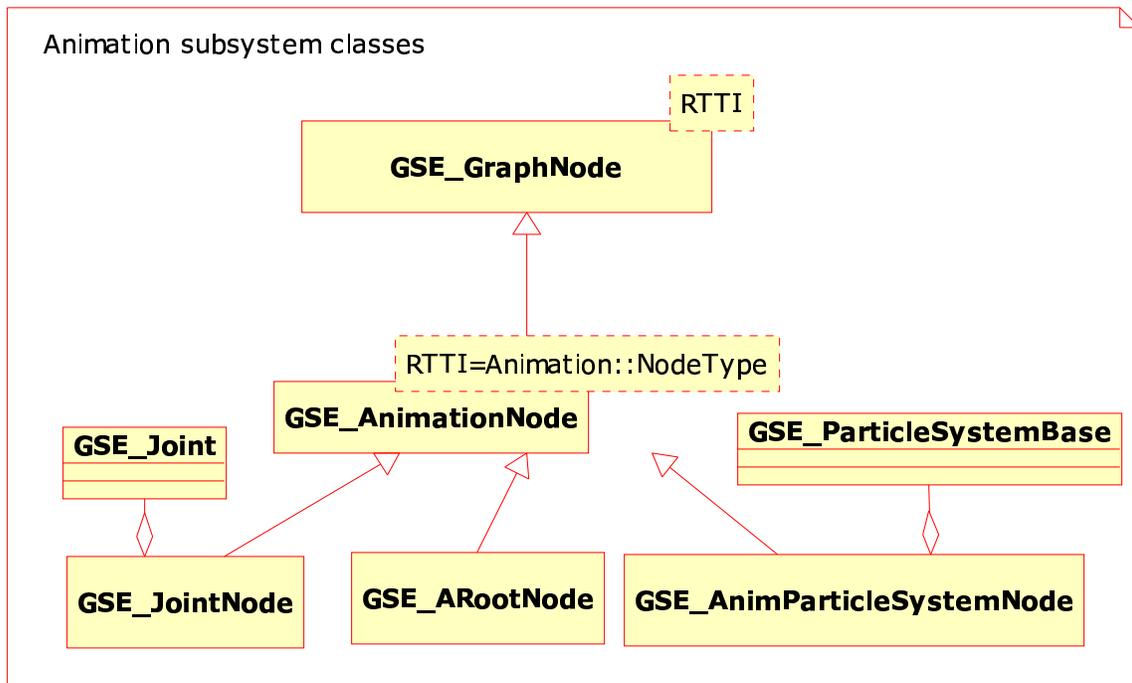


Figure 19: UML diagram of animation graph and nodes.

I designed animation renderer to be responsible for updating the animation in every joint. Each joint consists of three pointers; a pointer to a vertex array with original geometry data, a pointer to a vertex array with modified geometry data, and a pointer to an array of indices, that mark the vertices belonging to the joint. Each joint has also an array of keyframes, which express a rotation and translation for that joint at specific point in time. Vertices, marked by indices in the array, are transformed by animation renderer using interpolated parameters retrieved from keyframe data, and stored to the vertex array containing modified data. OpenGL renderer class uses the modified vertex data while rendering animated objects. I automated this rather complex process to the

point, where it is only necessary to design the animation, load it, and update it before rendering.

4.7 Scene Handling and Visibility Determination

Scene objects must be represented in hierarchical relation to each other, if scene is desired to be managed logically. Just by adding drawable objects into a dynamic list will impose problems when object-level culling is performed. If culling is done on each and every object, it works, but might be slow when the number of objects increases.

The scene graph is built to include an animation graph and a render graph, since in most cases they would be needed anyway - animation is needed for illusion of movement, and objects need to be rendered in order to produce an image. Scene objects are created via scene graph. This is because each graph node must belong to a graph, at least logically. Memory-wise speaking, it can reduce memory leaks when every graph handles all of its nodes, and graphs are able to delete their nodes when the graph itself is deleted. This is also easier for the programmer, since nodes are managed by the graph, and do not need an external storage.

Scene nodes contain a bounding sphere and an oriented bounding box for hierarchical frustum culling. Each node keeps track of its local transform, as well as world transform. The matrices representing transforms are used to update bounding volume positions and orientations. This way, it is only necessary to calculate bounding volume for vertex data in model coordinates, and later it is transformed using same operations that were applied to the vertex data. Class diagrams for scene graph are shown in Figure 20.

4.8 Materials, Textures and Lighting

This section describes the implementation of materials and textures, transparency and lighting.

