CHALMERS

An evaluation of techniques for use in a 3D computer game

Bachelor's Thesis

Rickard von Haugwitz
Daniel Lindén
Bartolomeus Jankowski
Magnus Olausson
David Sundelius

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2010

Abstract

In this thesis, we evaluate different techniques common in real-time rendering to see which are suitable for the purpose of implementing a 3D game application and what sorts of compromises would have to be made to make it feasible within a short timeframe and with little manpower. The case study application is intended to contain high graphical fidelity and entertainment level. To be able to evaluate this to any greater extent, we developed a racing game during the spring term of 2010.

During the development process, we found that time was of huge importance and that our end result could have been even more polished should we have had more time. However, we did implement many of the techniques and features that we were aiming for. Among these techniques were *deferred lighting*, *shadows*, *bloom*, *high dynamic range* and *particle systems*, all of which we consider to have turned out to be beneficial for the visual quality of the end result and suitable for our purpose. The final product is, however, clearly lacking content-wise, but content was not our main concern.

Although all planned effects and features could not be implemented within the given timeframe, the graphical fidelity of the application can, keeping the various limitations of the project in mind, be regarded as fairly high.

Acknowledgements

First of all, we would like to thank our supervisor Ulf Assarsson for providing invaluable help and feedback throughout the project, particularly during the writing of this report. Our thanks also go to former group member Alexander Göransson, who inspired the rest of the group with his enthusiasm for functional programming as well as his awesome beard, which is rumoured to be the longest in the Computer Science department. Furthermore, we express our thanks to groups 3 and 41, with whom we shared the project room. Lastly, we thank Anders Davallius for feedback and advice regarding game development, and other friends for their support.

Contents

Chapter 1

Introduction

1.1 Purpose

This thesis aims to discover and evaluate what techniques are suitable for engineering of real-time computer graphics in projects with a short time span and few developers. Its purpose is to highlight certain of these techniques in order to identify the aspects of a game application that make it enjoyable to watch and use.

Our aim was also to implement a game application to help support the intentions stated above, with the ambition of making it as entertaining as possible and with a high graphical standard.

1.2 Limitations

The research is limited by the computers which are used for development; a graphical application should be runnable on these since they are fairly representative to what our target audience has available. Also, since the project is aimed towards evaluating rendering techniques, our main focus will not be on different network technologies or physics. However, we have implemented basic networking and physics functionality in order to provide the user with a more complete gaming experience, and therefore such algorithms are covered, albeit briefly, in the report.

1.3 Background

Computer graphics is an area which has seen rapid development in recent years. Films use computer graphics to increasingly better effect; chances are good of having seen a computer-generated visual effect scene last time going to the cinema, regardless of it being a drama or fantasy film, although the frequency of the visual effect shots generally vary between genres. These scenes or shots are often very detailed and take a long time to render. In gaming, which in contrast to film involves rendering in real time, graphics have become more realistic, and convincing human characters are now possible and even common in modern state-of-the-art game applications. Here, completely different prerequisites apply due to the real-time requirement, and special algorithms and techniques have to be used in order to deliver a convincing scene while maintaining an acceptable rate at which screens are rendered to the display.

The computer generated scenes in games, and, for that matter, films, are built using triangles. Each triangle is shaded and/or textured individually and then drawn to the screen, which makes the scene come to life. This process is generally too heavy for the central processing unit (CPU) to handle all by itself, particularly when it is required to perform other tasks, so for shading and drawing these triangles to the display, a graphics card designed especially for drawing triangles is used. On this graphics card one finds, among other things, the *graphics processing unit*.

1.3.1 The graphics pipeline

The graphics processing unit (GPU) operates by pushing data down a *rendering pipeline* consisting of several stages, each of which performs a specific function. In modern GPU:s, this pipeline is implemented with the *vertex shader*, *geometry shader*, *clipping*, *screen mapping*, *triangle setup*, *triangle traversal*, *pixel shader* and *merger* stages, in that order (Akenine-Möller et al., 2008). Out of these, the vertex shader, geometry shader and pixel shader stages are fully programmable and therefore of foremost interest to the programmer; the clipping and merger stages are configurable but cannot be programmed, and the remaining stages are fixed. There are currently two widely accepted standard graphics pipeline models: Microsoft's Direct3D (part of the DirectX standards suite) and the open standard OpenGL.

Early commercially available GPU:s, although often providing highly configurable rendering pipelines, did not support programmable shaders; it was not until 2001 that the first GPU (NVIDIA's GeForce 3) supporting programmable vertex shaders was released. During the following year, this was expanded upon, resulting in DirectX 9.0 and Shader Model 2.0, the first to support fully programmable vertex and pixel shaders (Akenine-Möller et al., 2008). The introduction of programmable shaders gave the graphics programmer significantly higher control. Shader programming was originally done using assembly language, which quickly proved too cumbersome as shaders became increasingly long and complex. In order to alleviate the increasing burden placed on the programmer, higher-level shader programming languages were introduced. Included with DirectX 9 in 2002 was HLSL (High Level Shading Language), and OpenGL saw the introduction of GLSL (OpenGL Shading Language) at around the same time (Akenine-Möller et al., 2008). Graphics hardware are classified by shader capabilities within the DirectX model using the *Shader Model* concept, the current version being Shader Model 5, introduced along with DirectX 11.

1.3.2 Ever-increasing speed

The prime contributing factor that has made many of today's graphics effects possible is the continually evolving computation speed and memory size and bandwidth in the graphics hardware. To the end of aiding graphics computations, microprocessors have also had their floating-point units (FPU:s) improved, allowing more floating-point computations to be executed per clock cycle. The area of computer graphics has seen a consistent development of more and more functionality being implemented directly in the hardware in order to increase computation speed as much as possible. A major step in this direction occurred in 1999 and 2000 when NVIDIA and ATI, respectively, released the first commercially available GPU:s featuring hardware-implemented Transform and Lighting (T&L) capabilities, thereby freeing the CPU of substantial calculations (Sanford, 2002). This quickly became the industry standard, with most new games expecting hardware-implemented T&L, enabling significantly higher limits on performance and thus more advanced effects to be implemented. This development has continued to the present day, and graphics hardware manufacturers constantly strive to increase available memory and the amount of computations executed in parallel per clock cycle on the graphics card.

The current trend with both GPU:s and CPU:s is that they are built with more and more cores, enabling even more parallelism and hence more computing power. In certain cases GPU:s are becoming more like CPU:s in that that they are used for more than computer graphics. Intel's *Larrabee* GPU is one example of how GPU:s are moving towards a more general-purpose architecture. Its architecture is based on multiple in-order CPU cores and it has fewer fixed-function units, which makes it more programmable than current GPU:s (Seiler et al., 2009).

1.3.3 Graphics in games

In modern games, a lot of effort is put into delivering as good graphical content as possible. In order to achieve this, developers often try to add large amounts of effects as well as utilise techniques with high capability for visual realism. However, trade-offs in the use of different techniques and algorithms have to be made due to hardware limitations to maintain acceptable frame rates, system response times and in some cases network load.

1.4 Problem

1.4.1 The rendering equation

How to render realistic graphics is described in the rendering equation (Kajiya, 1986) that normally is written on the form:

$$L_o(\mathbf{p},\mathbf{v}) = L_e(\mathbf{p},\mathbf{v}) + \int_\Omega f(\mathbf{l},\mathbf{v}) \otimes L_i(\mathbf{p},\mathbf{l}) cos\theta_i d\omega_i, \tag{1.1}$$

where $L_o(\mathbf{p},\mathbf{v})$ is the outgoing light from a point $\mathbf{p}$ in a direction $\mathbf{v}$. The result of equation 1.1 is represented by the final colour of each pixel. $L_e(\mathbf{p},\mathbf{v})$ is the emitted light from a point $\mathbf{p}$ in a direction $\mathbf{v}$. $\int_\Omega$ is an integral that integrates over all incoming directions in the hemisphere. $f(\mathbf{l},\mathbf{v})$ is the *bidirectional reflectance distribution function* (BRDF), which is a function that describes the reflectance of the material based on the incoming direction $\mathbf{l}$ and outgoing direction $\mathbf{v}$. $L_i(\mathbf{p},\mathbf{l})$ is the light from an incoming direction $\mathbf{l}$.

The rendering equation is far too complex to be used for real-time rendering. Instead, it must be approximated by algorithms that can be executed must faster on modern graphics hardware. Our problem is mainly to find algorithms that can do this effectively in real time, and, if there are several options, investigate these algorithms in order to find the one that is best adapted to implementation in a real-time application.

Emitted light

The emitted light component is not a part of the integral and hence fairly trivial to implement. However, there are a number of ways in which this can be done, and we must compare the performance loss to the gain in visual quality in order to find the most appropriate technique.

BRDF function

There are several BRDF models available, and one of the most common is the Phong lighting model. Each of the different BRDF models provided a visual look that suits a special kind of material. It is most likely that several of these are needed in a single application in order to provide the maximum realism when rendering the different materials visible in a scene.

Incoming light

Approximating the incoming light is a problematic task as we need to consider many different phenomena.

Shadows There are techniques available for displaying high quality shadows, but those might not be applicable to a real-time fast racing game. We have studied several shadowing techniques in order to determine how well they are suited for the purpose of this report.

Indirect illumination When light hits a point, it is partially reflected, in either a diffuse or in a specular reflection. These reflections cause surfaces that are not directly hit by light to still be lit up. This also means that the colour of the surface that reflected the light will "bleed" into the receiving surface.

Ambient occlusion When an object is close to a point, some of the ambient lighting in the scene is occluded. This is called ambient occlusion.

1.4.2 Post-processing effects

There are several types of effects that can occur as artifacts in lenses, and, when simulated in a game, increase the sense that the image was produced with a camera rather than having been computer generated.

Bloom

Bloom refers the phenomenon where a very bright point "bleeds" light onto its surroundings in image space, reducing contrast.

Depth-of-field blur

Depth-of-field blur occurs when a lens has its focal length set at a certain distance, and objects closer and farther from the focal point get blurred.

Motion blur

When an object moves relative to the camera, the time in which the shutter is open can cause the light receptors to capture the positions of parts of the object at several locations. This causes the effect called motion blur.

1.4.3 Other effects

In a racing game, there are several other types of effects that might be considered for inclusion in order to achieve as striking graphics as possible, *e.g.* a particle system that can handle a lot of particles in order to simulate, for example, gases, fluids, collision sparks or exhaust plasma from a racing ship.

1.5 Method

The research for this thesis was partly done by implementing and testing some of the techniques that were to be evaluated. To be able to evaluate the techniques available to render convincing graphics, the decision was made to implement a game application and use it as a platform. There are many different game genres, ranging from sport games to role-playing games, each of which comes with its own set of challenges and possibilities. Out of several different alternatives proposed for the genre of the game application, the racing game was decided upon. The racing game alternative was among the project group members considered to have a large potential for implementing interesting shading, particle and post-processing effects in a 3D world. The vision was that of creating a game with hovercrafts rather than racing cars. The decision to build a racing game, and moreover one with hovercrafts, also meant that the game application could be built without worrying too much about large amounts of animation (for example, there would be no need to animate wheels). This was considered a good thing since the main focus was decided to be upon creating and evaluating the use of different rendering and effect techniques.

The rest of the report describes this case study, from idea to finished product. The development of the game is divided into three main chapters: *Game Design*, *Graphics* and *Game Engine*. Game Design describes the planning stage, development methods and interaction design of the game. The Graphics chapter describes our work of choosing the correct algorithms and techniques for creating a visually appealing game. The last chapter, Game Engine, provides information about how the game engine is implemented. Finally, the results of the research are presented to show how all parts of a game affect the visual quality of the end product.

SLERP 3D

The name of the case study project is *SLERP 3D* (*Space Laser Epic Racer Power 3D*). It is a fast paced racing game that takes place in space. The game has one track on which the user can compete against other over LAN, or the user can choose to beat a record time with the in-game race timing system.

1.5.1 Agile Development

The software engineering method chosen for the project was *Agile development*, AD. AD is a develompent model that evolved as a reaction to the waterfall model. AD is defined by the *Agile Manifesto* (Beck et al., 2001), a document consisting of twelve principles dictating how a development team should do their work.

AD began forming in the 1990s as an alternative to the waterfall model. In the waterfall model development of software can be broken down into four main components: *requirements*, *design*, *implementation* and *verification*. This model works when you have a solid requirement set. The four steps are then done in sequence, *i.e.* starting on the implementation is not done before the whole design phase is completed. This way of developing makes requirement changes hard and expensive to implement.

With AD, development is done in smaller time frames (1-4 weeks instead of the waterfall model's, which is one time frame for the entire project). For each iteration a full software cycle is done (from designing to testing). This short iteration period enables the project to dynamically adapt to changes without the large cost of refactoring that the waterfall model has. In the Agile Manifesto, the second principle states:

> *"Welcome changing requirements, even late in development."*

AD was chosen as a development method since it strongly encourages face-to-face meetings between the members, something we considered necessary for this project. From the Agile Manifesto:

> *"The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."*

This also worked well since the group had to use computers in one specific room, making daily meetings (as encouraged by AD) easy to have.

1.5.2 Design and UML

During the design process, the Unified Modelling Language (UML) was used. The UML provides good visualisation of the software components, which greatly helped getting a general view of how the different parts of the system will interact.

1.5.3 Programming language and framework

The choice of programming language, as well as the possible framework, is very important for the project and a considerable amount of time was spent choosing the right one for our intentions. Also, the question of which 3D API (application programming interface), OpenGL or DirectX, should be used had to be answered.

The language has to be high-level enough to free the programmer from such mundane tasks such as memory allocation/deallocation, implement basic data structures, etc., while retaining high performance and flexibility.

The framework should provide the project with ready-to-use components such as network libraries or sound libraries and relieve the programmer from having to write these components from scratch.

A couple of different languages and Integrated Development Environments (IDE) were considered. Among the discussed options discussed were: Java/Eclipse, Java/NetBeans, C++/MSVS, C#/MSVS and C#/XNA; also, Haskell and Python were brought up in the discussions.

Finally, a Microsoft-oriented environment was agreed upon; the language to develop the game in was chosen to be C#, the framework XNA Game Studio and the IDE to be Microsoft Visual Studio. Regarding 3D-API, this choice left DirectX as the only option.

The reason for choosing C# was its simplicity and generality. C# provides support for such software engineering principles as object orientation, strong type checking, array bounds checking and automatic garbage collection. It also integrates very well with XNA.

XNA is a framework developed by Microsoft, based on the native implementation of .NET Compact Framework 2.0 for Xbox 360 development and .NET Framework 2.0 on Windows. It is targeted at game development, and as such includes an extensive set of game development specific class libraries and tools. Together with C#, XNA felt as a natural choice of platform for our game development.

1.5.4 Version control

Software development in larger teams demands some kind of a version control software; for this purpose Subversion (SVN) was chosen.

Chapter 2

Game Design

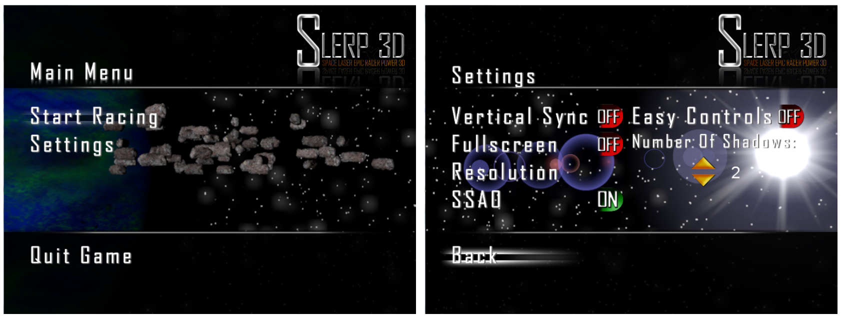
2.1 Game Presentation

For the game to feel polished and of high quality, care needs to be taken with even the fine details. With smooth transitions between different states of a game comes a seamless feeling and, if done well, can give a game the extra flair needed to really stand out. Such a transition can, for example, be a sweeping motion or a fading of menu alternatives, revealing the next state of the game.

Hargreaves (2007a) talks about this in one of his blog posts and notes that transitions are everywhere, from the selection of menu alternatives to HUD (heads-up display) changes. He also highlights, in another blog post, the importance of always having the menus responsive to user input, so that they are easily skipped. This is to avoid the users' annoyance of having to endure time-consuming transitions or displays when in some kind of impatient mood (Hargreaves, 2007b). In addition, as Hargreaves hints at, it is convenient for debugging purposes.

