CHALMERS

Road Kill

– Game development with limited resources

Bachelor's Thesis
Computer Science and Engineering Programme

Viktor Arvidsson
Jonathan Gustafsson
Per Jamot Johansson
Christoffer Nilsson
Adam Sällergård
Robin Ytterlid

Institutionen för Data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2011

Kandidatarbete/rapport nr 2011:039

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Gothenburg
Sweden

Sammanfattning

Det här är ett kandidatarbete som beskriver en studie där ett spel utvecklas med begränsade resurser och moderna tekniker. Fokus ligger på att utforska de olika stegen och de olika tillvägagångssätten som man stöter på under utvecklingen av ett spel ur ett perspektiv där spelutvecklaren har begränsade resurser. För att kunna undersöka dessa olika steg noggrant så har ett spel, vid namn Road Kill, utvecklats vid sidan av de teoretiska studierna under vårterminen 2011.

Trots att vi siktade högre än vad vi först var bekväma med så är vi väldigt nöjda med slutresultatet. Vi har i detalj utforskat flera av spelutvecklings olika steg, så som utveckling och implementering av fysik-, med tekniker som *bounding volumes* och *spatial data structures*; och grafiklösningar, med tekniker som *shadow volumes, culling* och *particle systems*; spellogik, flerspelarläge över nätverk och modellering. Dessa steg har utvärderats och presenterats så att det ska vara lätt för utvecklare med begränsad erfarenhet ska kunna få en snabb inblick i vanliga tekniker som kan användas för spelutveckling.

Abstract

This bachelor thesis presents a case study, where the development of a game is done with limited resources and modern techniques. The focus lies in examining the different steps and the many small choices that are continually made during the development of a game, from the perspective of a game developer that is limited in resources. To be able to study these different steps in detail, a game, Road Kill, has been developed in parallel with the theoretical studies during the spring semester 2011.

Even though we aimed higher then we, at first, were comfortable with, we are very happy with the end result of this thesis. We have studied, in detail, several of the different steps of game development, such as, development and implementation of physics-, with techniques such as *bounding volumes* and *spatial data structures;* and graphics solutions, with techniques as *shadows volumes, culling* and *particle systems;* game mechanics, multiplayer over network, and modeling. These steps have been evaluated and presented in a way that it should be easy for a developer with limited experience to get gain insight quickly, in to the usual techniques that can be used for game development.

Acknowledgements

There are a few people that have helped to heighten the quality of this thesis, to which we are very grateful. As our supervisor, Ulf Assarsson continually supported us with much needed advice and feedback. Our friends, who have continued to support us, despite our long working hours. Two friends, Jesper Westerberg and Alexander Eriksson, supplied the soundtrack to Road Kill, and to them, we are extra thankful.

Table of content

1 Introduction

Large game development studios have the resources to develop new games with the best techniques and, because of this, the potential to produce the highest quality games. Smaller game studios have a harder time to compete in the same market due to their more limited resources, such as development time, funds, personnel and experience. It is crucial, due to their limited resources, that small game studios develop their games as efficiently as possible. Arguably, it is even more critical for smaller than larger game studios, as they have a larger capacity to correct errors in the development process without compromising the final product.

This thesis will be particularly interesting for small game developers, who are starting a new project where the goal is to produce a modern game that can compete with games in the same market, produced by larger game studios. Furthermore, large game studios should be able to use this thesis to enhance their knowledge about game development, smaller game projects, or how to do a fast prototype for a game. Even corporations other than game developers should have use for our study, as graphics- and physics engines can be used in many different situations, such as movie making or vehicle collision testing.

1.1 Purpose

The purpose of this thesis is to test the different techniques game developers are able to use and to find simple solutions for the problems that occur during game development. Developers should be able to use our study to learn more about all the stages in game development, and different solutions for programming a game of high quality in short time.

This thesis examines some of the main parts of the game development process, to make the process easier for a small development studios and developers with less experience. Sections of this process that are of higher significance are examined more extensively, and this might provide value, not only to beginners but also, to experienced developers. The thesis brings up pros and cons for different development methods, analyzes which techniques are suitable for which game type, and also considers how difficult and time consuming different algorithms are to implement in a game.

In parallel with the theoretical research made during the writing of this thesis, a car racing game has been developed to enable the testing and evaluation of different techniques in a real time graphics environment. This game was named Road Kill, and was mainly inspired by Death Rally (Remedy Entertainment, 1996).

Road Kill was developed by six students with programming experience from 2.5 years of computer science studies, but no prior experience in game development. Each person contributed with approximately 200 hours of programming time. C++ was chosen as the programming language for the development of Road Kill, as it is used by many game titles with modern graphics. Road Kill was targeted for the Windows platform, but uses the graphics library OpenGL instead of Direct3D and DirectX, which keeps the option to port Road Kill to other platforms than Windows in the future. All software

used in this project was free of charge, with the exception of the modeling tool 3D Studio Max and the sound library FMOD.

1.2 Problem

This thesis describes the possible solutions to the following subset of problems:

- Implementing an efficient and precise physics simulation, with many objects, in real time. That is able to make the objects behave lifelike and interact in a realistic way.
- Choosing and developing a multiplayer solution with good flow and satisfying game play over the local area network and the World Wide Web.
- Creating a complete set of objects, to fill the game world and make it look realistic.
- Developing a complete racing game with limited resources, such as money and time. The game should contain features such as full multiplayer support and four different cars to choose from, each with a different set of physical features. Players should be able to destroy other players' cars with weapons mounted on their own cars.

1.3 Delimitations

This thesis focuses on game development with limited resources, such as development time, personnel, experience and funds. Therefore, delimitations have been made to focus on the study of the most fundamental parts of the game development process.

The marketing aspect is of large importance when it comes to creating a successful game, as Braben claims (Braben, 2011) that a typical $50 million dollar title spends about 30% of its budget on marketing. Due to the technical nature of this thesis and due to the limitations of the thesis budget, the marketing aspect was not included, which allowed for more focus on creating a more advanced game and investigating its development techniques in more detail.

During the development of Road Kill, it was discovered that the development of a physics engine, with a pleasing enough result, would be too time consuming. Therefore, an existing physics engine, that supplies the features that was required, was implemented with the rest of Road Kill's components.

Because of financial and time limitations, this thesis has more focus on producing a general answer to our problem, as described in Section 1.2, than creating the best game engine components or describing all possible techniques with the greatest possible depth. Due to this focus, each section has been limited in its extent.

To simplify the modeling process of Road Kill, this study does not feature any of the other modeling software other than 3D Studio Max.

For a game developer, there are many choices when it comes to choosing programming language, hardware and software platforms. This thesis describes only a limited selection of external libraries, all of which are compatible with the C++ language. Although Road Kill was

developed exclusively in C++, and targeted only for the Windows platform, most of the techniques and algorithms described in this thesis are general enough to be used for other programming languages and platforms.

1.4 Method

In this thesis, each step of the game development process is divided into different chapters. Chapters 2-7 each has a background, followed by a study of different techniques or algorithms that can be used in the corresponding step of the game development process. Each chapter describes problems for the current step in the process and presents different approaches to reach a solution, with focus on both time and resource efficiency. Result and Discussion sections at the end of each chapter further evaluate some techniques that have been tested and used in Road Kill.

While chapters 2-7 describe specific details about different parts of game development, Chapter 8 describes the game Road Kill, which is a result of game development with some of the techniques and algorithms described in the other chapters, and with the limitations described previously in this chapter. Section 8.2 evaluates the development process of Road Kill on an abstract level. A reader of this thesis is therefore advised to see chapters 2-7 for details about and evaluations of different techniques and algorithms, and to see chapter 8 for an abstract summary of Road Kill, its features, and its development process.

1.4.1 Development

There are several different types of development processes that, step by step, describe how a small or big project should be managed. There is the Waterfall model, Incremental delivery, and Reuse-software model (Royce, 1970). The Waterfall model concept is to plan and build up a structure of the project before the development process is started. Early in the process, the project goals are established: prepare a time schedule, and deadlines for each step. To move forward in the project, each phase has to be signed off before the next phase can commence. With this approach, it is hard to deal with changes and problems that are discovered during the process and there is a good possibility that a large amount of code has to be rewritten to suit the new criteria.

Choosing the Reuse-software method, which focuses on the reuse of old code from previous projects, is a good way to save time and to avoid reinventing the wheel. This method, however, is only useful if there is any relevant code already available.

Incremental delivery is a good choice for a beginner developer because the project is broken down into small wieldy versions, and the most important parts are the first to be developed, to get the product going. After that, new features are applied in each new version of the product. Using this method, a new version should be ready in 2-4 weeks and this will lead to a flow that is tightly correlated to specification, and even possible changes to it, during the progression of the project. One major downside with this process model is that building a new version on top of the old

one, tends to increase the complexity of the product, making the structure and maintainability more difficult.

For Road Kill, the incremental delivery technique was used, where basic specifications are decided in the beginning of the study. All features that the finished product should have decided early and where new features are implemented in each sprint.

In the beginning of each sprint, a short meeting was held where all of the group members emphasize problems and exchanged thoughts, so that all the members of the group were kept up to date with the progress. After that, a jointly decision was made on how to proceed in the next sprint.

A group leader was elected to book meetings, tracks the function list, write down decisions, and supervise the progress in the development.

The Waterfall-model requires experience of the process before hand, to plan each part of the game development process, and this was why this model was not chosen for Road Kill.

2 Program structure

A game is a very large software engineering project, which can contain several thousands of rows of programming code. Therefore, it is important for a developer to keep the code structured and modular. A part of the modularity is keeping the abstraction of the game engine as simple as possible, as illustrated in Fig 1.

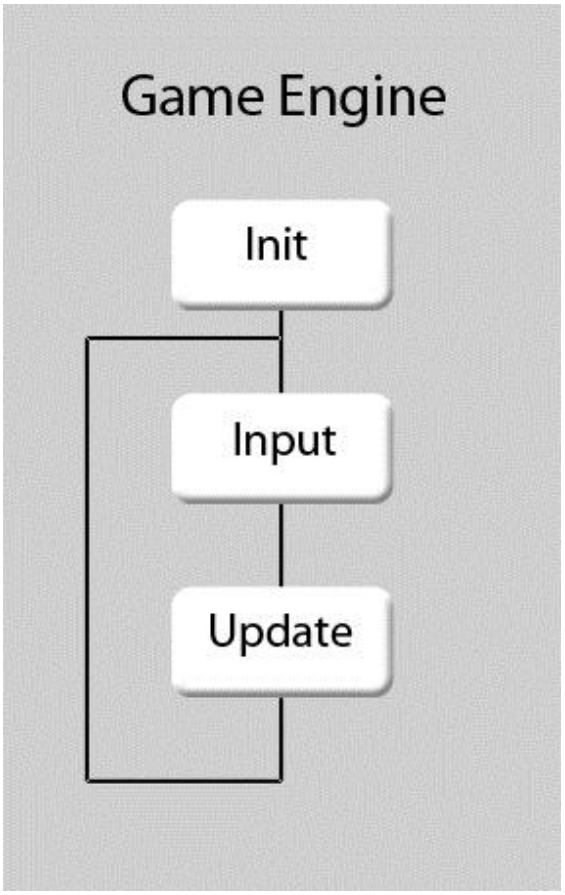

Figure 1: The game engine used in Road Kill.

2.1 Game initialization

The first step in the game engine is to initialize the game and this is done in the following steps that are illustrated in Figure 2:

- Create the track.
- Prepare for multiplayer.
- Add physics.
- Build the cars.

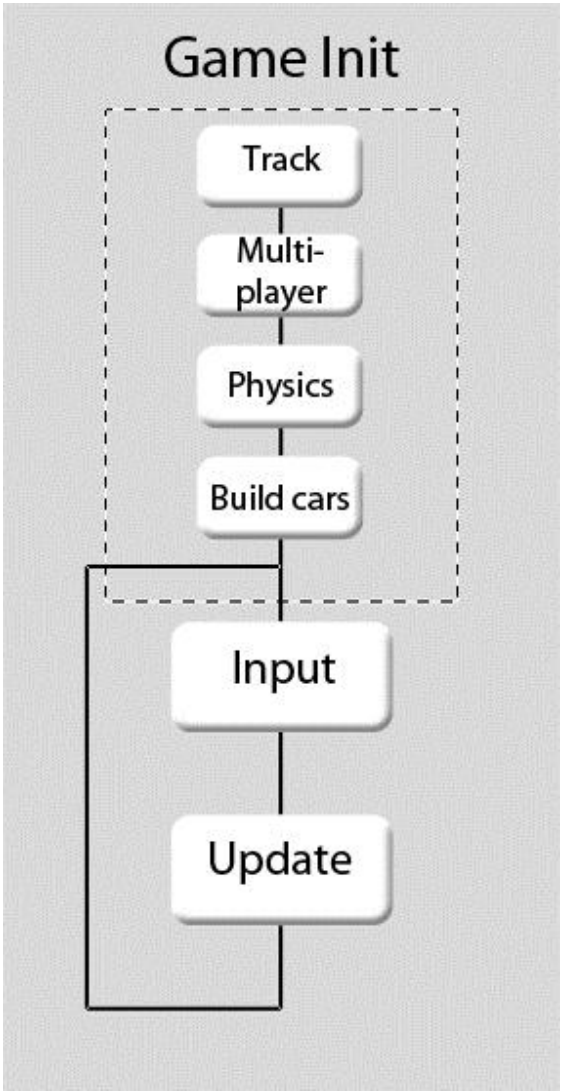

Figure 2: The game engine, with expanded initialization-section.

After the track has been created, Road Kill prompts the players to choose their cars. When the players are ready, Road Kill connects to the server and establishes the multiplayer communication. Now that Road Kill has a set number players, the physics engine and the players' cars are initialized. When the initialization is finished Road Kill enters the main game

loop, which begins with the input loop.

2.2 Input loop

When input loop has begun, Road Kill looks for inputs, such as keyboard events and window changes. All these events are processed before the game state is updated and the input loop is exited.

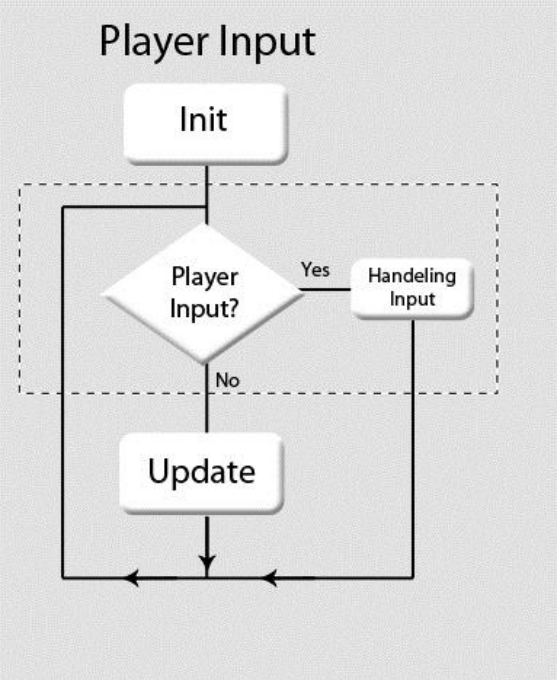

Figure 3: An illustration of the loop that handles the different inputs.

2.3 Game updates

When inputs have been handled, Road Kill is updated in five different steps, as shown in Fig. 4. Each step updates the players' game states variables, such as location, velocity, and position in the race.

Transfer data to and from the server
This step updates the multiplayer state by sending the client's information and receiving new information from the opponents, if any. This step might be limited to only sending data, since Road Kill can receive data packets at a lower rate than the frequency of which the game state is updated. The information being sent and received ranges from car positions to lap times.

Step physics
Whether or not there was any new information in the data transfer step, the physics simulation can now be updated by updating all the players' physical states.

Apply game logic
After Road Kill has updated all of the players' physical states, the logic step checks all new game state variables for new information. This step takes the new information and applies appropriate game mechanics, like the car losing health points when a car is hit by a missile or if the player finished the race.

Render with effects
The new game state, with effects like fire and explosions, is now drawn on the client's screen.

Sounds
New sounds are added where it is needed and old ones are removed, depending on the game state.

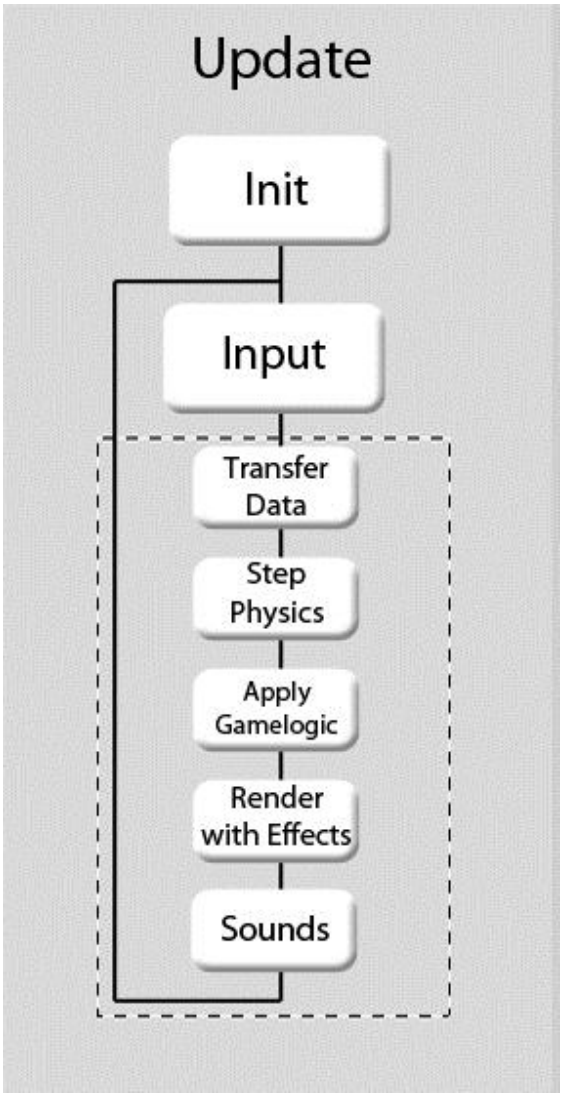

Figure 4: Game update steps in Road Kill.

3 Modeling

When developing a computer game, the visual style depends a lot on the objects that are contained in the game. There would be no point in having good-looking reflections, shadows and other effects if the objects cannot convey them in a way that has high visual appeal. Modeling is the process where objects are created and in this chapter we will explain, in detail, some techniques that are useful. To be able to give more concrete examples these following sections have used 3D Studio Max (Autodesk Inc., 2011a) as the modeling software, this is presented in the delimitation Section 1.3. The techniques can be used in other modeling software but some details may differ.

With modeling, the goal is to create an object using something called vertices, which is a point in the x,y,z space. These vertices can be connected and form all sorts of different objects.