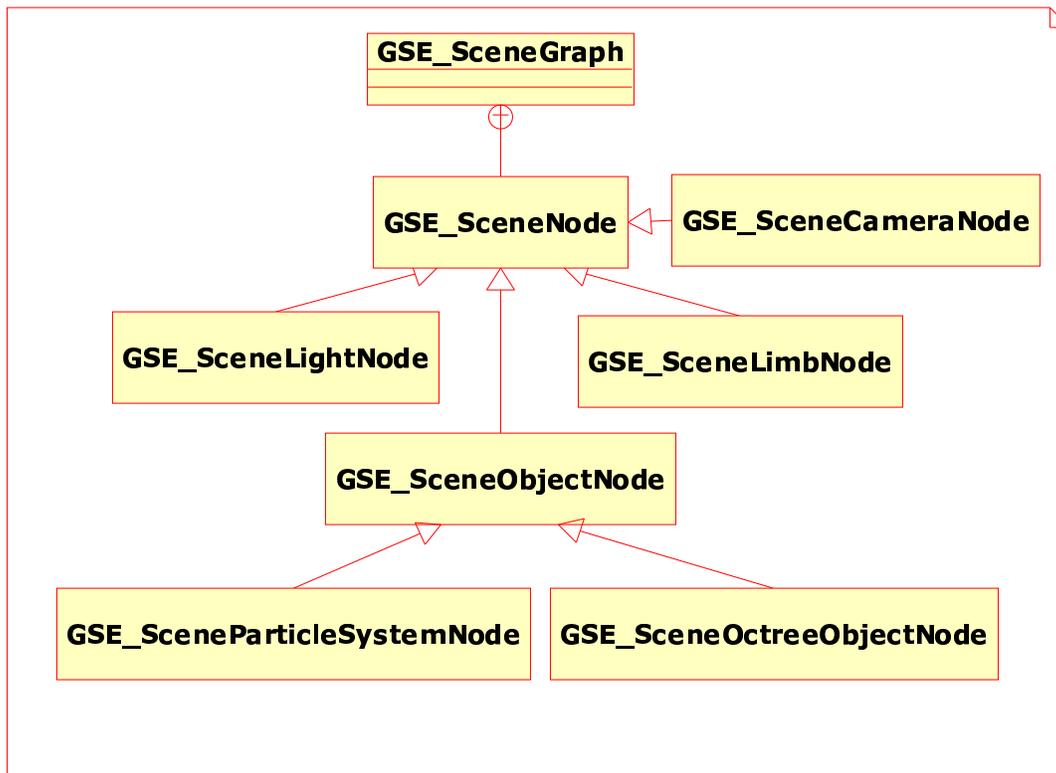


Figure 20: Class diagram of scene graph system.

Materials and Textures

Materials are represented by *Material* classes, and are managed by *MaterialMgr* class. Textures follow this same principle. I designed materials and textures to be applied to objects in following manner. Each material (or texture) is assigned to a material node (or texture node). Geometry objects are assigned as child nodes of material or texture nodes in a render graph. Render graph is processed by OpenGL renderer class, which interprets nodes one after another, and last material (or texture) applied before geometry rendering node, is taken into account. Materials can be either *opaque* (completely solid) or *(semi)transparent* or *translucent*, which means that underlying graphics are somewhat visible behind the object made of transparent material.

I also designed configuration file utility class, which is able to read text files and store name-value pairs. I included the configuration file utility class into *MaterialMgr* and *OglTextureMgr* classes, and wrote methods that read a configuration file and create new material (or texture) object from it. This allows materials and textures to be defined by configuration files, and read by their respective manager objects. Material

configuration files contain color values for material ambient, diffuse, emission, shininess and transparency values. Texture configuration files contain a path to an image (or video) where texture is generated from, and the texture type. Figure 21 shows class diagrams of material and texture objects, and their managers.

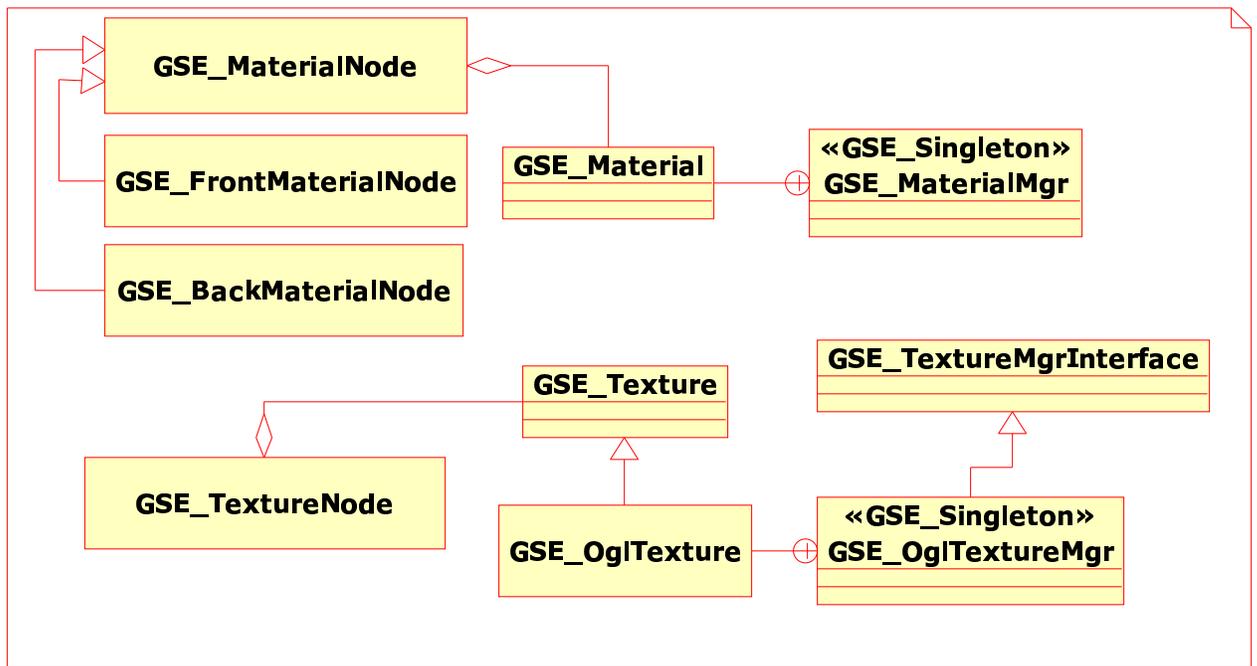


Figure 21: Class diagram of material and texture classes.

About transparency

The transparency effect is an expensive operation if it is used a lot, and if a realistic result is desired. True transparency effect can be achieved by determining the correct order of transparent pixels and rendering them from back to front. However, this is very time-consuming process. Transparency can be faked in variety of ways, depending on the level of desired realism. One way is to render transparent polygons in arbitrary order (after opaque polygons have been rendered) using additive pixel blending, where incoming fragment color values are added to existing ones. Disadvantage of this technique is, that values might become saturated, and final result is white and not transparent at all. Another way is to sort polygons from back to front, and render them in that order after opaque polygons.