2.1.1 Usability

As with any interactive product, it is important in a computer game to consider how to design the interface for human interaction; everything from the menu system to the controls of the in-game unit (for a racing game this would most likely be a vehicle of some sort) controlled by the player. These issues are baked into the concept of usability, which is built up from the following usability goals according to Sharp et al. (2007): *Effectiveness* describes the effect of the product (application) in relation to what it set out to accomplish. How well and efficiently the functionality of the product can be used is referred to as the *Efficiency*. *Safety* refers to the characteristics of a product that enables it to be interacted with without any hazard to the user; be it a real world hazard or loss of work, data or other effects of correct or incorrect user behaviour. *Utility* describes the product's ability to provide the user with appropriate and expected features in order to satisfy the user's needs. People do not generally want to spend large quantities of time trying to learn how to use a product. How easy a product is to learn to use is described by the *Learnability* goal. *Memorability* is connected to how well the interface is designed in order to make the user remember how to operate the product.



Vertical Sync OFF
Fullscreen OFF
Resolution
SSAO ON
Easy Controls OFF
Number Of Shadows: 2

(a) The main menu of SLERP 3D. The default option selected will lead the user one step closer to playing the game.

(b) The settings screen of SLERP 3D. There are several features that can be turned on or off.

Figure 2.1: Examples of menu screens in *SLERP 3D*

Sharp et al. (2007) also mentions in their book Interaction Design how these rules can be used to form usability criteria, which in turn can be used to actually test the product and get numbers from it. Another important aspect that they discuss in their book is the user experience. There are many different experiences related to the interaction with a product, and also the experience varies between people; therefore there are naturally also many different goals. This subject essentially involves reactions to different situations encountered when using the product, be it a negative, positive or neutral reaction.

Computer games suffer a little in this area due to a delicate dilemma, since many players desire the challenge of a difficult video game. This goes against many of the usability goals described above, which also has to be considered(Sharp et al., 2007).

2.1.2 Result

When it comes to the usability of SLERP 3D we have made several deliberate decisions.

The menu system basically works as follows: The user is shown a screen presenting the available choices. When picking an alternative a new screen is shown, in turn presenting the alternatives for the user's next action. We have designed the menu flow in such a way that the button highlighted on default (that is when you are facing a new screen) is always the button that will get you fastest into the game. This makes it easy to start a new game fast and not only is this good for debugging purposes and testing, but also for users who know the menus well and want to enter the game without wasting time manoeuvring the menus.

In the menus, the buttons are arranged so that the option to go up (from now on referred to as the "back button") in the menu hierarchy is only one keyboard interaction away as default (two interactions if counting the actual selection of the option) if the user chooses to utilise the menu alternatives displayed. There is also an option to use the *Escape* button on the keyboard to achieve the same result. This is to make it as fast and convenient as possible to navigate with ease, which is related to the *efficiency* usability goal. Also, the layout is designed in such a way that the back button is positioned some distance away from the other menu options in order to avoid that the user mistakenly presses this button trying to do something else. Using the keyboard, it is true that the back button is still only one press of a button away from the button selected by default, but in this case this navigational route was deemed convenient. However, *SLERP 3D* also supports mouse input, and when using the mouse to navigate the menus, the separation of the back button and the rest of the buttons is indeed convenient and strengthens the *safety* of the menu.

The menu is in its presentation quite simple and shows only what is necessary to guide the user forward and has the user select as few alternatives as possible to begin playing the game. This increases the *learnability* of the product. In-game, the game is interacted with through the keyboard only. We have chosen to implement a fairly complicated standard control which we thought would enrich the gaming experience and give the user very good control of the vehicle. However, we discovered that this layout was quite difficult to learn quickly, so we decided to include an alternative easy scheme in order to provide choice and increase the game's *learnability*.

The ability of *SLERP 3D* to deliver what is expected of it is described by the *utility* property. In the case of a racing game, there are several things that we believe the game should provide. It should offer different game modes to enable several kinds of interactive experiences. For example, a user who prefers to play solo would perhaps expect a racing game to have a campaign mode of sorts where they can race against AI-controlled opponents, and also a time trial mode. *SLERP 3D* features no campaign mode, however, and although a form of time-trial is implemented, it does not feature a leader board or any other means of comparing oneself to others or one's own previous achievements.

When it comes to expectations on multiplayer racing games, we believe the following game modes could be expected: LAN, Online and Split screen multiplayer. A user looking for multiplayer races will find that *SLERP 3D* features LAN multiplayer races, but lacks the other two modes mentioned.

Concerning all games for the PC platform, we believe that a certain amount of customisation is not only expected by the user, but also necessary due to the vast amount of different hardware setups on the market and in the consumers' hands today. Therefore *SLERP 3D* provides options to change many graphical settings (see figure 2.1b), as well as the control layout.

Clearly *SLERP 3D* is not a complete package and does not offer everything expected from a full-fledged racing game. This is due to several factors, among them the limitations we have had on this project. For example, our choice of development platform restricted us to LAN networking, we had a limited development time, and our aims with the project made us prioritise features to implement.

2.2 Software Architecture

When the size and complexity of software systems started to increase, a new problem emerged for the developers; the overall system design. To be able to create a complex system such as a computer game, a good structure of the program and the process is essential. The software architecture of a given software is a description of this structure, what the different components the software consists of and how they interact (Garlan and Shaw, 1994).

2.2.1 Results

The software architecture of *SLERP 3D* is based on design patterns, and was planned from the beginning to be extensible to the work we were planning to perform. The subsystems for rendering, game logic, reading/writing to hard drive, network interfacing and sound are clearly separated into different namespaces and classes. The base of the application is the state handling system. It contains a stack on which the program pushes game states, *e.g.* a new in-game state, the main menu or the title screen. The state machine then sends update signals down hierarchically to each updatable object (defined by the `IUpdatable` interface) in the current state. If a state returns the boolean value true from the update call, signaling that it is finished, it is popped from the stack and the previous game state reappears. The rendering is handled by the `Graphics` namespace, which contains classes for management of the effects that can be used for shading, as well as lighting and the particle system. The main class for rendering is the `GraphicsManager`, which uses the *Singleton* design pattern to ensure that only one instance is initialised, and the `Scene` class, which contains information about all the nodes (renderable objects) in a scene. The network is using the singleton `P2PManager`, which controls peer-to-peer networking functionality. Sound is modulated with the class `SoundManager`, which is used as an interface to the XACT framework of XNA.

The *SLERP 3D* architecture is extensible and modulated so that new functionality is easy to implement, especially in the graphics namespace. This was essential to our development process, since our research demanded use of evolutionary development. The design patterns used helped our development and cooperation during the project.

Chapter 3

Graphics

3.1 Rendering

3.1.1 Shading

The shading of a surface is the process where the outgoing light from a shaded point is calculated based on the type of material and the lighting conditions in the point. Shading can be evaluated at different resolutions as a balance between performance and visual quality, where higher resolution results in slower execution of the application. In modern games that strive for state-of-the-art graphics, shading is performed per pixel, a technique known as *Phong shading* (Phong, 1975). In older games, it was more common to evaluate the shading per vertex, called *Gouraud shading* (Gouraud, 1971), and then interpolate the result, or even to perform the shading per polygon (*flat shading*). In order to simulate the type of material used for the surface, a certain BRDF (Bidirectional Reflectance Distribution Function; see section 1.4.1) model based on the type of rendered material is used. These BRDF models can be quite expensive to evaluate, so in some cases it might be preferable to implement a faster BRDF model at the cost of a less correct result.

When calculating the lighting, it is common to divide the light reflected off the material into two terms: *diffuse* and *specular* (Akenine-Möller et al., 2008). Specular refers to the high-frequency reflection of the light source by the material. The diffuse term corresponds to the light reflected by the material as a result of transmission, absorption and scattering of the incoming light. The Blinn-Phong BRDF model (Blinn, 1977) is likely the most common model in real-time computer games, and produces materials that look quite plastic. For metals and similar materials, the Cook-Torrance model (Cook and Torrance, 1982) can be used instead in order to provide a more realistic look. Other BRDF models include, for example, Oren-Nayar (Oren and Nayar, 1994), Ward (Ward, 1992) and Ashikhmin-Shirley (Ashikhmin and Shirley, 2000). A reference of many BRDF models can be found in Hoxley (2008).

Transmission, absorption and scattering

When light travels and reaches a media with a new refraction index, it will *scatter*, causing the light to change direction. The phenomenon when the light is scattered and travels into the matter is called *transmission*. Inside the matter, the light can either continue to scatter until it travels out of the matter, or get converted into another form of energy due to *absorption*.

3.1.2 Forward rendering

The traditional way of rendering graphics is to use the *forward rendering* technique, where objects are rendered directly to the screen and shaded while they are rendered. This technique is good for older hardware, which does not have a large amount of video memory available to hold the intermediate buffers used by other rendering techniques. Forward rendering is also capable of handling transparency if the transparent objects are sorted from front to back before being rendered. If the objects are not sorted, the final result might be incorrect depending on the chosen blending operator.

3.1.3 Deferred rendering

Deferred rendering is a rendering method where one or several *geometric buffers* (*g-buffers*) are used in order to store data about the visible pixels on the scene. Depending on the needs of the application, these g-buffers can contain information about position, normal, texture or other surface and material data. This data is then used to produce the final image, and thus the shading is calculated only once per visible pixel on the screen, which is not normally the case for forward rendering techniques. However, transparent objects are problematic to render in the deferred pipeline as the object that is visible through the transparent object would also need to be shaded, something that deferred rendering algorithms do not support. Instead, a fallback to traditional forward rendering algorithms for the transparent objects is necessary.

Deferred shading

Traditionally, using forward rendering with the so called *multi-pass* technique, the object is rendered and shaded once for each light that affects it; if objects in a 3D scene is affected by several light sources this can be quite costly (Akenine-Möller et al., 2008). Another method when utilising forward rendering is the *single pass* technique. Here, all lights affecting an object are calculated and then one shader is used to render the lighting and material, which results in having to use a unique shader for each light/material combination possible (Valiant, 2007). Also, redundant shading may be applied. For example, if a pixel contains more than one fragment and one fragment covers another completely, the shading has still been computed on both fragments (Akenine-Möller et al., 2008).

The main deferred rendering algorithm has so far been *deferred shading*. Deferred shading is possible due to the availability of *multiple render targets* (MRT), which are used to store information in geometric buffers. Firstly, a rendering pass is performed to gather information about the closest visible surface. This information, which usually includes z-depth, normals, texture coordinates and material parameters, is then stored to multiple render targets. These g-buffers are then used by the shader programs to compute lighting and post-processing effects (Akenine-Möller et al., 2008). This gives the method several advantages. Light-shading can be optimised to only be computed on areas that are affected by the lights using geometrical shapes. This makes the method very efficient when handling many light sources. Deferred shading also separates lighting from material definitions, reducing the amount of shaders needed (Akenine-Möller et al., 2008).

However, deferred shading has three major disadvantages. Firstly, the g-buffers demand large amounts of video memory and memory bandwidth. Secondly, the technique cannot make use of hardware antialiasing on older hardware. This is due to the fact that the lighting is handled as a post-processing effect and therefore added to the back buffer after antialiasing has already been applied. This will nullify the effect the antialiasing had on the image, since aliasing appears after the lighting is done. Finally, alpha blending is not possible due to the fact that only one object per pixel is stored in the g-buffer (Akenine-Möller et al., 2008) (Koonce, 2007).

Nevertheless, there are solutions to the antialiasing and transparency problems. Both Shishkovtsov (2005) and Koonce (2007) suggest using an edge-detection technique, while *Guerilla Games* found a introduction of hardware supporting the Direct3D 10 standard, supplying means of accessing data needed to perform multi-sample antialiasing (NVI, 2006).

To solve the problem of alpha-blending, transparent objects are often rendered using a forward renderer after the deferred shading has been completed and then added to the scene. This is the case in the shipped games *Tabula Rasa* and *Killzone 2* (Koonce, 2007; Valiant, 2007), as well as *StarCraft II* (Filion and McNaughton, 2008).

Deferred lighting

The deferred shading algorithm requires large amounts of available render-to-texture memory in order to store all the g-buffers. This is viable on mid- to high-end PC systems as well as on the PlayStation 3 console. However, in the Xbox 360 there is only 10 megabyte of available render-to-texture memory (Akenine-Möller et al., 2008) and thus deferred shading is not always feasible. *Deferred lighting*, also known as *pre-pass lighting*, is an algorithm proposed by Engel (2008) that tries to solve this by using a much lower amount of g-buffers. There are three main passes in the light pre-pass algorithm: generating the g-buffers, performing the lighting, and shading. Similarly to the deferred shading technique, the g-buffers are generated by drawing the geometry, the only necessary data for the next pass in the technique being the normal and the depth of each pixel on the screen.

The second pass uses these g-buffers in order to generate a *light accumulation texture* (see figure 3.1) which contains the light of each pixel on the screen. Each light to be added to the scene is rendered with a bounding volume that fully bounds the light volume. A full screen quad would also be possible, but in order to run the pixel shaders only on the pixels which actually can be lit, a bounding volume is preferred. Each light is rendered in this fashion, and the light contribution is additively blended into a light accumulation buffer where the diffuse and the specular components are stored in separate channels.

The last pass renders all geometry again and performs lighting based on the light accumulation buffer in order to produce a lit image of the scene. This is also the pass where algorithms such as screen-space ambient occlusion, described in section 3.4.2, are integrated into the image.

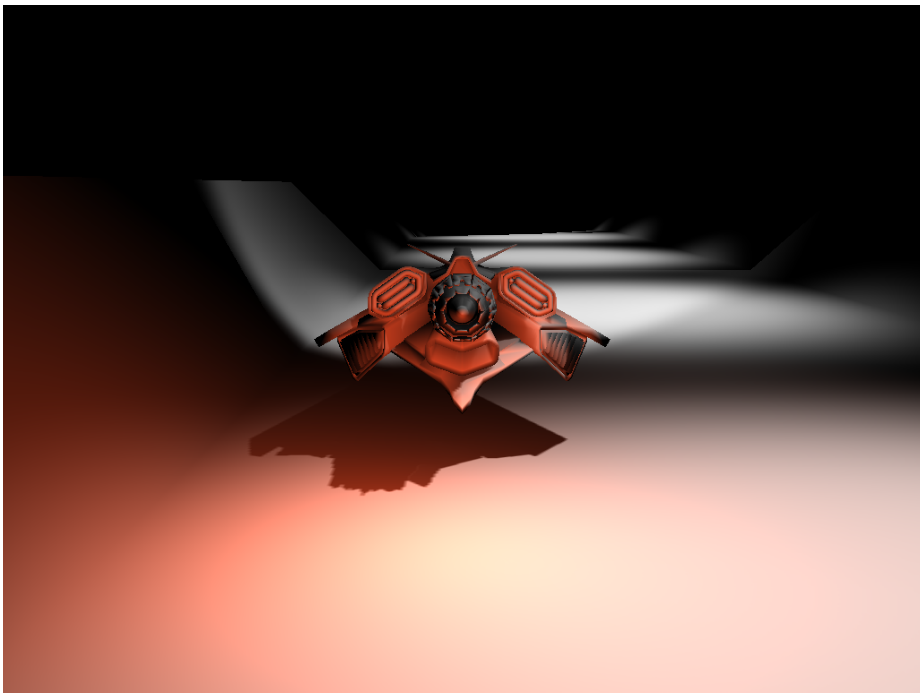


Figure 3.1: The light accumulation buffer used in *SLERP 3D*. The reddish tint is caused by light emitted from a red light source placed behind the ship, which is visible in the centre of the image. A number of spotlights are shining white light down on the track, and this light is accumulated with the illumination from the red light source and the white ship headlights.

Deferred lighting is, just as deferred shading, a very fast technique for rendering large amounts of light sources, and scales very well with increasing amounts of lights. The downsides are, like for the deferred shading technique, that transparent objects are not handled very well and require special treatment. Another algorithm, which is similar to deferred lighting, is *inferred lighting* presented by Kircher and Lawrance (2009).

3.1.4 Gamma correction

CRT displays are affected by a non-linear response curve from the input colour due to the relation of input voltage and the displayed intensity[1]. Thus, in order to produce the correct image, *gamma correction* needs to be taken into consideration during the whole graphics pipeline. Gamma correction is when intensity values of pixels are converted from or to linear space. Images that are created by artists are normally stored in non-linear space, and in order to properly use them in calculations this has to be taken into account. Modern graphics hardware has built-in functionality to perform this conversion, and the textures can then be used normally. All lighting calculations should be performed in linear space in order to avoid incorrect calculations that could

[1]This is also emulated on modern LCD displays.

