Crankshaft

A 3D Racing Game with Network Support

Jonas Gustafsson
Lin Loi
Fredrik Norén
Michael Sandén
Patrik Sjölin

Bachelor thesis
Department of Computer Science
Chalmers Univeristy of Technology
Gothenburg, Sweden 2008

Abstract

This bachelor thesis describes the development of a 3D computer game. While one of our more important goals was to produce a graphically attractive game a lot of effort also was put into network, physics and general game design. Different techniques that we use are described together with other possible solutions implemented in commercial games and described in scientific papers.

Contents

1 Introduction **3**
1.1 Purpose . . . 3
1.2 Background . . . 3
1.3 History . . . 3

2 Method **4**
2.1 Mapeditor . . . 4
2.2 Tools . . . 4

3 Program **5**
3.1 Crankshaft . . . 5
3.2 Server . . . 5
3.3 Client . . . 5
3.4 Results and Discussion . . . 6

4 Graphics **7**
4.1 Design . . . 7
4.1.1 Structures . . . 7
4.1.2 Discussion . . . 7
4.2 Lighting . . . 7
4.2.1 Background . . . 7
4.2.2 Techniques . . . 8
4.2.3 Result . . . 9
4.2.4 Discussion . . . 10
4.3 Shadows . . . 10
4.3.1 Background . . . 10
4.3.2 Techniques . . . 10
4.3.3 Result . . . 11
4.3.4 Discussion . . . 12
4.4 Particle system . . . 12
4.4.1 Introduction . . . 12
4.4.2 Available techniques . . . 13
4.4.3 Chosen techniques . . . 13
4.4.4 Results . . . 13
4.4.5 Discussion . . . 13
4.5 Ground . . . 13
4.5.1 Techniques . . . 13
4.5.2 Result and Discussion . . . 14
4.6 Bumpmapping . . . 14
4.6.1 Techniques . . . 14
4.6.2 Result and Discussion . . . 15
4.7 Culling . . . 16
4.7.1 Techniques . . . 16
4.7.2 View frustum culling . . . 16
4.7.3 Quadtrees . . . 17
4.7.4 Result and Discussion . . . 18
4.8 Optimizations . . . 18
4.8.1 Background . . . 18
4.8.2 Techniques . . . 18
4.8.3 Result . . . 18
4.8.4 Discussion . . . 19

5 Physics **20**
5.1 Introduction . . . 20
5.2 Integration . . . 20
5.3 Rigid Body Dynamics . . . 22
5.4 Collision Detection . . . 22
5.5 Collision Response . . . 23
5.6 Discussion . . . 24

6 Network **25**
6.1 UDP and TCP . . . 25
6.2 Game state . . . 25
6.3 Quake 3 networking . . . 26
6.4 Crankshaft . . . 26
6.5 Results . . . 27
6.6 Discussion . . . 27

7 Discussion **28**

1 Introduction

Crankshaft is a 3D computer car racing game. In this thesis we will describe the essential parts of what knowledge we have gained during this project.

We have divided this report into five main chapters which each describe an essential part.

Method This chapter describes one of the major issues while developing a game, namely the method that we used during the project.

Program The core of the program is also an important part of the project. Without a functional core the rest of the game will never reach its highest potential. The main structure of the game as a whole is described and explained. Our choice of programming language is also discussed.

Graphics A very essential part of a computer- or console game of today is the visual appearance. In this chapter we first describe the structure and design of the graphics. Different techniques when it comes to lighting and shadows are also discussed. This is followed by explanation of our particle system which is used to simulate fuzzy objects such as smoke and explosions. We also use a technique to render the ground in the game which is explained. Then the bumpmapping, culling and choice of optimizations are discussed.

Physics Physic systems are becoming more and more frequently used in modern games today. The system is used to simulate the behaviour of objects which exists in the game. In our game this mostly affects the cars. We describe how different techniques work and how we have implemented this system.

Network One of our goals where to be able to let players on different locations compete against each other. This meant that we needed to implement network support for the game. This is presented in this chapter.

1.1 Purpose

The purpose of this bachelor assignment is to learn how to efficiently develop a functional computer game within a tight timeframe. Our goal with the project was to develop a relative graphically advanced game. Requirements include a game that is visual attractive and supports multiplayer using network connection.

1.2 Background

The computer- and console games industry is growing larger for each year that passes. The Swedish market increased impressively 31.5 percentages during 2007 [1]. While the business is growing it receives increasingly more attention from corporations and researchers. Both hardware and software are rapidly developed which makes this a very interesting industry.

1.3 History

Racing games are often divided into racing simulators and arcade racers. The simulators are as the name implies trying to be realistic when it comes to how the car handles and interact with other objects. The arcade games on the other hand try to attract players by high speed, indestructible cars and such.

Our choice was to implement the arcade style of game, using checkpoints on the maps which gives the map makers the possibility to build maps with alternative roads.

2 Method

We decided early that we wanted to run the project in an iterative manner. To put up new goals and to check that old tasks were completed weekly meetings were held. To avoid problems with merging our code during meetings we put up an SVN where we could easily update and commit our piece of code. When proceeding with our project we all applied ourselves to eXtreme programming. The advantages of this would be that we wouldn't have to spend time on structuring the program in a way that later would be considered bad. Instead of that one will realize over time what is needed to be changed and/or rewritten.

2.1 Mapeditor

The mapeditor was developed by us using windows forms. This was done simply because we needed to store a lot of information concerning the map. This can be things like the cars' starting positions, checkpoints and props etc. To do this work manually would be too time consuming and probably even close to impossible since the world consists of around half a million objects. Our mapeditor loads the heightmap into a picturebox. Then one can select what type of object to add, and simply click the picturebox and a dot will appear representing that object. To store this information we are using the xml-format since it's very easy to operate with.

2.2 Tools

The console helped out when needing to dump information for finding out whether some certain code did what it was supposed to do. It also helped out a lot when trying to isolate what parts of a certain section of code that was causing a bug. To extend the debugging even further we used debugview that can display bugs detected by directx not visible to visual studio.

To find out about where the bottleneck was we used many different profilers.

Crazybump is an application, that was used for creating normalmaps. This application simply takes a picture and calculates its normals depending on the colors. We decided to grey-scale our textures and feed crazybump with that. Those normalmaps we then used in the shader to perform bump mapping.

3 Program

Producing quality software has always been a central problem to programmers. As time progresses higher level programming languages evolved which reduced bugs and simplified structuring. The introduction of Microsoft's C# and .NET has been a great step towards the next generation of languages, and provides many advantages over older languages such as C++, Java and C. As languages evolve so does the code and many design patterns[2] become obsolete as they are incorporated in the languages.

An article series that describes many of the old concepts of software design for games is the "Enginuity" series[3].

A brief introduction to game engine design (and ideas to extend it into a multi-threaded environment) can be read in "Multi-threaded Game Engine Design"[4].

3.1 Crankshaft

Crankshaft consists of two largely separated parts, the server and the client. The server is a threaded subsystem which runs on the hosting players machine in a network game. It is responsible for handling the game state and physics, as well as communicating this with the clients and handling their input. The client on the other hand is responsible for displaying the game on the screen and interacting with the server, as well as handling things that is not shared with other players (such as menus).

3.2 Server

The server is state based with three states, Lobby, Sync and Race, each implementing a common state interface, IServerState. The main server loop reads data from it is socket and lets the current state handle the input. 30 times per second it invokes a Update method on the current state as well, which sends data to the client.