A pseudo code for rendering sorted transparent polygons is shown in Listing 5. But in most cases, this is not applicable, since the drawing primitives are not stored as a

Listing 5: Pseudo code for rendering transparent polygons by sorting.

```
void Render( Polygon polygons [] )
{
    Polygon opaque [];
    Polygon transparent [];
    SortOpaqueAndTransparent( polygons , opaque , transparent );
    SortBackToFront( opaque );
    SortBackToFront( transparent );
    DrawPolygons( opaque );
    DrawPolygons( transparent );
}
```

Listing 6: Pseudo code for rendering transparent polygons using alpha test.

```
void Render( Polygons polygons [] ):
{
    Enable( DEPTH_TEST );
    Enable( DEPTH_WRITE );
    Enable( ALPHA_TEST, ALLOW_ALPHA_EQUAL_TO_ONE );
    DrawPolygons( polygons );
    Disable( DEPTH_WRITE );
    Enable( ALPHA_TEST, ALLOW_ALPHA_LESS_THAN_ONE );
    DrawPolygons( polygons );
}
```

polygon soup, but they are grouped by objects they belong to. Implementing the transparency algorithm using this method would require all drawing primitives of visible objects to be merged into one array of polygons, which would be first sorted to transparent and opaque, then each subset would be sorted back-to-front order and finally rendered. This yields a working, but very slow solution, which is not applicable to more advanced real-time rendering.

Woo et al. (1999: p.441) present an implementation for transparency algorithm using the hardware-accelerated depth and alpha testing¹¹. The algorithm shown in Listing 6 is pixel-perfect, but halves the frame rate from optimal situation since visible geometry is sent twice over the pipeline.

Speed-wise speaking, a quite viable solution is to sort objects by transparency, and draw them in two batches utilizing depth testing. This allows nearly-optimal speed and quite realistic results in most cases. It must be noted, that resolving back-to-front order of overlapping arbitrary-shaped objects is nighly impossible without splitting their intersecting polygons.

¹¹See Section 3.2 for details

Lighting

Lighting can be categorized as dynamic lighting and static lighting. *Dynamic lighting* means that lighting equations are calculated from currently active light sources, which contribute to the scene. Light sources in a scene can move and change brightness, color, shape, and so forth. This means that the pixel values will be updated to reflect these changes. *Static lighting* is a lighting scheme, where lights are stationary and do not change their parameters.

OpenGL library imposes some restrictions to the lighting. It is specified that OpenGL supports at least eight different light sources (Woo, & al., 1999). This means that the scenes cannot be lit with more than eight lights at a time, since most implementations support only the mentioned eight lights. But scenes might have more lights than that - think street lights, for example - how can eight lights manage that? Well they cannot.

By using multiple rendering passes and specific blending operations, contribution of each light could be calculated, but this solution is not very effective, because

1. lights may be so far away from the object that their contribution very small.
2. lights have different brightness.
3. each rendering pass consumes time.

Therefore, the solution is to reduce the number of lights at the cost of accuracy in lighting, which rarely imposes a problem. A pseudo-code for this algorithm is given in Listing 7, where lights are selected on object-basis.

Let n be the maximum number of lights supported in 3D graphics library. The set of active lights is sorted in descending order by brightness when observed from the renderable object. From the sorted set, the first n lights are made active during the rendering of current object. Using this procedure, an arbitrary number of lights can be inserted into the scene, and implementation uses only the ones with greatest effect on the object. The algorithm handles all lights as point lights, and will produce incorrect results if multiple light types are included in the evaluated set of lights.

There exist pre-processing techniques, which can emulate the effect of multiple static lights without actually using them, namely *light maps*, which are a texture-based so-

Listing 7: Pseudo-code for selecting most contributing lights for an object.

```
Lights[] SortLights( Light lights[], VectorXYZ position, integer NumLights )
{
    for each light l in Lights
    {
        l.brightness := |(l.position - position)|;
    }
    SortByBrightness( lights, DESCENDING_ORDER );
    Lights lightarr[];
    for i=1 to NumLights
    {
        lightarr[i] = lights[i];
    }
    return lightarr;
}
```

lution for emulating lighting effect. In this technique, on top of each texture is put another texture and their colors are blended. The top-most texture contains only different shades of gray and the output yields an image with darker and lighter areas. (Akenine-Möller & Haines, 2002: p.150). This technique can be used to speed up rendering, since actual lighting calculations take more time. GSE supports this technique via textures.

It is easy to picture a situation of a scene, which has many stationary objects, each lit by same set of lights. Every light might not contribute to every object in a scene, if their distance is large enough. Therefore, it would be unnecessary to sort every light for every object. Instead, some lights could be left out of altogether for objects that are definitely out of range. I implemented this optimization by using an illumination set for each object. An *illumination set* contains an arbitrary subset of lights in a scene. When object is rendered, only those lights in object's lighting set are considered. Illustration of illumination sets is seen in Figure 22.