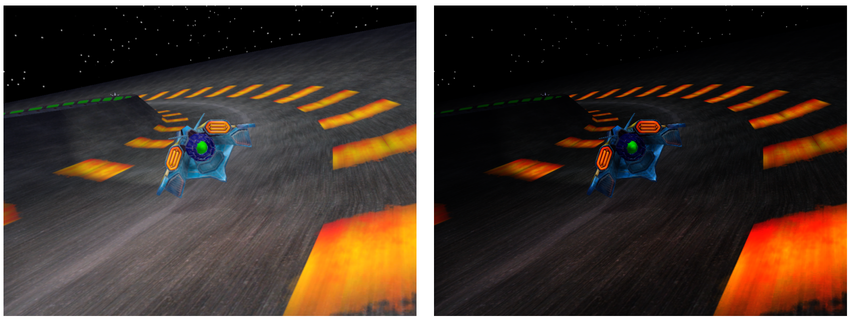

(a) Gamma correction on (b) Gamma correction off

Figure 3.2: Comparison of gamma corrected output in *SLERP 3D*.

result in some areas becoming too dark (Gritz and d'Eon, 2007), see figure 3.2. When sending the final image to the display it is important to make sure that the image is in non-linear space; this can be performed efficiently by modern graphics hardware. The current standard gamma in computer graphics is known as sRGB, and modern graphics hardware has support for conversion to and from this standard when sampling texures or when writing to render targets.

3.1.5 Results

In *SLERP 3D* we have implemented deferred lighting, enabling us to have a very large number of light sources on the screen. Tests carried out with around 300 light sources showed no significantly reduced performance. We do also have a way to render transparent objects in our renderer. However, those objects will not be sorted, so in certain conditions artifacts may occur. Deferred lighting also enables us to add minor light sources on objects which are expected to emit light into the surrounding environment in order to increase the perceived realism of the scene. The algorithm itself imposes little to no compromises on the quality on the lighting. Since we are only using phong shading, the material of our objects does not have quite the same characteristics as the corresponding materials in reality when considering *e.g.* specular and diffuse reflections or sub-surface scattering (Akenine-Möller et al., 2008).

We have taken some measures to make sure that our pipeline was gamma correct. In XNA, the application cannot directly change the gamma correction output mode on the GPU. Instead, this has to be done through the DirectX effect framework when writing the shader code.

Because of the fact that we only developed graphics for the PC platform, we could have chosen to use the deferred shading algorithm. However, we did not quite need the added g-buffers for most of our other visual effects, and thus we decided to implement deferred lighting.

The main work for us when trying to make sure that our pipeline was gamma correct was to correctly specify for all texture samplers in the shaders whether to use sRGB conversion or not.

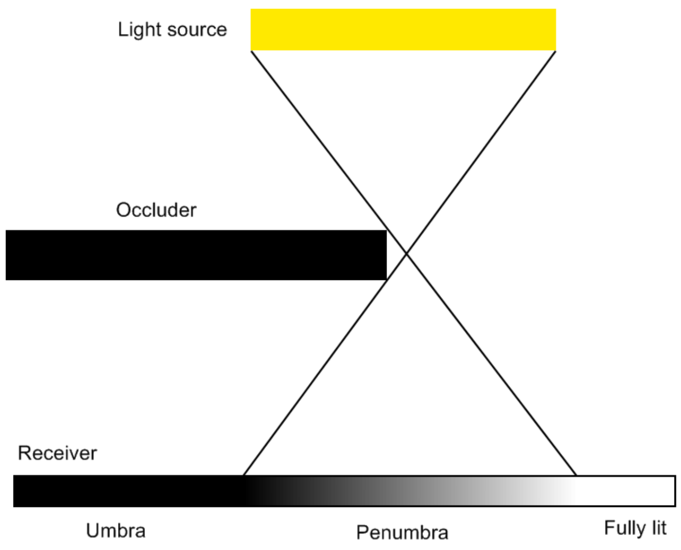


Figure 3.3: Illustration of a light source that gives rise to an umbra and a penumbra on a receiving surface.

3.2 Shadows

In the rendering equation, the lighting of a surface is dependent on the number of direct photons that reach the surface point. Therefore, any occluder that prevents photons from reaching this surface will impact the lighting of the surface point. Real-time dynamic shadows on arbitrary surfaces is something that is desirable to include in order to improve the visual quality of the game; such shadows also help the user to perceive depth and the relative position of different objects in the scene correctly. An important problem that some of these algorithms try to solve is how to correctly produce the soft shadows that occur due to the fact that light sources has a certain area. This, in turn, gives rise to an *umbra* and a *penumbra*, as can be seen in figure 3.3. The penumbra varies in size depending on the distance from the occluder, the distance from the light source and the size of the light source.

There are two main techniques available for implementing dynamic shadows in real times: *shadow mapping* (Williams, 1978) and *shadow volumes* (Crow, 1977). This text focuses on various shadow mapping techniques, since shadow mapping scales well with high geometrical complexity and is well suited for use with deferred lighting, wherefore it is very popular in modern computer games.

3.2.1 Shadow volumes

Shadows can be thought of as volumes where no direct illumination from a light source reach due to occluders blocking the photons. The shadow volumes technique uses this in order to create sharp dynamic shadows which can be cast on arbitrary geometry. These shadow volumes are extended from the edges of all the occluding geometry in the direction away from the light sources towards infinity. Every rendered pixel that is inside a volume is considered to be in shadow. This technique was later extended to make use of stencil buffers in order to improve the rendering speed (Heidmann, 1991). Also, Assarsson (2003) has presented an algorithm based on shadow volumes that can be used render soft shadows in real time.

3.2.2 Shadow mapping

Shadow mapping is an image-based technique for rendering real-time dynamic shadows on arbitrary surfaces (Williams, 1978). The algorithm is based on the creation of a *depth texture* from the shadow point of view, where each pixel contains the distance from the light to the rendered geometry. When rendering geometry, the position of the pixel that is being rendered is transformed into light projection space where the depth is compared to the depth stored in the shadow map. If the depth of the pixel being rendered is greater than the depth stored in the depth texture, the rendered pixel is in shadow.

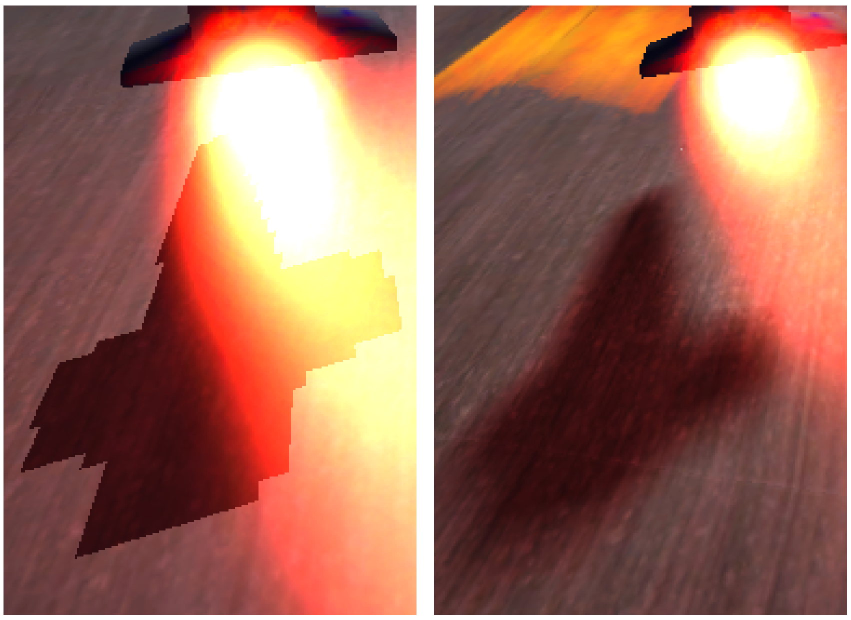
There are, however, several problems with this technique, some of which are related to the use of a depth texture. The first problem is called *surface acne*, and occurs due to imprecision in the depth texture; quantisation imprecision as well as depth imprecision. This problem can be significantly reduced by adding a bias to the stored depth, and with a finely tuned bias few artifacts can be noticed. However, as the bias increases the shadow line gets smaller and some accuracy is lost, which is why it is important to find a bias as small as possible. Another problem is aliasing which also occurs due to quantisation of the depth texture (see figure 3.4a). Aliasing can be reduced by increasing the resolution of the depth texture as well as using percentage-closer filtering (described in section 3.2.2) when sampling from the depth texture (Reeves et al., 1987).

Since the introduction of this algorithm by Williams (1978), several algorithms based on this technique have been presented. Some of these algorithms produce a better visual result and try to solve mainly the aliasing problem present in the original shadow map algorithm.

Perspective shadow maps

When several pixels in camera projection space cover the same pixel in the shadow map, aliasing artifacts occur (see figure 3.4a). Such artifacts can be reduced by increasing the resolution of the shadow map, but doing so will require more memory and makes the rendering of the shadow map slower due to the increased number of pixels that need to be processed by the graphics hardware. *Perspective shadow maps* (PSM, presented by (Stamminger and Drettakis, 2002)) is a technique that adapts the shadow map's projection to better fit the visible scene in order to attempt to make each pixel in the shadow map cover as little space as possible on the screen. This complicates the shadow map biasing problem because of the double projection, and Stamminger and Drettakis (2002) do not present a way of solving this. The algorithm is best suited for light sources that illuminate possibly large areas; such light sources will in most cases be directional lights, of which the most commonly used example is the Sun.

There are other, related techniques that attempt to reduce the aliasing problem by using several shadow maps of lower resolution as well as providing a



(a) Aliasing artifacts, visible as blocky edges on the shadow, caused by shadow mapping.

(b) Little to no aliasing visible when using variance shadow maps.

Figure 3.4: Comparison of standard shadow mapping and variance shadow mapping at the same resolution of the shadow maps in *SLERP 3D*.

better fit to the visible area compared to the standard shadow mapping algorithm. One such technique is *cascaded shadow maps*, presented by Dimitrov (2007). It uses several, cascaded, shadow maps, where shadow maps closer to the camera are set up to cover a smaller area than shadow maps further away from the camera. Doing this will make sure that most precision in the shadow maps is used for rendering of objects close to the viewer. This technique implies no further biasing problems, but the cost of rendering the shadow maps increases slightly since more shadow maps are needed.

Percentage-closer filtering

Special filtering techniques are necessary when gathering samples from the shadow map in order to reduce aliasing and creating a slightly blurry shadow. The blurry edge of such shadows resembles the penumbra of soft shadows, with the exception that the blurry edge is of constant width in most of the algorithms, which is not physically correct. Using normal bilinear filtering when sampling from shadow maps is not possible since they are depth textures, and averaging depth does not produce correct results. *Percentage-closer filtering* is a technique that allows correct filtering of shadow maps in order to reduce the aliasing problem (Reeves et al., 1987). This filtering technique averages the occlusion at the receiver by performing the shadow map test on many pixels and then averaging the result.

Variance shadow maps

Modern graphics cards have built-in functionality for performing filtering on textures. Using *variance shadow maps* (VSM), introduced by Donnelly and Lauritzen (2006), enables this hardware to be used for correctly filtering shadow maps to produce softened shadows (see figure 3.4b). When sampling an area in the variance shadow map, the first two moments of the depth distribution are retrieved, as can be seen in equation 3.1.

$$\begin{array}{l} M_1 = E(x) = \int_{-\infty}^{\infty} xp(x)dx \\ M_2 = E(x^2) = \int_{-\infty}^{\infty} x^2p(x)dx \end{array} \tag{3.1}$$

These two moments are subsequently used for calculating the mean and the variance of the depth distribution in the filtered region of the variance shadow map (see equation 3.2).

$$\begin{array}{l} \mu = E(x) = M_1 \\ \sigma^2 = E(x^2) = M_2 \end{array} \tag{3.2}$$

In order to resolve the occlusion factor, Chebyshev's inequality (equation 3.3) is used for determining the average occlusion at the rendered pixel. This also means that the shadow map can, aside from being filtered with graphics hardware, be pre-blurred or processed in other ways before being used. Furthermore, Donnelly and Lauritzen (2006)'s algorithm has been extended by Dong and Yang (2010) to be able to render soft shadows.

$$P(x \geq t) \leq p_{max}(t) \equiv \frac{\sigma^2}{\sigma^2 + (t-\mu)^2} \tag{3.3}$$

VSM produces light bleeding artifacts in certain situations where the depth complexity is high. This means that penumbrae are visible in areae which should be fully shadowed. In order to reduce these artifacts, a technique can be used that cuts off the darkest penumbrae and interpolates the rest of the penumbra between 0 and 1 (Lauritzen, 2007). Doing this will darken the rest of the penumbra a bit and only reduce the light bleeding, but the overall visual quality is improved.

Percentage-closer soft shadows

The soft shadows produced by the previously described algorithms have a fixed penumbra width. This is not very realistic, however, since the penumbra gets wider the farther away the receiver is from the occluder. *Percentage-closer soft shadows* (PCSS) present a solution by changing the width of the PCF kernel used for sampling the depth buffer (Fernando, 2005). In order to find the depth of the occluder, several samples are gathered from the shadow map. The depths of the pixels that are closer to the light than the rendered pixel are then averaged and used as the depth of the occluder. The kernel size of the percentage-closer filter is then approximated with the following function:

$$w_{Penumbra} = (d_{Receiver} - d_{Occluder}) * \frac{w_{Light}}{d_{Occluder}} \tag{3.4}$$

The shadow at the pixel is determined by sampling the shadow map using a PCF filtering technique.

The main difference of the PCSS algorithm compared to other shadow map algorithms is that it approximates the filter kernel width. This means that PCSS can be used in combination with other types of shadow map filtering techniques, such as CSM, VSM, SAVSM (*Summed-Area Variance Shadow Maps*, presented in Lauritzen (2007)) or ESM (*Exponential Shadow Maps*, presented in Annen et al. (2008)). The PCSS algorithm can also be combined with a technique for increasing the quality of the shadow map closer to the camera position such as PSM or cascaded shadow maps in order to further increase the visual quality of the shadows.

3.2.3 Results

We implemented variance shadow maps since it is a technique which is fast enough for our game, yet produces excellent results. Pre-blurring of the variance shadow map was attempted, but the resulting performance was significantly lower than if we instead performed the averaging when sampling the variance shadow map; however, the results were slightly better. Averaging in this manner gives rise to a slight penumbra of constant width. The main purpose of this gradient (visible in figure 3.4b) is to reduce the aliasing artifacts, and thus the penumbra is an additional advantage.

Since we have chosen to use a shadow map-based technique we will suffer from aliasing artifacts due to the limited resolution of the shadow map. However, due to the nature of the variance shadow map algorithm, these artifacts are reduced and instead traded for another visual artifact known as light bleeding (Lauritzen, 2007), but that artifact is only visible under certain conditions. One trade-off that we have chosen to make, in order to have acceptable performance, is to limit the number of light sources that can cast shadows. When rendering the scene, only the light sources that are closest to the viewer are used as shadow-casting lights. Generating a shadow map is rather costly, so we have provided an option for the user to choose the number of shadow maps which should be used each frame.

Soft shadows is an effect that generally increases the realism of the scene. However, the fact that basically the only occluder in the scene is the ship, and that the ship travels at a relatively constant distance from the shadow receiver, makes soft shadows a bit redundant.

3.3 Particle Systems

A particle system consists of several independent objects called *particles*. Each particle has a set of properties: position, velocity, colour, etc., as well as system-specific properties, for instance gravitation. A particle system is mainly used for simulating objects that are not solid geometry. Water, sparks, explosions, fire and beams are some effects that quite easily can be represented using this method. The basic functionality of a particle system should, in practically all cases, include: spawning, updating and removal of particles (Reeves, 1998).