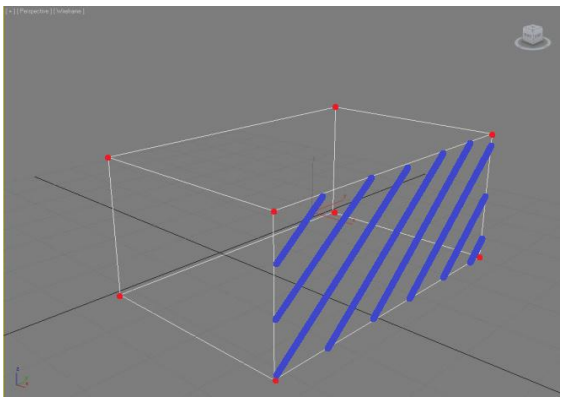

Figure 5: Vertices (the red dots) that are connected by edges to form a box.

Most 3D modeling software has an option to create predefined standard geometric forms such as cylinders, cones and spheres automatically. These can be modified and combined to create objects, making this the easiest way for beginners to start modeling.

3.1 Blueprints

To start modeling objects, a convenient approach is to use something called blueprints. A blueprint consists of an image file including a view of the chosen object from the sides, top, front and back.

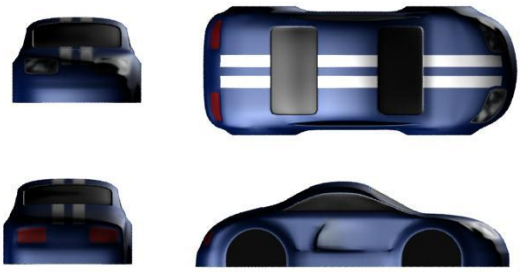

Figure 6: Example blueprint of a car.

When a blueprint has been acquired, an image editing program can be used to cut out the different viewpoints and put them in separate files. These images can then be loaded onto planes in the 3D modeling software, and should be setup like figure 3.

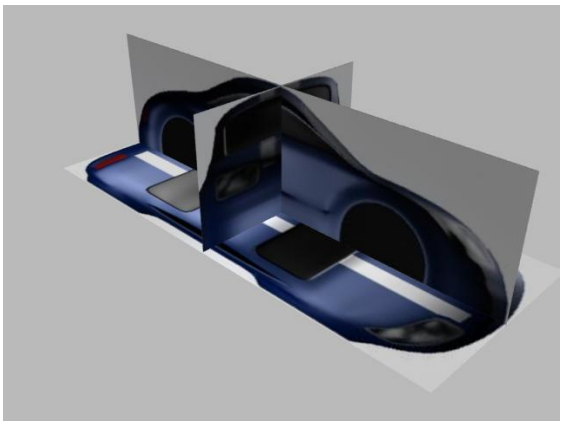

Figure 7: Example of how the blueprint should look in the 3D modeling software.

As seen, it is now a lot easier to make sure that everything is aligned the way it should be and that everything is in the correct scale compared to the real object.

3.2 Modeling a car

When modeling a car from scratch there are several different approaches that can be considered. One of them is to start with a box that consists of many vertices. A vertex is one of the three corners in the triangles, which build up the box. The whole box is built up by small triangles so the graphics card can draw them to the screen. For example, a plane with four vertices is usually divided into two triangles. The surface of objects in 3D Studio Max is called polygons, and the surface is bounded by edges, which have two vertices in each end. Another technique that one can use to model a car is to manually place every polygon so they build up a car. One fast manner is to begin with a box and extrude the polygons. For beginners this is a good way to learn 3D modeling fast. Common for all techniques is that all of them have a trick to get the car to look the same on right and left side. First, one half of the car is modelled, either the right or left side. Then, there is a modifier called symmetry that can take a half car and mirror it to the other side to make it complete.

The box-technique

A technique that can be used is to start off with a box that has many vertices. It is important to keep all vertices organized because when there are as many as 20000 vertices, such as one of the cars in Road kill, it becomes too difficult to move around the vertices one by one. An easy way to keep them organized is to keep the vertices aligned in rows so that an entire row can be selected and the vertices can be moved all at the same time. It is important to think about that the box should have the same amount of vertices from the beginning as the finished car. If one vertex is moved at a time, it is hard to get the vertices in a level, and the finished car would get uneven and dented.

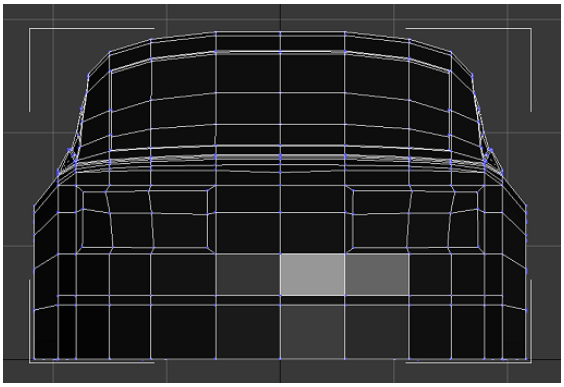

Figure 8: Organized vertices in rows with edges aligned straight seen from the Front view in 3D Studio Max.

To start sculpting the car, one view in 3D Studio Max is selected (front, left or top), and then all the vertices are arranged along the blueprints outline region. After this the car should look such as a car from the selected view but still have the shape of a box in the other two views. The next step in this technique is to do this method once for each view in 3D Studio Max.

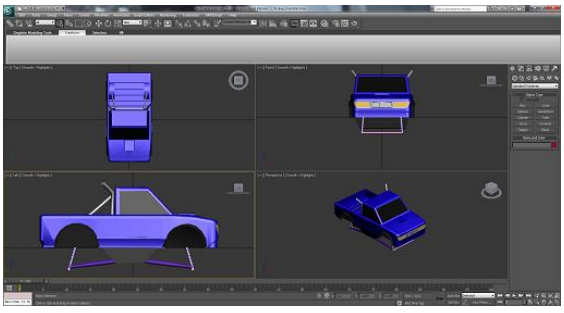

Figure 9: Picture that shows the 4 different viewports: top, front, left and perspective.

The polygon-procedure

A more tactful technique is to position every polygon, one at a time. Usually the start point is at a corner of the object and one side of the object is modeled. The polygons are placed directly where they should be, contrary to the Box technique, and they do not have to move afterwards. This is a good way to make a nice looking car because much time is spent on every part of the vehicle and the focus is only at a small part of the vertices at a time. For this reason the complexity is often reduced and if the vertices are under control it will be easier and faster to get good results (Gahan, 2010).

The extrude-method

Working on a box with fewer vertices can be a good way to model an object fast. Then new vertices are placed where they should be and extrude the polygons to form new parts. Even if the box is simple at the beginning, complexity is added per introduced vertex. Hence, the final result is rather simple and does not look like a real car. Because of the complexity it is difficult to get a lifelike result and the disorder of vertices prevent from continued adding of new parts to the model.

3.3 Painting a model

To make a model look realistic, the parts of the model need to be painted in different colors. In 3D Studio Max, there is a material editor that helps the user choose color, select reflection, or decide on a texture for the model. The fastest and easiest way is to paint a part in only one color but this can make the object look too clean. For example, a car only looks perfect directly after it has been produced.

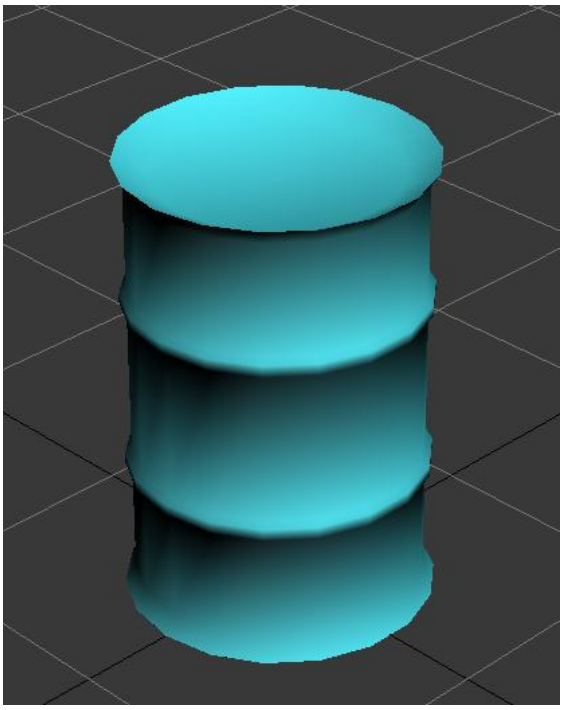

Figure 10: First the barrel has only one color and some shadow.

The best painting technique is to use a texture to cover the whole model. Texturing is like painting a picture on a paper and then wrapping it around the car (Heckbert, 1986). But when a model is wrapped, the paper gets all wrinkled, which does not look nice. An example of this can be seen in Fig. 12, where the top of the barrel is stretched, and the texture from Fig. 11 is distorted. Therefore, much time is spent on cutting this paper to fit nicely on the model; this is called texture mapping (Heckbert, 1986). To do this in 3D Studio Max, the tool modifier Unwrap UVW can be used. With this tool, the various parts of the object can be spread out on the texture and then resized, moved or transformed so that 3D Studio Max is able to cut the texture along the green wires in Figure 13.

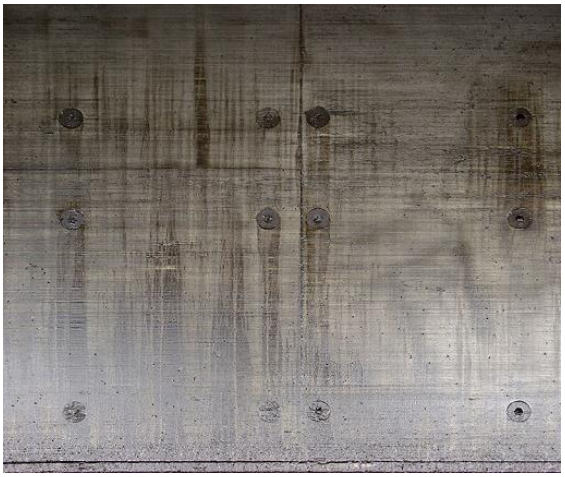

Figure 11: Barrel texture (CGTextures, 2011).

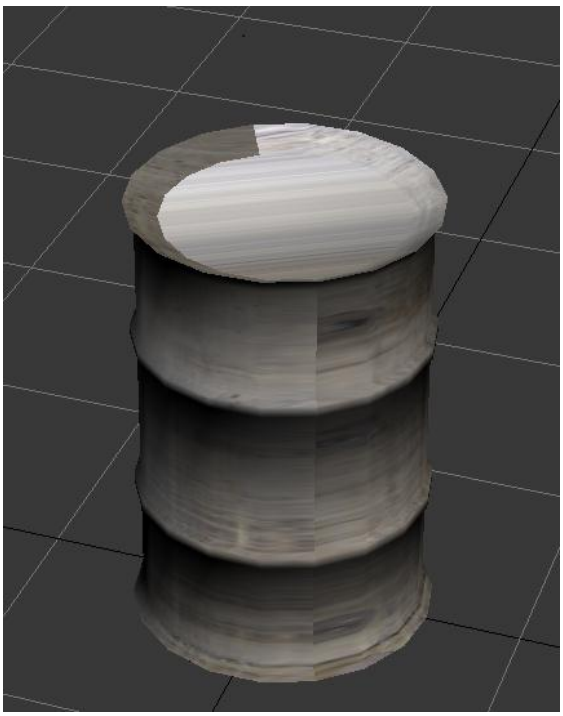

Figure 12: Barrel before the unwrapping, stretched and distorted texture.

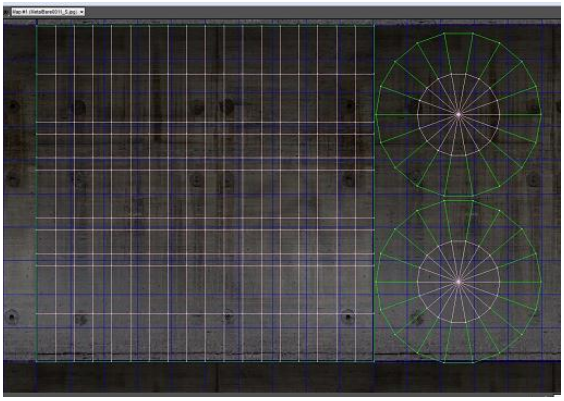

Figure 13: Unwrapping the surfaces of the barrel on to the texture. Here the barrel is represented as green and pink wires.

Figure 14: The barrel after the unwrapping.

This helps explaining to 3D Studio Max how to take a part on the texture and fit it on the car so it does not get stretched out or wrinkly. Something that gets stretched out easily when modeling for a racing game is the race track, because it has to be very large. Graphics cards have a limit on how large a texture can be, and even the largest texture allowed would not be enough to cover the landscape without

being stretched out.

3.4 Modeling the world

A big part of modeling is the tracks to drive on, and it is one of the most important parts in a game. There are several challenges when it comes to modeling a landscape, one of them is to simplify the real world to fit in a model and still keep the realistic appearance. As a game developer, the simplicity and the realism has to be balanced to keep the simulation look realistic and still avoid a low frame rate.

There are two common ways to create a landscape, for instance, by using a height map (Finney, 2004). A height map is a 2 dimensional grayscale picture that maps directly to the surface of the landscape where the brightness of each pixel represents the height of that area in the landscape. Downside with this technique is the resolution on the height map. For example, if a game developer wants to create a landscape that is 1 square kilometer and the developer wants to represent every square meter with one pixel, then a height map with 1000 x 1000 pixels is needed. With 8 bits per pixel the result is a height map at 1 MB, which is large for a small landscape.

Figure 15: An example height map used for creating the alley in Road Kill

Another way to proceed is to use 3D Studio Max to create the landscape with much flexibility to do changes. With this approach, all the modifiers in 3D Studio Max will be available for the developer's advantage and the height map technique can even be used as one of the tools. While this is a much more fine tuning way to go by, it is inefficient and complex and, when it comes to large landscapes, too heavy for 3D Studio Max because the number of vertices is too many for the program to handle and modify.

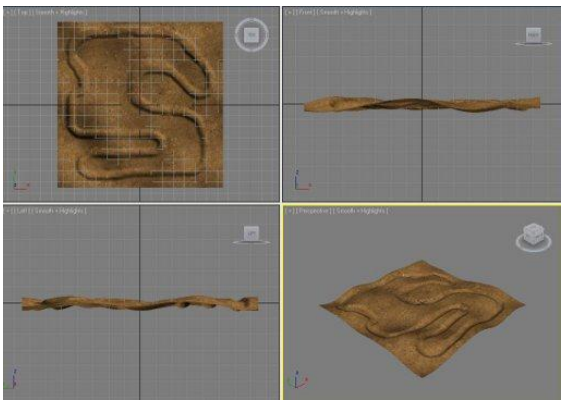

Figure 16: Example track in 3D Studio Max.

3.5 Results

For Road Kill, a large set of models were created. These included: cars with weapons, plant and a landscape. The different techniques, discussed earlier in Section 3.2, were all used in the modeling of Road Kills different objects.

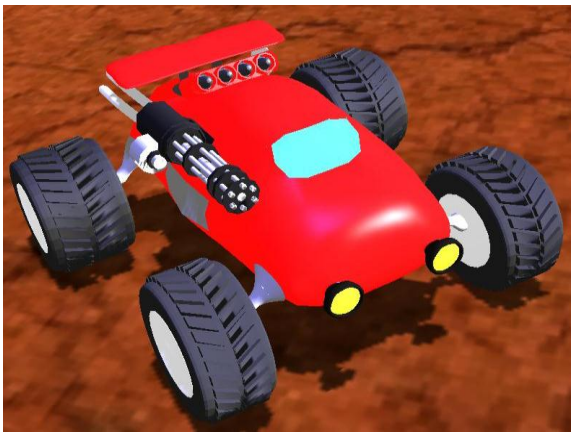

Figure 17: The rendered go-kart car. Modeled with the extrude technique.

Figure 18: The rendered monster truck. Modeled with the box technique

Figure 19: The rendered truck modeled with the box technique

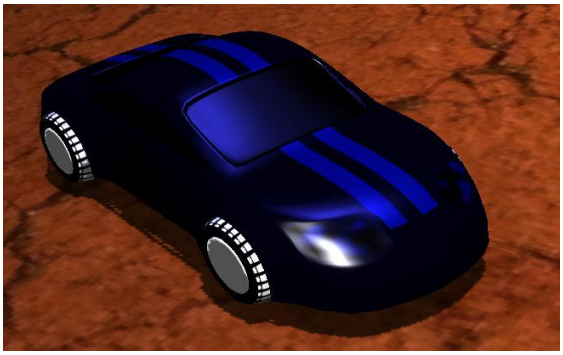

Figure 20: The rendered speedster car. Modeled with the polygon technique

3.6 Discussion

One decision that was made early in the study was that we wanted to make all of the objects ourselves, to make sure that Road Kill was so solely owned by the members of the thesis group to make a possible commercialization of Road Kill easier. This made modeling a time-consuming part of the study. Initially we did not plan for the modeling group to model the whole duration of the study. Due to the amount of time needed to create as many unique objects that are in Road Kill, a decision was made to continue modeling. The original plan was deviated from to solve the main thesis problems, as discussed in Section 1.2.

The modeling group decided to use 3D Studio Max to create the objects in Road Kill, Other alternatives that were taken into consideration were Blender (Blender Foundation, 2011) and Maya (Autodesk Inc., 2011b). Due to the fact that none of the members had any previous experience of modeling, we wanted to use the most beginner-friendly software. Based on our first impressions, 3D Studio Max was the best alternative for this purpose. As we were limited in time we were unable to study different modeling software

extensively. Therefore, there is a good possibility that we did not spend enough time using the other alternatives to fully appreciate their characteristics. One thing that has to be taken into consideration here is that as students, we had access to a student license of 3D Studio Max. If we had not been able to, we would have chosen Blender as it is open source, and therefore free to use,

The 3D Studio Max community is strong enough, with over 20 000 topics in the discussion groups (Autodesk Inc., 2011c), to answer any problems a beginner could have. Because of this, finding help on the internet was easy and it was a good help.
Having beginner-friendly software was important for the development of Road Kill, because it was significant that we started with the modeling process as early as possible. The plan was to learn to use the software during the first two weeks of the study and then start modeling objects for Road Kill. As we found out, there are many things to learn about modeling and it would take far more than two weeks to get a grasp of everything. Instead we learned most of the more advanced features, for instance adding textures using UVW-unwrapping, as discussed in section 3.3, during Road Kill's development process. The process of modeling for Road Kill was not a linear one, since the modeling group found themselves returning to old models and improving them with new proficiency.

One of the problems we faced during the first weeks of the study was when we loaded our objects into Road Kill, as there had not been much consideration about the scale of the objects in 3D Studio Max, and this led to a problem. The scale problem was made obvious in the Road Kill environment, by objects not behaving as expected. To illustrate the scale problem, a car that is 20 meters tall will appear to be falling in unnaturally slow, compared to a normal sized car, with earth's normal gravitational pull of 9.82 m/s^2. In order to compensate for this, we could calculate our own values of these types of physical constants to fit the scale of our objects, which would take unnecessary time. Instead this was handled by trial and error rescaling of the models in 3D Studio Max until we found a size that worked correctly with our physics-library.

We also had some trouble with applying textures to our models at first. Later on we found out about the UVW-unwrapping feature in 3D Studio Max, which made it easier to scale, move and resize the textures to fit the objects.