4.9 Shaders

Shaders were briefly mentioned in Section 3.2, and they can be used to create more complex lighting effects, for example. There exists several shader languages, namely assembler, Nvidia Cg, HLSL and GLSL. I implemented shader support in GSE using OpenGL Shading Language (GLSL). Shaders are divided into two parts, namely vertex shaders and fragment shaders. Fragment shaders are usually data-driven by vertex

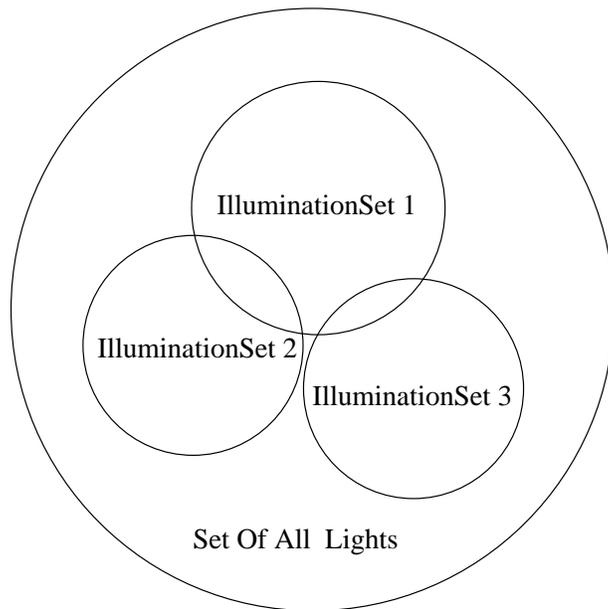


Figure 22: Illustration of the illumination set scheme.

shaders, but they can be used also individually. Vertex and fragment shaders can be combined into a *shader program*, which is used to reprogram some parts of the rendering pipeline. Since a program can be constructed by combining several different shaders, it is necessary to manage vertex shaders, fragment shaders and shader programs separately.

Each shader can be passed shader parameters, either *uniform parameters* (same parameters for all vertices) or *vertex parameters* (different parameters for each vertex). Shader code is loaded to shader object using a separate text file, and rendering primitives are drawn using a shader by assigning primitive drawing commands as child nodes of a shader node - in the same manner as materials and textures are used.

The shaders are powerful tools, but they also come with a price: shaders must implement main functionality in the fixed-function pipeline, such as vertex transformation, lighting and texture mapping, if any shader is used. This means that the mathematical model for those operations, such as the lighting equations described in Section 3.6, must be understood. Figure 23 shows class diagrams of a shader subsystem (shader parameters are excluded to maintain some clarity).

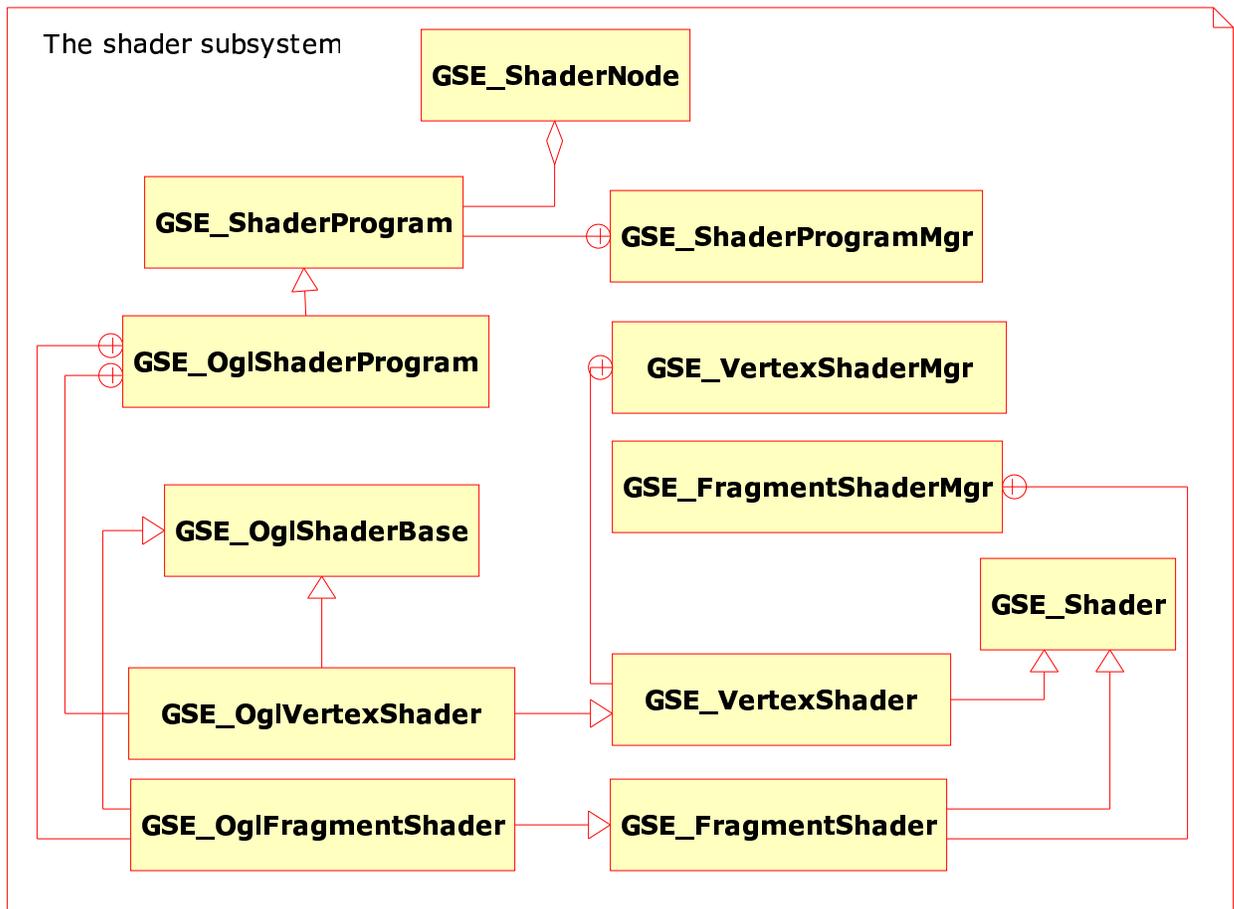


Figure 23: Class diagram of the shader subsystem.