The Lobby state is the only state in which players can connect. The state's purpose is to wait until everyone that wishes has connected and they have all sent a Ready message, indicating the connected players have loaded the game on their local machines.

Once all players are ready the states advances into the Sync state, which sends the initial game state to all the players. The clients respond with a Synched message when they have received the message. The game then advances into the Race state.

The Race state simply reads input from the users and uses it to update it is game state, and sends the updated game state to the clients. This is covered in greater detail in the Network section, see 6. In this state the physics simulation runs, see 5.

3.3 Client

The client is state based as well, but the number of states is greater than in the server. The client has several states for different menus and states for starting, running and waiting for player to finish a race. They all implement the same abstract interface, IClientState, and everything happens within the states (see Figure 1). All menus derives the abstract class ClientStateMenu, which contains methods and structures to easily create menus.

```
nextstate = state.Update()
if(nextstate != null)
{
  state.Release()
  nextstate.Init()
  state = nextstate
}
state.Draw()
```

Figure 1: *Crankshaft main loop, same principle in both client and server*

The first state the program enters is the MainMenu state, which eventually leads to the player selecting whether to host or join a game. If the player chooses to host a game a server is started locally.

After this the player loads all content (such as models, textures, shaders, sound, the map, etc.) in a Loading state. Models are the objects which gets drawn on the screen, such as the car and trees. Part of the models are loaded from files, and assigned certain attributes describing how they should be drawn, such as textures and shaders (more on this in **??**). All these models is loaded and assigned their attributes in the Models class, which also contains methods for retrieving them again later. Sound is handled by XNA's Audio Engine and sound banks.

The map with the game state is then loaded in our GameState object. Maps are simple xml files with information on where and what is in them, such as trees and checkpoints, and it also contains information on what height map to use and where to place the cars when the race starts. When loading the map it starts with creating the ground (see **??**). It then builds a quadtree (see 4.7) with the loaded width and height of the map. The next step is to load all the entities in the game, the trees, bushes and checkpoints etc. These are stored inside the game state, and their graphical representation (model) is inserted in the quadtree.

```
Update network
Read input
Update audio
Update display
Update gui
Update gamestate
Send input to server
Draw gamestate
Draw gui
```

Figure 2: *Crankshaft game loop*

When this is done, the game then first enters the InitRace state which is a countdown to the start of the race, then the Race state launches. The Race state is the main game loop state, in which all the interesting things happen. The game loop is quite simple, it starts with handling input from the user and the network, then it updates its game state and finally draws everything to the screen (see Figure 2). The game state update is equally simple, advancing the cars and checking if anyone passed a checkpoint.

3.4 Results and Discussion

Although the design is extremely simple, it does work very well with the task at hand, we had little trouble adding new subsystems. The main drawback with this approach is the user modability, which is basically non-existent.

The physics simulation on the server takes it game state from the client. This means the game only has to load the game state once, and in one place, which saves both runtime resources and time, and development time. The downside is that it couples the code unnecessarily, which in the long turn makes it more complex and harder to maintain.

4 Graphics

Computer graphics have always been a important part of games to strengthen the interaction between the player and the game, but it can also improve the gaming experience. For example games like Doom 3 (a first person shooter game developed by Id software), where the graphics is a very important part to give a really intense atmosphere. Graphics in games are for the most used to show visuals like: shadows, reflections, lighting etc. and the algorithms used to implement these are often mathematical models.

Visuals in computer graphics have developed from simple hardware with a few pixels like in the game Spacewar (assumed by many to be the first computer game) to today modern games with millions of polygons like in the game Crysis (a first person shooter game developed by Crytek).

4.1 Design

The design for the computer graphics part in Crankshaft consist of how to structure the meshes in a appropriate way. With appropriate means taking the consideration of factors like: which type of game you are making, how complex the software will be, time limitations, how easy the software should be to modify and how reusable the software should be.

A good way of structuring meshes in a bachelor project as in Crankshaft is to implement a tree structure with parent children relationship, where changes in a parent node propagates to the children nodes. By implementing such a structure changes can easily be made without rewriting a lot of code.

4.1.1 Structures

XNA's Model structure The XNA Framework have a built in Model class [5] for the structuring of meshes. the meshes is separate objects and can be transformed independently. Each mesh consist of a ModelMeshPart which descibes the material properties of the mesh (for example: which effect the mesh is using).

The structuring of meshes in the XNA's Model class resembles of the one in Crankshaft, Where the structure is a tree with parent children relationship.

The advantage of using the XNA Model class is that the class is easy to use and has support for the most common features of a Model class. Another advantage is that by using XNA's Model class one will not have to implement a content loader to the vertex data.

Crankshaft's Structure The Crankshaft implemntation of the structure for models and its meshes is similar to that in XNA's Model class. We also had, as in XNA's Model class, a tree with parent children relationship beetwen models.

The Crankshaft structure used the ASE (ASCII Scene Export) format for holding the necessary data for a model. The reason to why we chose the ASE format was because the format is easy to use and understand with decent functionality. The disadvantage is that in the implementation a content loader have to be written, compared to XNA which had one built-in one in the framework.

The structure in Crankshaft is such that when changes occurs in a parent node, the changes will propagates to the children nodes. The implementation also supported cloning of models which will faciliate the loading of models. Instead of loading a new instance of a model which will require reading from a file and therefore take many cycles, the interested model could just be cloned (which will save many expensive cycles).

4.1.2 Discussion

The reason why the Crankshaft team did not choose XNA's Model class and made our own was that in the Crankshaft project we wanted to understand how the structure of meshes in a model could be done.

4.2 Lighting

4.2.1 Background

Adding lighting to a game can drastically improve the atmosphere and realism in games. Implementing lighting in the past has been done in the application (for example: lightmapping). Today the hardware is more advanced which makes it possible to program shaders that runs on the GPU and hence doing the ligthing in the GPU, which is faster.

With the powerful hardware today, it is possible to implement advanced Lighting techniques like per pixel lighting and reflections without burden of the GPU to much.

4.2.2 Techniques

Light mapping Lightmapping[6] is an old technique used in games like Quake to shade the scene. The concept behind lightmapping is that you have a texture of your model and blend it with another "'lightmap"' texture (the shading texture) by using multitexturing, which gives the result of the object beeing shaded. lightmapping has been widely used in games in the past, but are now used less and less in modern games because of the support for dynamic lighting is hard to implement. Modern GPUs now has the capacity to do per pixel lighting dynamically.

Lighting by using lightmapping can also be done dynamically as in [7]. The advantage are as in static lightmapping, that you won't need the GPU to do the mathematics. the disadvantage is the precision of the lighting, and making shadows are considered really hard (ref). because of the limitations of making lighting dynamically this techniques is not the primary technique to use when having dynamic scenes.

Gouraud shading The idea behind Gouraud shading [8] is to give smooth shading to objects without heavy computations as in per pixel lighting.The principle behind Gouraud shading is that surface normals for each vertex is calculated and then using Phong reflection model to computate color intensities at vertices. Then screen pixel intesities can be calculated by interpolating the calcualted color intensities.

The advantages with gouraud shading is that it does not require heavy computations. the problem with Gouraud shading is when applying specular highlighting to a low polygon model, and the model is rotating, the model's specular lighting will give the illusion of blinking, which is due to the highlight passing over a neighbouring vertex.