4 Graphics

The role of graphics in games grows more and more important as the quality of computers' graphics hardware and players' demands increase. Better graphical effects in games can give players a better gaming experience and a stronger illusion of reality. This chapter will describe some basic properties of modern graphics hardware and some algorithms that can be utilized to create graphical effects in games.

4.1 Graphics Pipeline

Fixed and Programmable Graphics Pipelines

One of the most important features of graphics cards today is that they have a pipeline, the so called graphics pipeline, which consists of mainly three different steps: application, geometry and rasterizer.

The Application step is executed on the CPU and is responsible for creating graphical objects. This means that, in both fixed and programmable pipelines, the application step is programmable for the developer.

The Geometry step is responsible for, among other things, moving objects in the world, some lighting computations on triangles and projecting the world from 3D to 2D.

The rasterizer step finds out which pixels are inside each triangle and applies textures and colors among other things to the pixels.

A couple of years ago, graphics cards used a fixed graphics pipeline for rendering graphical scenes, which basically meant that programmers were very limited in the ways they could implement different graphical effects. The fixed graphics pipeline is not used almost at all in modern games and consoles, and Nintendo's Wii console that came in late 2006 almost certainly is the last console that is not using a programmable pipeline (Akenine-Möller, 2008).

A Programmable Graphics Pipeline differs from its fixed counterpart in the way that in addition to a programmable application stage, some parts of the geometry and rasterizer stages are also programmable for the developer. The programs written for these two stages are called vertex shaders and pixel shaders. A vertex shader handles parts of the geometry stage while the pixel shader handles parts of the rasterizer stage of the programmable graphics pipeline. A vertex shader program executes for every vertex that is rendered by the graphics card and a pixel shader for every pixel. This means that operations for every pixel or vertex are pretty simple to add.

Matrix operations

The main task of graphics cards is to perform matrix operations, and the shaders of modern graphics cards are so efficient and optimized for doing this that some matrix operations have turned into single instructions. Because it has been shown that graphics cards today are better suited for executing matrix operations than CPUs (Fan Wu, 2010), overall system performance can be improved by letting graphics cards handle those operations. This can be done by using, for example, NVIDIA's parallel computing architecture, CUDA (NVIDIA Corporation., 2011).

4.1 Lighting

Lighting in a scene is a crucial part of the visual experience and the absence of light in a 3D game makes a dramatic difference, as illustrated in Fig. 21 and 22.

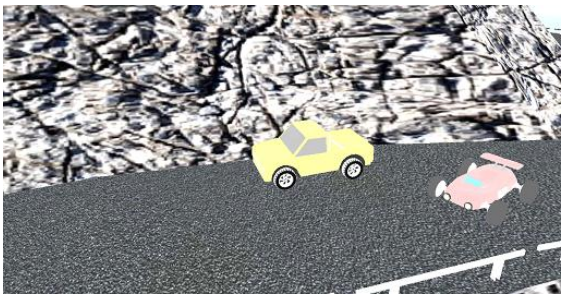

Figure 21: A scene rendered in a world without light sources. Each object is rendered with its original textures and colors, without the influence of lighting.

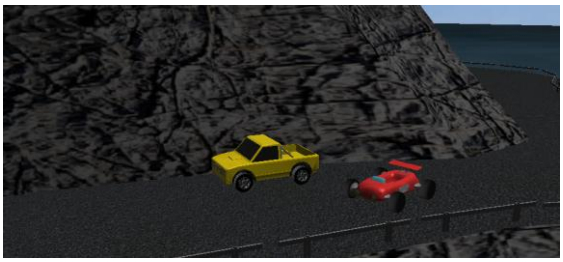

Figure 22: A scene similar to Fig. 1, but this time with a light source.

Shading

Shading is the process of using an equation to compute the lighting of an object based on the properties of surrounding light sources and material properties of the object's surface. There are three different shading models that are used for computing light in games: flat shading, Gouraud shading and Phong shading. In flat shading, the lighting is calculated per triangle, which might give objects an angular and artificial appearance. Gouraud shading **(**Gouraud, 1971) instead calculates light per vertex, which gives a better result than flat shading but still very unrealistic. Phong shading **(**Phong, 1975) calculates light per pixel, which is computationally expensive but will give a better end result than Gouraud shading.

There are numerous different kinds of equations used for shading calculations. Some of them consist of four different parts, which are called ambient, diffuse, specular and emission, where each part has its own color.

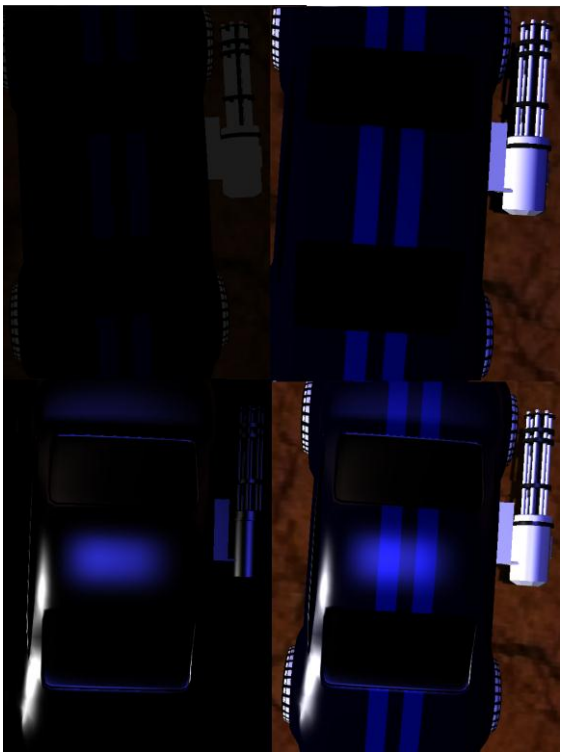

Figure 23: The car in the upper left corner has only the ambient part of the light The one in the upper right corner has only the diffuse part. The one in the lower left has only the specular part, and finally, the one in the lower right has all three of them. There is no emission part, since the car is not self-luminous.

The ambient part represents the light that is not coming directly from the light source but is reflected from other objects in the world. The diffuse part comes directly from the light source and is not reflected or highlighted but spread out evenly on the surface. An example of a very diffuse

material is a blackboard. The specular part represents highlights on reflective surfaces, and the emissive part represents the light coming from self-luminous objects. Fig. 23 shows a graphically rendered car with all lighting parts except emission. The color of the ambient light is usually a weighted sum of the diffuse and specular colors, but in some cases the ambient and diffuse colors are the same (Cook & Torrance, 1981) (Cook & Torrance, 1982)**.**

Types of light sources

The light sources that surround us in the real world illuminate objects in different ways. The sun, for example, is very far away, and illuminates our world uniformly. The light from a burning candle, on the other hand, spreads out evenly in all directions, and grows weaker when the distance to illuminated objects increases. Because real light sources behave in such different ways, realistic lighting in computer graphics applications is not efficiently approximated by a single algorithm. Instead, light sources are divided into categories, such as directional lights, omni lights, and spotlights; and implemented separately.

A directional light source does not have a position from where it emits light, but only a direction. An example of a directional light source from the real world is the sun: it is so far away that its rays can be considered parallel, and in local scale it illuminates the world uniformly. An illustration of directional light can be seen in Fig. 24.

Omni lights emit light equally in every direction, from a certain point in the world. The light's intensity also fades with greater distance from the light source. A real world example of an omni light source is a light bulb that is small enough to be considered a point source. Fig. 25 shows an example of how omni light is spread evenly in every direction.

A spotlight is a light source that unlike omni lights is not emitting light equally in every direction. Instead, it emits light in a cone-shaped fashion, where the light intensity is greatest near the cone's axis. An example of a spotlight could be a car lamp. One way to implement a spotlight is to assign it a source point P, direction vector L and maximum angle θ, as shown in Fig. 26. The light intensity will decrease the larger the angle is between the vector L and the illuminated point, and when the angle exceeds the maximum angle θ; the point will be in complete darkness.

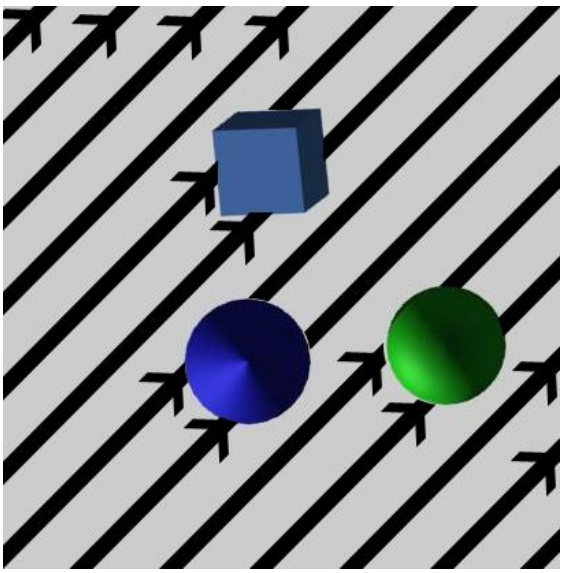

Figure 24: A scene lit by a directional light source.

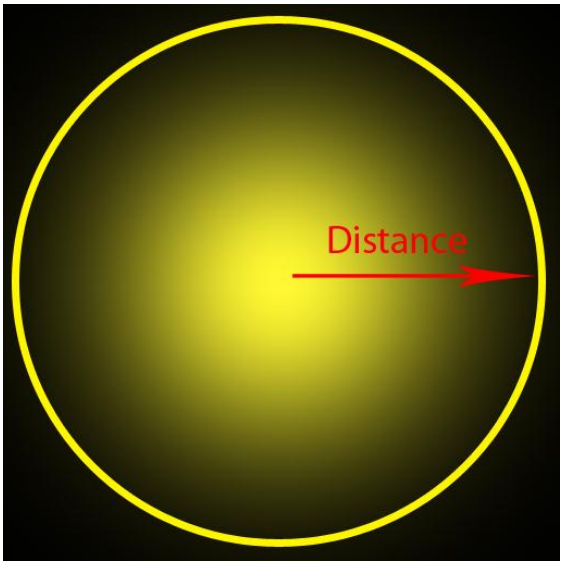

Figure 25: The top picture shows how an Omni light works. The light is faded out according to the distance from the source.

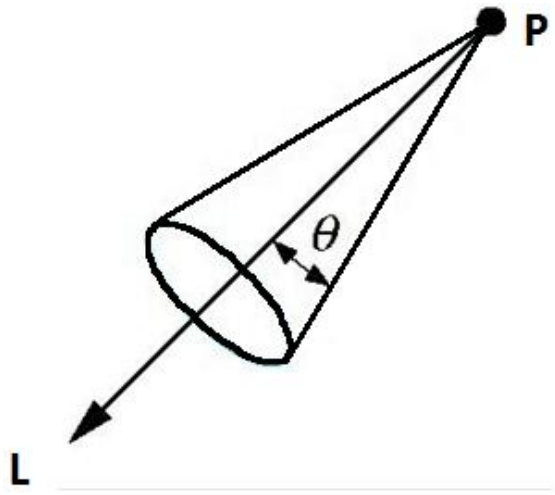

Figure 26: An illustration of a spotlight.

A mathematical formula for a more advanced algorithm for spotlights is shown in Fig. 27. This algorithm has two angles: θ_u and θ_p, which define two areas: an umbra and a penumbra. While inside the umbra, a point is lit by the maximum possible amount of light. When outside the umbra, the point becomes less lit depending on the angle θ_s between the point and the spotlight's direction vector. When the point is outside the penumbra, it is not lit at all by the spotlight. This algorithm gives a smoother transition from maximum light to complete darkness. An illustration of the algorithm is shown in Fig. 28.

$$L = \begin{bmatrix} if\ (\theta_s < \theta_p) & L_{Max} \\ if\ (\theta_p < \theta_s < \theta_u) & L_{Max} * \frac{\cos(\theta_s) - \cos(\theta_u)}{\cos(\theta_p) - \cos(\theta_u)}^{S_{exp}} \\ if\ (\theta_s < \theta_p) & 0 \end{bmatrix}$$

Figure 27: Spotlight algorithm with umbra and penumbra. L is final light intensity and L Max is provided with the spotlight and the other values are illustrated in Fig. 11.

Although all of the previously described algorithms are useful for graphics applications, they are still poor approximations of real light sources. Verbeck and Greenberg's article describes different measures of real light intensities and how to apply them to real-time rendering (Praun, 2001)

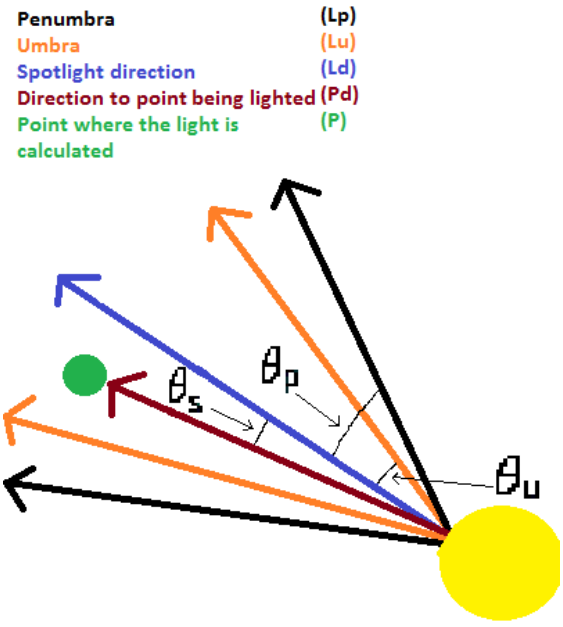

Figure 28: Spotlight algorithm with a penumbra. This algorithm gives a smoother transition between light and darkness than the one illustrated in Fig.

26. The three angles in the figure are the same as in Fig. 27.

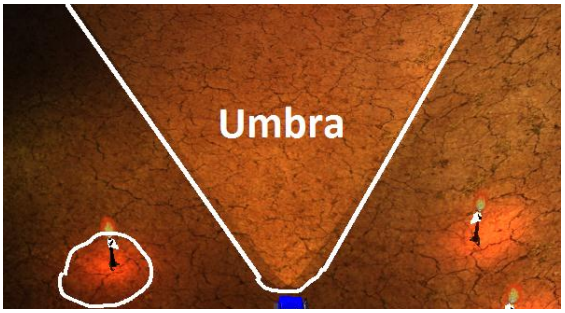

Figure 29: A picture from Road Kill where the fire has an omni light source and the car lamp has a spotlight with an umbra, but no penumbra. The umbra is the area where the light is, as illustrated inside the white lines.

4.1.1 Results

For the lighting in Road Kill, we used the Phong shading model together with omni lights and spotlights with only an umbra (see Fig. 26). Omni lights were used for the moon, fires and explosions; and spotlights were used for the streetlights and car lamps (see Fig. 29). The ambient light has the same color as the diffuse light for all light sources in Road Kill.

4.1.2 Discussion

We chose Phong as the shading model to use for Road Kill, with the reason that it should provide a better visual result than flat shading or Gouraud shading. Graphics cards today give developers the ability to program some parts of the graphics pipeline themselves (as mentioned in Section 4.1), and implementing Phong shading in the fragment shader proved to be fairly simple. The flat and Gouraud algorithms should be as easy to use, but the code must be written in the application layer for flat shading, and in the vertex shader for Gouraud shading.

The main disadvantage of having the shading calculations in the application or vertex shader, instead of in the fragment shader, is that the resulting rendered scenes would be of a lesser visual quality. Flat shading and Gouraud shading can still be good options, however. This is particularly true if a large amount of light sources are needed, since shading can be done many times faster with these methods than with Phong shading.

Despite the relatively high number of calculations done by Phong, we noticed that the graphics cards in our development computers could handle shading for all our light sources in a satisfactory way. Therefore, we decided to keep using Phong through the whole development process.

As mentioned in the results, we decided to approximate the moon as an omni light instead of a spotlight or directional light. The reason for this is that we wanted the moon to cast shadows, and therefore needed a position for the light source, as will be explained in Section 4.2. We did not use any fading for the moonlight since we wanted to illuminate the whole world uniformly. Since the distance between the moon and the Earth in Road Kill is so great, using a directional moonlight instead of an omni light would have made little visual difference in local areas, and would have been less computationally expensive. Therefore, it might have been more efficient to use a directional light source for the moonlight and add a specific additional point where the moon is and from where shadows should be calculated.

We decided to use omni lights for the fires and explosions in Road Kill, since they tend to spread light evenly in all directions and fade out according to distance. Some results can be seen in Fig. 12.

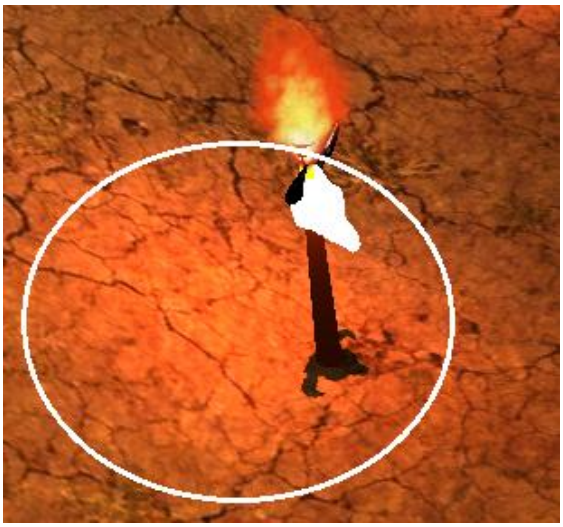

Figure 30: Omni light from fire

For the streetlights and car lights, we used spotlights with only an umbra (see Fig. 26). An in-game picture of the spotlight algorithm depicted in Fig. 26 is shown in Fig. 31.

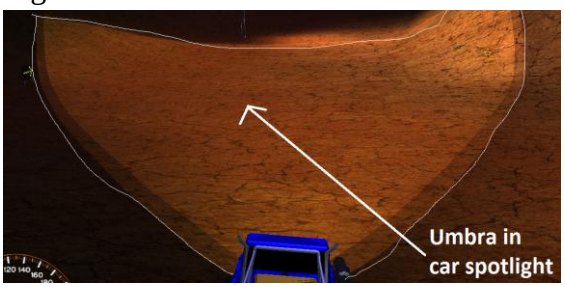

Figure 31: Car lamp approximated as a spotlight with only an umbra, as illustrated in Fig. 26. The white lines roughly show where the light has faded out completely.

4.2 Shadows

While lighting does improve the visual appearance of a scene, without shadows the scene will still seem unrealistic. Fig. 32 and 33 show the difference between a scene rendered with and without shadows.

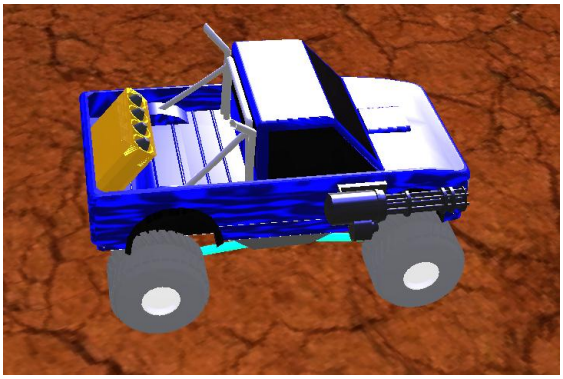

Figure 32: A car from Road Kill on the ground without any shadows.