4.10 Optimizing the performance

This section explains three optimization techniques used in the GSE, namely avoiding use of virtual functions, vertex buffer objects and C++ template metaprogramming. They have effect on the performance and also to work related to maintaining source code.

Avoid using virtual functions

When programming in C++, some OOP design patterns must be avoided in order to reduce unnecessary function call overhead. An example of such a case, is the use of inheritance interfaces. Inheritance interfaces, which are commonly used design pattern in OOP, are implemented using pure virtual functions in C++. *Pure virtual functions* are base class methods without implementation. This means that they must be imple-

mented in a child class inheriting the base class, in other words, through *direct specialization*. Virtual method call is determined run-time, using a virtual function table, if it is called from a base class Schildt (1998: p.349). This feature has been shown to cause significant overhead (Driesen & Hölzle, 1996) and should be avoided in time-critical sections of GSE, which happen to be everything related to rendering a single frame; drawing calls, visibility determination, animation, and so forth. The old truth, OOP is not a silver bullet¹², is apparently valid also today.

Vertex Buffer Objects

When large amount (tens or hundreds of thousands) of consequent calls to functions describing the vertex position, color, and texture coordinates are made, frame rate comes down to a grinding halt. One of the problems is the *function call overhead*, which means the time to enter the function and leave it. Another problem is the state change required in OpenGL to submit different types of data (Marselas, 2000; Woo & al, 1999: p.67).

To keep frame rate sufficiently¹³ high and render highly detailed models, the data must be optimized before sending it to rendering pipeline. By cleverly grouping the data, and sending larger batches using specific array function calls, is more efficient than calling individual functions that define vertex positions, color and surface normals.

Vertex arrays are the key to perform this kind of optimization. Data is packed into arrays, which are sent to rendering pipeline using a single function call. Vertex data (vertex coordinates, texture coordinates, colors and normal vectors) may be packed tightly into a single array, or be split among many arrays. The former way is referred to as *interleaved data* and latter as *streamed data*. According to Marselas (2000), using streamed data is about 30% faster than interleaved data. Using this information as basis, I implemented GSE drawing primitives entirely with streamed vertex arrays. This reduces the function call overhead, and provides better performance when large vertex arrays are used. However, Marselas stated that using vertex arrays with array

¹²The term in this context was first used by Brooks (1987), who stated that OOP is not a silver bullet that could kill the werewolf of software engineering problems.

¹³This depends on application, but subjective view of the author is: over 60 FPS=good, less than 30 FPS = bad.

rendering functions requires 10-50 vertices to be submitted in order to overcome the array function call overhead.

Marselas also stated, that when vertex arrays are used, vertex (and other) data is copied into graphics card memory during rendering of each frame. This takes time, and if the data does not change, it is more viable to store it into the memory of graphics card, where it can be used directly without copying. This process is referred to as *caching*, and in GSE, it is realized by using OpenGL vertex buffer objects.

Vertex buffer objects (VBOs) allow application to store vertex (and other) data on the server-side high-performance memory of the OpenGL architecture (NVidia Corporation, 2003). The VBO is an extension to the OpenGL, and since OpenGL version 2.0, it is promoted as core functionality. By using VBO, vertex, normal, and texture coordinate data is transferred to the memory of the graphics card, and can be retrieved from there when rendering calls are made.

Adding VBOs to current engine architecture required slight compromises to isolation of graphics library code. Each VBO requires an OpenGL-compliant name(which is expressed with a specific OpenGL data type). This name is stored into GeometryData class as a member variable, because it was the only reasonable place. Since GeometryData objects should not know anything about the underlying 3D graphics library, this is a problem. If similar support exists for another 3D graphics library, and that library is included into GSE, its name storing scheme must be added to GeometryData class.

Controlling the way how VBOs are generated and altered, is more complicated matter. GeometryData class itself should not know anything more about the VBOs than a name. Actual controlling of VBO functionality is left for the OpenGL renderer. To solve this problem, I developed a communication method between OpenGL renderer and GeometryData class, which allows renderer to know when to create, regenerate or delete VBOs. A GeometryData object may hold one of the messages listed in Table 5 to the renderer at any time.

First item in Table 5 is the initial state. This state is also set if VBOs are not supported by underlying hardware, or an error has occurred during cache generation process. Second item is the principal item for creating the VBO cache. Third item is set when cache as been created. Fourth item in Table 5 is used to refresh a VBO cache by recreating it from vertex data. Last item is obviously used for deleting the VBO cache.

NO_CACHE	VBOs are not used, data is copied to graphics card memory for each render call.
REQUEST_VBO_CACHE	Next time the OpenGL renderer accesses GeometryData object, cache is created to graphics card memory, and message is set to CACHED_IN_VBO.
CACHED_IN_VBO	Data is already cached on graphics card memory, VBO is used render it.
REGENERATE_VBO_CACHE	Data in the vertex array is changed, cache in the graphics card memory is updated and message is set to CACHED_IN_VBO.
DELETE_VBO_CACHE	Delete cached data from graphics card memory, and set message to NO_CACHE.

Table 5: The VBO messaging scheme between OpenGL renderer object and GeometryData object.

C++ Template Metaprogramming

Matrix mathematics is one of the key elements in 3D programming. Isensee (2000) stated, that by optimizing the matrix functions using C++ templates, a 20% increase in performance was discovered in conjunction with 3 x 3 matrices. The improvement was accomplished by unrolling loops in multiplication, a trick which can be explained how loops are implemented using the assembly language. However, the importance of profiling was also stressed by Isensee: same optimization technique applied to 4 x 4 matrices caused over two-fold performance degrade. The algorithm itself is not changed when using these methods, so asymptotic time-complexity remains the same.