Phong illumination Is an empirical model by Bui Thong Phong in [9]. A model that gives very realistic lighting, but requires heavier computation than in the Blinn Phong shading model.

The basic principle of this model is that the shading of a object consist of adding up light components: Ambient lighting, diffuse lighting and specular highlighting, which together will give the final image of the object. See figure 3 for clarification.

One major problem with this model is that it does not take to account that there can be occlusions from other object (see figure 4). So implementing a shadow algorithm will make the shading more credible.

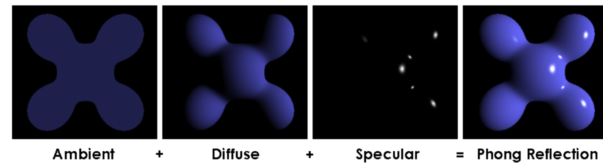

Figure 3: *Light components for Phong illumination. Picture is taken from [10]*

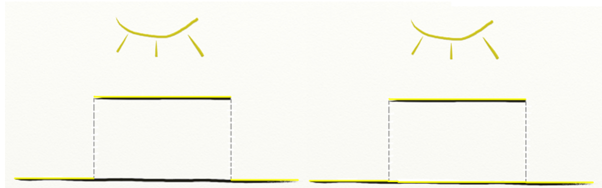

Figure 4: *The left picture shows the correct shading (the yellow color means a lit surface) where a occlusion does not give a lit part below the occluder. The right picture instead is a implemented Phong shading where the part that should not be shaded are shaded. This shows the failure of the empirical model. To get rid of this problem a shadow algorithm must be implemented. See section 4.3 for futher details on shadow algorithms.*

Blinn Phong shading The Blinn Phong model is a simpler model than the original model and is invented by James F. Blinn and described in [11]. The basic principle is the same as in Phong illumination. The model does not give as good result as Phong illumination but the result are very close.

The model is a modified model of Phong's model but it is more computing friendly. It is more friendly in the sense of: When there are a directional light in your scene and the light and viewer is infinitly far away then the halfvector in the Blinn Phong model can be calculated just once and used on the entire frame

As in Phong illumination, this model does not take to account occlusions from other models when calculating the shading, which makes it preferable to implement a shadow algorithm for a correct result.

Environment Mapped Reflection The environment mapped reflection gives a nice feeling to a model of reflecting the environment and gives a better visual impression. Especially in racing games where the car paint gives a nice reflection of the world. The problem with this simple model is that there is no reflections other than the skysphere. The technique works so that you have a texture map to the car (in our case) and then calculates the dot product between the normal and the eye direction.

To get a physically correct reflection of the scene where one way is to use ray tracing (referens), but due to the cost of doing it is to expensive (referens), environment mapped reflections seem to be the best solution for our needs. Figure 5 shows only when the environment mapped reflection of the skysphere is implemented.

Figure 5: *Figure showing the added effect of reflection from the skysphere. Notice the brown and blue parts which is the reflection of the skysphere.*

4.2.3 Result

In Crankshaft we used Blinn Phong shading which gave us realistic lighting. We also implemented reflections on the car and together with the ligthing the result was very satisfying. The reason of using Blinn Phong shading instead of Phong illumination is that the Blinn Phong shading gives really good result but also is more computation friendly than the Phong illumination. And the reason why we chose Blinn Phong shading instead of lightmapping and Gouraud shading was that per pixel lighting can be done in GPU easily and lessen the workload for CPU.

The use of environment mapped reflection was obvious because reflection of the skysphere gives a better visual impression of that it is a car which is made of metal. We limited the number of light-sources to one and the type of light source to a directional light. We found that this is enough because of rally games is limited to outdoor environments where the sun is the only light source. To see the final result of the lighting in Crankshaft see figure 6.

Figure 6: *The upper image is the final result of the lighting in Crankshaft where ambient lighting, diffuse lighting, specular highlighting and environment mapped reflection was added. Notice the difference with the lower image where no lighting at all was implemented. The result shows that implementing lighting will give a realistic look to the game.*

4.2.4 Discussion

Implementing Blinn Phong shading and environment mapped reflection did not give us any serious problems. The implementation of the techniques was not hard. Everything went on well.

4.3 Shadows

4.3.1 Background

In modern games implementing shadows is almost an essential to give realism to your scene. Calulating shadows is expensive and much research on optimizing shadow algorithms and research in improving the quality of generated shadows has been done. For a futher description and survey of shadows see the excellent paper in [12] and the more up to date article [13] about shadow algorithms and their variants.

The most widely used shadow algorithms today are Shadow volumes and Shadow mapping.

4.3.2 Techniques

Shadow volumes The basic principle behind shadow volumes [14] is, that sillouette edges is created which is the boundary between the front facing polygons (lightsource to polygon) and the back facing polygons (Polygons facing away from the lightsource). Then the silouette is extended in the direction of the lightsource. The extended sillouettte is then capped to form a shadow volume.

The advantage of using shadow volumes instead of shadow mapping is that the accuraacy is to pixel level. but the disadvantage of shadow volumes is that the technique can be CPU intensive when the complexity of the geometry increases.

For a futher clarification of the technique see figure 7

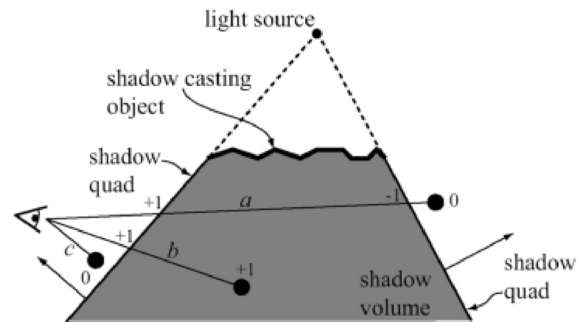

Figure 7: *clarification picture for the shadow volume algorithm. Picture is from [15]*

Shadow mapping In shadow mapping (SM) [16] the scene is rendered through the view of the light source to a depth texture every frame. To determine if a pixel should be shadowed or not one have to compare the depth from the pixel to the lightsource with the depth texture and see if the pixel has a greater depth than the value in the depth texture. If so is the case then the pixel should be shadowed.

Below is the pseudocode for the basic shadow mapping algorithm:

```
shadow(depth, depthTexture)
{
  get the corresponding color c
   from depthTexture given depth

  if depth < c then
   return not shadowed
  else
   return shadowed
}
```

Figure 8: *Pseudocode for the basic shadow mapping algorithm*

The use of shadow mapping reqiures a texture to compare the depths, and therefore the quality of the generated shadows are determined by the resolution of the texture. Another disadvantage of the algorithm is the aliasing problem caused by using floating points for comparison. Yet another disadvantage is that this technique doesn't give as good precision as the shadow volume algorithm. The advantages compared to shadow volumes is that it is mostly

faster than the shadow volume algorithm and fairly easy to implement.

Implementing the basic shadow mapping algorithm alone won't give a good visual impression because of the aliasing problem. Although one can increase the texture resolution to reduce the aliasing problem, but this is not a preferable solution because the texture will be to big and writing the rendertarget to a texture will take many cycles, so the solution of only implementing the basic shadow mapping algorithm will not yield a satisfacting visual impression due to aliasing.

One way to tackle the problem of aliasing is to implement variants to the shadow mapping algorithm which will give a better visual impression.

below we will only discuss some of the variants to the shadow mapping algorithm that the Crankshaft team have been looking into.