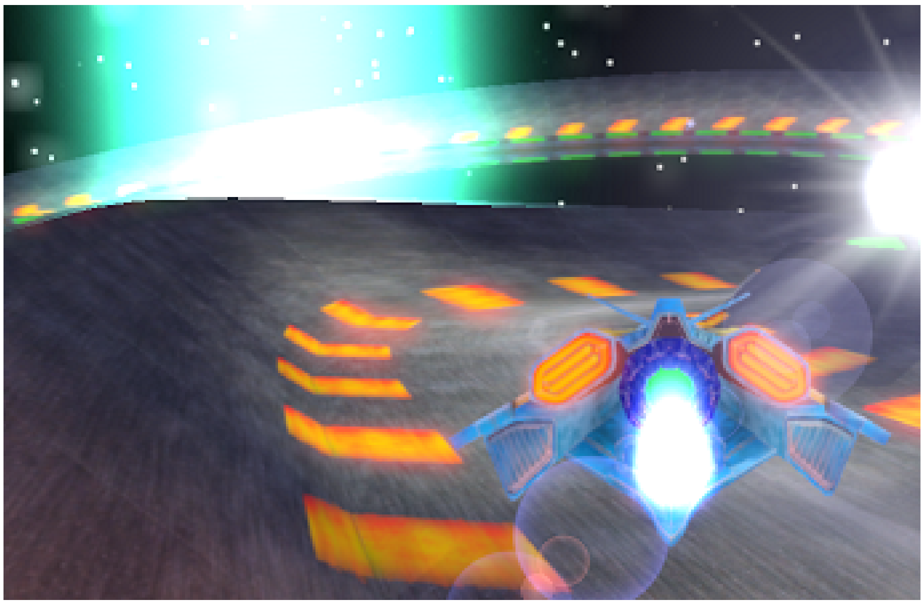


Figure 3.5: The particle system in use in *SLERP 3D*.

3.3.1 Billboards

To simulate objects that are normally too complex to be rendered using geometry or simply does not gain any visual quality by being rendered with geometry, such as grass or clouds, billboards can be used. A billboarded object is a flat and textured polygon that is always facing the viewer in order to create the illusion of a 3D object.

There are several different ways of achieving a billboarding effect. Screen-aligned billboards are implemented so that the polygon is rotated according to the camera's up vector. This is a simple technique that is both cheap to use and provides good results. A problem is that the billboards will be rotated if the camera rolls. However, since particles are rotationally invariant they are not affected by that problem.

World-oriented billboards use the world's up vector instead of the camera's, making it possible to roll the camera. Axial billboards are billboards that are rotated around their own axes to get the desired illusion. The world's up vector and the viewport's direction vector are used to rotate the billboard to the right viewing angle (Akenine-Möller et al., 2008).

It is common to represent the particles in a particle system with billboards since particles and billboards share the property that they should be facing the screen at all times. Billboards are also fast enough to enable the particle system to contain the large quantity of particles necessary to be able to display the wide range of effects that can be visible on the screen.

3.3.2 Result

In *SLERP 3D*, we have chosen to implement a fairly simple CPU-based particle system that uses screen-aligned billboards, since this is the easiest and one of the most efficient ways of implementing the effects we wanted to achieve.

The particle system is used to create sparks on ship collision, show the power beam behind all ships, display background effects and create explosions. We have also decided to use the particle system for visual feedback when the player respawns or uses certain power-ups. The end result improves the visual quality of the game and adds to the sense of movement and speed in the game.

3.4 Global illumination

Global illumination is when all significant light bounces and refractions is taken into consideration when computing the lighting for the scene. When global illumination is used, there are a few important effects that can be seen in the final image: indirect illumination, ambient occlusion, refraction, colour bleeding, soft shadows and caustics.

3.4.1 Real time implementation

The global illumination problem is inherently hard to solve due to the integral over the hemisphere, which can be seen in the rendering equation (equation 1.1). There are currently no algorithm that can achieve full global illumination in a real-time game. A very fast recent algorithm is *image-space photon mapping* (McGuire and Luebke, 2009) which takes advantage of the GPU in order to render images in interactive frame rates and is based on the photon mapping technique (Jensen, 2001). However, this algorithm is still too slow for being used in a fast racing game. Since there is no real-time algorithm for global illumination, focus must be shifted to simulate one or more of the most important visible effects of global illumination. What effects are most important depends on the type of application, the layout of the scene and the desired type of visual effects.

3.4.2 Ambient occlusion

Ambient occlusion, also known as *contact shadows*, occurs when an object close to a surface occludes the incoming ambient light and thus darkens an area at the surface. The effects of ambient occlusion can be seen in figure 3.6 and in figure 3.7.

Ambient occlusion volumes

Ambient occlusion volumes is an algorithm that produces results that are very close to the ground truth in real time (McGuire, 2009). The algorithm extrudes volumes from all polygons on the screen and for each pixel which are contained in these volumes calculates the ambient occlusion based on the distance of the occluding polygon as well as how large part of the pixel's hemisphere is covered by the polygon. The visual results of this algorithm are of high quality and considering the speed of the algorithm it is certainly plausible for real time use as well. Under some conditions in the tests performed by McGuire (2009), the ambient occlusion volumes were faster than Crytek's screen-space ambient occlusion algorithm (Mittring, 2007) depending on the settings used.

Screen space ambient occlusion

A technique which has been used for ambient occlusion in games is the screen space ambient occlusion (Mittring, 2007). This algorithm operates on the stored depth values of the scene in order to calculate occlusion from nearby pixels. This data is then used to lower the ambient lighting contribution to the rendered pixel. The algorithm can also be slightly modified to also take into consideration the normals at each surface point in order to get a slightly different result. Since the algorithm operates in screen space, it has a constant cost each frame and the performance is only dependent on screen resolution. In order to improve performance it is possible to make this algorithm work on a down-sampled version of the depth buffer, and since ambient occlusion is primarily a low frequency phenomenon the quality of the ambient occlusion will still be sufficient. However, as noted by McGuire (2009), this technique is not very accurate and contains a few visual artifacts, but the final results which is achieved when using this algorithm is of sufficient quality to be used in modern computer games.

3.4.3 Indirect illumination and colour bleeding

When photons hit a surface, they are either reflected or transmitted[2], depending on the characteristics of the material. Reflected or transmitted photons will then illuminate other surfaces in the scene. This phenomenon is called indirect illumination. Another phenomenon called *colour bleeding* occurs when outgoing photons from a surface are coloured by the material and spread this colour to the surfaces they reach. Indirect illumination is a very important part of the global illumination as it is responsible for the ambient lighting in the scene.

These types of effects are very slow to render fully correctly in real time, and thus one needs either to make approximations which can give slightly incorrect results or choose to pre-compute all the lighting information in the scene.

Cascaded light propagation volumes

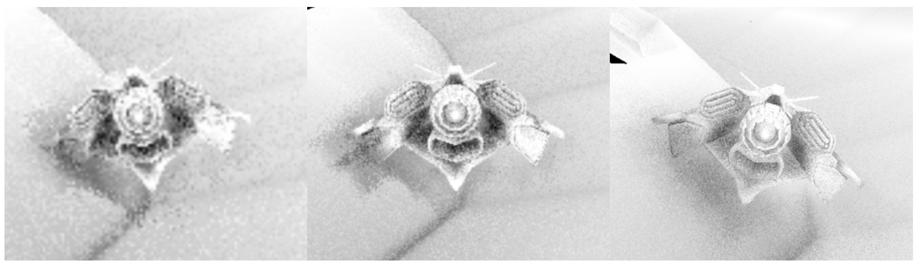
Cascaded light propagation volumes (Kaplanyan and Dachsbacher, 2010) can simulate indirect illumination as well as colour bleeding from the first bounce of the light, which is the most important bounce, as on each bounce a large fraction of the photons are absorbed. The algorithm works by creating virtual point light sources in the scene by rendering reflective shadow maps from each light. These virtual point lights are then injected into a light propagation volume, where they are stored using spherical harmonics. The resulting light in the light propagation volume is subsequently propagated in several iterations where the light can be occluded as well as bounce on geometry, that is injected into a separate volume. The final light propagation volume represents the indirect illumination in the scene and can be used for rendering the ambient lighting in the scene. This algorithm is designed for real-time use and runs very fast on modern hardware, but because of the limited resolution of the light propagation volumes, the results are not completely accurate. The authors present a possible improvement by using cascades of the light propagation volumes where volumes of higher resolution are used closer to the viewer.

[2]There are also other phenomena such as subsurface scattering and absorption, which occur when the photon is transmitted.

Precomputed radiance transfer

Some algorithms, such as the precomputed radiance transfer (Sloan et al., 2002), pre-compute the lighting information prior to rendering in order to display good results in real time. The precomputed radiance transfer algorithm stores a large number of spherical harmonics for each object, which then are used to calculate lighting. These spherical harmonics represent the occlusion as well as indirect illumination across the hemisphere at the point where the spherical harmonic is sampled. Rendering by using these spherical harmonics is quite fast, and the technique can correctly handle dynamic light sources since the entire hemisphere for each point is stored. However, this also means that animated objects are not supported very well since these spherical harmonics would change for each point in the animation. This algorithm also requires the storage of extra data; depending on how many bands are used for the spherical harmonics, it can be as much as 36 to 100 bytes per vertex. The number of bands affects how detailedly the spherical harmonic approximation can reconstruct the original data.

3.4.4 Results



(a) Low resolution **(b)** Mid resolution **(c)** High resolution

Figure 3.6: Comparison of different resolutions when calculating SSAO in *SLERP 3D* in similar conditions. The result of these images are subsequently blurred which lowers the difference slightly.

The only algorithm of the above mentioned that we have implemented is the screen space ambient occlusion algorithm. We found that there are a large number of slightly different versions of this algorithm with different sampling patterns and different depth fall-off functions that alter the look of the generated occlusion. What we finally used was a slight modification of an algorithm that also operates on the normal of each point. This algorithm runs rather fast, and there is in fact little performance difference with the algorithm deactivated on the computers which we tested it on. This is also probably due to the fact that we perform the SSAO using a sixteenth of the screen resolution. Using such a low resolution does not affect the visual quality very much, however (see figure 3.6). It is important to note that we found out that SSAO gives very little gain in visual quality on the kind of level that we have at the moment. Since the geometry is quite flat, there are few situations where the effect is clearly visible. A comparison can be seen in figure 3.7.

We had an idea of another ambient occlusion technique which utilised the

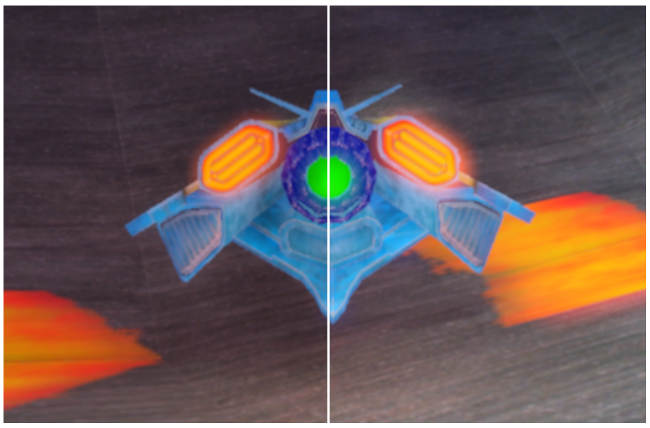


Figure 3.7: Comparison of with and without SSAO in *SLERP 3D*. Left: SSAO is on. Right: SSAO is off.

fact that our lighting system can handle a large number of concurrent light sources. In this technique, we place a negative light source in the centre of each object. The light source will then darken nearby objects, but since the normals of the object is directed outwards of the object, it will itself not receive any darkening on surfaces pointing away from the centre. Surfaces pointing slightly towards the centre of the object will, however, receive a slight amount of darkening, which is to be expected of ambient occlusion as well. However, this technique does not work very well for simulating ambient occlusion of large-scale hollow geometry, such as a room, for obvious reasons. The technique is, however, not at all physically based, but provides plausible results. Also, it executes very fast and required only a few lines of code to implement in order to get our ship to cast occlusion on nearby geometry.

We have not, implemented any kind of indirect illumination. Partly, because these algorithms are rather complex and would require a lot of time to implement with little gain in visual quality and we felt that time could be better spent on other areas of the game.

3.5 Post-processing effects

3.5.1 Motion blur

Motion blur is a physical effect that occurs due to the fact that cameras has a certain shutter speed. During the time the shutter is open, fast moving objects will be captured at several different locations in the picture, and the result will be blurred. Motion blur is an important visual effect in order to increase the sense of speed, as can be seen in figure 3.8, and to make the game feel smoother to play as well as help the player perceive fast moving objects better (Wloka and Zeleznik, 1996). Akenine-Möller et al. (2008) notes that a game running in 30 frames per second with proper motion blur often looks smoother than a game running in 60 frames per second without motion blur.

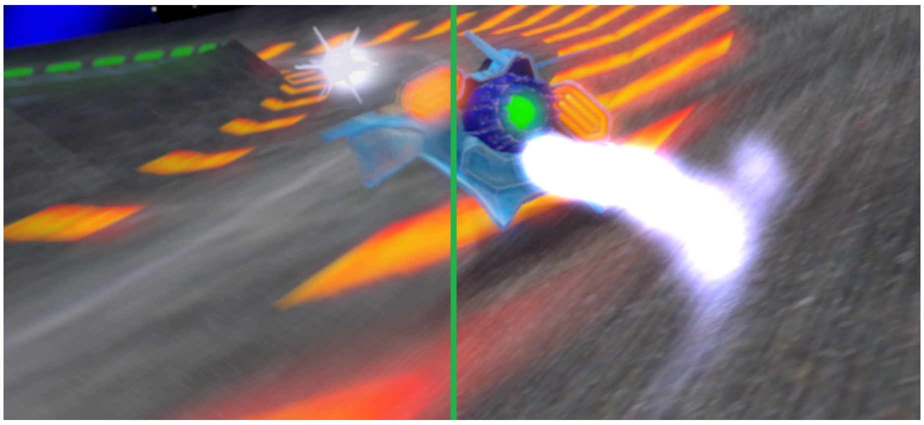


Figure 3.8: Comparison of motion blur on (left) and off (right) in *SLERP 3D*.

Current algorithms

One option that creates very realistic results is to use the accumulation buffer to accumulate several renderings over a frame and move the objects slightly between each render (Haeberli and Akeley, 1990). This will, however, be too slow for real time applications and needs a large number of renderings in order to converge to the correct result.

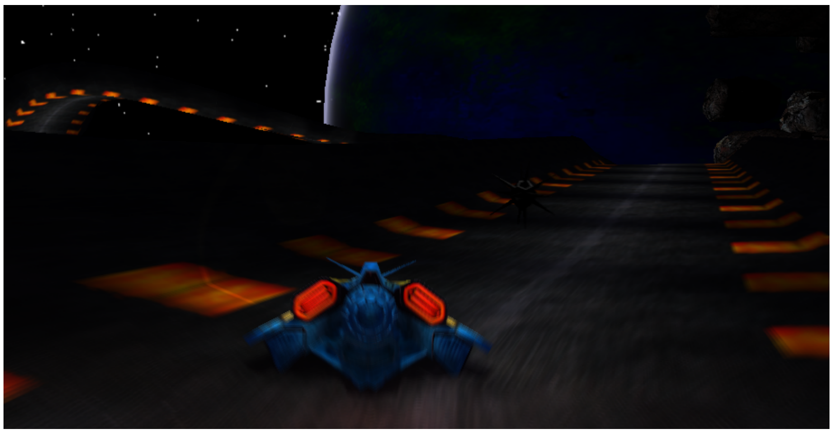
Performing motion blur as a post-process is preferable, as most post-processing effects operate in constant and known time, since the input[3] is of the same size every frame. Rosado (2007) proposes an algorithm which only uses the current depth buffer and the view projection matrix of the previous frame in order to calculate the motion blur. The depth buffer is used to calculate the world space position which is then transformed by the view projection matrix of the previous frame in order to find where in projection space the position would be in the previous frame. Using the current and the previous projection-space positions, a velocity is calculated and subsequently used to determine the kernel size of the blur that is used.

Another technique, which is also calculated in screen space but results in a more correct result, uses a *velocity buffer* to store the velocity of each pixel on the screen (Green, 2003). Since all velocities are known per pixel no incorrect blurring will occur, unlike the algorithm proposed by Rosado (2007), but will cause more overhead and is more complicated to integrate in an existing rendering pipeline due to the larger changes involved in rendering the velocity buffer.