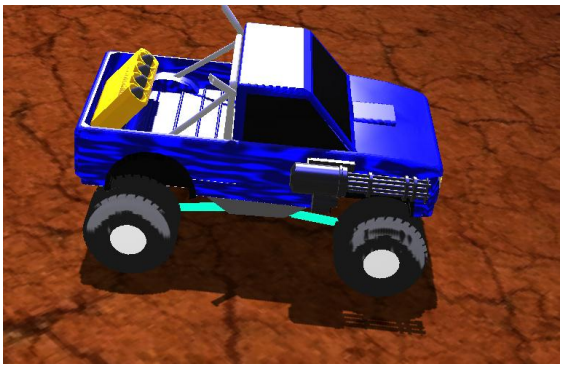

Figure 33: The same car on the same place as in Fig. 32, but this time with shadows.

The purpose of a shadow algorithm is to decide for every pixel if it is in shadow or not. There are mainly two different shadow algorithms for real-time rendering: shadow maps and shadow volumes.

4.2.1 Shadow Maps

In 1978 Williams introduced the Shadow Map technique (Williams, 1978). The algorithm works by rendering the scene with the camera's position equal to the position of the light source. Every pixel that can be seen by the camera should then be illuminated by the light source, and all other pixels should be in shadow.

Before the scene is rendered, the position of each pixel seen by the light source is transformed into a coordinate system

where the light source lies in the origin. The scene is the rendered from the perspective of this coordinate system, and the depth of each rendered pixel (i.e. the distance between pixel and light source of each pixel that can be seen by that light source) is stored in a so called depth buffer. The elements of a depth buffer that is used for shadow calculations are called shadow map samples.

When shadows are calculated in the world, a comparison is made between every pixel's depth–as seen through the light source's coordinate system–and the corresponding shadow map sample's depth in the depth buffer. If the depth of the pixel is the same as the depth of the sample, the pixel is seen by the light source and should be lighted. If the depth of the pixel is greater than the depth of the sample, however, another object must have obstructed the pixel, and therefore, the pixel should be in shadow (see Fig. 34).

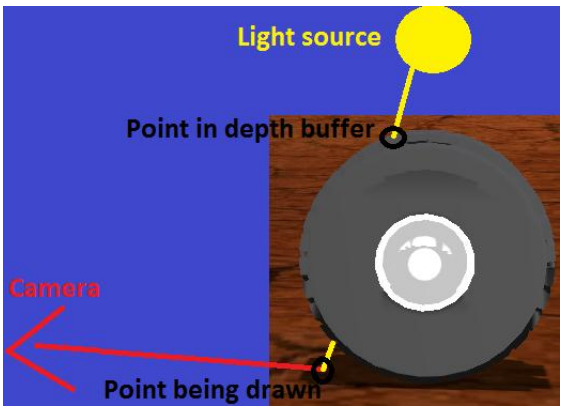

Figure 34: Shadows are calculated for a pixel below the car tire. The depth of the pixel is first compared against the depth of the corresponding shadow map sample in the depth buffer. Since the depth of the pixel is less, the pixel cannot be seen from the light source, and is therefore rendered in shadow.

The shadow map technique is a relatively efficient method. Using shadow maps will reduce the overall performance of rendering a scene by a constant factor equal to the number of shadow casting light sources. This is because a scene will always be rendered at least once per frame, and the creation of one shadow map for each light source leads to the scene being rendered one additional time per light source. There are, however, two problems with shadow maps that need to be considered during their implementation. Both of these problems originate from the fact that the sizes, or resolutions, of depth buffers are limited. Because of this, a shadow map sample may correspond to multiple pixels in the world.

Perspective Aliasing problem

Because more than one pixel are represented by the same shadow map sample, either all of those pixels will be shadowed, or none of them. Because shadows are cast on pixel groups, instead of individual pixels, the shadow image will be aliased and pixelated. This phenomenon is called Perspective Aliasing (Scherzer & Drettakis, 2005) (Stamminger & Drettakis, 2002). There is no clear solution to this problem except to limit the view of the shadow map (King, 2004), but doing so will decrease the size of the area where shadows are cast.

Bias problem

Because of the way nearby pixels are grouped together before their depths are stored in the depth buffer, the depth of a pixel might be slightly different than the depth of its corresponding shadow map sample, even if the pixel is actually seen by the light source, i.e. not obstructed. The effect of this is that some pixels that should

be lighted are instead cast in shadow (see Fig. 35). This problem is called the bias problem, or self-shadowing. An example of self-shadowing in a scene can be seen in Fig. 36.

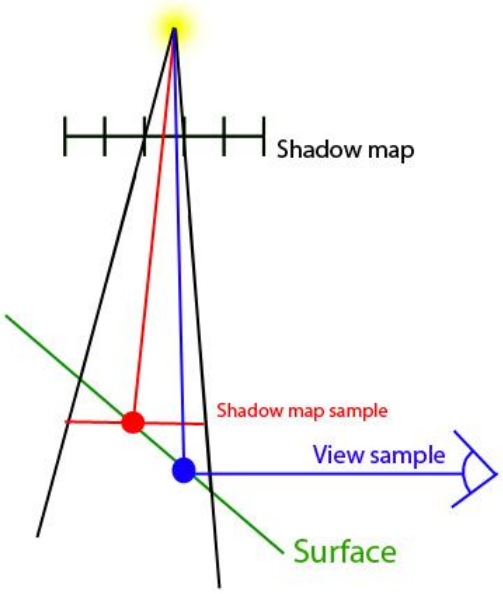

Figure 35: All the pixels above or under the red line will have the same depth value in the depth-buffer. The pixel in the blue sphere will be self-shadowed because its depth is greater than the corresponding shadow map sample in the depth-buffer.

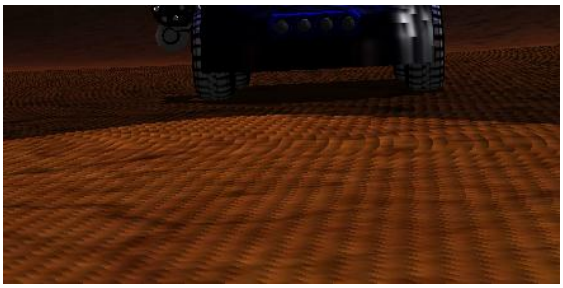

Figure 36: all the small black dots and lines are results of self-shadowing.

A solution to the self-shadowing problem is to add a so called bias (Shüler, 2005). A bias is a constant value (Lengyel, 2000) that determines a region where shadowing is prohibited. Having a bias stops pixels from casting shadows on nearby pixels that are within this region.

4.2.2 Shadow Volumes

Shadow Volumes was introduced by Heidmann in 1991 (Heidmann, 1991). In one version of the Shadow Volume technique, a volume is created for every triangle. The volume is defined as the region that is obscured from the light source by the triangle. Fig. 37 illustrates such a volume.

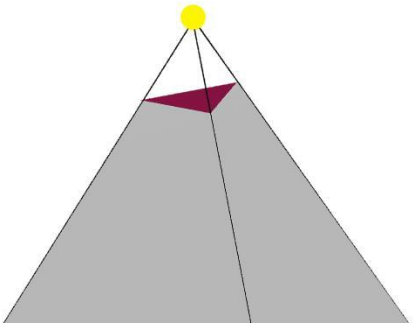

Figure 37: A Shadow Volume created for a purple triangle.

A pixel that is inside one or more shadow volumes will be rendered in shadow.

Shadow volumes from multiple triangles can be merged into so called silhouette edges. Doing this reduces the total number of shadow volumes, and therefore, the number of calculations made when shadowing objects.

The main advantage of shadow volumes is the sharpness of the resulting shadows. Disadvantages include having to create the numerous volumes and testing whether objects are located inside them or not.

4.2.3 Results and discussion

We have chosen to use the Shadow Map technique as the shadow algorithm for Road Kill. Because we had some previous experience with shadow maps, we knew that the technique would be relatively easy for us to implement quickly. By implementing a shadow algorithm as quickly as possible, we could save development time for other kinds of game features. Another reason for using shadow maps was that we wanted a way to render shadows as quickly and efficiently as possible.

In Road Kill, the moon is the only light source that casts shadows. We have reduced the effects of the resolution problem by only rendering shadows within a constant distance from the player's own car, but we still need a small bias to prevent self-shadowing.

Figure 38: Scene in Road Kill with shadows and enough bias to prevent self-shadowing.

Figure 39: The same scene as in Fig. 38, but this time with a lower bias, which leads to self-shadowing artifacts.

Finding the right balance between how much of the world should be affected by shadows, how high the bias should be, and which resolution the depth buffer should have, proved to be a delicate task.

Shadowing a greater part of the world, while keeping the bias and depth buffer resolution constant, would lead to an increase in size of all shadow map samples, which means that more pixels would be mapped to the same sample. This would lead to a greater amount of self-shadowing (see Fig. 39).

By having a too high bias, some pixels that should be in shadow may not be shadowed. If, for example, an object was obstructed by another object, but the distance between them was less than the minimum distance indicated by the bias, the obstructed object would not appear in shadow.

While increasing the depth buffer resolution would reduce the effects of self-shadowing, it would also reduce performance, since a higher number of shadow map samples would have to be

processed.

4.3 Particle Systems

Some objects are more dynamic than objects with rigid bodies (which are discussed in Section 3.2), and are therefore ill-suited for modeling. To create these graphical effects, like fire, smoke, and explosions, a particle system can be, and generally is, used (Reeves, 1983). To create a more compelling game, it is important to have these graphical effects due to the added liveliness.

There are some characteristics of particle systems that limit how well real world phenomena can be simulated, e.g. a dust cloud. The main problem is simply that it is not possible to imitate a real dust cloud with real-time graphics due to the complex physical dynamics and the huge number of particles. However, with the right techniques it is possible to still produce a visually pleasing result.

With two-dimensional particles, a developer is able to use a larger number of particles at the same time, improving the look of the graphical effect. This increase in particles is due to the much reduced complexity of drawing a two-dimensional object compared to a complex three-dimensional one. This leads to the problem of having two-dimensional particles in three-dimensional space – rotating around them would ruin the imitation of the real world phenomena.

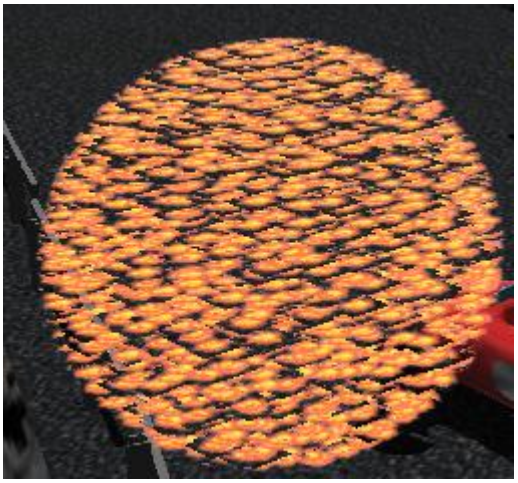

Figure 40: A collection of particles that do not stay perpendicular to the camera.

Using two-dimensional particles is preferable in real-time graphics, and to solve the rotation problem, a technique called billboarding is used.

4.3.1 Billboards

When a textured polygon is rotated based on the view direction, a technique called billboarding (McReynolds, T., Blythe, D., 2005) is used. To solve the rotation problem, which is discussed in Section 3.5, each object is always drawn perpendicular, by rotating them around their own axis, to the screen to hide the fact that they are two-dimensional.

Flattening of complex objects in to two-dimensions has been used extensively in real-time graphics, and the results are generally quite good as long as a distance is kept between the camera and the flattened object. As soon as the object is approached, it becomes obvious that it is not a real three-dimension object, especially if the object is only rotated around a single axis, like trees.

Billboards in particle engines

All particles in particle engines generally each have some properties, e.g. velocity, position, and whether or not they are alive and should be rendered. At a distance, this technique produces a good looking result as seen in Fig. 41.

Figure 41: A torch fire consisting of three different particle systems (each spawning a different particle color) and a total of 17000 particles.

With billboarding, the number of particles that are available can be increased by not using three-dimensional objects, and maximizing the number of particles that are available is critical when it comes to creating realistic fuzzy objects.

4.3.2 Soft particles

When a billboarded particle intersects a surface, a common artifact appears where the half intersected particle has an unnaturally sharp edge against the surface. These sharp edges are unwanted, because they ruin the graphical effect by acting in an unnatural way.

By implementing soft particles (T. Lorach, 2007), this artefact is avoided by fading the pixels, of the particle, near the intersection. As shown in Fig 42, the level of fading is correlated to the distance to the nearest object behind the particle.

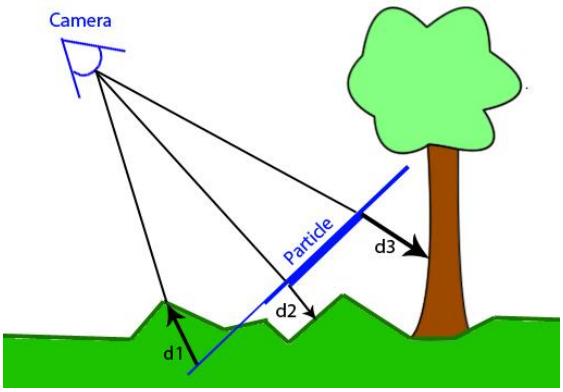

Fig 42: Here, the particle is faded depending on the distance to the nearest object behind the particle. The intersecting part, near d1, is under the ground and is therefore never drawn at all.

Even though this technique can improve the look of particle systems, especially if the particles of the system are quite large, it requires reading of the depth buffer, to get the distance between the particle and the object behind it. The reading operation can be quite costly if the buffer is large, depending on the resolution of the screen.

4.3.3 Results and discussion

Due to the limitation of memory access on the graphics card it is important to limit the number of particles in each effect. An example of the significance of this is that the torch shown in Fig. N was not included in Road Kill due to its high number of particles, despite its good look. So there was a large part in the development of the effects that was merely optimization.

Road Kill has 13 particle engines that

supply effects such as missile explosions, muzzle flashes from miniguns, gravel spray from the tires, and exhaust fumes. Fig. 43 and Fig. 44 illustrate a number of them.

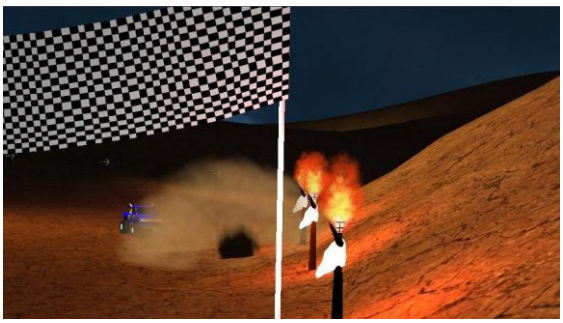

Figure 43: An illustration of the in-game explosion and torch particle effects of Road Kill.

Figure 44: An illustration of exhaust and gravel spray effects of Road Kill.

Because of the added memory accesses and the lowered frame rate due to soft particles, we decided not to use this technique because we felt that the artifact, which is discussed in Section 3.5.2, was never really that prominent. We also felt that we would benefit more from being able to use a higher amount particles compared to having fewer that were soft.

4.4 Culling

(Cohen-Or, 2003) Even today, with graphics cards that can draw incredible amounts of triangles very fast, rendering entire 3D worlds will still result in a bottleneck if the worlds are large enough. By only rendering the objects that will actually appear on the computer screen, a graphics card can avoid doing unnecessary calculations.

Culling is a method for filtering out objects in the world that will not be seen on the computer screen at a particular time. There are many different culling techniques that can be used, and which will be explained in this section.

4.4.1 View frustum culling

In 3D graphics, the view frustum is a volume that defines a camera's field of view. Only primitives that are inside the view frustum can be seen by the camera. The view frustum consists of six planes, as illustrated by Fig. 45.

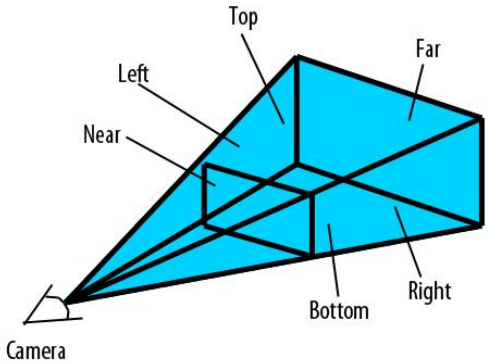

Figure 45: A view frustum defined by the six planes called Near, Far, Top, Bottom, Left and Right.

Since the primitives that are outside the view frustum will end up not being

displayed on the screen, it is unnecessary to send them to the graphics card at all. Therefore, they can be culled.

4.4.2 Backface culling

Another, very simple culling technique is called backface culling. It can be turned on in OpenGL with one command, and then, triangles that are facing the camera with their back side will not be sent to the graphics card for rendering. Fig. 46 illustrates two primitives which are back face culled.

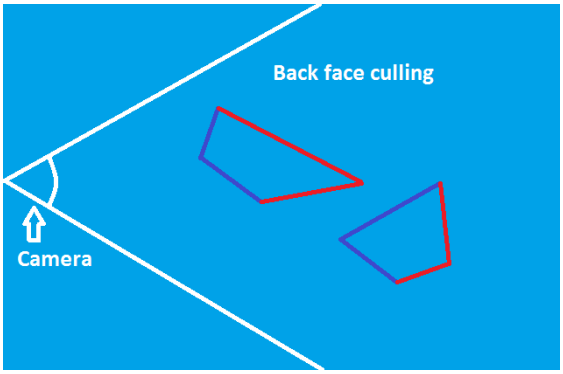

Figure 46: A view frustum with two primitives inside. The primitives are drawn after view frustum culling has been applied. The red lines mark the triangles that will be back face culled because they are facing the camera with their back side.

4.4.3 Occlusion culling

Even with both view frustum culling and backface culling enabled, objects that are hidden behind other objects would be drawn unnecessarily. For example, all the three spheres in Fig. 47 would be drawn, but only the leftmost one would be seen by the camera.

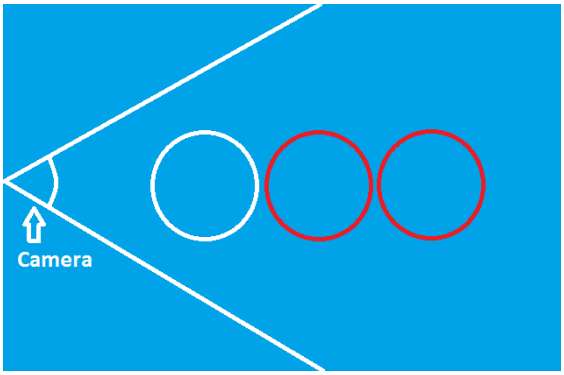

Figure 47: The two red spheres are hidden behind the white one, and will be unnecessary to draw.

Occlusion culling deals with this problem by filtering out primitives that are fully hidden behind other primitives. Nevertheless, most techniques for occlusion culling are inefficient, and the time taken to cull an occluded primitive is in most cases greater than the time saved by not rendering it.