Parallel Split Shadow Mapping The principle behind parallel split shadow mapping (PSSM) as described in [17] is that instead of as in the basic shadow mapping algorithm, where one uses one depth texture, in PSSM one splits the view frustum in parts with a suitable strategy (more about strategy and how the splitting is done in [17]). Then smaller depth maps are created (ex: 512 x 512 pixels in resulotion) from the splitted parts.

The main advantage is that the aliasing problem is reduced. the disadvantage with this technique is that multiple rendertargets have to be created, and writing the targets to multiple textures to create the depth maps, will take many expensive cycles. For a clarification of the algorithm see figure 9. (finslipa)

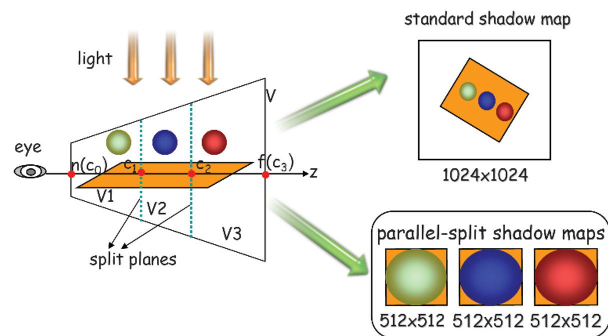

Figure 9: *clarification picture for the PSSM algorithm. Also shows the splitting by using the frustum plane. Picture is taken from [17]*

Variance Shadow Mapping The principle of variance shadow mapping as described [18] is that one has to render the depth to one channel buffer and the squared depth to another channel buffer. After that preprocessing is done to make it easier for filtering.

After that the shadow map can be blurred to reduce the aliasing. This technique requires a lot of processing power and are not suited for implementation in Crankshaft.

Percentage Closer Filtering Shadow Mapping In percentage closer filtering (PCF) the surrounding texels are taken from the depth map and the values are then compared to the depth of the current pixel. After that one will know if that texel is shadowed or not. Figure 10 (after the compare step) shows the process of doing PCF. After that the filtering is done and the value is returned.

The advantage of this technique is that it is easy to implement and does not require as much processing power as in VSM.

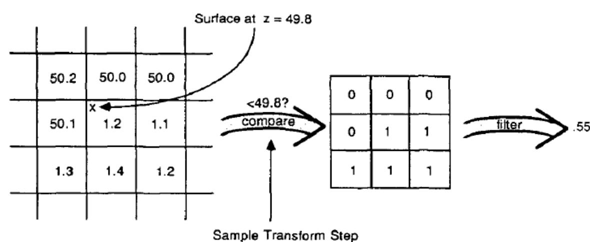

Figure 10: *The process of doing PCF. Picture is taken from [19]*

4.3.3 Result

In the Crankshaft implementation we decided to use the shadow mapping algorithm for the directional light source that we had, combined with PCF (Perentage Closer Filtering) to reduce the aliasing problem on the generated shadows. We also used a orthogonal view matrix instead of a perspective view matrix. We were very satisfied with the final result when using shadow mapping with PCF. See figure 11 for the final result in Crankshaft.

Figure 11: *The upper image is the final result of the shadows in Crankshaft where percentage closer filtering was applied. Notice the difference with the lower image where only the basic shadow mapping algorithm was implemented. Although the depth texture is of resolution 2048 x 2048 pixels, the aliasing problem can clearly be seen (especially shadows cast by the trees)*

4.3.4 Discussion

The reason that we used shadow mapping was that shadow mapping is more suited for our scene where we have a large environemnt with many trees. By using shadow volumes we would have to make heavy computations. and the use of PCF was due to that we found that inplementing shadows to our game gave a pretty much framerate drop and later on we found that the reason of that drop was due to CPU bottlenecks.

The reason why we didn't choose PSSM and VSM is that those two algorithms requires more processing power in the CPU. So implementing PCF was was a good idea because that can be done in the Pixel shader which won't burden the CPU.

Because of our scene beeing very static (only the car moving) we thought of creating the depth textures first for the whole scene without the car and then creating the shadows. This idea seem to be a good idea but we have to create many textures to map the whole scene and creating the depth textures will make it not worth implementing, and also the GPU may not have support for so many textures which would be needed.

4.4 Particle system

4.4.1 Introduction

To further improve the game experience at the graphical level we decided to implement a particle system [20]. The particle system is used to simulate fuzzy objects as for instance smoke, fog and rain. The basic flow consists of two major stages.

Simulation In the simulation stage the new particles to render are being created. They are all given a 3D position in the world of the game. This position is based on the emitter's position. The emitter is the core of the system because all particles initially originate from this object. During this update phase all existing particles are also checked to see if they have exceeded their time to live and in that case removed from the simulation. In the case of their existence their new position and characteristics are calculated. This calculating can be implemented with different techniques but the essential part is to move the particle a little bit each frame to simulate some kind of real behaviour. Particle systems often perform collision detection between particles and 3D objects in the game but collisions between particles are more often ignored.

Rendering When the calculations during simulation stage are done all the particles are rendered. The particles can be rendered in different ways. There are techniques that render all particles as a single pixel. More often the particles are rendered as a textured billboarded quad. This is an object (quadrilateral) that is always facing the viewer, in this case the player's screen.

4.4.2 Available techniques

GPU It is very important that the system is efficient in the sense that it doesn't demand too much time per frame. One way of solving this problem is to let the GPU on the graphic card handle the update procedures that are part of the particle system [21]. You can do this by making the vertexshader and pixelshader calculating position, velocity, angle etc of the objects in the system.

Billboard The model we use for the particle system represent a simple plane. The texture is applied to this plane and it is important that the object at all times is faced directly to the camera. This technique implements the idea of billboards. The big advantage of using billboards is that it consists of only two triangles and a texture instead of a more complex and more processor-demanding 3D model with a lot of triangles. The disadvantage is that the image will look the same from every direction and therefore this technique is not always suited. To display a car as a billboard simply wouldn't work. However for our particle system this works fine, by using a lot of particles with different rotation and transparency the system makes a realistic effect.

4.4.3 Chosen techniques

A particle system often consists of several of thousands or more small particles. Instead we use a technique where one particle equals one model with a texture applied to it. This way a couple of hundreds of particles are enough to simulate a realistic effect. The models are rotated and given a transparent value, alpha value. This gives a surprisingly smooth and realistic effect. By using models we can add all particles to our renderer and automatically use the z-sorting algorithm used in our renderer.

4.4.4 Results

The use of billboards greatly improves performance because of the substantial lesser amount of triangles to render in each frame.

4.4.5 Discussion

Because of the z-sorting we sadly weren't able to implement the calculations of the particles positions and transparency values.

4.5 Ground

The ground in a game can be thought of as a surface. When coloring surfaces there are several ways to proceed about this. The most common way to do this in video games, is to map textures down to the surface using UV-coordinates. By just coloring the surface by mapping textures to it, will cause the outcome to look very flat since all light reaching the ground will be reflected back to the eye in the very same way. This problem is very well known, and has a whole area of techniques available for bending the normals in a proper way. Some of those together with the one Crankshaft is using is brought up in the bumpmapping section. The only remaining problem is that the ground is large consisting of a lot of vertices which means that rendering it all in each and every frame would be very expensive. This is solved by splitting the ground up in different patches, so that the frustum culling will ignore the patches not viewed by the camera.