3.5.2 Tone mapping

The luminance (amount of light passing through a given solid angle, measured in candela per square metre) that can be perceived by the human visual system ranges between $10^{-6}\ cd/m^2$ for below starlight and $10^{6}\ cd/m^2$ for sunlit snow; a range of approximately ten orders of magnitude (Reinhard et al., 2002; Durand and Dorsey, 2000). In a single real scene, the range from the darkest shadows

[3]The input is based on the rendered image and thus only its content changes


(a) Original image

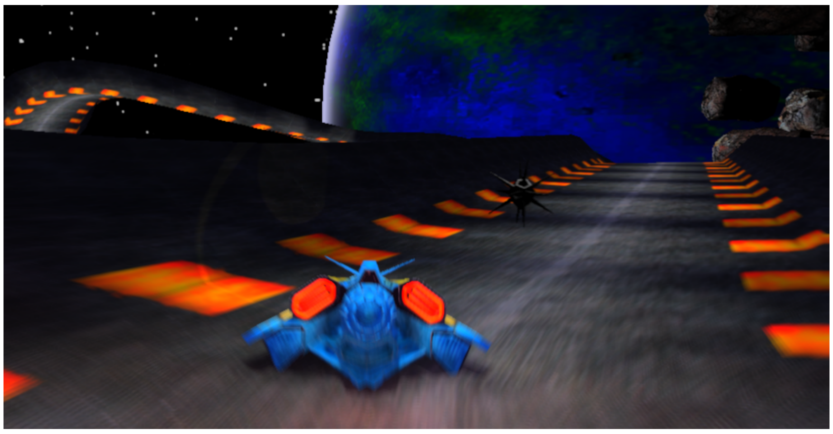

(b) Tone-mapped image

Figure 3.9: The tone mapping operator is applied to the original HDR image (a) to produce the tone-mapped image (b), where colours have been converted to the output range $[0, 1]$.

to the brightest highlights may span up to four orders of magnitude. The ratio between the highest and the lowest luminance in a scene is referred to as the *dynamic range* of the luminance (Reinhard et al., 2002).

A computer display is limited in its output capabilities and is unable to reproduce the high dynamic range (HDR) lighting of a real scene. While lighting calculations may be performed in high dynamic range on today's graphics hardware with high-precision render targets and no upper limits on luminance, colours must therefore be converted to the lower dynamic range of the output device. This process of compressing dynamic range is known as *tone mapping* (Calver, 2004; Akenine-Möller et al., 2008; Reinhard et al., 2002; Durand and Dorsey, 2000). Most applications of tone mapping aim to achieve good *tone reproduction*; that is, the mapping of scene luminances to display luminances in such a way as to produce a (subjectively) pleasing image (Reinhard et al., 2002). An evaluation of a number of tone mapping operators can be found in Čadík et al. (2008).

Visual adaption

The high dynamic range of real scenes is handled by the visual system through a process known as *visual adaption*. Visual adaption is itself constituted of three separate processes: *light adaption*, *dark adaption* and *chromatic adaption*.

Light adaption is responsible for recovering visual sensitivity when moving from a dark to a bright setting such as when exiting a tunnel, or when moving from a bright to a slightly less bright setting such as when coming indoors from sunlight. This is a rapid process, normally taking a few seconds. Daylight, or *photopic*, vision is provided by the *cone* photoreceptors in the retina (the other type being the *rods*), which adapt quickly to changes in light intensity and are very sensitive to colour.

Dark adaption is the change of sensitivity that happens after a dramatic decrease in light intensity and is considerably slower than light adaption, taking up to tens of minutes. The photoreceptors responsible for dark-adapted (*scotopic*) vision are the rods, which are considerably more sensitive to light than the cones but provide less precise chromatic vision and lower visual acuity.

The perception of objects as having constant colour when observed in differently coloured light is due to chromatic adaption. Chromatic adaption depends on the colour of the illuminant rather than the colour of the object and causes the visual system to adjust the perceived hue of the stimulus based on the composition of the illuminating light, approximated from the visual information of the surroundings. The colour of the illuminant is then said to be *discounted*.

The mechanics of visual adaption is described in fuller detail by Durand and Dorsey (2000), who also provide references to in-depth texts on the subject.

Implementation

The easiest way of outputting HDR images is to simply clamp the colour values to the displayable range (Akenine-Möller et al. (2008, p. 476) note that this simple method can work surprisingly well in applications with predictable lighting and camera conditions). However, it is often desirable to use a method that adapts to the overall illumination in the scene. Although simple adaption can be achieved by setting the luminance of the brightest pixels in the scene to the highest displayable value and linearly scaling the luminance of the rest of the scene accordingly (an operator known as *maximum to white*), this can result in the overall scene becoming very dark in the presence of a single extremely bright pixel (Akenine-Möller et al., 2008). A better approach would be to use a model of the human visual system.

Reinhard et al. (2002) present a tone mapping algorithm based on a method long used in conventional photography called the Zone System, developed by the photographer Ansel Adams for use in his black-and-white photographs. The algorithm first calculates the *key* of the scene, a value indicating whether the scene is subjectively light, dark or normal (a light scene would be high-key while a dark scene would be low-key), at full dynamic range and sets the white point to the maximum luminance in the scene. It then uses the key and the white point to scale the results to the displayable range. Lastly, a technique known as *dodging-and-burning* is used to lighten or darken regions in the final image. However, following the presentation given in Calver (2004), the method described in this text omits the dodging-and-burning feature for the sake of brevity.

The key of the scene is approximated by calculating the logarithmic average luminance $\bar{L}_w$:

$$\bar{L}_w = \exp\left(\frac{1}{N}\sum_{x,y}\ln(\delta + L_w(x,y))\right), \tag{3.5}$$

where N is the total number of pixels, δ is a small constant value to avoid the introduction of $\ln(0)$ in the case of black pixels, and $L_w(x,y)$ is the "world" luminance of the pixel at the 2D coordinates (x,y). The luminance function L_w derives from the equation for conversion from RGB to XYZ colour space as given in Dutré (2003, p. 63) and Akenine-Möller et al. (2008, p. 215-216) and can be written as:

$$L_w(x,y) = 0.212671R_{xy} + 0.715160G_{xy} + 0.072169B_{xy} \tag{3.6}$$

The constants in equation 3.6 are weights based on the sensitivity of the human eye to different wavelengths of light as well as the spectra of the three phosphors in modern CRT and HDTV displays (Akenine-Möller et al., 2008).

Next, the luminance of the current pixel is scaled depending on the key of the scene using the following function:

$$L(x,y) = \frac{a}{\bar{L}_w}L_w(x,y), \tag{3.7}$$

where $a \in [0,1]$ is the *middle-grey* value of the scene. The middle-grey is closely related to the key of the scene (and is in fact usually referred to as the *key value*), and is commonly assigned the value 0.18 for moderate illumination conditions (*i.e.* normal-key) (Reinhard et al., 2002; Krawczyk et al., 2005). Krawczyk et al. (2005) suggest using the following formula to dynamically interpolate the middle-grey value between a set of empirically determined key values for several illumination conditions:

$$a(\bar{L}_w) = 1.03 - \frac{2}{2 + \log_{10}(\bar{L}_w + 1)} \tag{3.8}$$

Lastly, the scaled luminance obtained from equation 3.7 is compressed to within displayable range:

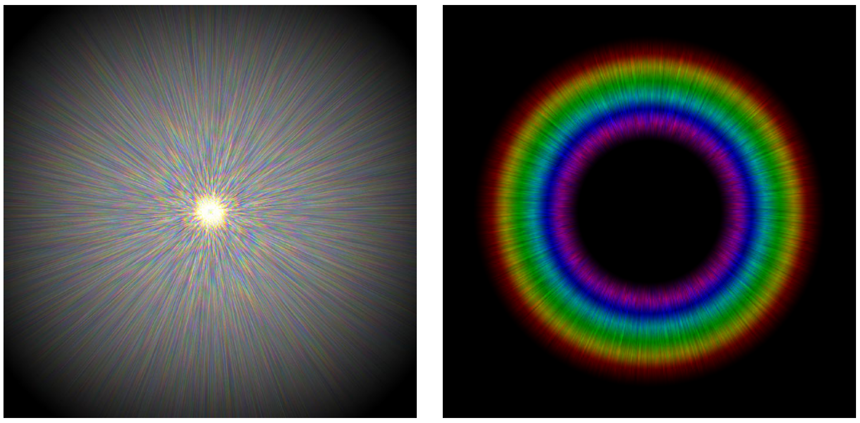
$$L_d(x,y) = \frac{L(x,y)}{1 + L(x,y)} \tag{3.9}$$

Using this method, $L_d(x,y)$ is guaranteed to fall within $[0,1]$. However, as noted by Reinhard et al. (2002), it is usually desirable to let high luminances burn out in a controllable fashion. This is achieved by extending equation 3.9 to the following form:

$$L_d(x,y) = \frac{L(x,y)\left(1 + \frac{L(x,y)}{L_{white}^2}\right)}{1 + L(x,y)}, \tag{3.10}$$

where L_{white} is the white point; that is, the lowest luminance value that will be mapped to pure white. L_{white} is set by default to the maximum luminance in the scene, causing a subtle contrast enhancement for low dynamic range scenes. However, it will need to be set lower to result in burn-out, and setting the white point to infinity will revert the function to equation 3.9.

An example of the tone mapping operator described above can be seen in figure 3.9.



(a) The ciliary corona (picture by van den Berg et al. (2005)) **(b)** The lenticular halo

Figure 3.10: In the eye, the flare effect consists of the ciliary corona (a) and the lenticular halo (b).

3.5.3 Glare effects

Glare is a collective term for a number of related visual effects caused by scattering and diffraction in the eye, as well as by microscopical defects and impurities in a camera lens. Computer displays are, however, unable to output the high luminances required to directly produce these effects. Instead, digitally adding glare to an image produces an illusion of increased luminance of very bright pixels by mimicking the way the human eye perceives bright light in various brightness settings and makes it possible to simulate a camera lens, adding to the perceived realism of the scene if done right (Akenine-Möller et al., 2008; Spencer et al., 1995). For the purposes of rendering, glare effects can be divided into the major components *lens flare* and *bloom*.

Lens flare

A lens flare (or simply *flare*) is composed of a *ciliary corona* and a *lenticular halo*. The ciliary corona appears as radial rays emanating from the centre of the light source, as illustrated in figure 3.10a, and is caused by scattering due to local variations in the density and thus the refractive index of the lens. As the solid angle subtended by the light source decreases, the individual rays of the ciliary corona become brighter and more easily discernible, while as the subtended solid angle increases the ciliary corona appears to blur due to superimposition of the rays. The lines are barely visible for light sources with a visual angle greater than 20 arcminutes (Spencer et al., 1995).

The lenticular halo, shown in figure 3.10b, takes the form of concentric coloured rings in the full visible spectrum surrounding the ciliary corona (although the radial streaks of the ciliary corona may extend beyond the lenticular halo). It is caused by diffraction in the crystalline lens of the eye due to radial fibres forming a circular optical grating surrounding the central clear part of the lens. The optically homogenous central portion is 3 mm in diameter, so any beam of

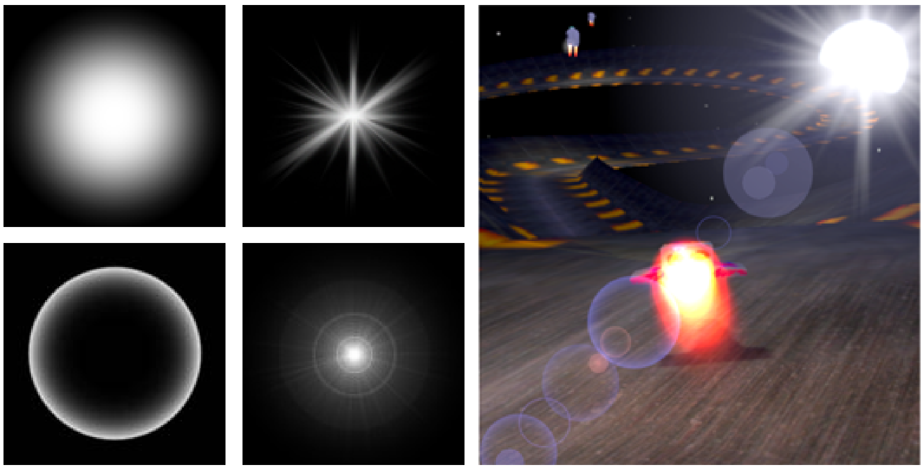


Figure 3.11: Left: some of the constituent lens flare textures used in SLERP 3D. Right: an ingame example of a lens flare from the Sun.

light with a diameter less than this can pass unhindered through the lens. This is the case in average daylight conditions, as the pupil is contracted to 2 mm across. Beams wider than 3 mm, however, pass through the circular grating and give rise to a lenticular halo. The lenticular halo always subtends the same solid angle at the eye, creating an illusion of more distant light sources having larger halos; however, the intensity of the lenticular halo decreases with distance (Spencer et al., 1995).

Further, secondary effects can also be caused by camera lenses as light is refracted or reflected internally by parts of the lens. These effects manifest as patterns such as rings or circles arranged in a row across the view through the image centre. The shape of the lens's aperture also affects which patterns are seen; a six-bladed aperture, for example, may produce hexagonal flare patterns (Akenine-Möller et al., 2008).

Implementation Lens flares can be algorithmically generated and added to an image by a shader, but it is significantly more efficient to instead use a set of textures for the different parts of the flare and add these where appropriate. Each texture is applied to a black square facing the viewer, and treated as an alpha map when blended into the scene. These squares are given colour, typically pure red, green or blue, when rendered and additively blended when they overlap to obtain other colours. To make the lens flare change with the light source, the squares are arranged on a line going from the light source through the middle of the view, and may change size and intensity as the distance to the light source varies correspondingly. This method can be quite convincing if applied skilfully (Akenine-Möller et al., 2008).

Bloom

The visual phenomenon of very bright areas "bleeding" light onto their surroundings and thereby reducing contrast (see figure 3.12) is referred to as bloom or *glow*. The bloom effect has become tightly associated with HDRI (High Dynamic Range Imaging) due to it routinely being used together with tone mapping (described in section 3.5.2) when rendering to approximate the look of HDR images (Sousa, 2004). Bloom may be used without HDR, however, and HDR may likewise be used without bloom.

In the eye, bloom is caused by the scattering of light by the lens, cornea and retina in roughly equal contribution. Scattered light from one source A is added to the light from another source B, increasing the perceived luminance of B and decreasing the contrast ratio since light is added equally to light and dark parts (Spencer et al., 1995). In a camera, the effect is due to charge sites in the camera's *charge-coupled device* (CCD), which captures images by converting photons to charge, becoming saturated and overflowing into surrounding sites (Akenine-Möller et al., 2008).

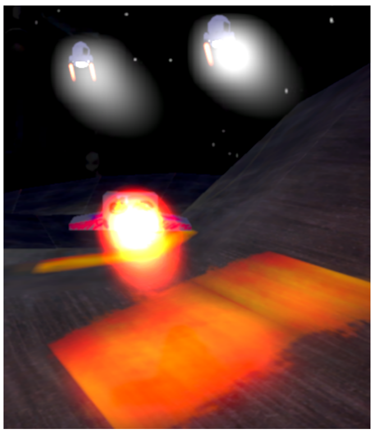


Figure 3.12: Extremely high bloom visible around the light sources and on the ground texture, with streaking caused by an after-image effect

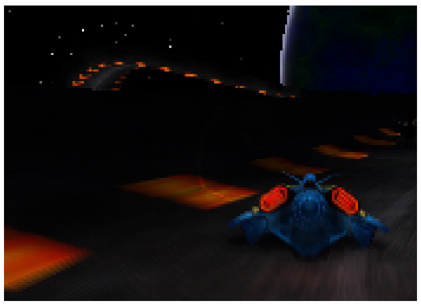
Implementation There are several ways to implement bloom. One common way is to first downsample the original image to a smaller size, *e.g.* a quarter of the original, and then perform a simple thresholding operation to keep only high-luminance pixels. The resulting texture is then blurred using a Gaussian blur filter to create the bloom texture with glow, which is then added back to the original image. It may be desirable to adapt the level of bloom (or glare effects in general) to the average luminance of the scene; one such algorithm for *adaptive glare* is presented by Sousa (2004) and is described in the following paragraphs.

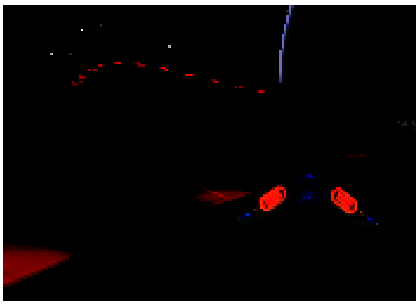
The adaptive glare[4] algorithm works by first downsampling the original image to a 1×1 texture, which is converted to a scalar value representing the average luminance of the scene. The conversion to luminance is done using equation 3.6. In order to minimise flickering due to fluctuation of sampled luminance from frame to frame, the computed luminance is linearly interpolated with that of the previous frame.