4.4.4 Portal culling

The portal culling technique can be described as a mixture between occlusion culling and view frustum culling. When the camera is positioned inside a building or similar structure (such as the one in Fig. 48), it is possible to create multiple view frustums that perfectly fit door openings and similar gaps in the walls. The primitives that are located inside any of those view frustums can be seen by the camera, and are rendered as normal. The primitives that are outside all of the view frustums are completely hidden behind the building's walls, and can be culled by using the relatively efficient view frustum culling technique.

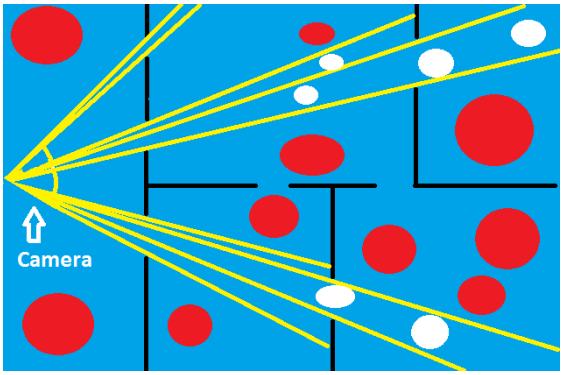

Figure 48: A view frustum has been created for each door opening. The red spheres are hidden behind the building's walls, and have been portal culled. The white spheres are visible by the camera, and will be drawn.

An algorithm for portal culling was first introduced by John M. Airey in 1990 (Airey et. al., 1990) (Airey, 1990). Since then, Teller and Sequin (Teller & Sequin, 1991) (Teller, 1992) and Teller and Hanrahan (Teller & Hanrahan, 1994) have created more efficient algorithms for portal culling.

4.4.5 Level of Detail

Just like in the real world, objects far away from the observer of a scene will appear smaller than objects that are close. When an object is far away enough, the viewer cannot tell whether the object consists of many vertices, or just a few. Rendering objects with their full amount of vertices is therefore unnecessary when the objects are sufficiently far away. Level of detail is a technique that utilizes this fact to reduce the number of vertices rendered in a scene.

To be compatible with the level of detail technique, a program must contain multiple versions of the same model. Each version of a model should be made up of a different number of vertices. When an object is about to be rendered to the screen, a suitable model version is picked based on the object's distance to the camera. For example, when the object is very close, the version with the highest number of vertices is rendered. Fig. 49 illustrates a situation where different versions of a cylinder are drawn depending on its distance to the camera.

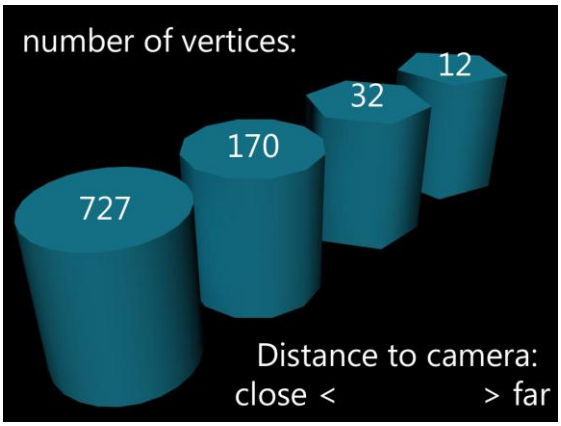

Figure 49: A cylinder is approximated by models with fewer vertices when the distance to the camera increases.

Some versions of the level of detail technique increase system performance even further. Objects that are sufficiently far away can be rendered as 2D images, also called imposters, instead of real 3D models (Maciel, 1995). If they are small enough, and also far away, some models may not be drawn at all. Memory can also be saved by using textures of lower quality for models that are further away (Cebenoyan, 2004).

4.4.6 Results and Discussion

Road Kill uses the relatively simple backface culling and view frustum culling techniques to increase performance during the rendering of scenes. Since these two techniques are both simple to implement and efficient to use, we recommend all 3D

game developers to use one or both of them in their games. The only reason for not using these two methods is the time needed to implement them. If, for example, a game is developed, where the number of objects rendered in the world is low, no performance increasing algorithms may be needed at all. In such a case, development time may be better spent implementing other features.

Occlusion culling was not implemented in Road Kill, because we decided that the time needed to find and implement an efficient algorithm was better spent on other game features and effects. The benefit of having occlusion culling would also be marginal, since, at the time of writing of this report, Road Kill consists mainly of open landscape, where few objects obstruct each other.

There are no indoor environments in Road Kill, but portal culling may still have been useful if applied to the road valley that makes up most of the race track (see Fig. 50). However, once again we decided that the development time required for implementing portal culling was better saved for other, more important features.

Figure 50: This picture illustrates that by treating cliffs as building walls, and the valley as a door opening, portal culling could have been implemented and used successfully in Road Kill.

Level of detail was not used in Road Kill, mainly because of the additional time needed to model different versions of objects. Even though we made only one version of each object in Road Kill, we still consider the amount of time that was spent modeling to be significant. This is further discussed in Chapter 10.

4.6 Reflections

Most models in a 3D world are made to approximate real-world objects. Some of those objects, such as mirrors, water pools, and flat, shiny metal sheets, are highly reflective. To render those objects in a realistic way (compare Fig. 51 and 52), a developer needs to implement a way to approximate and cast reflections.

Figure 51: A car without reflections looks too clean and unrealistic.

Figure 52: The same car as in Fig. 51, but with reflections added. This car does not look realistic either, but heavy reflections have been added to illustrate the difference.

Cube maps

To cast reflections in a satisfactory way, a reflection algorithm must be able to map an image of the surrounding world onto the surface of a reflective object. Such maps were introduced by Blinn and Newel (Blinn, 1977), and are called environment maps.

In real-time rendering, the cube map technique is the most widely used environment mapping technique today (Akenine-Möller, 2008). The technique was introduced by Greene in 1986 (Greene, 1986), and, as the name implies, it uses a cube as an environment map. The cube is first created around the reflective object. Images of the surrounding world are then projected and stored in each of the cube's six sides

To calculate the color of reflections in a pixel, a reflection vector is calculated from the pixel's normal vector and the eye vector, i.e. the distance vector from the camera to the pixel, according to the laws of optics. The reflection vector will point to a pixel in one of the cube's sides. The color of the reflection will be determined by the color of this pixel.

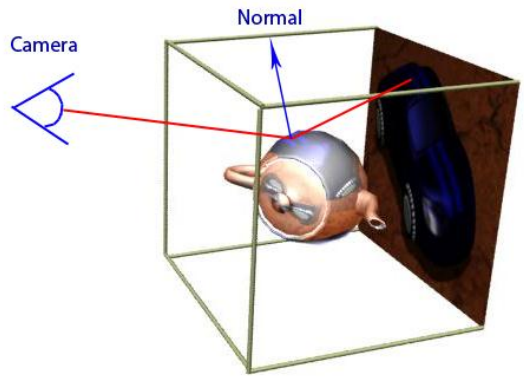

Figure 53: Reflections for a certain pixel in a teapot is calculated using cube maps. A reflection vector is calculated from the eye vector and the pixel's normal vector. The color of the reflection is determined by the pixel pointed to by the reflection vector.

Incorrect Reflections

Due to performance and memory limitations, a unique cube map cannot be created for every pixel. Since the reflections in multiple pixels are calculated from the same cube map, this method does not provide completely realistic shadows.

When the environment is projected onto the cube map, a 3D world is reduced to six 2D images. During this transformation, information about the depth of different parts of the world is lost. An improvement that can be made to cube maps and environment maps in general, is to add a depth value to each pixel stored in the map (Szirmay-Kalos et al, 2008). These depths can be used to create reflections of increased realism.

4.7 Texture Blending

When creating models in modeling software, such as 3D Studio Max, it is possible to apply several textures together on the same object. Directly placing different textures next to each other will, however, result in ugly transitions, as illustrated in Fig. 54.

Figure 54: Several textures are used without texture blending. A particularly ugly transition can be seen between the grass and mud textures.

One solution to this problem is to simply use only one texture. Doing this will introduce another problem: texture repeating. This results in a very unrealistic world, such as the one depicted in Fig. 55. Another solution is to blend textures together in the programmable graphics pipeline, as has been done in Fig. 56.

When blending two textures together, a grey scale map is used to determine how much of each original texture should cover each point of the resulting texture. A bright area of a grey scale map could, for example, correspond to a grassy texture, while darker areas correspond to a rocky texture. A grey area would then correspond to a texture which is both fairly grassy and fairly rocky.

Figure 55: A small part of a world with only one texture. This results in an artificial visual appearance due to tile repeating.

Figure 56: The same part of the world as in Fig. 55, but this time with texture blending between four textures.

4.7.1 Results and discussion

The world of Road Kill consists of a valley located in a desert landscape. Inside the valley lies the race track. The ground is covered by a combination of four textures: grass, dirt, sand, and cracked rock. The race track is mostly covered by the cracked rock texture. The open desert parts of the landscape and the walls of the valley are cover by the sand texture. Patches of grass and dirt are also scattered around the game world.

Texture blending is performed to create a more realistic and visually pleasing terrain. Since we have four different ground textures, we need an additional three grey scale textures for the blending. Because texture blending of this amount of textures takes a lot of time, we initially performed the blending only once, during Road Kill's loading phase. We did this by blending all textures onto a plane. This plane would in turn be used as one large texture tile to cover the whole game world. We soon discovered that a texture tile of that size would require a very large resolution in

order to provide a visually pleasing result. Such a high resolution proved to be impossible to use due to memory constraints. Therefore, we changed our program to perform blending each frame, which means that we no longer have to store any resulting textures between frames.

Some results of our final texture blending method are shown in Fig. 56.

4.8 Camera

The control of cameras in a game is just as important as the control of cameras in movies. The movement of the camera can influence how we think of the content in a game and enhance the game experience. Therefore it is important to implement a well-functioning and realistic camera. There are several different alternatives when it comes to observing a game. The two most common classifications are the first person perspective and the third person perspective.

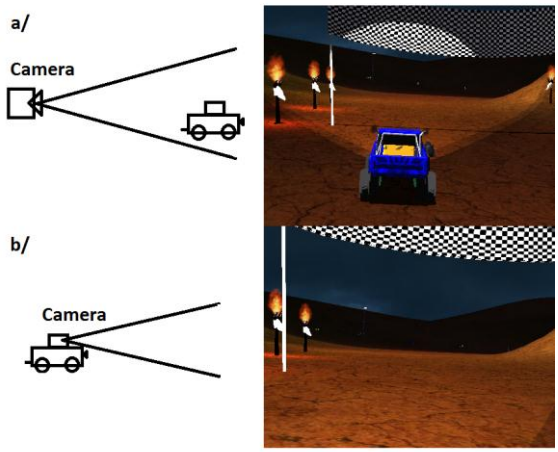

Figure 57: (a) Third person camera position and view. (b) First person camera position and view.

The first person perspective is implemented by setting the position and rotation for the camera to the position and rotation of the character. Ideally the camera position should be the same as for the player's character's eyes, but there can be a problem that the own character concealing the view of the player. To avoid this problem the character can be excluded from the rendering loop.

In the third person perspective, there are several options how the camera will move. The camera can be moved by the player regardless of the position of the character, which is usually used in strategy games, or the camera can move and rotate depending on the characters. The third person perspective is more expensive for the performance because of the need to check if the camera collides with other objects each frame. This collision detection is needed to avoid that the camera, for example, ends up on the wrong side of a wall, i.e. not the same side as the character. If the camera shall follow the characters movement and rotation even more calculations are needed calculate the position and rotation of the camera.
One way to achieve the effect that the camera is following the character is to get the position and rotation for the character. Then an offset is added to the position of the character in a direction depending on the rotation of the character and then make the camera to look at the character. This seems to work in theory, but in practice the camera becomes unsteady and unrealistic. A better way is to simulate that the camera is following the player attached to a spring with dampening. This will lead to smoother movement for the camera, much like a real camera in cinema productions with accelerating movement, but it is still

possible for the camera to end up inside other objects, and consequently there is a risk not being able to see the character at all times. A solution for that problem is to make the camera a part of the physics world but without rendering it so that it would stay invisible for all players. Now another problem arises, the camera can influence the other objects in the world and if an object comes in the way of the camera and prevents it from moving forward then, because of the spring attached to the character, the character will not be able to move forward either.

Another way to implement a camera, which follows the character with a smooth movement, is to calculate the distance to the desired new position but only move the camera a fraction of the total distance. This will make the camera move smoother and much like a damped spring.

A more sophisticated way to follow the player is to use an interactive camera that, like in cinema, focuses on important parts in the story and makes sure that the player does not miss anything by looking in the wrong direction. When an interactive camera is used, there are one or more preset routes that it will follow, depending on the area that the character is located in. In this way, there is more assurance that the camera will behave satisfactory. In this method, some colorful effects like zooming and rotating the camera can be used with good results.

4.8.1 Results

Road Kill uses a third person perspective camera that follows the car with the method of calculating the distance of the desired new position, but moving it a fraction of the distance for smoother movement.

To save some time, we decided to use the existing spring constraints in the Bullet physics engine to attach a slim cylinder reaching from the car and diagonally backwards. The camera was then positioned in the end of the cylinder looking at a point a little bit ahead the car. We used a cylinder to avoid getting stuck with the camera behind an object.

5 Physics engine

The physics engine handles the simulation of all physics for example collisions and applying forces to the objects.

Figure 58: The blue truck is driving right through the lamp post when the lamp post is not a part of the physics engine.

A physics engines tasks are to calculate if, where and when objects collide with each other and then respond to the collisions according to the laws of physics. To be able to achieve this, generally physics engines include two main parts, collision detection/collision response and dynamic simulation.

For each frame there are many tasks that need to be done and therefore each task needs to be done as fast as possible for a game to run smoothly. Because of the many complex calculations in the physics engine much time can be saved by applying various simplifications and speed-up techniques.

5.1 Detecting collisions between objects

As the name indicates, collision detection is responsible for detecting the collisions between different objects. The most difficult part with collision detection is not to calculate the actual collisions but to do this really fast. Collision response uses the collisions detected earlier and applies the corresponding forces to the objects involved.

As mentioned in the beginning of Section 5, the problem is not to detect the collisions but to do this within a very small time limit. The frame rate to strive for is roughly 60 frames per second (Watson & Luebke, 2005). This means that every frame has to be calculated in less than seventeen milliseconds.

As described in Section 3.2, every object consists of a collection of triangles. For example, one of the cars in Road Kill consists of 20 000 triangles as shown is Fig. 59.

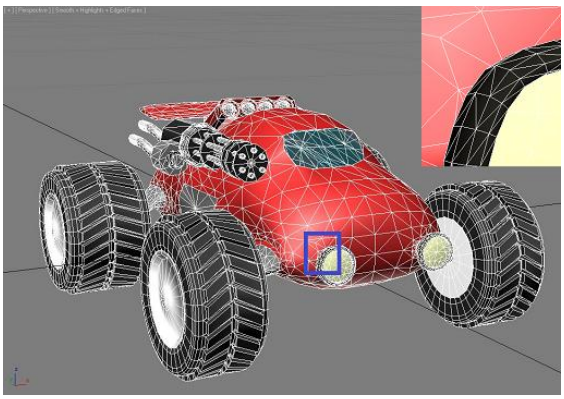

Figure 59: Illustration of how many triangles one of the cars in Road Kill contains.

With unlimited time it would be possible to actually calculate intersections between

every single triangle but in real time graphics this is not possible due to the time constraint, mentioned earlier, and the large amount of triangles in the objects that need collision detection. The solution to this problem there are lots of speed-up techniques.

5.1.1 Bounding volumes

Since it is not possible to check every pair of triangles for intersections every frame, the triangles need to be divided into geometric shapes, surrounding the triangles of an object. These shapes are called bounding volumes (BV) (Akenine-Möller, 2008). There are a few common BVs: bounding sphere, axis-aligned bounding box (AABB) oriented bounding box (OBB) and k-discrete oriented polytope (k-dop).

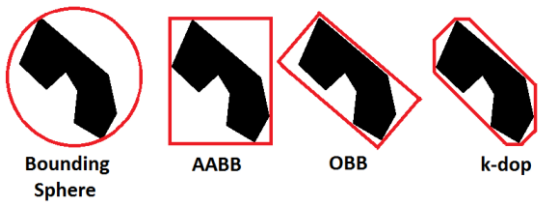

Figure 60: Examples of bounding volumes: Sphere, AABB, OBB and 8-dop.

There is no right or wrong when it comes to choosing BV because all of them have different trade-offs. Either the BV is tight around the triangles of the object, which demands a complex geometric shape, or a larger fit with a common geometric shape according to Fig. 60. If a BV has a small fit then fewer triangles will need to be tested for intersections since fewer BVs would intersect. On the other hand the complex geometric shape of the BV will cause the computation of the intersections to take longer time. If the BV is a simple geometric shape the complexities are reversed with a higher amount of intersections and faster computations instead.

Bounding Sphere

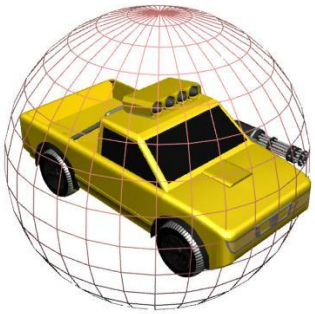

Figure 61: Car approximated with bounding sphere; the easiest BV to rotate and translate.

A bounding sphere is a bounding volume that surrounds all the triangles of an object. The advantages with bounding spheres are that they are easy to create (Ritter, 1990), translate, rotate and scale. These properties fit nicely for objects that are moved frequently. The drawback is that the volume inside the sphere is much larger than the encased object as illustrated in Fig. 61.

Axis-aligned bounding box

AABB is box that always stays aligned with the x-, y- and z-axis. This is an even easier BV to create than a bounding sphere.

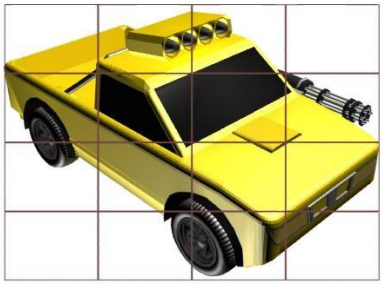

Figure 62: An AABB containing the vehicle object.

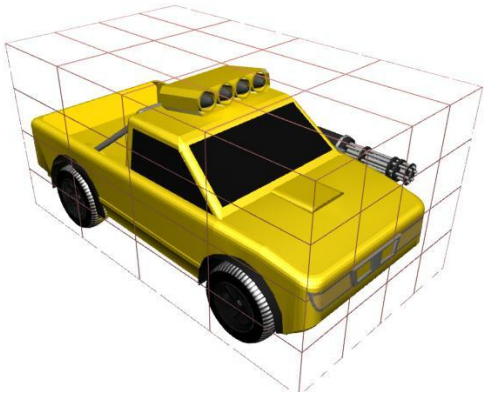

Figure 63: An OBB containing the vehicle object.