Templates have also other advantages than performance - they can reduce the amount of code. For instance, by declaring a matrix template with SIZE and TYPE parameter, arbitrary-sized square matrices can be created on the fly with the desired data type just by typing one line.

```
template <std::size_t SIZE, typename TYPE>
class Matrix
{
protected:
```

```
    TYPE m_aValues[SIZE*SIZE];  
public:  
    ...  
};
```

And to create 2 x 2, 3 x 3 and 4 x 4 matrices with float, following lines must be written:

```
Matrix<2,float> m_2x2matrix;  
Matrix<3,float> m_3x3matrix;  
Matrix<4,float> m_4x4matrix;
```

All of the matrices now have similar implementation and their functions are optimized during the compilation phase. The matrix class can now be implemented with double data type just by replacing the word *float* with *double* - all data types with one implementation. I implemented matrix classes in this way, and so far no performance problems have been noticed.

4.11 Libraries utilized

GL and GLU

OpenGL library provides the core 3D graphics drawing commands. GLU is an utility library for OpenGL library. It allows, for instance, rendering of simple geometric objects with simple function calls.

GLee

GL Easy Extension library handles loading of various OpenGL extensions. It provides run-time mechanisms for checking support for specific extensions in several platforms.

lib3ds

The lib3ds library provides support for loading and saving data into 3DS format. It is used to load models, materials and textures from 3DS files to the engine.

SDL

The Simple Directmedia Layer is a multimedia library for several platforms. It provides support for keyboard and mouse event handling, creating windows with OpenGL context, rendering 2D graphics, playing sounds, etc. It is primarily used for creating windows with OpenGL context and handling keyboard and mouse events.

SDL-image

SDL-image library is an extension to SDL, and allows loading of various image formats, such as JPEG, PNG, BMP, and GIF. It is used to create textures from images.

Ffmpeg

Ffmpeg library is a cross-platform audio and video streaming library, and is used to provide video textures. It supports various codecs, allowing various formats to be used in videos.

Freetype

Freetype library is used to render text on 3D screen with desired truetype font. It is used only for displaying 2D messages on screen. Fonts are not converted to transformable 3D objects, and are used in texture-like manner.

5 I Have Created a Monster! Time to Flee, But Where?

In earlier chapters, I have told that my motivation for doing this research had risen from a need to design a 3D engine for netWork Oasis project. I have explained that real-time 3D graphics are used in various places, such as in military training simulations, collaborative virtual environments, medical applications, games, 3D window managers, and so on. I have also briefly covered the history of hardware-accelerated 3D rendering.

I defined the 3D engine to be a two-edged sword; it is a higher-level abstraction of 3D graphics library calls where data is handled on object-level, and at the same time, it works as a combination of several techniques which accelerate the rendering process. I have introduced three different 3D engine projects available from sourceforge web site, and cross-compared their rendering features - concluding that their features are very equal.

I have explained the concepts involved in real-time 3D graphics, observing them from mathematical and computer graphics perspective. I have clarified the design of my 3D engine, starting from requirements, going through implementation details and optimization techniques, finally reaching the point where research questions must be answered. In this chapter, the research questions presented in Section 1.1 are answered by providing a list of necessary features and explaining them more closely. In the final paragraphs, directions for further work is discussed.

Questions and Answers

Question one - **which set of features is needed in generic 3D graphics engine library to provide basis for further development?**

1. **Vector and matrix manipulation package.** Each of the 3D engines will have to deal with linear algebra, mainly matrices and vectors.
2. **Scene culling.** Since all objects are not frequently on screen at the same time, there is no use to draw them all, so a method for determining their visibility is needed. By designing proper bounding volumes, and intersection test methods, visibility determination can be made faster.

3. **Texturing, materials and lighting support.** Various surfaces look more detailed with images and materials are applied over them. Also lighting improves the visual quality in some situations.
4. **Hierarchical scene representation.** Usually scenes consist of more than one object. Managing the relative and absolute positions of each object will be a nightmare without a logical structure.
5. **Animation support.** Some applications, such as games, use animated 3D models. Skeletal animation is popular and flexible method of character animation and must be supported.
6. **Spatial sorting of geometry.** When dealing with large objects, which have only a portion of their geometry displayed at a time, there is no need to draw them completely. Therefore, a method which sorts the vertex data into sufficiently small subsets and allows fast selection of an arbitrary set from them, is needed.
7. **Shader support.** Shaders can be used for various effects and techniques, ranging from skeletal animation to physics simulation calculation and lighting effects. The shader support is essential in order to be able to add various features to the engine in the future.
8. **Properly documented API with tutorials and examples.** This is required for any developer to be able to utilize the engine.

Question two - how the minimal set of features must be implemented in order to provide maximal reusability in different projects?

There exists numerous ways to implement these features, and these features probably have been implemented in somewhat different ways in every engine. And it is very hard to say, which ways are best or most efficient. In GSE, I projected my efforts towards reusability, and implemented the features in following ways.

Vector and matrix manipulation package

The mathematic operations are needed in more places than a humble programmer is able to count, and the operations are used frequently when rendering frames. This leads to the fact, that the mathematics package must be as efficient as possible to provide

a high frame rate. The mathematics package can be further optimized by processor architecture-based optimizations, such as Streaming SIMD Extensions (SSE), which may reduce the practical execution time by factor of three. This optimization does not affect the time complexity itself, but takes advantage of the pipeline provided by the processor, and enables execution of several commands within one clock cycle. My current implementation does not use SSE optimizations, but attempts optimize mathematics package using C++ templates combined with unrolling loops in multiplication process. I have designed the mathematics package so, that it can be extracted from the rest of the engine and used independently, if such a need should arise.