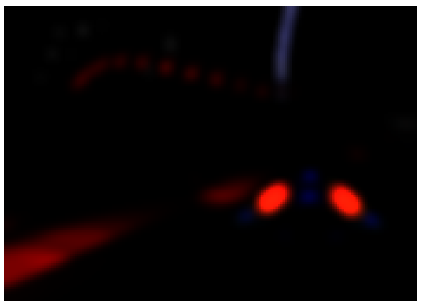
Secondly, the bloom texture is computed. The original scene image is downsampled to a low-resolution texture, from which the brightest pixels are extracted by subtracting a threshold value and clamping (*saturating*) the result to a $[0, 1]$ interval[5] (see figure 3.13b). This method of reducing all dim pixels to black and retaining brighter pixels is called *bright-pass filtering* (Akenine-Möller et al., 2008). By multiplying the result with the inverse luminance $1 - L$ (where L is the luminance value obtained in the previous step), the amount of bloom is decreased for very bright scenes. The bloom texture is then obtained by blurring the thresholded image using a simple separable Gaussian blur filter (see figure 3.13c), which is applied in two one-dimensional passes (one horizontal and one

[4] *Glare* here refers only to the bloom effect, so to avoid confusion this text uses the latter term.

[5] Sousa (2004) suggests downsampling to 1/64 of the original texture's size and using a threshold value of 0.4


(a) Downsampled image


(b) Thresholded image


(c) The blurred bloom texture with after-image

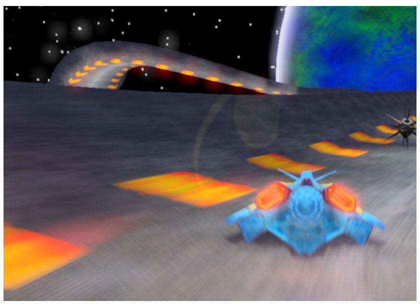

(d) Composited final image

Figure 3.13: The original image is first downsampled to (a), from which the thresholded image (b) is obtained by applying a bright-pass filter and adding glow on areas designated by the alpha values. The thresholded image is then blurred to create the bloom texture (c), to which an after-image effect has been applied by blending in the faded bloom texture from the previous frame. Lastly, the bloom texture is composited with the original image, yielding (d) (which has also been tone-mapped as shown in figure 3.9b and gamma-corrected as this is the last pass applied before the image is output to the screen).

vertical) for efficiency; the number of samples required are reduced from d^2 to $2d$, where d is the blur diameter, compared to performing one two-dimensional pass (O'Rorke and James, 2004).

Lastly, the final image with bloom is created by compositing the bloom texture with the original scene image. This can be done using simple additive blending (*i.e.* adding both textures together), but this may cause undesirable side effects if the bloom texture is not carefully generated. The approach suggested by Sousa (2004) is to instead make the contribution of the bloom texture proportional to the luminance of the original pixel.

An extension to the bloom effect suggested by O'Rorke and James (2004) is to use the alpha channel of certain textures to store a "glow texture". Essentially, the glow texture is used to designate areas of the main texture that should glow in the final image, and is taken into consideration when computing the bloom texture by adding the term $rgb \cdot \alpha$. Another possible extension is the *after-image* effect, simulating the lingering vision of an image caused by bright areas "burning" onto the retina. After-image is implemented by saving the texture from the previous frame and blending it with the current scene image using an appropriate ratio, thus letting it fade out. While this can be applied to the bloom texture to generate light streaks as the observer moves, O'Rorke and James (2004) caution that this may cause the bloom to bleed out of control if done carelessly, and suggest instead to use the original texture without bloom. The use of a glow texture and an after-image effect on the bloom texture is visible in figure 3.13.

3.5.4 Results

For motion blur, we have chosen to implement the algorithm presented by Rosado (2007) because of its speed and the fact that it provided a better fit to our current graphics pipeline than the algorithm described by Green (2003). The performance impact is quite low compared to the improvements of the results. However, we did not implement the masking described in the algorithm for objects moving at a relatively constant speed compared to the camera. An example of such an object in *SLERP 3D* would be the player ship. The ship does now receive some incorrect motion blurring (see figure 3.8; note that this is also one of the worst cases we have been able to produce) in some cases, but in most cases this is not visible, and that is one reason why we have not implemented the masking. Another reason is that it fits very badly in our pipeline, and in the case that a fix had been necessary, we would probably have chosen to implement the Green (2003) algorithm instead.

In *SLERP 3D*, we perform all lighting and shading calculations in high dynamic range until the very end of the rendering pipeline. Before outputting the image to the screen, tone mapping and bloom are applied as part of the same effect technique. Tone mapping was implemented using the algorithm by Reinhard et al. (2002), with the extension of the method presented by Krawczyk et al. (2005) for dynamically computing the middle-grey value for a scene. Visual adaption was approximated using a simple model of light/dark adaption that interpolates the average luminance for one frame with the value retained from the previous frame, causing the luminance to vary smoothly and make a transition over the course of a few seconds. This was primarily done to avoid flickering between frames due to rapid fluctuations in scene luminance, and it was decided that a correct implementation of dark adaption, with an adaption time of up to thirty minutes before optimum scotopic vision is obtained, did not make sense in a fast-paced racing game.

The luminance is averaged according to equation 3.5 from a fixed number of samples performed over a downsampled version of the original scene texture and written to a 1×1 render target as described by Sousa (2004). The sampled luminance is then used to separately perform tone mapping on the original image and compute the bloom texture from the downsampled image. For the bloom effect, which is otherwise implemented using the adaptive glare algorithm by Sousa (2004), we have chosen to add streaking with the after-image effect using the bloom texture saved from the previous frame, as this gives an increased sense of motion blurring and was considered visually pleasing. Furthermore, a glow texture stored in the alpha channel as suggested by O'Rorke and James (2004) is used to allow the artist to add a glow effect to specific areas in the texture, the use of the alpha channel for transparency having been rendered ineffective due to our use of deferred lighting. However, since the alpha values are by default set to 1, we use $1 - \alpha$ when adding glow, meaning 0 (black) in the glow texture designates full glow.

With the tone-mapped original image and the bloom texture in hand, those two textures are composited together in the manner described by Sousa (2004) to form the output image.

Chapter 4

Game engine

4.1 Collision detection

Collision detection is very important, not only for racing games, but also most other types of dynamic games where the player interacts with the environment. The uses of collision detection are very game dependent; some games can, for example, provide a fully dynamic world where all objects can collide and interact, while other games use collision detection in order to keep the player from entering certain areas. A racing game typically uses collision detection to provide believable vehicle physics and to keep the player from going through the ground. Collision detection can also be used to add another game element where players can collide with each other on purpose in order to get an advantage in the game.

There are several different types of collision detection algorithms, where the differences can be *e.g.* speed and correctness. The difference in speed is also affected by the type of collisions that are expected to occur in the game, since some algorithms handle certain scenarios better (Akenine-Möller et al., 2008).

4.1.1 Collision detection using rays

One very efficient method of performing collision detection is to approximate a moving object using a set of *rays*. The rays are placed at fixed positions on the object's surface and tested for intersection along their respective paths (Akenine-Möller et al., 2008). A ray R is defined by its *point of origin* $\mathbf{o}$ and *direction vector* $\mathbf{d}$, yielding the following expression for computing any point lying on the ray half-line:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}, \tag{4.1}$$

where t is the *signed distance* from the ray's point of origin to the point $\mathbf{r}(t)$ along the direction defined by the direction vector, which is then assumed to be normalised (Havran, 2000; Akenine-Möller et al., 2008) (see figure 4.1). An intersection query should return the closest intersection, *i.e.* the lowest value of t for any intersecting point on the ray half-line, and the object which the ray intersects.

This method of performing collision detection and determination with rays is commonly referred to as *ray casting* or *ray tracing*, although the latter term

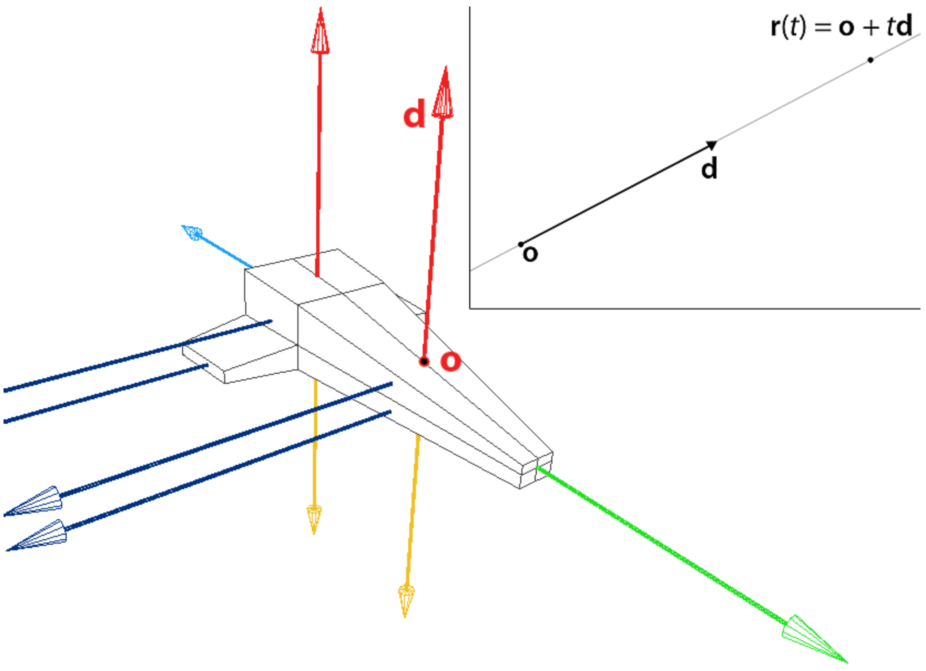


Figure 4.1: A set of rays on a ship, used for collision detection in different directions. **o** is the point of origin, and **d** is the direction vector. The inlay illustrates equation 4.1: $\mathbf{r}(t)$ is a point lying t units of distance away from the origin along the normalised direction vector.

correctly applies to a slightly different technique used to render photo-realistic scenes in which an intersection might spawn new rays as the original ray is reflected off a surface. When used in the context of collision detection, however, it is not enough to know whether the ray intersects somewhere down the ray from the point of origin; if a collision has already occurred, it is also necessary to determine where the extended ray intersects the object, potentially yielding negative values of t as the solution to equation 4.1. For this reason, a slightly modified version of the ray casting algorithm (which only handles $t \geq 0$) has to be used, where the ray's point of origin is first moved back along the extended half-line defined by the ray to outside the scene's bounding box, from where intersection testing is performed (Akenine-Möller et al., 2008).

4.1.2 Intersection testing

Ray-triangle

As triangles are the most common rendering primitive, there are many tests for ray-triangle intersection available. The algorithm used in *SLERP 3D* and presented here is from Möller and Trumbore (1997) (and is also presented in Akenine-Möller et al. (2008, p. 746-750)) and does not presume that normals are precomputed, as storing precomputed triangle plane normals can be quite costly for large triangle meshes in terms of memory. The presentation here follows the one given in Akenine-Möller et al. (2008).

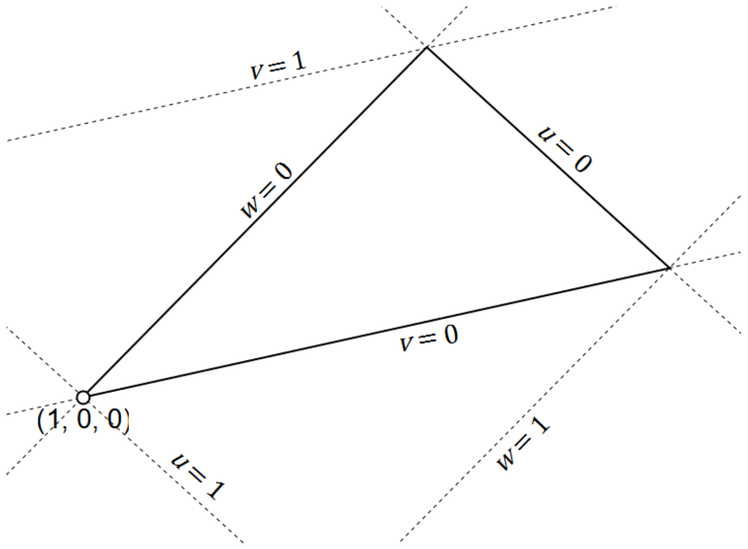


Figure 4.2: Barycentric coordinates for a triangle.

A triangle is defined by three vertices $\mathbf{p}_1$, $\mathbf{p}_2$ and $\mathbf{p}_3$ as $\triangle\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$. A point $\mathbf{f}(u, v)$ on the triangle can be computed using the formula

$$\mathbf{f}(u, v) = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2, \tag{4.2}$$

given two *barycentric coordinates* (u, v) for the triangle. Barycentric coordinates can be seen as weights showing how much each of the triangle's vertices contribute to a given point on the triangle; at each vertex one of the three values u, v and $w = (1-u-v)$ is 1 and the other two 0, and along the edges one value is always 0, as shown in figure 4.2. It follows that a point is inside the triangle if and only if $u, v, w \in (0, 1)$ (Dutré, 2003).

Using the definition of a ray $\mathbf{r}(t)$ given in equation 4.1, the intersection between the ray and the triangle, if it exists, can be computed by equating their equations:

$$\mathbf{r}(t) = \mathbf{f}(u, v) \Leftrightarrow \mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2 \tag{4.3}$$

Rearranging the terms gives a linear system of equations:

$$\begin{bmatrix} -\mathbf{d} & \mathbf{p}_1 - \mathbf{p}_0 & \mathbf{p}_2 - \mathbf{p}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{o} - \mathbf{p}_0 \tag{4.4}$$

Solving this yields the distance t from the ray origin to the intersection point as well as the barycentric coordinates (u, v). Using Cramer's rule, the solution to equation 4.4 is

$$\begin{aligned} \begin{bmatrix} t \\ u \\ v \end{bmatrix} &= \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{bmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{bmatrix} \\ &= \frac{1}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \begin{bmatrix} (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{e}_2 \\ (\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{s} \\ (\mathbf{s} \times \mathbf{e}_1) \cdot \mathbf{d} \end{bmatrix}, \end{aligned} \tag{4.5}$$

where $\mathbf{e}_1 = \mathbf{p}_1 - \mathbf{p}_0$, $\mathbf{e}_2 = \mathbf{p}_2 - \mathbf{p}_0$, and $\mathbf{s} = \mathbf{o} - \mathbf{p}_0$. The factors $\mathbf{q} = \mathbf{d} \times \mathbf{e}_2$ and $\mathbf{r} = \mathbf{s} \times \mathbf{e}_1$ can be computed first, saving two unnecessary (and costly) cross products. Equation 4.5 then becomes

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\mathbf{q} \cdot \mathbf{e}_1} \begin{bmatrix} \mathbf{r} \cdot \mathbf{e}_2 \\ \mathbf{q} \cdot \mathbf{s} \\ \mathbf{r} \cdot \mathbf{d} \end{bmatrix} \tag{4.6}$$

If storage is not an issue, there are faster algorithms available. Havel and Herout (2010) present a very fast (when implemented using Intel's SSE4 instruction set, which has an instruction for calculating dot products) algorithm for ray-triangle intersection. This algorithm requires the triangle's normal plane as well as two planes perpendicular to the triangle, used for computation of the barycentric coordinates, to be stored. It should be noted, however, that storing the triangle vertices is not necessary in this case, as the triangle is fully described by these three planes. The total memory cost of storing a triangle in this way is thus twice that of storing the three vertices (storing a plane as a point and a normal vector).

4.1.3 Space-partitioning data structures

As performing intersection testing on every primitive for each ray on every update is quite expensive, some sort of data structure is needed to store the triangles in such a way as to decrease the number of tests needed for each ray. There is a range of options available for this purpose, the most prominent being *quadtrees* for two dimensions, their three-dimensional counterpart *octrees* and *k-d trees*, which can be applied to any number of dimensions.

These data structures are all examples of *space-partitioning trees*, and work by decomposing a scene into progressively smaller parts according to a certain algorithm. Each node in such a tree then represents a geometric subdivision of the cell represented by the parent node (the root node, then, corresponds to the entire original scene) and contains, in addition to other data needed by the particular structure, a list of references to the primitives fully or partially contained in the node's cell. This definition is given in fuller detail in the discussion on spatial subdivision structures in Havran (2000, p. 10-11).