4.5.1 Techniques

Generating geometry The most commonly used technique for doing this is the generation of heightmaps. Those are generated from a grey scaled bmp file where each rgb-vaule at a certain pixel serves as a height. White usually present high points while black stands for the opposite. A problem that easily occurs to this is that the ground can look very angular and rough. To solve this the heightmap can be sent through a filter that evens out the height values. In our implementation of this, 9 samples were taken for each and every vertex. This is a very expensive way of doing it, but since the ground is generated only once, performance is not an important issue.

Texturing Just like the generation of a heightmap, texture splattning[22] operates on a bmp file. This colormap consists of a red, green, blue, black and a alpha value that represents different textures. In the shader each color component for a certain pixel serves as a weight on how much of each texture that is supposed to be drawn.

```
Input: Texture Spattmap,
    Grass, Earth, Road;

float3 color = (Splattmap.Green*Grass +
    Splattmap.Red*Earth +
    Splattmap.Black*Road)
```

Figure 12: *This is run for each pixel in the pixel shader. The more red there is in the map for that pixel, the more will be taken from the earth texture. The more black there is, the more of the road texture will be taken, and it keeps on going like that.*

Figure 13: *In the upper image the splatt map representing different textures is shown, in the lower image the textures have been applied in the pixel shader.*

Level of Detail As mentioned in the introduction the ground needs to be split into patches so that the areas not viewed by the camera can be culled out. Now let's consider the parts actually viewed by the camera. The idea of LOD is that areas close to the camera will be rendered using objects with high resolution while areas far away from the camera will be rendered using objects with low resolution. To reduce the LOD of an object there are several LOD-algorithms available.

4.5.2 Result and Discussion

The Geometry The result using heightmaps gave the ground a very pleasant look. By splitting it up into different patches the rendering of the ground ended up to be very cheap.

Texturing Splatting gave the exact look we were after, the textures melted together without giving an unnatural look.

Lighting Out of the three major techniques for doing this we picked bump mappnig. It gave the objects using it an impression of deepness in the surface and it didn't cause any loss in performance.

Level of Detail Rendering the ground in Crankshaft didn't result in any decrease of performance since the camera is always directed slightly down towards the ground. Also some areas that could have been considered far away from the view, was covered by trees anyway. Therefore we decided to skip implementing any LOD technique for rendering the ground.

4.6 Bumpmapping

When viewing surfaces that only has normals based on the geometry of the surface itself the result looks very flat and unnatural. Since textures are in 2D there are a lot of height information that gets lost. So a way of solving this is to compute normals out of those textures and apply them to the surface in some way. There are various techniques available for doing this that is brought up in this section.

4.6.1 Techniques

Normal mapping is a quite cheap technique while also being very easy to implement. It simply applies

a map of normals on to a certain surface. It doesn't consider the precomputed normals of the surface. It just swaps them out to the ones in the map.

Bump mapping[23] can be implemented in some different ways but the technique behind those are very similar. In the Silicon Graphics version used in Crankshaft, 3 vectors are needed to create an orthogonal matrix used to transform the bump normal into one fitting the ground. The 3 vectors are computed as seen below and then put into a matrix.

$$v1 = groundNormal \times bumpNormal$$

$$v2 = groundNormal \times v1$$

$$\begin{pmatrix} v1 \\ v2 \\ groundNormal \end{pmatrix}$$

Figure 14: *The matrix used for transforming the bump normal.*

To get this set of bump normals needed to do those computations a normalmap is used. The normalmaps was created using a tool called "crazybump" that is explained in detail in the tool section. Crazybump takes a grey scaled texture and returns a normalmap for the given texture. When those calculations are done this bumpmap is used in the shaders to reflect the light hitting the ground in their proper directions.

Parallax mapping [24] is slightly more complex. In difference from normal mapping and bump mapping it actually displaces the geometry. This is done using a heightmap for describing the roughness in the surface. This technique makes the surface of objects look even more natural than bump mapping and normal mapping. This technique is still fairly new and would go to hard on this game's performance. There are other techniques for displacing the geometry. Some also operating in the vertex shaders. If the resolution is high it will give a similar result to parallax mapping which operates in the pixel shaders.

4.6.2 Result and Discussion

Bump mapping made the surfaces reflect the light more accordingly to the hidden geometry within the textures. The reason we used bump mapping instead of normal mapping was only because the material we found about bump mapping seemed easy to implement and the examples shown of it looked promising. The difference in performance wasn't brought up to be of any importance either. In Crankshaft bump mapping was used on the car and the ground.

Figure 15: *The car above uses bump mapping in the shader while the one below doesn't. Notice the shininess of the grill in the car above.*

Parallax mapping would have been the best when only thinking about looks, but for creating displacement in the surfaces we estimated to lose performance. Also the implementation seemed to be more complex.

4.7 Culling

Scenes in a 3D game often consists of millions of millions of triangles. No modern graphics cards can handle this much data and thus, if all triangles would be rendered each frame, it would drastically lower the frame rate of the game. To combat this there is various techniques to remove triangles which would not affect the final scene, known collectively as culling techniques[25].

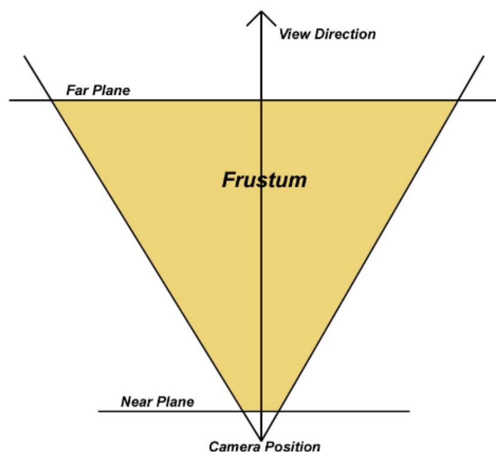

Figure 16: *Example of a view frustum in 2D. Picture from [26].*

4.7.1 Techniques

View frustum culling Removing objects which is outside the view frustum, more on this later.

Backface culling Backface culling determines whether a face is facing the viewer or not, and ignores rendering those which does not face the viewer[27]. It is handled entirely on the GPU on modern hardware. Mathematically it is a very simple test, described by the following formula:

$$n = (v_1 - v_0) \times (v_2 - v_0)$$

Where $v_{0,1,2}$ is the vertices of a triangle. If the z value of the n vector is > 0 the face is pointing away from the screen and does not need to be drawn (provided it is counter clockwise ordering).

Distance culling Objects which are very far away from the viewer may not be large enough to be represented on the screen, and can thus be removed.

Occlusion culling Occlusion culling removes objects which are behind non-transparent objects, and hence would not be visible anyhow. It can be done using a skyline algorithm[27].

Portal culling This technique is used when looking through a door or a window, in which case the frame and its surrounding occlude the scene behind the door. On way is to simply forms a new view frustum from the current view frustum and the frame of the door or window, and then rendering the scene on the other side of the door/window using this new frustum[27].

4.7.2 View frustum culling

Objects which are outside the view field of the screen, ie the view frustum, is unnecessary to render and can be removed, which is what is done by view frustum culling[26] (see Figure 16). In it is most basic form the program keeps a list of objects and each frame goes through the list, deciding which objects are visible and which are not by doing a frustum-bounding volume intersection. The bounding volume can be of any type, but commonly is a axis aligned bounding box (AABB). The frustum consists of 6 planes, the near, far, left, right, top and bottom planes of the cone constructed from the cameras view.