The drawback with an AABB is that it has to be recreated when rotated since it needs to stay axis-aligned. Although this is not that much of a problem since it is a very easy BV to create. The volume for AABB is less than the volume for a sphere so there are fewer intersections between the BVs. However the more complex shape for AABB requires more calculations when the intersection occurs.

Oriented bounding box

An OBB is a box that is rotated to best fit the objects within as seen in Fig.31. It is hard to create and to manage but do have a tight fit around the object. An OBB needs much memory to be stored and have rather expensive collision testing but because the tight fit there are not many OBB-intersections without actual triangle intersection. Because of the free rotation of the OBB there is no need to recalculate, just rotate it as the object.

K-discrete oriented polytope

A k-dop is defined by the tightest set of k/2 slabs. A slab is the volume between one pair of parallel planes as illustrated in Fig. 64 (Ericson, 2005). In Fig. 64 the black lines represent the planes that limit slab 1 and slab 2. If a third dimension were to be added, two additional slabs that are parallel to this paper are needed. In this three dimensional case it would be a 6-dop and if the slabs are axis-aligned it would be an AABB.

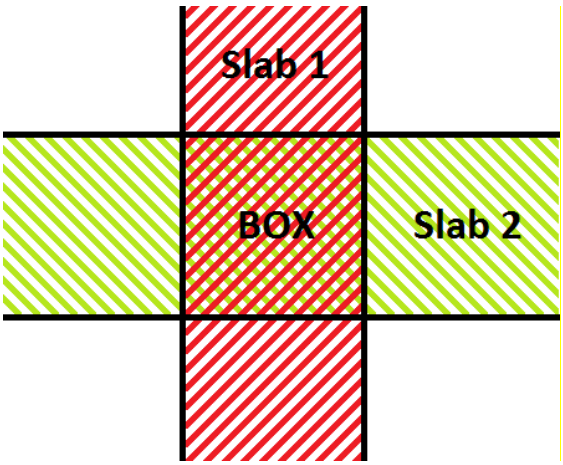

Figure 64: A box in two dimensions defined by two slabs. The black lines represent the planes that limit the two slabs.

Figure 65: A 10-dop containing the vehicle object. The number 10 in 10-dop means that there are 10 planes that limit the object. The last two planes are parallel to this paper and are not shown.

Convex hull

In mathematics, a convex hull shape is the shape that contains the points with a minimal convex set. A good metaphor to illustrate the convex hull in two dimensions is a large rubber band tightening around the object (see Fig. 66). In three dimensions, the object is inside an inflated balloon and the hull is minimized by letting the air out of the balloon (Barber, 1996).

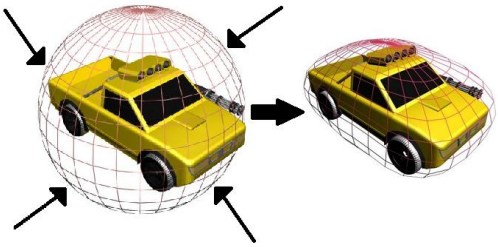

Figure 66: A metaphor for how a convex hull contains an object.

A convex hull is a complex shape that is hard to create and rotate but the big advantage with it is that the shape can enclose a generic convex object. Therefore it is very useful when, for example, the objects are created at run-time. If the object is not convex the enclosed area will be larger than wanted. A solution to this problem is to test whether it is convex or not and if it is the object can be decomposed into convex objects.

Triangle mesh

When all triangles of an object are stored one by one a translation needs to change three vertices for every triangle. If instead a mesh is used to store the triangles the common vertices that connect the triangles only need to be changed once (see Fig. 67).

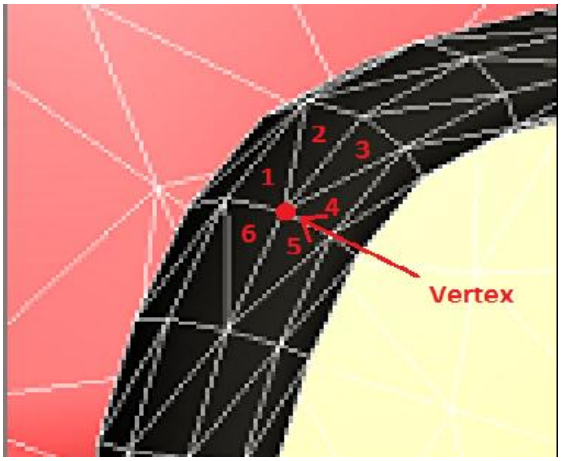

Figure 67: A triangle mesh where the red dot highlights a vertex that six triangles have in common. If the triangles where to be moved one by one instead of as a mesh then this vertex would be changed seven times instead of one.

If a triangle mesh is to be moved there are nevertheless many vertices needed to be calculated for a new position and therefore a triangle mesh is best suited for static objects.

An interesting feature for triangle meshes is that it can be deformed by changing the position of some vertices relatively the other vertices. Picture a fishing-net hanging in the air. If the net is pushed at one point the net surrounding that point will also be moved but not that much as the point itself. This will simulate

deformable objects very well if there are enough vertexes to make the dent look smooth. If the net has very few vertexes it will be as if the meshes are very large and the deformation will look angular and non-realistic.

Compound of BVs

Sometimes it is natural to group several objects into one big object, for example, a car has a chassis and four wheels. A compound is also useful when an object cannot be represented with an easy geometrical shape or is concave. The object can then be decomposed into several easy geometrical shapes or convex objects and bound together in a compound.

A decomposition of an object gives the opportunity to bind them with constraints. For example, the four wheels and the chassis are all different shapes. The wheels can now be attached to the chassis with constraints so the wheels are allowed to rotate around one axis.

5.1.2 Spatial data structures

Even with perfect bounding volumes it still takes too much time to calculate the intersection between all pairs of BVs every frame. A solution to this problem is to create data structures for the BVs.

According to Fig. 9 the structure divides the world in volumes, each containing BVs. By doing this we do not need to check collisions between every pair of BVs, only the ones that are in the same area of the spatial data structure.

Bounding volume hierarchy

The most common spatial data structure is bounding volume hierarchy (BVH) and is built like a tree data structure (Sweeney, 1999). The root of the tree is a BV that contains all other BVs. The children of a BV in the tree contain smaller BVs which was included in the parent. Therefore it is a hierarchy of BVs as illustrated in Fig. 9 (Cormen, 1990).

If all collisions between a BV A and the rest of the world are wanted, see Fig. 68 and 69, the first intersection test is computed between BV A and BV 1. Since they intersect intersection testing continues by testing A against BV 2 and 3. Here BV A only intersects BV 3 so all testing against the BVs in BV 2 can be discarded. The procedure is then continued by intersection testing BV A against BV 7 and 8. BV A intersects both 7 and 8 and therefore all BVs inside 7 and 8 needs to be checked. Further, BV A intersects BVs 9 and 11 and since they are leaf nodes in the hierarchy BV A needs to be tested against all primitives inside them. Finally the result is that A intersects the primitive inside BV 9.

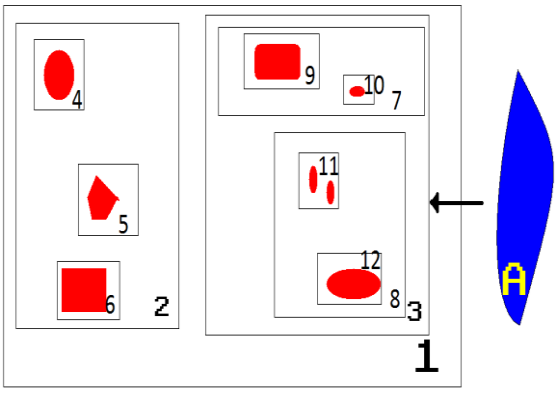

Figure 68: A world that is surrounded by BV 1 and the BV A that moves towards the world.

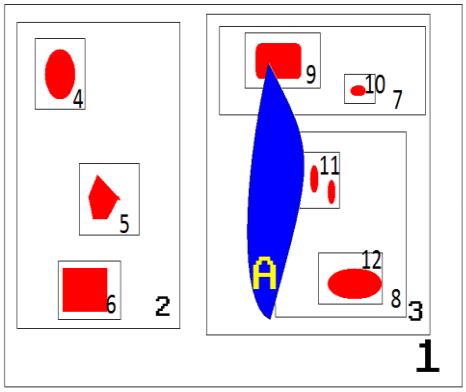

Figure 69: Same as Fig. 68 except that BV A now intersects the world.

When using BHV, the time complexity will be O(log n) (Akenine-Möller, 2008), where n is the number of BVs, tests instead of O(n) as it would have been without the BVH.

A BVH is either built top-down or bottom-up. If it is built top-down, first a BV that contains all other BVs is created and then split by a plane on the axis where the bounding volume is longest. The two new BV are then minimized so that they fit their BV children as tightly as possible. The procedure is then continued again until a stop criterion is fulfilled, for example two splits as shown in Fig. 70.

A tree built bottom-up starts with containing all objects in BVs and making them the leaf-nodes in the tree. The BVs will then be paired together, two or more, by some merging criterion and be enclosed in new BVs. The new BVs will be the parent nodes, to those BVs paired, in the tree. Those steps will be repeated until all objects are enclosed in one BV. The bottom-up build takes more time to create and is harder to implement, but usually produce better trees than the top-down build (Omohundro, 1989).

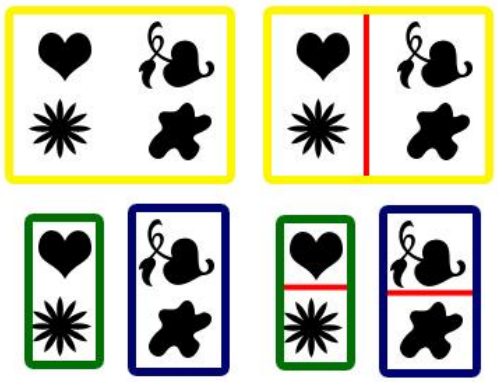

Figure 70: Example of creating a BVH using AABBs as bounding volumes. First an AABB is created to contain all objects. Then it is split by the red-dotted line on the x-axis because that side was the longest on the AABB. The new AABBs are minimized and then split again.

Binary space partitioning tree and Kd-tree

A Binary space partitioning tree (BSP-tree) (Samet, 1989b) is a binary tree where the parent node a box is split by a plane and which child they end up in depend upon which side they are of the plane. This is continued recursively until the stop criterion is reached, for example, two BVs in each child.

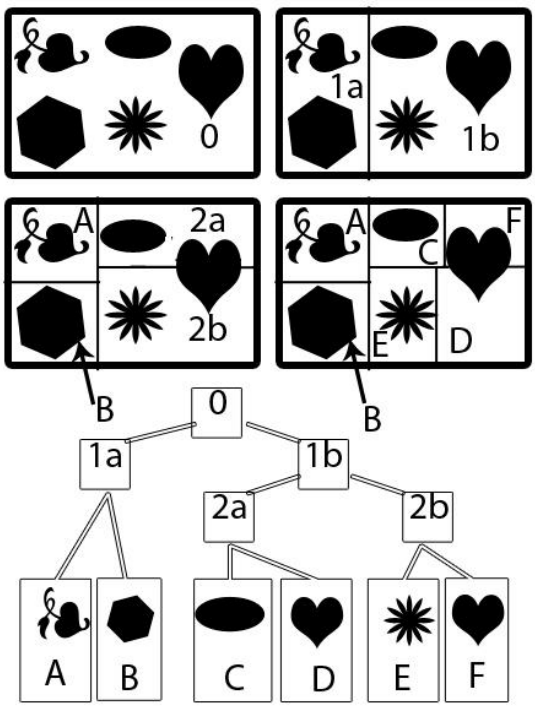

Figure 71: The picture illustrates how to make a BSP-tree by splitting the box with a plane recursively. Pay attention to the triangle ending up in both C and E because it is on both sides of the splitting-plane.

One problem that can occur when the splitting plane is poorly chosen is that all geometry is intersecting the plane and all geometry will be placed in both children, this is illustrated with the triangle that exists in both the C and E child in Fig. 10. This problem can lead to an infinite loop if the geometry continues to end up in both children and the end criterion never is met.

A nice feature of BVHs is that only one vector and one point for each plane needs to be saved in the graphics-memory and very little memory is therefore occupied.

The Kd-tree is very similar but has an advantage that it has fixed order of the splitting planes (Samet, 1989a). With the Kd-tree the only thing needed to be saved in the graphics-memory is a point. Usually BVHs are used for static scenes but may be used for dynamic as well.

Octree

An octree is similar to a BSP-tree. A box is split along all three axes by three planes that intersect each other at the center of the box. This creates eight new boxes, hence the name octree.

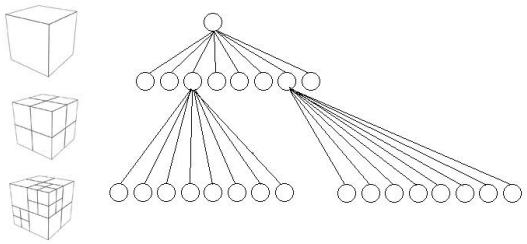

Figure 72: Example of an octree

The splitting is then recursively repeated until a stop criterion is fulfilled for example a maximum depth of the tree is reached or maximum amount of objects in the box (Samet, 1989a) (Samet, 1989b). A negative property of octrees is that objects often intersect several splitting planes and end up in several child boxes. A solution for this problem is called loose octrees (Ulrich, 2000). With loose octrees the risk for objects to end up in more than one box is decreased.

5.1.3 Intersection test

To this point no real collision detection has been discussed, only speed up techniques that will make it possible to detect intersections in real-time in a large scene. There are many ways to determine the intersections in a scene. There are mainly four different techniques: analytical,

geometrical, separating axis theorem and dynamic tests.

Analytical testing

Analytical testing calculates the intersection between two objects by inserting the mathematical representation, an equation, of one object into the other objects equation and then tries to solve the new equation. If there is a solution then the objects intersect in that point. Calculating these intersections in real-time is not an easy task because many intersection algorithms are very ineffective and they need to be improved significantly to meet the timing requirements of a real time game.

Geometrical testing

Geometrical testing is suitable for testing intersections between boxes and rays. To test intersections geometrically between a box and a ray, three slabs are needed. As mentioned earlier, a slab is defined as the volume between two parallel planes illustrated in Fig 73.

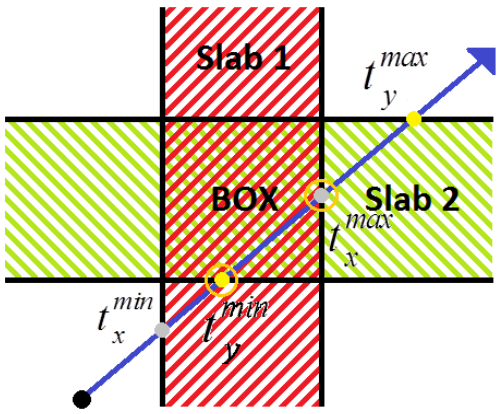

Figure 73: A ray intersecting the box in two dimensions.

The picture in Fig. 14 describes the algorithm. t_x^{min} denotes the time when the ray enters the vertical slab and t_x^{max} is the time when the ray exits the same slab. Similarly, t_y^{min} and t_y^{max} represent the same timing but with the horizontal slab instead of the vertical slab. If the later min value is less than the first max value an intersection has occurred, otherwise there is no intersection. The point for intersection is not given by this algorithm although it is rather simple to expand the algorithm to give the exact point. The point is given by inserting the time of intersection into the ray equation. In three dimensions three slabs are used instead of two. The reasoning stays the same.

Separating axis theorem

The separating axis theorem (SAT) is a very important tool for intersection testing (Gottschalk, 1996) (Greene, 1994). The SAT is built upon the fact that there can be no intersection between two convex primitives if there is a plane where the orthogonal projections of the primitives do not overlap. The only planes that really need to be tested against are the planes that are orthogonal any face of the two objects and the cross product between one edge from each primitive.

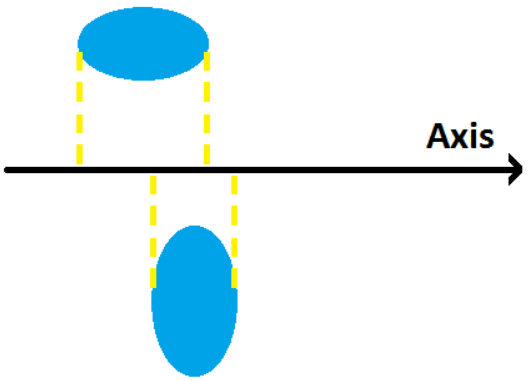

Figure 74: Two boxes projected to a plane where they overlap.

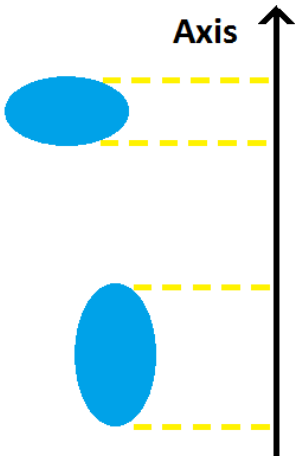

Figure 75: The same two boxes projected on to a plane where they do not overlap. Therefore the primitives do not intersect with each other.

Dynamic testing

All the intersection testing methods mentioned earlier have been static which means that the intersection testing is done on objects that do not move during testing. The problem with static testing is that if the object move to fast between two frames intersection can be missed. Dynamic testing is designed to solve this problem. Dynamic testing takes more time than static testing and is therefore seldom used in real time collision detection systems.

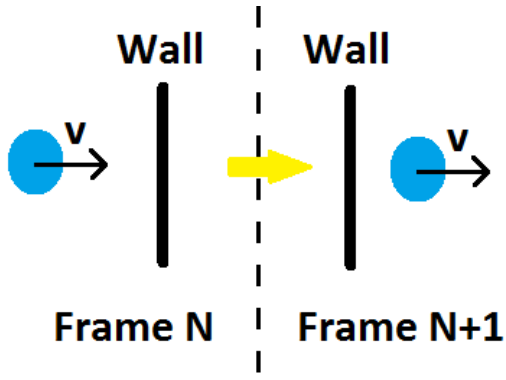

Figure 76: The sphere is going right through the wall. This is possible when static testing is used instead of dynamic testing.

5.2 Existing physics engines

If a game developer has limited resources, using an existing physics engine should be carefully considered. There are a few stable and capable open source alternatives, which can be a good alternative to proprietary physics engines when resources are limited. Even though the open source physics engines do not come with any guarantee or support, as with commercial physics engines usually do, they generally have a high standard and a big helpful community so it can be a satisfactory option even for projects with large budgets.

Bullet Physics Library

Bullet (Game Physics Simulation, 2011) is an open source 3D game multiphysics library, available under the ZLib license (Roelofs et al, 2005), which provides collision detection, soft body and rigid body dynamics. It is possible to use only the collision detection library and apply other dynamics for custom fit. Both discrete and continuous collision detection are supported in Bullet which suit software with both precision and high speed body requirements.

The BVs that are included in Bullet are several simple geometrical objects as boxes and cones. Bullet also supplies generic convex hull and triangle mesh. Additionally Bullet has limited constraints for rigid bodies, such as ball-socket and hinge constraint, and deformation of non-convex triangle meshes.