Quadtrees and octrees

Quadtrees, introduced in 1974 by Finkel and Bentley, are data structures most commonly used for decomposition of two-dimensional space. In the two-dimensional case, every node in the quadtree represents a rectangle. This rectangle might then be divided into four sections corresponding to the quadrants of the rectangle relative to the coordinates of a given point (chosen to optimise search time) in the rectangle, forming four subtrees (Knuth, 1998). An example of space subdivided by a quadtree can be seen in figure 4.3.

The octree (octal tree) is an analogous construction in three dimensions having eight-way branching with each subtree corresponding to an octant in a cuboid. The quadtree structure might thus be generalised to k dimensions, with a k-dimensional quadtree having 2^k branching (making a one-dimensional quadtree a normal binary search tree) (Knuth, 1998; Havran, 2000).

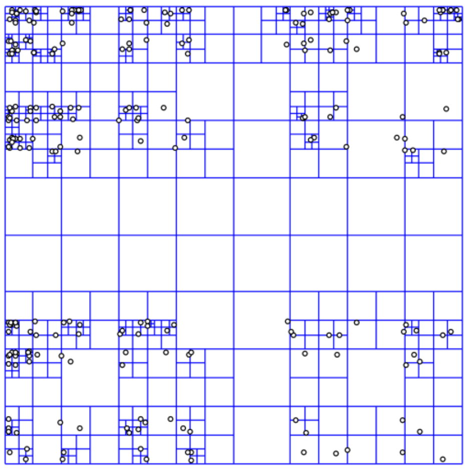


Figure 4.3: Two-dimensional space subdivided by a quadtree. Picture by Eppstein et al. (2005).

Whether or not to perform further (recursive) subdivision of a node's cell is usually decided based on the recursion depth compared to a maximum allowed depth or the number of objects occupying the cell; cells containing many objects might be further split if it is computationally optimal to do so (Havran, 2000).

k-d trees

The k-d tree (k-dimensional tree) was introduced by Bentley in 1975 and is a form of binary search tree and a simplification of the quadtree structure. However, instead of comparing on every coordinate at each branching, k-d trees compare on only one coordinate at a time. On branching, a hyperplane called a *splitting plane* is selected, splitting the k-dimensional hyperspace represented by the parent node at a certain value of the branching coordinate into two nodes, making the basis vector of the branching coordinate the normal vector of the splitting plane. In general, branching at the lth level might be performed on coordinate number $(l \bmod k) + 1$, yielding $x, y, z, x, y, \ldots$ in a 3-d tree (with the root node as level 0) (Knuth, 1998).

Since the splitting planes are always perpendicular to one of the coordinate axes, the k-d tree is said to be *axis-aligned*. This property greatly simplifies intersection testing between a ray and the splitting planes, requiring about one third as many elementary operations as for arbitrarily oriented splitting planes, as shown by Havran (2000, p. 52-53). Furthermore, testing of whether a given object belongs to one of the halfspaces induced by an arbitrarily oriented splitting plane is considerably more computationally demanding, and the build time of the k-d tree increases from $O(N)$ to $O(N^3)$.

Positioning of the splitting plane The positioning of the splitting plane in non-leaf nodes is an important problem in the construction of k-d trees, and Havran (2000) proposes a cost model, based on the model introduced by MacDonald and Booth in 1989, for this. Other possible methods are the spatial median and object median methods, which position the plane so as to balance the space and number of objects, respectively, on both sides. The cost model, however, performs statistically better and will therefore be explained in greater detail. It attempts to algorithmically estimate the average cost, in terms of computation needed, of an arbitrary ray traversing the k-d tree, under certain assumptions, during the construction of the tree. The most important underlying observation is that the conditional probability $P(Y|X)$ of an arbitrary ray intersecting the convex spatial area Y given that it passes through the enclosing convex spatial area X (such that $X \cap Y = Y$) is proportional to the surface area of Y divided by the surface area of X:

$$P(X|Y) = \frac{SA(Y)}{SA(X)} \tag{4.7}$$

For axis-aligned bounding boxes X and Y, this yields the expression:

$$P(X|Y) = \frac{Y_{width}Y_{height} + Y_{width}Y_{depth} + Y_{height}Y_{depth}}{X_{width}X_{height} + X_{width}X_{depth} + X_{height}X_{depth}} \tag{4.8}$$

Havran (2000) goes on to show that the upper bound of the estimated total cost $\hat{C}_T[s]$, then, of shooting an arbitrary ray through a k-d tree can be expressed as follows:

$$\hat{C}_T[s] = \frac{1}{SA(root)} \cdot \left[\hat{C}_{TI} \cdot \sum_{N_i}^{i=1} SA(i) + \hat{C}_{TL} \cdot \sum_{N_l}^{l=1} SA(l) + \hat{C}_{IT} \cdot \sum_{N_l}^{l=1} SA(l)N(l) \right], \tag{4.9}$$

where

N_i	the number of interior nodes in the tree,
N_l	the number of leaf nodes ($N_l = N_i + 1$),
$N(l)$	the number of objects stored in the leaf node l,
$SA(n)$	the surface area of node n (*root*, then, being the whole scene),
$\hat{C}_{TI}$	the estimated cost of traversing an interior node,
$\hat{C}_{TL}$	the estimated cost of traversing a leaf node,
$\hat{C}_{IT}$	the estimated cost of performing a ray-object intersection test.

Because of the relation between the cost and the surface areae of the node cells, an algorithm that tries to minimise the estimated total cost is referred to by MacDonald and Booth as a *surface area heuristic* (SAH). For a more comprehensive description and analysis, as well as algorithms, see Havran (2000).

Traversal Wald (2004), building primarily upon previous work by Havran, discusses a few algorithms for fast ray traversal of k-d trees. Throughout traversal, the *current line segment* $[t_{near}, t_{far}]$ is maintained and updated in each step. This is the parameter interval of the ray that actually intersects the current cell, and is first initialised to $[0, \infty)$ and then clipped to the root node's cell (*i.e.* the bounding box of the entire scene). At each step, the signed distance d to the current node's splitting plane is calculated and compared to the distance to the current line segment. Three distinct possibilities arise:

$d \geq t_{far}$	The ray segment is completely in front of the SP
$d \leq t_{near}$	The ray segment is completely behind the SP
$t_{near} > d > t_{far}$	The ray segment intersects both sides of the SP

In the first two cases, where the ray segment lies completely on one side of the splitting plane, the subtree on the other side can be "culled" and only the subtree corresponding to the intersected cell traversed. If the ray intersects both children, however, both subtrees need to be traversed in turn. First, the "near" side is traversed with line segment $[t_{near}, d]$, and then the "far" side is traversed with $[d, t_{far}]$. This use of parametrised ray segments allows all computations to be performed in one dimension, which significantly simplifies the process. Since the cells are traversed in front-to-back order, traversal may be terminated as soon as intersection with an object is detected inside a cell, yielding potentially large performance gains. This algorithm naturally lends itself to a recursive implementation, however Wald (2004) and Havran (2000) note that an iterative implementation gives better performance.

Performance According to Havran (2000), the k-d tree performs statistically better than its competitors when used with common heuristic ray shooting algorithms. Furthermore, he argues that the k-d tree has relatively low memory complexity, with the number of cells increasing roughly linearly with the number of objects in the scene. Construction of a static k-d tree for orthogonal rectangles in three dimensions can, using the right approaches, be achieved in $O(N \log N)$ time (Wald and Havran, 2006).

4.1.4 Results

Due to the simplicity of the objects involved in collision detection in a racing game, rays are suited very well for this particular application. In essence, the player ship, which cannot be deformed in any manner and thus can have rays attached to fixed points on its surface, needs only be tested for collision against the environment, which is static, and other ships.

The ship class in *SLERP 3D* stores one two-dimensional array of rays for each side of the ship (up, down, right, left, forwards and backwards). The placement of those rays may either be defined by the ship model in a way efficient for the particular model or determined dynamically by the program upon initialisation by shooting a given number of rays inwards from the six sides of the ship's bounding box and affixing rays where intersection with the ship is detected. The stored rays are defined in object space, and a world-space copy is created for each ray on every update using the ship node's transformation. In order to detect negative values of the parameter t in the ray equation (equation 4.1), an attempt was made to move the origin of the world-space ray back to the opposite side of the ship before evaluation of the ray. It was not moved back further as to avoid detecting intersections with obstacles lying on the other side of the ship, since no assumptions might be made about the environment or orientation of the ship. However, this caused unpredictable side effects such as the ship being erratically relocated even when not colliding with the ground, and falling through was still possible (although less likely) and occurred seemingly at random. Since the precise cause could not be determined, it was decided that the reduced risk of driving through the walls and thus falling off the track was not worth the problems caused by this method.

As other intersection test methods are provided by the XNA framework, ray-triangle intersection testing is the only method explicitly implemented in *SLERP 3D*, using the algorithm by Möller and Trumbore presented in section

4.1.2. The environment is stored in a k-d tree, the construction of which increases the loading time but which substantially decreases the load on the CPU, giving better performance during gameplay. In order to reduce loading time, however, the construction of the k-d tree is performed in a separate thread from the moment the application starts. Other space-partitioning data structures were considered, such as the octree, but the k-d tree was settled upon due to its superior performance despite its complexity in implementing. The ray traversal algorithm implemented in *SLERP 3D* is the TA_{rec}^{B} algorithm presented in Havran (2000, appendix C, p. 157-159).

A further, albeit minor, improvement in performance was obtained by including an extra parameter to the ray traversal method of the k-d tree setting the maximum distance a ray should be traced (or `null` if the ray is to be traced until it intersects an object or leaves the scene), allowing early termination of traversal. This parameter is obtained once for all the rays pointing in the same direction by calculating the dot product $v_{\mathbf{d}} = \mathbf{v} \cdot \mathbf{d}$ of the ship's velocity $\mathbf{v}$ and the rays' direction vector $\mathbf{d}$ before sending the array containing the rays for evaluation. If $v_{\mathbf{d}} \leq 0$, none of the rays need to be tested against the environment as it is known to be static and the ship is not currently moving in the direction $\mathbf{d}$. It should be noted that this is not done for the rays going down from the ship, as the current distance to the ground needs to be measured on every update in order to maintain height.

4.2 Network Gaming

Support for multiplayer functionality in games is becoming an increasingly critical feature to implement for game developers. With a huge amount of games entering the market each year, the publishers and developers are fighting for the money and spare time of the gamers. With most of the games tending to leave big holes in the gamers' wallets, the gamers are looking for lasting appeal in games, and multiplayer support can vastly improve the amount of time spent with a game. Also, with the current trend of games being rented or sold second hand, the multiplayer functionality of the game can be the difference of a game kept and a game traded in.

Networked multiplayer games are most commonly played either over the Internet (online) or on the local network (LAN). A game session held on a local network very much resembles a session held over the Internet when it comes to the communication between the involved units, but with a few important differences. Firstly, only computers connected to the same local network may join the network session. Also, due to the high proximity of the networked computers, a LAN offers a high connection speeds. This made the LAN a popular solution when network gaming was in its infancy, and it continues to appeal to a lot of gamers, alongside the networking possibilities of the Internet (Lecky-Thompson, 2008).

Online games, in contrast, are played with computers possibly distributed all over the world, with possibly hundreds or thousands of intermediate routers and links with varying bandwidth and quality. Therefore, online games face far more synchronisation problems related to latency than a local network game. *Latency* is the time it takes for a request to be answered, in other words the network delay (often called *lag*) (Lecky-Thompson, 2008, p. 91).

4.2.1 Client/Server Networking

One approach to networking is the so-called *Client/Server* model. The setup for this networking model is that one network node (this could be a computer or any other machine with network capabilities), called the server, is responsible for the communication between all of the connected computers. The other computers are called clients to this responsible node that services their requests (Kurose and Ross, 2007, p. 38). An application using this networking technique is an example of a so-called distributed application. Applications commonly using the Client/Server networking solution include e-mail programs and the World Wide Web (Kurose and Ross, 2007, p. 170).

The Client/Server networking solution has also been commonly used in variations for computer games, for example in *Half-Life*, *Counter Strike* and *Unreal Tournament* gngine games (Bernier, 2001).

Bernier (2001) describes, at an abstract level, how the client and server applications divide the tasks of real-time rendering. Basically it is done in the following way: the client samples the user input and sends the associated data over the network to the server. The server processes the user input and moves all involved objects accordingly. The server then responds to all connected clients with the relevant data of the objects/world involved in the scene. When this data is received by the client it determines visible objects and finally renders the scene accordingly.

A positive effect concerning latency with the Client/Server model is that the latency perceived by the server is roughly the same for all nodes connected to it. This can for example make fast-paced games fairer. Steed and Oliveira (2009) discusses this and claims that the most important benefit from using the Client/Server approach is that the server's activities can be controlled and secured, while the peers of a peer-to-peer network are unreliable and prone to perform activities such as cheating.

One of the most prominent negative effects of using a Client/Server approach is the need to maintain a centralised unit, which can be both economically costly and onerous (Isensee and Ganem, 2003).

4.2.2 Peer-to-Peer Networking

With the peer-to-peer networking model the nodes in the network communicate directly between each other, in contrast to the Client/Server model which uses a responsible unit called a server through which the communication data is distributed (See section above about Client/Server Networking) (Brookshear, 2006).

For game applications this means that game data has to be sent to all of the other nodes in the network, not just one node as in the Client/Server model. If communicating over the Internet, this will put strain on the Internet connection, demanding more bandwidth. This issue can however be managed by allowing one node to act as a *host*; each node will send its data to this host, and the host will gather all of the data into one package and send it to all of the other nodes. This makes it possible for a host with a reliable and fast Internet connection to handle and support several other nodes, which could suffer from poor bandwidth (Lincroft, 1999; Isensee and Ganem, 2003).

Steed and Oliveira (2009) performs a case study of Criterion's *Burnout Paradise*. The game uses peer-to-peer networking where one player acts as a *host*, which performs no extra networking functionality other than controlling the initialisation of different game-events.

A positive thing about the peer-to-peer model is that host migration is easily implemented, in contrast to the Client/Server model(Mic, Unknown year). Furthermore it is easy to administer a peer-to-peer implementation since no centralised unit such as a server needs to be maintained (Isensee and Ganem, 2003).

4.2.3 Results

In a racing game, there are several different game modes that are often present in games currently on the market, and historically. A set of commonly implemented game modes for racing games include: time trial, single player racing against AI, split-screen multiplayer racing and online racing (sometimes also over LAN). However, all of these are not present in *SLERP 3D*.

Split-screen multiplayer was a candidate for implementation; much so because we considered this functionality easy to implement. However, the implementation of this feature would contradict one of our main goals with the project, namely the ambition to make a game filled to the brim with graphical effects and 3D splendour. The reason for this was that an implementation of split-screen multiplayer would demand an impairment of the game's visual quality due to the fact that the 3D scene would need to be rendered two (or up to four [1]) times.

Our intention was to make the gaming experience both fun and compelling. The time trial feature was spoken of as a worst case game mode to implement, if no other game modes could be implemented adequately. That, and the fact that feedback from potential short-term users of the game requested it[2], led to us focus on implementing network functionality. From the beginning, however, we felt no urgency to implement this feature; the reason being that the focus of the project was the real-time graphics. A contender to the implementation for network multiplayer support was the inclusion of AI-controlled opponents. Here, the fact remains that both alternatives do not directly coincide with the aims of the project, but as already mentioned, an other intention was to make the game entertaining, which requires a certain amount of game play. What it finally came down to was, besides the feedback previously mentioned, that the LAN-multiplayer simply was considered by us as the most fun to play game mode of the two.

About choosing what network model to use, we reasoned that peer-to-peer would be most feasible in our situation. Mostly, because it demands very little support from our part, and, as a bonus, we were able to implement host migration with just one line of code in XNA. Furthermore, since we only provide LAN multiplayer support for *SLERP 3D*, the problem with latency is pretty much nonexistent.

[1] XNA allows a maximum of four players split-screen.

[2] During a Q&A session in connection to the half-time presentation of the project, the audience asked if multiplayer was to be supported.