Scene culling

I have implemented camera and frustum classes, which allow intersection tests to be performed easily. I have also written several bounding volume implementations, and intersection test methods. My implementation includes axis-aligned and oriented bounding boxes, bounding spheres and cones. I have implemented intersection tests for frustum-sphere-, frustum-box-, sphere-sphere-, sphere-cone- and box-cone-pairs. I have integrated hierarchical view frustum culling into a scene graph, which handles also the updating of bounding volumes when position and orientation of objects change. These visibility determination techniques can be used easily, since I have encapsulated them into a single package, so those tests can be used in various other projects as an independent part. The culling methods, however, rely on features of the mathematics package and therefore require for it to be used as well. I have written the camera and view frustum classes without any bindings to the underlying 3D graphics API.

Texturing, materials and lighting support

I have designed materials and textures so, that they can be applied to renderable geometry very easily. Material properties in GSE be defined in files, which in turn can be converted into run-time objects my material manager. GSE is able to use variety of image formats in textures. This includes also videos, which are supported via ffmpeg library.

I designed the lighting to be very configurable, and it works per-object basis. I've implemented a light virtualization technique, that includes only the lights which contribute the most light to an object. In GSE, lighting works by defining illumination sets (which may include an arbitrary number of lights) that are attached to objects. In this way, very complex lighting schemes can be created.

Hierarchical scene representation

In my implementation, objects are grouped using a scene graph, which provides a simple way to handle hierarchical object structures, and automates position and orientation updating. For instance, if there exists object A, which is positioned to a specific location on object B, programmer can move both objects simply by moving object B, since position and orientation changes are reflected from parent node to child nodes. This removes a great deal of updating burden from a programmer, and encourages to create complex models constructed from several 3D objects.

Animation support

I designed 3D model animation support using skeletal animation. I have written helper utilities, which assist in converting an MS3D animation into an animation graph. Animation is controlled by defining a passed time for each frame. I implemented animation so, that it is controllable on joint-basis, meaning that animation can be stopped in some joints while continuing it on others. I designed a feature to animation renderer, which allows to use only a portion of an animation stored in MS3D file. This way, several animations for a single model can be included in one file, and used easily. Animation system supports also animation pausing, continuing and looping.

Spatial sorting of geometry

Very large models require much processing power to be rendered. In GSE, the process is accelerated by using octrees, which split the data into smaller pieces. Hierarchical view frustum culling is used to determine which pieces are visible during run-time. This feature allows very large terrains to be rendered without stressing the bandwidth of

the graphics processing pipeline. I implemented octree class to be simple and effective, general-purpose data structure, which can be used in many places.

Shader support

Shaders can be used for various effects and techniques. I have implemented the shader support for GLSL, which is a part of the OpenGL 2.0 standard. Both shader types, vertex and fragment shaders, are supported. In this implementation, shaders can be read from source files and inserted as rendering commands like any other rendering nodes - they do not need any special treatment. The shader support in GSE allows several special effects added into the engine, without actually changing the actual implementation of GSE. An example of such an effect is *blooming*, where light sources “leak” their light around them, and generate a glowing effect. Shaders are especially useful feature in the sense of the engine extensibility.

Properly documented API with tutorials and examples

The importance of documentation in software projects cannot be stressed enough. An API can be used by examining the source code, but it is not very intuitive. Properly designed examples help in the learning process, and allow the programmers to study the features of the library. For later use, a programming reference is needed when a specific feature description is required. Tutorials are very hard to create in this particular case, since there is not exact purpose, where the library should be used. In the absence of a better tutorial, I have provided an amply commented reference implementation for one project, and documented the entire API using doxygen. Doxygen generates, among other formats, an HTML-documentation of API functions for easy viewing.

I Stand on the Shoulders of Giants

I find the overall implementation of the engine to be quite satisfactory. Graph data structure seems to work very well as a basis for nearly everything. It is very flexible and new features can be added using the graph-renderer scheme. The purpose of GSE was not to provide a solution for a specific visualization problem, although it was ignited using a specific problem as a basis. Neither it is a generic 3D library, which would work

well out-of-the-box in all possible projects. The result is a 3D engine library, which can be used as a basis to build more sophisticated solutions to visualization problems.

From theoretical and technical perspective, the future research could focus to find out how several sections of GSE can be optimized even further, and what implications do these optimizations have. It would also interesting to know, what optimizations be done automatically via compiler, and what must be written manually.

The engine is never actually ready. Future projects might consist of implementing support for other graphics libraries, such as Direct3D (which might allow better performance on Windows platforms) and adding more effects to GSE, such as dynamic shadows, reflective materials, mirrors and so forth. These development projects could also put GSE in trial to test its extensibility, and results could be used to create a better 3D engine base design.

References

Akenine-Möller, T., Haines, E. (2002). *Real-time rendering Second Edition*. A K Peters, Ltd., Natick, Massachusetts, USA.

Alexandrescu, A. (2001). *Modern C++ design : generic programming and design patterns applied*. Addison-Wesley, New Jersey, USA.

Apple, (2007). *Mac 101 Customize Your Mac*. <http://www.apple.com/support/mac101/customize/5/> (11.7.2006).

Baum, D. (1998). 3D graphics hardware: where we have been, where we are now, and where we are going. *ACM SIGGRAPH Computer Graphics*, **32**(1), 65-66.

Barry, B.M. (1990). Using OOP for real-time programming (workshop session). *Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications*. ACM Press, New York, USA. 57-65.

Bertails, F., Ménier, C., Cani, M-P. (2005). A practical self-shadowing algorithm for interactive hair animation. *Proceedings of the 2005 conference on Graphics interface*. Canadian Human-Computer Communications Society, Ontario, Canada. 71-78.

Beryl-project.org, (2007). *Beryl Feature Spotlight*. WWW-page. <http://beryl-project.org/features.php> (11.4.2007)

Bilas, S. (2000). An Automatic Singleton Utility. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 36-40.