Open Dynamic Engine (ODE)

ODE (Smith, 2006) is an engine designed for stability, robustness and to be fast

rather than supplying perfect physical accuracy in a simulation. Therefore ODE is best suited for real-time simulations and it is good for simulating articulated rigid bodies such as vehicles and legged creatures.

ODE uses hard contacts instead of virtual springs to respond to contacts. This means that there will be no penetration between objects and the error-prone virtual spring alternative is not needed. A virtual spring system is simply explained a representation of objects as a set of point masses connected with weightless springs. This method is popular when handling deformable objects.

Built-in collision detection is included in ODE but it can be replaced by external collision detection libraries. The collision primitives included in ODE are sphere, box, cylinder, capsule, plane, ray, triangle mesh and convex hull.

ODE is released as free software and can be redistributed or modified according to either GNU Lesser General Public License (Free Software Foundation, Inc., 2010) or BSD-style license (Open Source Initiative, 2011).

5.3 Results

We decided to use Bullet as the physics engine in Road Kill and to use the built-in vehicle class. The BVs we use for the dynamic objects are primarily convex hull. All static objects are a part of the world and are represented as a static triangle mesh.

5.4 Discussion

When we first started the project one goal was to create our own physics engine. We started to use sphere as BVs and used a BSP-tree as spatial data structure. At first all objects only had one BV but the performance was too poor, mainly because the many triangle-triangle intersection tests needed, so we decided to use BSP-tree as an inner spatial data structure too.

We realized after a few weeks that the physics engine would take a long time to implement and very hard to get the precision and performance wanted so we decided to use an existing physics engine instead. We wanted to have the possibility to commercialize Road Kill and because we were doing research on game development with limited resources we decided to use an open source physics engine with no restrictions on commercializing.

The two physics engines we found that fulfilled our demands and had been used in successful game-projects before were Bullet and ODE.

We decided to use Bullet because it has more features and also a pretty good built-in vehicle class. Overall we were satisfied with Bullet and it performs very well. One negative aspect of Bullet was the lacking documentation which caused some trouble a number of times during the process of Road Kill development.

6 Network and Multiplayer

When it comes to supplying challenging opponents in a game, there are only two real choices, either a computer controlled opponent with artificial intelligence (AI) is developed or communication over a network is implemented so that multiple players can interact in one game session. Multiplayer has a number of advantages over developing an artificial intelligence, mainly the two following; it is easier to implement and gives players more varied and unpredictable opponents. According to Ookla (Ookla, 2011) and Internet World Stats (Miniwatts, 2010), bandwidth and coverage of the internet has kept increasing all over the world, enabling game developers to focus more on multiplayer aspects in their games. One important question that game developers should ask themselves early during the design of a game is if and how to support multiplayer.

This section will, using a bottom-up approach, examine some key points when it comes to implementing network communication and multiplayer support. Some of the examination's results are general to any type of application, but the majority is game development specific.

6.1 Choosing the right Network Model

A key point to realize is that, different types of games have different network demands. Fast paced real-time games like first person shooters and racing games require data to be delivered between players with minimal time delay, also known as latency, to give the most realistic and enjoyable experience. While large scale turn-based, or slow paced real-time games have no such problems, they often need to transfer greater amounts of game data, and thus require a higher bandwidth instead.

To choose the right network model for their games, developers need to consider, among other things: the desired number of supported players, minimum tolerable response time between user input and action, the amount of data needed to be sent between players to update the game state and how often such updates need to occur.

This chapter will present a comparison of the two main transport layer protocols, in Section 6.1.1, and an examination of the different network topologies that can be used for network communication.

6.1.1 Transport Layer Protocols

One of the main influences on how a multiplayer game behaves is the choice of protocol and, as mentioned before, different games have different demands.

Transmission Control Protocol (TCP) is a connection-based protocol that is used when it is important that all data is not only delivered but also received in order and without any duplicate data. E-mail, file transfers and the World Wide Web rely on TCP since it supplies the features that these applications require. All TCP features have resulted in a large header, 20 bytes.

bits	0-15		16-31
0	Source Port Number		Destination Port Number
32	Sequence number		
64	Acknowledgement number		
96	Data Offset and reserved bits	Flags	Window Size
128	Checksum		Urgent pointer
160	Options		

Figure 77: A table of the TCP-header. Field in blue is optional. (Postel, 1981b)

User Datagram Protocol (UDP) has no focus on any of the aforementioned features and it is connection less. If there is any need for any of TCP-supplied features, developers have to implement them themselves. The upside to this is that UDP has less overhead as its header is only 8 bytes.

bits	0-15	16-31
0	Source Port Number	Destination Port Number
32	Length	Checksum

Figure 78: A table of the UDP-header (Postel, 1980).

For some developers, the choice might seem obvious. TCP is easier to implement and it supplies a number of useful features, but there are several drawbacks (Fiedler, G, 2006) to using TCP. Whenever a packet is lost, it is resent by TCP, and this incurs a delay in the game. When a packet is lost, there is no real need to resend the packet since a more current packet will have arrived, before the old packet is successfully resent, causing the first packet to become old and irrelevant. This is, however, only relevant to fast paced games where data is sent continually. In for example turn-based games, it would be easier to use TCP and the game would not suffer noticeably.

Critical data that is not sent continually will need a guarantee that it has been received successfully, and it might seem like TCP is the better choice again. But it is actually better to implement an custom own packet delivery guarantee on top of UDP, instead of mixing the two, as it has been shown that TCP can induce packet loss in UDP (Hori, 1998). As long as all of TCP's extra features are not needed, the extra overhead is best avoided.

6.1.2 Network topology

Beyond protocols, it is also important to consider how the flow of traffic should be shaped. Since a great deal of focus in real-time games lies in minimizing latency the next two sections will explore, among other things, the different latency impacts of the two most common network topologies used in game networking.

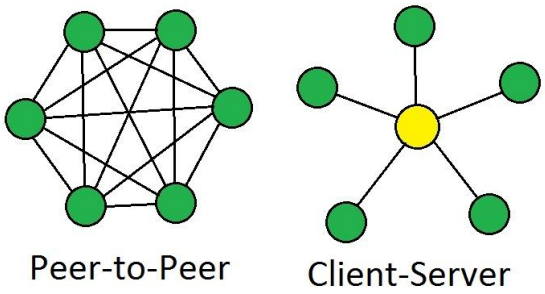

Figure 79: An illustration of the peer-to-peer and client-server topologies where each green node is a user and the yellow node is a server.

6.1.2.1 Peer-to-Peer

When it comes to reducing latency, using the peer-to-peer technique of connecting each client directly to every other client is

the best solution, since all data takes the most direct route to its goal. This topology was used early in online gaming in action games such as Duke Nukem and Doom (Sweeney, 1999), and even though it has been largely abandoned today, it is still a valid option when it comes to choosing the right topology.

However, this solution enables each client to be authoritative over the own game state while simply informing all the other clients continually about the changes in it. This leads to a couple of problems; first, it is difficult to handle conflicts in game state between clients in a fair way, since all clients are equal. Second, since each player is authoritative over the own game state, cheating will become trivial by reverse engineering and changing the client application or the network packets.

6.1.2.2 Client-Server

In the client-server topology, every client is connected to a server and all data traffic flows between the clients through the server. Having an authoritative server running its own version of the game based on all of the client data, will enable the game session to have a tie breaker when game states from different player create conflicts, which solves the problem of cheating. The price for solving the problem is increased latency, compared to peer-to-peer, due to the fact that the data needs to first travel to the server, secondly be processed by the server, and finally travel to the clients. Another drawback is the single point of failure that this topology creates. If the server is disconnected or if it cannot perform its duties in a fast enough pace, the clients will suffer without any means to solve the situation.

6.2 Limitations in network traffic – Bandwidth and Latency

To simulate a real-time world accurately in a multiplayer environment, with physics, interactive objects and multiple controllable characters such as cars or soldiers, the players need to regularly synchronize their game states. The question is how much and how often game state data should be sent.

In general, the more information a game client can get about an object, the more accurately it can simulate physics for that object. For example, the physics simulation in a car racing game can be improved significantly when the cars' velocities are sent in addition to their positions. Also, the more often players send data to update each other's game state, and the faster that data travels between them, the more coherent their game states will be.

Bandwidth

A multiplayer game works better if it is allowed to send more data between players. However, too much traffic over any network causes dropped packets and flooded buffers in network devices, which can ruin game play (Savage, 2009). Therefore, to get the best results, the amount of data being sent must not exceed the bandwidth of the players' network connection.

According to Frank Savage, a study made by Bungie Studios in 2007 shows that the median bandwidth available to their players was 352 kbit/s (Savage, 2009). Glenn Fiedler has another take on the median bandwidth available, as he claims

that a study by Sony in 2010 showed a higher median bandwidth of 1024 kbit/s (Fiedler, 2010a). Such median values are bad guidelines for how much data to send, however, since such a bandwidth requirement would make half of the game's potential players unable to play. Instead, developers are recommended to send no more than 8 kbit/s (Savage, 2009) in their games, which according to the studies by Bungie Studios and Sony would make 99% of the potential players able to play over the internet.

Some methods for optimizing a game to use bandwidth more efficiently are discussed in Section 6.2.2.

Latency

Action games should let players experience and influence the game world in real time, but this can be tricky to achieve in a multiplayer environment. Since data packets always take some minimum amount of time, τ, to travel from one player's computer to another, an action performed by player A at time T will never be noticed by player B before time $T + \tau$. An illustration of this can be seen in fig. 80.

Figure 80: An example of network latency. At time T, the blue race car is at the position of the red wireframe. At time T + τ, the race car has moved slightly forward. Due to latency, the player that is driving the yellow truck still believes that the race car is at the position of the wireframe.

While a perfect solution to real time network communication is not possible, since there is no way to completely remove latency in network traffic, it is possible to create an illusion of real time communication in games. One way of doing this is by using a technique called Client-Side Prediction (Fiedler, 2010b), which is discussed further in Section 6.2.1.

6.2.1 Solving Latency Problems – Client-Side Prediction

As was explained in Section 6.1.2.2, a server should have full authority over every player's game state, which means that all effects of actions performed by a client will be decided by the server. In a real time game, we want players to see the effects of their actions immediately, but due to latency, clients must wait some time before the server acknowledges their actions and tells them what has really happened.

Client-Side Prediction, also known as Dead Reckoning (Aronson, 1997), lets a client predict the server's response and the actions of other players. These predictions are then used to calculate the client's next game state, which can be displayed to the local player's screen in real-time. When an updated state arrives from the server, the client replaces its current, predicted state with the one received. Thus, erroneous predictions are quickly corrected, and the coherence between the client's and server's states is maintained.

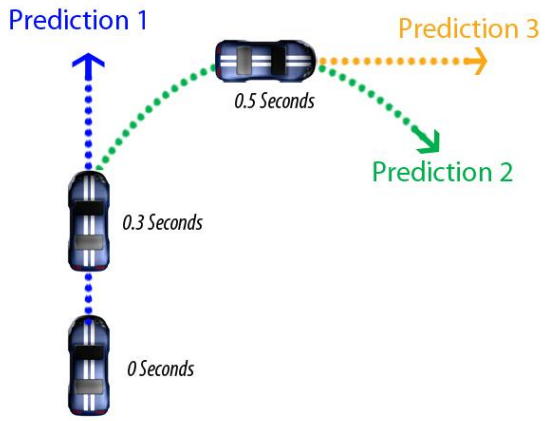

Figure 81: An illustration of Client-Side Prediction. At time 0, the client uses the last known user input, velocity and position of the car to predict its future movement according to the blue dotted line, prediction 1. At times 0.3 and 0.5 new data arrives from the server, so the client updates its game state and new predictions are made according to the green and orange dotted lines, prediction 2 and 3.

Interpolation

Although basic client side prediction is effective at hiding latency from players, it also introduces a stuttering effect when a client abruptly switches, "snaps", to the server's state (Fiedler, 2006). This phenomenon can be easily observed in multiplayer games, for example when a player controlled character suddenly jumps back in time or instantly moves to a different position in first person shooters or role playing games. A smooth transition between the old and new states would be preferable, and can be achieved by using interpolation.

Instead of directly switching to the state received from the server, a client can compare its own predicted and simulated state to the one received. If the difference is less than some value δ, a new game state is calculated by interpolating the two states. By using this interpolation technique, the clients can slowly adjust to the server's state, and unnecessary snapping can be avoided. If the difference between the states is greater than δ, however, the client has gone too far astray in its predictions, and snapping is performed to keep the client's and server's game worlds similar and to preserve the authority of the server (Fiedler, 2006).

6.2.2 Solving bandwidth problems

Games, such as the real time strategy game Rome: Total War (The Creative Assembly, 2004), can contain thousands of interactive objects, each with its own game state variables such as health, position, velocity, owner and current orders. Sending a whole game state over a network every frame, while keeping the bandwidth limit of 8 kbit/s, discussed in Section 6.2, would be impossible in such games. Even game states in other kinds of games, such as first person shooters, grow increasingly more complex with a greater number of players and more interactive environments, for example destructible buildings and drivable vehicles. Luckily, there exist special techniques for conserving bandwidth, compressing game data and updating game states without sending a whole new state over the network.

Trading bandwidth for latency

Bandwidth is a hard limit in the sense that it causes packets to be dropped if exceeded, which can ruin game play as important state updates are lost. While low latency is desirable in most games, a slightly longer delay between data being produced by one player and delivered to another might not ruin the game experience completely. A simple method

for conserving bandwidth is therefore to send game state updates less often (Savage, 2009).

For example, instead of sending position and velocity updates for an object every frame, send updates every 10th frame to reduce the amount of data sent by 90%. This will delay a physics state update by at most 9 frames, which corresponds to 1000/60*9=150 ms if the frame rate is 60 FPS. Latency hiding techniques such as Client Side Prediction, which was explained in Section 6.2.1, can then be used to deal with the increased delay.

Reduce network overhead by sending larger but fewer packets

Because the header size of a UDP datagram is 8 bytes (Postel, 1980) and the header of the encapsulating IP packet is at least 20 bytes (Postel, 1981a), the total overhead of the network will increase with the number of packets sent. By grouping many smaller messages together into one package, the number of UDP datagrams sent can be reduced, and the network overhead will shrink accordingly (Savage, 2009).

6.3 Results

For Road Kill, we implemented a client-server solution using UDP only. Instead of implementing an authoritative server with real client-side prediction we ended up with a server that simply echoes the messages that are sent to it by the clients that has a current session with the server. In addition to ignoring packets from players that do not have a current session with the server, it also ignores packets that have an incorrect header, and these two features make the server more robust and stable.

6.4 Discussion

We originally had a peer-to-peer solution which we experimented with before ending up with our current client-server solution. The decision to leave the peer-to-peer solution was made because we had plans to implement features, e.g. client side prediction presented in Section 6.2.1, that demanded an authoritative server.

For our critical non-game state data, we opted to use UDP exclusively instead of TCP or any mix of them both simply because we did not feel that we benefited from using TCP when taking into to account all of the pros and cons, as discussed in Section 6.1.1, related to it. We decided to implement our own supplements to UDP to get the features we wanted. However, features like delivery guarantee were down prioritized in favor of graphical effects and game logic.

The next step with our network solution would have been expanding the Road Kill game state data to include not only position and current speed of cars but also current user input. This would have allowed us to send even fewer packets per second, 10 being a good minimum number with our current implementation, and after that, adding a delivery guarantee on our non-game state messages, such as session initialization, and then focusing on real client-side prediction.

7 Sound

To create a compelling gaming experience, graphical effects alone might not be enough to arouse the player. However, graphical effects in combination with music and audio effects can create a more enjoyable gaming experience for the user. A study done by S. Wolfson (Wolfson, 2000) has shown that loud sounds with a red background intensify the aura of a game which leads to a more enjoyable game session.

The easiest way to add sound effects to a game is to use a library that is able to play interactive audio. There are several different libraries with this function, and every library has its own features to offer. These libraries are usually able to play the common sound formats, like MP3 and WAV and more extensive libraries also offer a toolbox of equipment to manage sounds, for example 3D sounds, multi-channel streaming, multiple outputs, recording, and effects. Libraries that support these features are OpenAL, DirectX Audio and FMOD. DirectX Audio has a wide market because this library is included in DirectX SDK (Seddon, 2005)

The difference between a high- and low-level Application Programming Interface (API) is that a low-leveled API manages chunks of sampled audio data that it stores in a secondary buffer and also the transfer from the small buffer to a master buffer for hardware mixing.

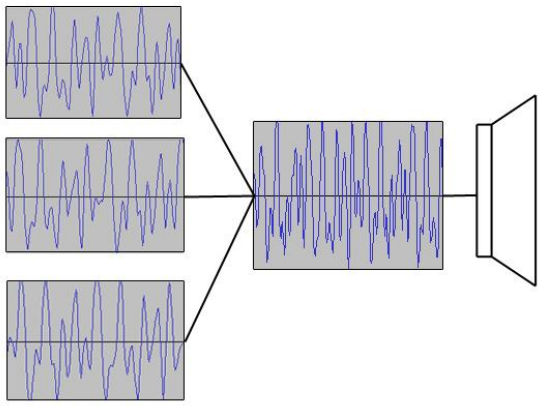

Figure 82: Combining many secondary buffers to one master buffer.

A high-leveled API controls the audio further away from the buffers than a low leveled API and has therefore less control over the audio but has more complex features to offer such as playback of MP3 or WAV. The high leveled API reduces the amount of code a programmer has to write and makes it easier to play simple sounds or music in a game. These features are possible to create with a low leveled API but require a bigger effort from the programmer.

In the next sections, the previously mentioned audio libraries will be presented and compared from the viewpoint of a developer with limited resources.

7.1 OpenAL

OpenAL is a cross platform 3D audio library made especially for sounds moving around in a 3D space, which suits a 3D game well (Creative Labs, 2010). This library has objects such as a listener, a source, and a buffer. With this library, all sound rendering is made from the location of the listener. The downside with this library is that it is narrow and does not include many features beside the 3D

Sounds (James, 2003).

7.2 DirectX Audio

DirectX Audio is a Windows based sound library and is a combined version of a low level API called Direct Sound and a high level API, Direct Music. As claimed by James (James, 2003), the DirectX Audio API is confusing for programmers to read. The upside with this combination is that DirectX Audio becomes one large API that can handle audio in many different ways. However, DirectX Audio is restricted to Microsoft-only products such as Windows and Xbox 360 (Microsoft, 2010). Some of the features are 3D sound effects, playback from multiple sources simultaneous and schedule the timing of music events with high precision (Hawkins, 2002).

7.3 FMOD

The world-leading sound API is called FMOD and it is widely used by the game industry (Firelight Technologies Pty. Ltd, 2011). FMOD has become widely popular due to of its simplicity and cross-platform support. For example, developers only need to include one header file to get access to all of FMODs features. FMOD has features like floating point calculations, output to mono, stereo, 5.1 and 7.1, advances compression algorithms and 3D sound effects. The downside with FMOD is that for commercial use, the library is very expensive (Firelight Technologies Pty, Ltd, 2011).