Chapter 5

Conclusion

5.1 Results

When implementing the algorithms in our case study, *SLERP 3D*, we found that certain algorithms provide a more visible result than others. We estimate that about 50% of our time spent on the project went into implementing graphics, and this has to be taken into consideration when reading the results of this thesis. Since the focus was to find what algorithms are suited when developing a game with a short time-frame as well as a limited amount of people, we believe that it is most important to focus on the effects that provide a significant gain in visual quality. The algorithms which we found added most visual quality to the game were deferred lighting, shadows, bloom, HDR and particle systems. Deferred lighting gave us the possibility to insert numerous lights into our 3D scene, giving us artistic freedom of placing light sources pretty much wherever we want them without having to consider their cost. Shadows give the 3D scene a great sense of depth, which makes the scene seem more realistic. The Bloom and HDR techniques both mimic the behaviour of either the human eye or a camera lens; these techniques give the effect of observing something through a real camera, and moreover have the potential to substantially affect the game's visuals. By mainly focusing on these algorithms, we think that it is very possible to create a game with relatively high graphical quality in a short amount of time.

5.2 Discussion

Since we used the XNA Game Framework when creating the game, we had a certain base framework to build our code upon. This certainly helped us save a lot of development time, but at the same time caused a few bugs due to a few non-intuitive features in the framework. Nevertheless, in the long run, the XNA framework saved us a lot of time by providing built-in functionality for *e.g.* mathematical functions, game loop updating and window creation and management. These gains do not mainly affect the development time for computer graphic algorithms, but mostly the code needed around the graphics and the game kernel.

There are other effects that we think should have been implemented should we have had enough time. Using different BRDF models for different materials is important in order to provide a convincing and realistic look of the rendered image. We should also have chosen a better motion blur algorithm and made sure that it could be properly integrated into our rendering pipeline early in the development process.

5.3 Future work

Since we did not focus on the creation of content to our game, we tried as much as possible to avoid algorithms that required additional work when creating the assets (*e.g.* bump mapping (Blinn, 1978)). However, these types of effects are also important and we think that future work could evaluate the efficiency of these techniques in a setting where more focus can be spent on creating assets for the game, both based on visual gain as well as considering the time it takes to implement these algorithms.

Bibliography

Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. ISBN 987-1-56881-424-7. URL `http://realtimerendering.com`. Cited on pages 2, 12, 13, 14, 16, 22, 26, 28, 29, 30, 31, 32, 33, 37, and 38.

Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential shadow maps. In *GI '08: Proceedings of graphics interface 2008*, pages 155–161, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0. Cited on page 21.

Michael Ashikhmin and Peter Shirley. An anisotropic phong brdf model. *Journal of Graphics Tools*, 5:25–32, 2000. Cited on page 12.

Ulf Assarsson. *A Real-Time Soft Shadow Volume Algorithm*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, oct 2003. Cited on page 18.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Principles behind the agile manifesto, 2001. URL `http://www.agilemanifesto.org/principles.html`. Cited on page 6.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361002.361007. Cited on page 41.

Yahn W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization. Wiki, 2001. URL `http://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization#Basic_Architecture_of_a_Client_.2F_Server_Game`. Cited on page 45.

James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA,

1977. ACM. doi: http://doi.acm.org/10.1145/563858.563893. Cited on page 12.

James F. Blinn. Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292, 1978. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/965139.507101. Cited on page 48.

J. Glenn Brookshear. *Computer Science: An Overview (9th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321387015. Cited on page 45.

Martin Čadík, Michael Wimmer, Laszlo Neumann, and Alessandro Artusi. Evaluation of hdr tone mapping methods using essential perceptual attributes. *Computers & Graphics*, 32:330–349, 2008. ISSN 0097-8493. URL `http://www.cgg.cvut.cz/members/cadikm/tmo/cadik08cag.pdf`. Cited on page 28.

Dean Calver. Deferred lighting on ps 3.0 with high dynamic range. In Engel (2004), pages 97–105. ISBN 1584503572. Cited on pages 28 and 29.

R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, 1982. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/357290.357293. Cited on page 12.

Franklin C. Crow. Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 242–248, New York, NY, USA, 1977. ACM. doi: http://doi.acm.org/10.1145/563858.563901. Cited on page 17.

Rouslan Dimitrov. Cascaded shadow maps, 2007. URL `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`. Cited on page 19.

Zhao Dong and Baoguang Yang. Variance soft shadow mapping. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 1–1, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-939-8. Cited on page 20.

William Donnelly and Andrew Lauritzen. Variance shadow maps. In *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 161–165, New York, NY, USA, 2006. ACM. ISBN 1-59593-295-X. doi: http://doi.acm.org/10.1145/1111411.1111440. Cited on page 20.

Frédo Durand and Julie Dorsey. Interactive tone mapping. In *Proceedings of the Eurographics Workshop on Rendering*. Springer Verlag, June 2000. URL `http://people.csail.mit.edu/fredo/PUBLI/EGWR2000/index.htm`. Held in Brno, Czech Republic. Cited on pages 27, 28, and 29.

Philip Dutré. Global illumination compendium, sep 2003. Available at http://www.cs.kuleuven.be/p̃hil/GI/. Cited on pages 30 and 39.

Wolfgang Engel, editor. *ShaderX^3: Advanced Rendering with DirectX and OpenGL*. Number 3 in ShaderX. Charles River Media, Inc., Rockland, MA, USA, 2004. ISBN 1584503572. Cited on pages 50 and 55.

Wolfgang Engel. Light pre-pass renderer, 2008. URL `http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html`. Cited on page 14.

David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In Joseph S. B. Mitchell and Günter Rote, editors, *Symposium on Computational Geometry*, pages 296–305. ACM, 2005. ISBN 1-58113-991-8. Cited on page 41.

Randima Fernando. Percentage-closer soft shadows. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 35, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1187112.1187153. Cited on page 20.

Dominic Filion and Rob McNaughton. Effects & techniques. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 133–164, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1404435.1404441. Cited on page 14.

Raphael Ari Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974. Cited on page 40.

David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994. Cited on page 11.

H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(6):623–629, 1971. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/T-C.1971.223313. Cited on page 12.

Simon Green. Stupid opengl shader tricks, 2003. URL `http://developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf`. Cited on pages 27 and 35.

Larry Gritz and Eugene d'Eon. The importance of being linear. In Nguyen (2007), chapter 24. ISBN 0321515269. Available at http://developer.nvidia.com/object/gpu-gems-3.html. Cited on page 16.

Paul Haeberli and Kurt Akeley. The accumulation buffer: hardware support for high-quality rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 309–318, New York, NY, USA, 1990. ACM. ISBN 0-89791-344-2. doi: http://doi.acm.org/10.1145/97879.97913. Cited on page 27.

Shawn Hargreaves. The importance of transitions. Blog, 2007a. URL `http://blogs.msdn.com/shawnhar/archive/2007/03/02/the-importance-of-transitions.aspx`. Cited on page 8.

Shawn Hargreaves. The importance of transitions. Blog, 2007b. URL `http://blogs.msdn.com/shawnhar/archive/2007/05/24/transitions-concluded-there-is-no-spoon.aspx`. Cited on page 8.

Jiří Havel and Adam Herout. Yet faster ray-triangle intersection (using sse4). *IEEE Transactions on Visualization and Computer Graphics*, 16:434–438, 2010. ISSN 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2009.73. Cited on page 40.

Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, nov 2000. URL `http://www.cgg.cvut.cz/~havran/phdthesis.html`. Cited on pages 37, 40, 41, 42, 43, and 44.

Tim Heidmann. Real shadows, real time. *Iris Universe*, 18:23–31, 1991. Cited on page 18.

Jack Hoxley. Lighting (summary), 2008. URL `http://wiki.gamedev.net/index.php/D3DBook:(Lighting)_Summary`. Cited on page 12.

Pete Isensee and Steve Ganem. Developing online console games. Web Article, 2003. URL `http://www.gamasutra.com/view/feature/2875/developing_online_console_games.php`. Cited on pages 45 and 46.

Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001. Cited on page 23.

James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20 (4):143–150, 1986. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/15886.15902. Cited on page 3.

Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 99–107, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-939-8. doi: http://doi.acm.org/10.1145/1730804.1730821. Cited on page 24.

Scott Kircher and Alan Lawrance. Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In *Sandbox '09: Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, pages 39–45, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-514-7. doi: http://doi.acm.org/10.1145/1581073.1581080. Cited on page 15.

Donald Ervin Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0. Cited on pages 40 and 41.

Rusty Koonce. Deferred shading in tabula rasa. In Nguyen (2007), chapter 19. ISBN 0321515269. Available at http://developer.nvidia.com/object/gpu-gems-3.html. Cited on page 14.

Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel. Perceptual effects in real-time tone mapping. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 195–202, New York, NY, USA, 2005. ACM. ISBN 1-59593-203-6. doi: http://doi.acm.org/10.1145/1090122.1090154. Cited on pages 30 and 35.

James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, fourth edition, 2007. ISBN 0321497708. Cited on page 45.

Andrew Lauritzen. Summed-area variance shadow maps. In Nguyen (2007), chapter 8, pages 157–182. ISBN 0321515269. Available at http://developer.nvidia.com/object/gpu-gems-3.html. Cited on pages 20 and 21.

Guy W. Lecky-Thompson. *Fundamentals of Network Game Development*. Course Technology PTR, 2008. ISBN 9781584506256. Cited on page 44.

Peter Lincroft. The internet sucks: Or, what i learned coding x-wing vs. tie fighter. Web Article, 1999. URL `http://www.gamasutra.com/view/feature/3374/the_internet_sucks_or_what_i_.php`. Cited on page 45.

J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. In *Graphics Interface '89*, pages 152–163, jun 1989. Cited on pages 41 and 42.

Morgan McGuire. Ambient occlusion volumes. Technical Report CSTR200901, Williams College Computer Science Department, Williamstown, MA, USA, dec 2009. URL `http://graphics.cs.williams.edu/papers/AOV09/`. Cited on pages 23 and 24.

Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 77–89, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-603-8. doi: http://doi.acm.org/10.1145/1572769.1572783. Cited on page 23.

Network Topologies and Host Migration. Microsoft Corporation, Unknown year. URL `http://msdn.microsoft.com/en-us/library/bb975826.aspx`. Cited on page 46.

Martin Mittring. Finding next gen: Cryengine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 97–121, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1281500.1281671. Cited on pages 23 and 24.

Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics, gpu, and game tools*, 2(1):21–28, 1997. Cited on pages 38 and 43.

Hubert Nguyen, editor. *GPU Gems 3*. Number 3 in GPU Gems. Addison-Wesley Professional, first edition, aug 2007. ISBN 0321515269. Available at http://developer.nvidia.com/object/gpu-gems-3.html. Cited on pages 51, 52, 53, and 54.

Microsoft DirectX 10: The Next-Generation Graphics API. NVIDIA Corporation, 2006. Cited on page 14.

Michael Oren and Shree K. Nayar. Generalization of lambert's reflectance model. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 239–246, New York, NY, USA,

1994. ACM. ISBN 0-89791-667-0. doi: http://doi.acm.org/10.1145/192161.192213. Cited on page 12.

John O'Rorke and Greg James. Real-time glow. Web article, 2004. URL `http://www.gamasutra.com/view/feature/2107/realtime_glow.php`. Cited on pages 34 and 35.

Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360825.360839. Cited on page 12.

William T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. In *Seminal graphics: poineering efforts that shaped the field*, pages 91–108. ACM, New York, NY, USA, 1998. ISBN 1-58113-052-X. doi: http://doi.acm.org/10.1145/280811.280996. Cited on page 21.

William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 283–291, New York, NY, USA, 1987. ACM. ISBN 0-89791-227-6. doi: http://doi.acm.org/10.1145/37401.37435. Cited on pages 18 and 19.

Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. Graph.*, 21(3):267–276, 2002. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/566654.566575. Available at http://www.cs.utah.edu/ reinhard/cdrom/. Cited on pages 27, 28, 29, 30, and 35.

Gilberto Rosado. Motion blur as a post-processing effect. In Nguyen (2007), chapter 27. ISBN 0321515269. Available at http://developer.nvidia.com/object/gpu-gems-3.html. Cited on pages 27 and 35.

Bradley Sanford. Integrated graphics solutions for graphics-intensive applications. White paper, 2002. Cited on page 3.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29:10–21, 2009. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2009.9. Cited on page 3.

Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2 edition, 2007. ISBN 0470018666. Cited on pages 8 and 9.

Oleg Shishkovtsov. Deferred shading in s.t.a.l.k.e.r. In *GPU Gems 2*, chapter 9. Addison-Wesley, 2005. ISBN 0321335597. Cited on page 14.

Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536, New York, NY, USA,

2002. ACM. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566612. Cited on page 25.

Tiago Sousa. Adaptive glare. In Engel (2004), pages 349–355. ISBN 1584503572. Cited on pages 33, 34, 35, and 36.

Greg Spencer, Peter Shirley, Kurt Zimmerman, and Donald P. Greenberg. Physically-based glare effects for digital images. In *SIGGRAPH*, pages 325–334, 1995. Cited on pages 31, 32, and 33.

Marc Stamminger and George Drettakis. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, New York, NY, USA, 2002. ACM. ISBN 1-58113-521-1. doi: http://doi.acm.org/10.1145/566570.566616. Cited on page 18.

Anthony Steed and Manuel Oliveira. *Networked Graphics: Building Networked Games and Virtual Environments*. Elsevier Inc., 2009. ISBN 978-0-12-374423-4. Cited on page 45.

Michal Valiant. Deferred rendering in killzone 2. Online, 2007. URL `http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf`. Cited on pages 13 and 14.

Thomas J T P van den Berg, Michiel P J Hagenouw, and Joris E Coppens. The ciliary corona: physical model and simulation of the fine needles radiating from point light sources. *Invest Ophthalmol Vis Sci*, 46(7): 2627–32, 2005. ISSN 0146-0404. URL `http://www.biomedsearch.com/nih/ciliary-corona-physical-model-simulation/15980257.html`. Available at http://www.iovs.org/cgi/content/full/46/7/2627. Cited on page 31.

Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/. Cited on pages 42 and 43.

Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). In Ingo Wald and Steven G. Parker, editors, *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, September 2006. Cited on page 43.

Gregory J. Ward. Measuring and modeling anisotropic reflection. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 265–272, New York, NY, USA, 1992. ACM. ISBN 0-89791-479-1. doi: http://doi.acm.org/10.1145/133994.134078. Cited on page 12.

Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, New York, NY, USA, 1978. ACM. doi: http://doi.acm.org/10.1145/800248.807402. Cited on pages 17 and 18.

Matthias Wloka and Robert Zeleznik. Interactive real-time motion blur. *The Visual Computer*, 12:183–295, 1996. Cited on page 26.

Appendix A

Contributions

The planning of the project was done by the whole group during the meetings throuhout the project. Information gathering was done individually depending on each member's needs and responsibilities.

A.1 Responsibilities

Rickard von Haugwitz

Collision Detection - k-d tree, Rays
Game Logic - Physics, Input Management
Post Processing - Bloom, Tone Mapping
Networking

Daniel Lindén

Graphics Management
Collision Detection - Intesection Testing Framework
Post-Processing Effects - Motion Blur, SSAO
Lighting System - Deferred Lighting, Shadows
Particle System - Mega Beams, Explosions
Effect System
Resource Loading

Bartolomeus Jankowski

3D Modelling
Texturing
Level Design
Resource Organization

Magnus Olausson

Menu System - Design, Implementation
Mouse Input System
Settings - Configurations
2D Graphics and Design - Menu, HUD

HUD - Reactions
Networking - Framework
Post Processing Effects - Lens Flare

David Sundelius

System Design - UML
System Architecture
Particle System - Respawn Aura
HUD - Implementation, Functionality
Game Logic - State Management, In-Game Logic
Networking - Powerups
Powerup System

A.2 Problem solving, synthesis and analysis

Problems were most often solved by the individual that encountered them, and if not, they were discussed and solved as a group. Everyone came with many good and bad ideas, causing constructive discussions within the group, eventually leading to the good ideas appearing in the game (at least some of them). Conclusions have been made as a group.

A.3 Report contributions

Editorial work was done by Daniel Lindén, Magnus Olausson and David Sundelius with special credits to our main editor Rickard von Haugwitz for his extensive work on editing and typesetting.

Rickard von Haugwitz Sections 1.3, the parts on glare effects and tone mapping in 3.5, and 4.1 (except the introduction).

David Sundelius Sections 1.1, 1.2, 1.5, 2.2 and 3.3.

Magnus Olausson Sections 1.3, 2.1, the parts on deferred shading in 3.1.3, and 4.2.

Daniel Lindén Sections 1.4, 3.1 (except the part on deferred shading), 3.2, 3.4, the parts on motion blur in 3.5, the introduction to section 4.1 and 5.