Calculating the view frustum can be done in several ways, one way is to back project points in the corners of the frustum, for example the top far left point would be $(-1, -1, 1)$ (depending on how the view matrix is defined, but assuming it is defined to be $[(-1, -1, 0), (1, 1, 1)]$ where z=1 is the far plane). This point is then back projected simply by multiplying the inverse view matrix with the point.

$$point' = inv(viewMatrix) * point$$

With all the eight corners transformed planes can be formed between them, for example the bottom plane

would be:

$$p = bnl, n = cross(bfl - bnl, bnr - bnl)$$
$$bnl = \text{bottom near left point}$$
$$bfl = \text{bottom far left point}$$
$$bnr = \text{bottom near right point}$$

The culling itself, in the case of an AABB, is done by intersecting each plane to the AABB. The outcome is either, inside, outside or intersecting. There are various ways of doing this but a decently fast approach is described in [28].

4.7.3 Quadtrees

In a modern game the maps can be enormous with millions of objects, and culling each one of them each frame would just be too slow. A technique to counter this is to insert all the objects into a so called spatial data structure, more specifically bounding volume hierarchies, which organizes the objects based on their bounding volumes. Culling can then be done to groups in this hierarchy, resulting in a lower number of cullings. There are many techniques to do this, such as BSP trees, Axis-Aligned BSP trees, Octrees and Quadtrees[27]. The technique described here will be Quadtrees.

A Quadtree is a tree structure where each node represent a bounding volume (see Figure 17). The tree is built by first placing a node covering the whole world, then this node has children recursively which each divide the world into 4 parts, one for each child node. When an object is inserted in the hierarchy it is simply culled against the node's bounding volume until it finds the smallest volume it can fit into, see Figure 19. Culling the objects in the scene then simply becomes culling the trees' nodes recursively and any children they hold. If a node is outside we know all it is children is outside too and it can be skipped safely. If it is on the other side inside we know all the children must also be inside and all of them can hence safely be rendered, see Figure 18.

The depth of the Quadtree depends on the size of the world, and since we always divide each node in four, we can use a logarithmic function to calculate it:

$$depth = \frac{log(\frac{max(mapwidth, mapheight)}{squaresize})}{log(2)}$$

Where squaresize is the desired smallest size of a node in the tree.

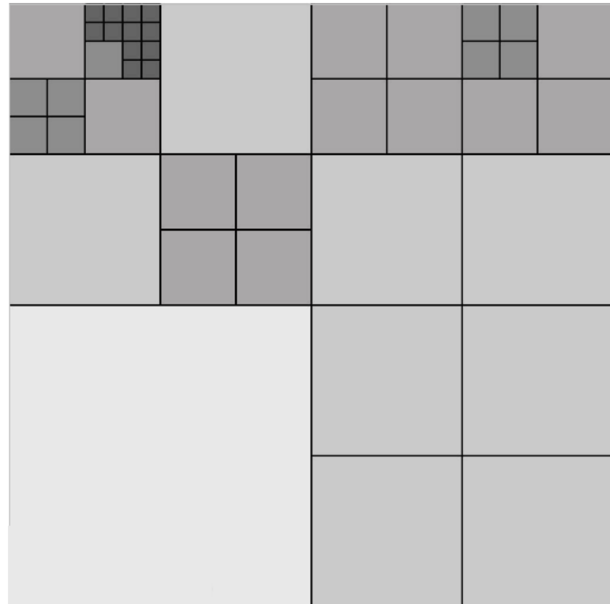

Figure 17: *Example Quadtree. Picture from [26].*

```
function cull(node, frustum)
{
  if(node is outside frustum) return

  if(node is inside frustum)
    draw all objects in node
      and its children
  else
  {
    foreach(o in node.objects)
      if(o is not outside frustum)
        draw(o)

    call recursively for each
      node.children
  }
}
```

Figure 18: *Quadtree culling pseudocode*

```
function insert(node, object)
{
  if(object fits into
      any node.children)
    insert it in that childnode
  else
    insert it into node
}
```

Figure 19: *Quadtree insertion pseudocode*

4.7.4 Result and Discussion

In Crankshaft we made use of Quadtrees, since Crankshaft is a very open terrain game with only one "level" (i.e. no bridges or other ways to drive above other cars, which would require another spatial data technique such as an Octree). The Quadtree greatly improved the performance of the application and enabled us to have huge maps with more than a million trees and bushes. A simple graphical representation of the quadtree was developed using boxes representing nodes, to help debugging, which proved very useful. Also backface culling was naturally used on much of the geometry, although not on some of the transparent geometry such as checkpoints as they can be viewed from both sides of the triangles.

4.8 Optimizations

4.8.1 Background

The often taken approach in the gaming industry for optimizations is that one developes a games and add features to a game until the performance of the game is not satisfying. After noticing that ones starts to profile the game to see what the bottlenecks of the game are. After discovering the bottlenecks, various optimization techniques for that bottleneck can be used to improve the performance. And so the cycle continues until the game runs as satisfied.

4.8.2 Techniques

Statechanges Statechanges in the code is expensive. For example: Where a setting of a vertexdeclaration can cost up to 6500 - 11250 cycles (in an average) (see [29]). Restrucutring what have to be rendered correctly will reduce the number of state changes (like changes of textures, models etc) and will save many cycles. By sorting the models with the same models after each other, the number of state changes will be reduced. And by reducing the changes of textures and effects will reduce the number of cycles lost even more. This technique widely used in almost all engines. (ej så övertygande mening)

There are many ways of reducing the number of state changes, but in Crankshaft we sorted all models, so the same models came after each other and then we rendered them in that order. We also kept track of which effect and texture that was used currently, and checked if a change was necessary.

By doing so the number of state changes were reduced.

Hardware Instancing The purpose of hardware instancing is to lessen the burden of the CPU and should be done when the CPU is the bottleneck and when your scene consist of many instances of the same model which is just placed in different places.

By using hardware instancing the time spent in draw calls are reduced because there is only one instance of a model per unique model.

The basic principles behind hardware instancing (see [30] for a detailed description) is that there is two streams to the GPU, but the GPU will interpret these streams as one stream. The first stream will contain the instance of a model to be rendered. The second stream consist of the world matrices for the models. The Vertexshader will then get the world matrices for all the models and place the vertices on the proper places. See figure 20.

Figure 20: *Figure showing how Hardware instancing is working. Two streams are sent to the vertexshader, where the first stream consist of your model and the second stream consists of world matrices which determines where your models should be placed in the world. Picture is taken from [30]*

4.8.3 Result

In Crankshaft we implemented both Hardware Instancing and reduced the number of state changes when rendering.

4.8.4 Discussion

Implementing the both techniques in Crankshaft was done without bigger problems. The biggest problem in the optimization part was that we did not properly profile the game before we started to do optimizations. reducing state changes can be done before doing some profiling, but optimizing shaders and implementing hardware instancing could have been waited until we had properly profiled the game.

We thought early in the development that the game was only CPU bound, because the CPU resources was at maximum. But it turned out to be almost right. Because after profiling the game with proper software (PerfHud, CLR Profiler and nProof), it turned out that Crankshaft was having problems with both the CPU and the GPU. The problem was that the CPU and GPU was often in idle time waiting for each other.