Brooks, F.P. (1987). No silver bullet: essence and accidents of software engineering. *Computer*, **20**(4), 10-19.

Byoungwon, C., Min Gyu C., Hyeong-Seok K. (2005). Simulating complex hair with robust collision handling. *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM Press, New York, USA. 153-160.

Crystal Space, (2007). *Main page - Crystal Space 3D*. WWW-page. http://www.crystalspace3d.org/main/Main_Page (3.6.2007)

Driesen, K., Hölzle, U. (1996). The direct cost of virtual function calls in C++. *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, New York, USA. 306-323.

Eberly, D.H. (2001). *3D Game Engine Design A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann Publishers, San Fransisco, USA.

Gamasutra, (2000). *Building an Advanced Particle System*. WWW-page. http://www.gamasutra.com/features/20000623/vanderburg_pfv.htm (23.6.2000)

Gamasutra, (2002). *A Real-Time Procedural Universe, Part Three: Matters of Scale*. WWW-page. http://www.gamasutra.com/features/20020712/oneil_pfv.htm (12.7.2002)

Gamedev.net, (2003). *Designing an Extensible Particle System using C++ and Templates*. WWW-page. <http://www.gamedev.net/reference/articles/article1982.asp> (19.8.2003)

Ginsburg, D. (2000). Octree Construction. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 439-433.

GNU Project, (2007). *GNU Lesser General Public License*. WWW-page. <http://www.gnu.org/licenses/lgpl.html> (14.4.2007)

Gross, M.H. (1998). Computer graphics in medicine: from visualization to surgery simulation. *ACM SIGGRAPH Computer Graphics*, **32**(1), 53-56.

Gröhn, A. (2006). *Collaborative Virtual Environments : When Real World Is Not Enough, We GLOW*. ftp://cs.joensuu.fi/pub/Theses/2006_BSc_Anssi_Gröhn.pdf (22.1.2006).

- Gwinn, J.M. (1992). Object-oriented programs in realtime. *ACM SIGPLAN Notices*, **27**(2), 47-56.
- Irrlight Engine, (2007). *Irrlicht Engine - A Free Open Source 3D Engine*. <http://irrlicht.sourceforge.net/faq.html> (20.2.2007)
- Isensee, P. (2000). Fast Math Using Template Metaprogramming. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 20-35.
- Kees, C.E, Miller, C.T. (1999). C++ implementations of numerical methods for solving differential-algebraic equations: design and optimization considerations. *ACM Transactions on Mathematical Software*, **25**(4),377-403.
- King, Y. (2000). Never let 'Em See You Pop-Issues in Geometric Level of Detail Selection. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 432-438.
- Lang, S. (1986). *Introduction to Linear Algebra Second Edition*. Springer-Verlag, New York, USA.
- Lengyel, E. (2004). *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, Massachusetts, USA.
- Linux.com Article DB, (2007). WWW-page. *Crystal Space: An Open Source 3D Engine* http://linux.omnipotent.net/article.php?article_id=11293 (20.3.2007)
- Lorensen, W.E., Cline, H.E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings of International Conference on Computer Graphics and Interactive Techniques*. (ed. Stone, M.C.). ACM Press, New York,USA. 163-169.
- Losh, E. (2006). The palace of memory: virtual tourism and tours of duty in

- Tactical Iraqi and Virtual Iraq. *Proceedings of the 2006 international conference on Game research and development*. Murdoch University, Perth, Australia. 77-86.
- Marselas, H. (2000). Optimizing Vertex Submission for OpenGL. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 353-360.
- Microsoft.com, (2007). *Windows Aero*. WWW-page. <http://www.microsoft.com/finland/windows/products/windowsvista/features/experiences/aero.aspx> (11.4.2007)
- NVIDIA Corporation, (2003). *Using Vertex Buffer Objects (VBOs)*. <http://developer.nvidia.com/attach/6427> (16.10.2003)
- OGRE, (2007). *About OGRE*. WWW-page. http://www.ogre3d.org/index.php?option=com_content&task=view&id=19&Itemid=79 (6.4.2007)
- Schildt, H. (1998). *Teach Yourself C++, Third Edition*. Osborne/McGraw-Hill, California, USA.
- SGI, (2007). *OpenGL Overview*. WWW-page. <http://www.sgi.com/products/software/opengl/overview.html> (11.4.2007)
- Sherrod, A. (2007). *Ultimate 3D Game Engine Design & Architecture*. Charles River Media, Boston, Massachusetts.
- Sun Microsystems, (2007). *Project Looking Glass*. http://www.sun.com/software/looking_glass/ (17.1.2007)
- Svarovsky, J. (2000). Quaternions for Game Programming. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 195-199.
- Sweet, M., Wright R.S. (2000). *OpenGL SuperBible, Second Edition*. Waite Group Press, Indiana, USA.

Ulrich, T. (2000). Loose Octrees. In *Game Programming Gems*, (ed. De Loura, M.A), Charles River Media, Inc., Massachusetts, USA, 444-453.

Wolfram Mathworld, (2003). *Polygon*. WWW-page. <http://mathworld.wolfram.com/Polygon.html> (17.6.2003)

Wolfram Mathworld, (2004). *Quaternion*. WWW-page. <http://mathworld.wolfram.com/Quaternion.html> (3.3.2004)

Woo, M., Neider, J., Davis, T.m Shreiner, D. (1999). *OpenGL programming guide: The official guide to learning OpenGL, version 1.2*. Addison-Wesley, Massachusetts, USA.

ZLib, (2004). *ZLib Licence*. WWW-page. http://www.gzip.org/zlib/zlib_license.html (3.10.2004)

Appendix 1: Source code of GSE

`ftp://ftp.cs.joensuu.fi/pub/Theses/2007_MSc_Anssi_Grohn.pdf.zip`