7.4 Results

Testing with the library FMOD was done for Road Kill with satisfying results. All that was needed was a header file and a DLL file from FMOD in Road Kill's directory. Initializing a FMOD system and loading a sound, for example music, required four function calls in C++. After these calls, the ability to choose from a list of effects was supplied, enabling the modification of sounds, such as echoing, pitching and 3D position.

To add a 3D effect to the sound, in order to hear where the sound came from, a sound source- and listener position, which usually is the camera, needs to be supplied.

7.5 Discussion

The only real downside with FMOD is the price for a commercial license, and since Road Kill is a game with potential for commercializing, we do not want use a library as expensive as FMOD when there are alternatives that are free to use. FMOD is expensive because it supplies so many built-in features, many of which are superfluous in a game. Therefore, it would be an unneeded expense to purchase a license for FMOD and all of these features. However, the convenience with the cross-platform support should not be underestimated when it comes to creating a game for many different platforms. FMOD is the only library in this study that features an active support to contact if problems occur.

The best library in our opinion, for a game, is OpenAL. This is true, especially if you develop the graphics part using OpenGL. Since these two libraries are siblings and built up with the same structure, it is easy to use either if you are familiar with the other. OpenAL is free software with a supportive forum for users to search, if

they are in need of help. 3D effects are supported by OpenAL and this enhances the gaming experience.

If you use DirectX for the graphics, then DirectX Audio is probably a better choice than OpenAL. This is because it is already in the same package as DirectX Graphics, and they have the same platform boundaries.

8 Development of Road Kill

This chapter describes the racing game Road Kill, which was developed by the authors of this thesis during a four-month period. While the previous chapters of this thesis describes and evaluates techniques for game development, the purpose of this chapter is to give concrete examples of some results that can be achieved during the development of a computer game with limited resources and short time frame.

During the development of Road Kill, the authors faced many problems related to game development and software development in general. Section 8.2 describes some of those problems that have not been mentioned previously in this thesis.

8.1 Results

Road Kill is in a playable state, where up to four players, a higher amount of players have not been tested, can compete against each other on one race track. The players must drive their cars through a sequence of checkpoint in order to complete laps, and, ultimately, beat their opponents by finishing first, by either driving the fastest or being the only player left with a working car. To gain an edge over the opposition during the race, players can utilize two kinds of weapons to destroy their opponents' cars: miniguns and rocket launchers.

8.1.1 Game Setting

Road Kill's setting is a moonlit desert landscape, lighting is added to by streetlights, car lights and torches. The ground is mostly covered by sand, rocks, and dirt, but also by small patches of grass. During the race and in the menu, the metal music soundtrack is played in the background. According to numerous test players, the combination of the above features and missile explosions, minigun fire and racing at high speeds; creates an action packed atmosphere.

8.1.2 Main features

Some of Road Kill's features are a HUD with race timing, a mini-map with real time position indication of the client and the opponents; a speedometer, a countdown to the start of the race, physics simulated in real time, graphical effects, such as explosions, reflections and fires; and full multiplayer support.

8.2 Discussion

A larger than expected part of the development process has been spent on cleaning up and structuring the code. Some examples of functions that have been restructured are the reading of car data from binary files and the importing of 3D Studio Max objects, light sources and different items needed for the physics engine; such as pre-calculated bounding spheres. This has reduced the size of class files and simplified both the modifications of the cars' attributes and adding new objects to the game world, simplifying the further development of Road Kill.

We benefited greatly from partitioning our code into smaller modules. Programming in a modular way makes development and debugging faster and easier. By creating

more testing modules, instead of the three we have now: network, modeling, and everything else, we could have simplified the development even more.

SVN was used for software versioning and revision control by the group. SVN keeps track of updates to the code base and also eased the synchronizing between the group members. SVN proved hard to get used to for some group members and since none of the group members had any experience with versioning control it was hard at times to find the solutions to our problems. However, we cannot imagine doing a project of equal size without revision control since it is very important that all group members have access to up-to-date files, documents and folders.

During the development of Road Kill, frame rate was the largest bottleneck, which limited the addition of extra effects, such as motion blur. We also learned the importance of developing on your target machine as higher frame rate was achieved on our personal computers then at the University. The feature that demanded the highest amount of optimization was the particle systems. In the end, the resulting particle effects for Road Kill were very appreciated by test players. Some effects, like exhausts and gravel spray from the tires, had a less significant impact on the general appearance of Road Kill than we initially thought, and we feel that there was a bit too much time spent on them.

After half the time, we started using our project room. This led to better group coordination and communication when working in the same room. It was much easier to make requests and demonstrate new functions and we feel that we should have started using the project room earlier.

Even though this thesis is about explaining which methods are suitable for inexperienced game developers, we have found that a better way of learning about game development is not through reading, but trying different methods and seeing their effects, pros and cons in practice.

Initially we felt that the goals we set for this project might be too high, but we consider the end result to be far above our expectations. The parts that seemed particularly difficult for us in the beginning, such as graphics engines, physics engines and network support, proved to be easier to implement than we initially thought. This was mainly due to our collective experience ranging from our different fields of interest. The really time consuming part of the project was not implementing basic versions of game components, but fine tuning them to provide an enjoyable and realistic car racing game.

References

W. T. Reeves (1983) A Technique for Modeling a Class of Fuzzy Objects. *ACM Transactions on Graphics*. Vol. 2, Iss 2. pp. 91-108.

Airey, J. M. & Rohlf, J. H.& Brooks, F. P. (1990) Towards image realism with interactive update rates in complex virtual building environments. *Computer graphics* vol. 24 no. 2 pp. 41-50

Airey, J. (1990) *Increasing update rates in the building walkthrough system with automatic mode-space subdivision and potentially visible set calculations*. University of North Carolina (department of computer science)

Akenine-Möller, T., Haines E., and Hoffman N., (2008) *Real-Time Rendering Third Edition*, Wellesley, Massachusetts

Aronson, J. (1997) Dead Reckoning: Latency Hiding for Networked Games. *Gamasutra*. http://www.gamasutra.com (17 Apr 2011)

Autodesk Inc. (2011a) 3ds Max – 3D Modeling, Animation, and Rendering http://usa.autodesk.com/3ds-max/

Autodesk Inc. (2011b) Maya – 3D Animation, Visual Effects & Compositing Software http://usa.autodesk.com/maya/

Autodesk Inc. (2011c) Area :: Discussion
http://area.autodesk.com/forum/autodesk-3ds-max/

Braben, D (2011) Death of the £50m game?
http://www.develop-online.net/blog/178/Death-of-the-50m-game (12 May 2011)

Remedy Entertainment (1996). *Death Rally.* [CD-ROM]. Garland, Texas, USA: Apogee Software.

Barber, C. B. & Dobkin, D. & Huhdanpaa, H. (1996) The Quickhull Algorithm for Convex Hulls. *ACM Transactions on Mathematical Software*, vol. 22, no. 4, pp. 469-483

Blender Foundation (2011) Blender.org Home http://www.blender.org/features-gallery/features/

Blinn, J. F. (1977) models of light reflection for computer synthesized pictures. *Acm Computer Graphics*, Vol. 11 no. 2 ss. 192-198

Bruno M. (2002) *Game Programming All in One*. [Electronic] Muska & Lipman/Premier-Trade.

Cebenoyan, C. (2004) Graphics pipeline performance. In: Randina Fernando, ed., *Gpu gems*, Addison-Wesley ss. 473-486

CGTextures, (2011) http://cgtextures.com/ (2011-05-17)

Cohen-Or, D et al. (2003) A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, no. 3, pp. 412-431.

Cook, R. L. & Torrance, K. E (1981) A Reflectance Model for Computer Graphics. *Computer Graphics (SIGGRAPH '81 Proceedings).*

Cook, R. L. & Torrance, K. E. (1982) A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics*, vol. 1, no. 1, pp. 7-24.

Cormen, T.H., C.E. Leiserson, and R. Rivest, (1990) *Introduction to Algorithms*, MIT Press, Inc., Cambridge, Massachusetts,

Creative Labs (2010) http://connect.creativelabs.com/openal/default.aspx (2011-04-15)

DeLoura M. (2000) *Game programming gems*, [Electronic] Cengage Learning

Ericson, C., (2005) *Real-Time Collision Detection*, Morgan Kaufmann

Fan Wu. Cabral, M. Brazelton, J. (2010). High Performance Matrix Multiplication on General Purpose Graphics Processing Units. *2010 International Conference on Computational Intelligence and Software Engineering (CiSE 2010)*. Dec 10-12, 2010, Wuhan, China.

Fiedler, G (2010a) Networking for Physics Programmers. *Game Developers Conference*. March 9-13, 2010, San Francisco. [pdf] http://www.gafferongames.com (31 March 2011)

Fiedler, G (2010b) What every programmer should know about game networking. *gafferongames.com.* http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking (17 Apr. 2011)

Fiedler, G (2011) Networking for Game Programmers. *gafferongames.com.* http://www.gafferongames.com/networking-for-game-programmers/udp-vs-tcp (31 Mar. 2011)

Fiedler, G (2006) Networked Physics. *gafferongames.com.* http://gafferongames.com/game-

physics/networked-physics (2 may 2011)

Finney K. (2004) *3D Game Programming All in One*, [Electronic] Course Technology PTR

FMOD Features, Firelight Technologies Pty, Ltd. http://www.fmod.org/index.php/products/fmodex (2011-04-15)

FMOD Licenses, Firelight Technologies Pty, Ltd. http://www.fmod.org/index.php/sales (2011-04-19)

Free Software Foundation, Inc. (2010) *GNU Lesser General Public License*. http://www.gnu.org/copyleft/lesser.html (13 May. 2011).

Gahan, A. (2010) *3D Automotive Modeling*, [Electronic] Science Direct

Game Physics Simulation (2011) *Game Physics Simulation*. http://bulletphysics.org (13 May. 2011).

Gottschalk, S. & Lin, M.C. & D. Manocha (1996) OBBTree: A Hierarchical Structure for Rapid Interference Detection. *Computer Graphics (SIGGRAPH 96 Proceedings)* pp. 171-180

Gouraud, H. (1971) Continuous Shading of Curved Surfaces. *IEEE Transactions on Computers*, vol. 20, no. 6, pp. 623-629

Greene, N. (1986) Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications.* vol. 6, no. 11, ss. 21-29

Greene, N. (1994) Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. In: Heckbert P. S., ed., *Graphics Gems IV*, Academic Press.

Haigh-Hutchinson, M., (2009) *Real Time Cameras: A Guide for Game Designers and Developers*, Morgan Kaufmann

Hawkins K., Astle D. (2002) *OpenGL Game Programming*. [Electronic] Course Technology PTR.

Heckbert, P. PIXAR (1986) Survey of texture mapping. *IEEE computer graphics and applications*. Vol: 6 Iss: 11 pp: 56

Heidmann, T. (1991) Real Shadows, Real Time. *Iris universe*. no. 18, pp. 23-31

Hori et al. (1998) Performance evaluation of UDP traffic affected by TCP flows. Special Issue on Multimedia Communications in Heterogeneous Network Environments. *IEICE Transactions on Communications*, Vol. E81-B, No. 8, pp. 1616–1623, August 1998.

Internet World Stats (2010) *Internet Usage Statistics, The Internet Big Picture*. http://www.internetworldstats.com/stats.htm (17 Apr 2011)

James R. (2003) *Game Audio Programming.* [Electronic] Cengage Learning.

King, G. (2004) *Shadow Mapping Algorithms.* Gpu jackpot presentation, oct.

Lengyel, E (2000) Tweaking a vertex's projected depth value. In: Deloura, M. (ed). *Game programming gems 3*. Charles River media.

Lorach, T. (2007) NVIDIA White Paper, 2007 http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf

Maciel, P. & Shirley, P. (1995) Visual navigation of large environments using textured clusters. *Proceedings symposium on interactive 3d graphics*, Monterey, CA, USA — April 09 - 12, 1995 pp. 96-102

McReynolds, T., Blythe, D (2005) *Advanced graphics programming using OpenGL.* San Francisco, CA: Elsevier Morgan Kaufmann Publishers.

Microsoft. (2011) *Core Audio Overview*. http://msdn.microsoft.com/en-us/library/ee415698(v=VS.85).aspx (19 Apr 2011)

NVIDIA Corporation. (2011). What is CUDA?. http://www.nvidia.com (14 May 2011)

Omohundro, S. (1989) *Five Balltree Construction Algorithms*. Berkley: International Computer Science Institute.

Ookla (2011) *Net Index*. http://www.netindex.com (17 Apr 2011)

Open Source Initiative (2011) *Open Source Initiative OSI - The BSD License: Licensing Open Source Initiative.* http://www.opensource.org/ (13 May. 2011).

Phong, B. T. (1975) Illumination for Computer Generated Pictures. *Communications of the ACM*, vol. 18, no. 6, pp. 311-317

Postel, J. (ed.) (1980) User Datagram Protocol. *RFC 768*. USC - Information Sciences Institute. http://www.ietf.org (18 Apr 2011)

Postel, J. (ed.) (1981a) Internet Protocol. *RFC 791*. USC - Information Sciences Institute. http://www.ietf.org (18 Apr 2011)

Postel, J. (ed.) (1981b) Transmission Control Protocol. *RFC 793*. USC - Information Sciences

Institute. http://www.ietf.org (18 Apr 2011)

Praun, E. et al. (2001) Real-time Hatching. *Computer Graphics (SIGGRAPH 2001 Proceedings).*

Reeves, W. T. (1983) Particle Systems—a Technique for Modeling a Class of Fuzzy Objects.

Ritter, J. (1990) An Efficient Bounding Sphere. In: Glassner, A. S., ed., *Graphics Gems*, Academic Press.

Roelofs, G., Gailly, J., and Adler, M., (2005) *zlib License.*
http://www.gzip.org/zlib/zlib_license.html (13 May. 2011).

Royce, W. W. (1970) *Managing the Development of Large Software Systems*, Concepts and Techniques. IEE WESTCON, Los Angeles CA: 1-9.

Samet, Hanan, (1989a) *The Design and Analysis of Spatial Data Structures*, Addison-Wesley.

Samet, Hanan, (1989b) *Applications of Spatial Data Structures*: Computer Graphics, Image Processing and GIS, Addison-Wesley.

Savage, F. (2009) Networking, Traffic Jams and Schrödinger's Cat. *Game Developers Conference*. March 23-27, 2009, San Francisco. [ppt and wma] http://microsoft.com. (31 March 2011)

Scherzer, D., Drettakis, G. (2005) Robust Shadow Maps for Large Environments.
Central European Seminar on Computer Graphics. May 9-11, 2005, Budmerice castle, Slovakia, pp. 15-22.

Seddon, C. (2005) *OpenGL Game Development.* [Electronic] Jones & Bartlett Publishers

Shüler. C. (2005). Eliminating Surface Acne with Gradient Shadow Mapping. In:
Engel , W. ed. *ShaderX^4*. Charles River media.

Smith, R. (2006) *Open Dynamics Engine*. http://www.ode.org (13 May. 2011).

Stamminger, M. & Drettakis, G. (2002) Perspective Shadow Mapping. *ACM Transactions on Graphics (SIGGRAPH 2002),* vol. 21, no. 3, pp. 557-562.

Sweeney, T. (1999) Unreal Networking Architecture. http://unreal.epicgames.com/Network.htm (1 apr. 2011)

Szirmay-Kalos, L. & Aszodi, B. & Lazanyi, I. (2008) Ray tracing effects without tracing rays. In: Engel, W. ed., *shaderX^4*, Charles River Media.

Teller, S. J. & Sequin, C. H. (1991) Visibility Preprocessing for Interactive Walkthroughs, *Computer Graphics* Vol. 25, No. 4, ss. 61-69

Teller, S. J (1992) *visibility computations in densely occluded polyhedral environments* University of Berkeley. (Department of Computer Science).

Teller, S. J, & Hanrahan, P. (1994) Global Visibility Algorithms for Illumination Computations. *Computer graphics*, ss. 443-450

The Creative Assembly. (2004). *Rome: Total War.* [CD-ROM]. San Francisco: SEGA

Ulrich, T., (2000) *Game Programming Gems*, Charles River Media pp. 444-453

Watson, B. & Luebke, D. (2005) The Ultimate Display: Where Will All the Pixels Come From? *Computer*, vol. 38 no. 8 pp. 54-61.

Williams, L. (1978) Casting Curved Shadows on Curved Surfaces. *Computer Graphics (SIGGRAPH '78 Proceeding).* pp. 270-274

Wolfson S. (2000) The Effects of Sound and Colour on Responses to a Computer Game. *Interacting with Computers* Vol. 13, Iss 2, Dec 2000, Pages 183-192

Appendix A

Contributions

A report of the individual contributions is included in this appendix, containing the contributions to this thesis and to the development of Road Kill.

A.1 Development

The project was planned and executed by all members of the group as a whole. The different responsibilities were divided to the subgroups that were formed and reformed, as features were completed and development of new features was started. The following list contains the contributions to features from each individual.

Viktor Arvidsson

Modeling – Creating models, loading screen
Game logic – Checkpoints

Jonathan Gustafsson

Graphics – Particle effects design
Network – Framework and peer-to-peer solution

Per Jamot Johansson

Game engine
Graphics engine
Physics engine – Our own, Bullet
Game mechanics – Checkpoints, missiles, Game GUI
Network
Graphics – Shadows, light, culling, texture blending, reflections, sprites
Particle system engine
Optimization and testing
Game restart – Restarting clients
Physics – balancing car parameters
Game menu

Christoffer Nilsson

Modeling – Creating models, textures, bounding volumes
Level editor

Sound – Creating sounds
Game mechanics – Checkpoints
Optimization
Graphics – Texture blending
Physics – Balancing car parameters

Adam Sällergård

Physics engine – Our own, Bullet
Game mechanics – Checkpoints, weapons, Game GUI, race synchronization
Network – Peer to Peer solution
Graphics – Camera, sprites
Optimization and testing
Physics – Balancing car parameters

Robin Ytterlid
Network – Implementation, sending of game state variables, synchronization, handling of different message types, partially reliable data transfer
Game mechanics – Placements, timekeeping, Game GUI
Game restart – Self restarting server
Physics – Balancing car parameters

A.2 Thesis writing

Editorial work was shared between all members of the group during the project. During the end of the project the majority of the editorial work was moved to Jonathan Gustafsson and Robin Ytterlid, while other members focused more on the formalities. The following list contains the main sections that each member has contributed to.

Viktor Arvidsson
Writer:
- Modeling
- References

Jonathan Gustafsson
Writer:
- Network
- Graphics – Particle systems

Corrector:
- Introduction
- Program structure
- Appendix
- Modeling
- Development of Road Kill

Per Jamot Johansson
Writer:
- Graphics – all sections except particle systems and camera
- Physics
- Program structure
- Development of Road Kill
- Appendix

Christoffer Nilsson
Writer:
- Modeling
- Introduction
- Program structure
- The majority of the figures in this thesis

Adam Sällergård
Writer:
- Physics
- Graphics – Camera
- Development of Road Kill – Results

Robin Ytterlid
Writer:
- Network – section 6.2 with sub sections.
- Development of Road Kill

Corrector:
- Introduction
- Graphics – all sections except particle systems and camera

A.3 Other

Per and Jonathan shared the role of group leader. Per was the leader up until the second half of the project when Jonathan assumed the role.