The best approach (as described in the introduction of this section) would be to first profile the game and then see what bottlenecks there were and then optimize those parts. And then continue the cycle until we are satisfied.

Due to lack of time and the Crankshaft team beeing inexperienced with optimizations we could not optimize the game as much as we wanted.

5 Physics

5.1 Introduction

In just about every computer game there is an underlying physics engine. Older physics engines do just about enough to prevent objects from falling through the floor or travelling through each other but as processing power increases the physics simulation in games become more and more advanced. Lately we've been introduced to games with much higher physical realism such as Half-Life 2 which uses the Havok Engine and more recently the game Crysis. Both are high paced action games where gun shots will cause barrels to tilt over and explosions will cause nearby objects to fly in all directions and bounce around the game world in a very believable fashion. A good physics engine is capable of enhancing the gaming experience and has almost become mandatory in modern games.

5.2 Integration

A physics engine stores the states of all the objects in the game. This state can contain many quantities such as position, velocity and acceleration among others. To simulate the motion of an object, the quantities stored in the object state are numerically integrated over a given time step. The two main integration methods used are *Euler Integration* and *RK4 Integration*[31].

Euler Integration Euler integration is the most basic form of integration. The main idea behind this technique is to approximate the change of a function over an interval by using the derivative of the function at the beginning of the interval. If you have a differential equation $f(y,t)$, and its derivative $f'(y,t)$ and a time step $deltaT$ you can approximate $f(y,t+deltaT)$ (which is the value of the function after the time step has passed) by using the following formula:

$$f(y,t+deltaT) = f(y,t) + f'(y,t) \cdot deltaT$$

This approximation can be applied on many physical quantities. For example, the position quantity has velocity as its derivative so we can approximate the position of an object after a certain time step by using Euler Integration:

$$newPosition = currentPosition + currentSpeed \cdot timestep$$

The major drawback with Euler Integration is that it introduces an error if the derivative of a quantity changes over the integration time step. This will also cause the error to grow larger over time.

RK4 Integration RK4 stands for Runge-Kutta order 4 and it is an integration method that samples the derivative of a quantity four times over the time step and makes a weighted average of these derivatives. The mathematical formula for RK4 Integration is:

$$f(y,t+deltaT) = f(y,t) + \frac{deltaT(k_1 + 2k_2 + 2k_3 + k_4)}{6}$$

where k_1 ... k_4 are the four derivatives sampled by the method. The four derivatives are sampled in the following way:

- $k_1 = f'(y,t)$
- $k_2 = f'(y + deltaT/2 \cdot k_1, t + deltaT/2)$
- $k_3 = f'(y + deltaT/2 \cdot k_2, t + deltaT/2)$
- $k_4 = f'(y + deltaT \cdot k_3, t + deltaT)$

By doing this the RK4 method is capable of detecting the curvature of the quantity over the given time step and this gives a much better approximation than for example Euler's method. The error of RK4 is so small that it can be used for any numerical integration given that the time step is reasonable.

Time Step Independent Physics One problem with any type of integration is that it becomes very unpredictable when the time step grows very large. For instance, imagine that an object is travelling in the direction of another object and is already very close to it. If the frame rate should drop to very low numbers the time step will grow large and this might cause one object to penetrate the other or maybe even pass through it since the integration step is too large. This can cause an erratic collision response or there may never be a response at all if the object passed through. This problem will exist even when an RK4 integrator is used since RK4

```
deltaT = TIMESTEP
while(deltaT > 0)
{
        Test if the object is in a penetrating state after deltaT
        if(Object is penetrating)
                Find point in time collisionT where the collision occurred
                Integrate with timestep collisionT
                Calculate collision response
                deltaT-=collisionT
        else
                Integrate with the timestep deltaT and break
}
```

Figure 21: *The psuedo-code for time division*

doesn't do any type of collision detection when it samples its derivatives

One solution to this problem is to use a fixed time step and a time accumulator[32]. When the physics are updated the time interval between the previous and current frame is given as a parameter. This interval is added to the time accumulator and if the accumulator is larger than the fixed time step it means that enough time has passed to make a physics update. The pseudo-code for the update method looks like this:

```
void Update(float dtime)
{
        accumulator+=dtime
        while(accumulator >= TIMESTEP)
        {
                UpdateTimeStep()
                accumulator-=TIMESTEP
        }
}
```

Using the above algorithm and a small time step will produce nice results but will not remove penetrations completely.

Collision Resolver To further avoid penetrations there exists a technique called *time division* . The pseudo-code for the time-division algorithm is shown in Figure 21.

When a penetration has been detected it means that a collision happened somewhere between t and $t + deltaT$. To find what point in time the collision occurred we use a algorithm called *bisection* . Bisection performs a binary search over the time step to approximately find the point of collision. Given a time interval $deltaT$ the method samples the state of an object at $t + deltaT/2$ and $t + deltaT$. If it turns out that the object is in a penetrating state at both $t + deltaT/2$ and $t + deltaT$ it means that the object collided somewhere between t and $t + deltaT/2$. If the object is in a penetrating only at $t + deltaT$ it means that the collision occurred somewhere between $t+deltaT/2$ and $t+deltaT$. The algorithm will recursively search the time interval $(t, t + deltaT/2)$ or $(t + deltaT/2, t + deltaT)$ depending on where in time the collision occurred. The binary search will end when the algorithm has found a point in time where the object is within a certain distance from the colliding object while not actually penetrating the object. This point in time will be the $collisionT$ seen in the above pseudo-code.

Resting Physics Another problem arises when objects have very low velocity and are close to resting on a surface. In this case, a collision will be registered every frame and the repeated collision responses make it impossible for the object to truly rest unless special measures are taken. One way to solve this problem is to use a velocity threshold. If objects have a velocity that is smaller than a certain value they do not move.

Results In the end we decided to go with an Euler integrator instead of an RK4 Integrator. This is because we thought that absolute precision was not necessary for our game and that speed was more important. Also, by using a small time step together with the algorithm used for time-step independent

physics the error caused by Euler integration becomes much smaller. Crankshaft also uses a time-division algorithm much like the one described in a previous section and the problems with resting physics have also been solved by using a velocity threshold. When a collision response is generated the velocity of the colliding object is set to zero if the velocity of the object is less than the acceleration of the object. This means that the object was in a resting state before the update and that it was accelerated by the forces acting upon it.

5.3 Rigid Body Dynamics

A *rigid body* is a physical object that cannot change its shape unlike for example jello. In Crankshaft we have used the dynamics of rigid bodies to simulate the cars and this is pretty realistic under the assumption that the cars can not be damaged. The four main physical quantities needed to simulate the motion of a rigid body are acceleration, velocity, position and orientation.

Acceleration The acceleration of an object is based on the forces that act on the object. Forces are represented with two vectors: one vector representing the direction and strength of the force and one vector representing the point where the force is applied on the object. The reason for having a vector for point of application is because the rotation of an object is effected differently depending on where the force is applied as will be shown in a later section. To get the acceleration of an object, the sum of all forces is divided by the mass of the object. This is the same as using the well know formula $F = ma$.

Velocity and Position Velocity has acceleration as its derivative so when the acceleration of an object is known we can numerically integrate over the time step to find out the new velocity of the object. Once we have the velocity we can numerically integrate to find the new position of the object since position has velocity as its derivative.

Orientation and Angular Effects Orientation is the most complex of all physical quantities since it consists of multiple sub quantities. First of all, orientation itself can be represented in two different ways. One way is to represent orientation by using a 4x4 matrix.

Orientation and Angular Effects is not complete and will be fully explained in another version of the report.

5.4 Collision Detection

Collision detection is necessary in every physics simulation since the results would not be believable without it.

Separating Axes Theorem One method of collision detection between objects is the so called *Separating Axes Theorem (SAT)*[33]. This theorem states that two convex, disjoint objects do not intersect each other if there exists a separating axis where the projections of both objects on the axis do not overlap. Given a shape A and a shape B the following are considered separating axes:

1. The normal of a face in shape A
2. The normal of a face in shape B
3. The cross product between an edge in shape A and an edge in shape B

SAT could be used to do exact collision detection but it is not very efficient when objects consist of many polygons. What you often do in reality is an approximation of an object's shape. Three common approximations are so called *Oriented Bounding Boxes (OBB)* , *Axis Aligned Bounding Boxes (AABB)* and *Bounding Spheres*[33]. An OBB is a minimal box that contains the entire object and is oriented according to the orientation of the object itself. An AABB is a minimal box that contains the object and is aligned with the x, y and z axes. This approximation is very fast but it can be very loose-fitted depending on the shape of the object Finally, a bounding sphere is a minimal sphere that will contain the object for every possible rotation. The advantage with using bounding spheres is that the bounding volume does not need to be recalculated whenever an object rotates unlike AABBs and OBBs.

Results In Crankshaft, AABBs are used when doing car-tree collisions and car-car collisions. This is because XNA contains a BoundingBox class that represents an AABB and it also contains methods for collision detection. Using the BoundingBox class was much easier than writing our own classes for bounding boxes and collision detection so there

$$j = \frac{-(1+e)v_1^{AB} \cdot n}{n \cdot n(\frac{1}{M_A} + \frac{1}{M_B}) + [(I_A^{-1}(r_{AP} \times n)) \times r_{AP} + (I_B^{-1}(r_{BP} \times n)) \times r_{BP}] \times n}$$

Figure 22: *The impulse formula for collision response*

seemed like there was no reason to reinvent something that was already at our disposal. However, the BoundingBox class cannot be used for car-to-ground collisions since the AABB of the terrain would be very loose fitted. For car-to-ground collisions, we use an OBB to represent the car and every frame the Z-value of every corner of the OBB is tested towards the Z-value stored in the height map. If the Z-value of any corner is less than the corresponding Z-value of the height map a collision has occurred.

5.5 Collision Response

When an object has collided with something in the game world, it should bounce off in some way or at least be prevented from penetrating the collision surface. This step is called *collision response* and is essential for a believable physics simulation.

Before one can calculate a collision response you have to find the collision normal[34]. Our height map is capable of returning a terrain normal given an x and y coordinate and this normal is used for car-to-ground collisions. For car-to-car and car-to-tree collisions we calculate the distance on the x/y plane between the two objects. The distance vector then becomes the collision normal.

Spring-based Collision Response When the collision normal has been determined it is finally possible to calculate a collision response. Collision responses can be modelled using spring physics and in this case you apply a spring and a damper force along the collision normal between the two colliding objects causing them to move apart[34]. The formula for a spring-based collision looks like this:

$$F = Nkd - bN(N \cdot V)$$

where F is the separating force, N is the collision normal, V is the relative velocity between the two objects, k and d are the spring dampener constants and finally d is the desired separating distance between the two objects. This gives a spring force that dampens the velocity along the collision normal and pushes the two objects apart.

There are a few drawbacks with using springs to simulate collision responses. One of them is that the effects of the applied force take time to propagate since a force does not effect the velocity or the rotation of an object instantaneously. This means that objects can penetrate each other regardless of a collision response. Spring constants have to be tuned in order to get the desired effect and they are also dependent on various environment variables such as gravity. All in all, using springs is not a very robust solution.

Impulse-based Collision Response Another approach to modelling collision responses is by using impulses[35]. Impulses can be seen as very large forces acting over a very short time span that cause an immediate change in an object's velocity and rotation. To calculate an impulse a restitution constant is needed. The restitution constant tells how much of the kinetic energy is absorbed in the collision. This constant has to be tuned to get the desired effect, however, it is independent of gravity and other environmental variables. The collision impulse j is a scalar value that is calculated using the formula in Figure 22.

- A and B are the two colliding objects.
- e is the restitution constant.
- v_1^{AB} is the relative velocity between the two objects.
- n is the collision normal.
- M_A and M_B are the masses of the two objects
- I_A^{-1} and I_B^{-1} are the inverse inertia matrices for the two objects
- r_{AP} and r_{BP} are the vectors between the centres of mass of the two objects and the point of collision P

For a full derivation of this formula, check [35] for details. The formula can be applied for collisions with objects that are meant to be stationary (such

as the terrain and trees) and this is done by giving these objects infinite mass and zero velocity. This will cause many factors to fall out of the equation.

Once the impulse has been calculated the new velocities of the objects can be calculated in the following way:

$$v_2^A = v_1^A + \frac{j}{M_A} n$$

$$v_2^B = v_1^B + \frac{j}{M_B} n$$

The impulse only acts along the collision normal and this is because the collision model is very simple and does not take friction into account.

The impulse also acts on the angular velocity of an object and it is recalculated in the following way:

$$w^A = I_A^{-1}(L^A + r_{AP} \times jn)$$

$$w^B = I_B^{-1}(L^B + r_{BP} \times -jn)$$

Results In Crankshaft, we decided to use impulse based collision response. This is because the solution did not seem to be robust at all. It seemed likely that we would want try different values for the gravity force to get the right feel of the game and having to tune the spring constants whenever gravity is changed seemed like too much of a hassle. Also, spring-based collision responses allow some amount of penetration which we did not want in the game.

5.6 Discussion

The physics of Crankshaft do not behave as well as we could have hoped for. The car does not really behave like a car at all, instead it resembles and ice cube sliding over the terrain. This is probably because we do not take tire friction into account and that is probably why the car is always skidding around. Another problem is that the rotational effects are heavily dampened. As it is now we apply a torque whenever the player steers the car and since the rotational effects have no friction it would cause the car to rotate infinitely if it weren't for the dampening effect. The rotational effects only manage to keep the car aligned with the road and not much else which makes the physics look a little dull. This could be solved by handling steering differently so that a torque is not applied when a players steers the car.

6 Network

The PC gaming industry is focusing more and more on multi player games, and although the area is already widely explored in the most basic setups, it still expands and evolves in a rapid pace. As graphics and game play improves by each year, so has the network systems also had to adapt to handle the new requirements, such as handling more players, higher precision and more data (for example from physics simulations).

The keyword for any network solution for a game is latency. Latency is the time it takes for a package of information to reach the recipient. Games with high latency has a slow and indirect feel to them, as well as making it harder to react to the information on the screen as it is always old information. In some games high latency means less (such as turned based games or strategy games) and in others it is of highest importance to the player experience (for example first person shooters and racing games)[36].

6.1 UDP and TCP

The first thing to decide upon when designing a networking solution is what protocol use. The two most common networking protocols today are UDP and TCP. In gaming both are used, but for different situations and games.

UDP (User Datagram Protocol) is a connection less protocol based on having small packages and almost no reliability, for example it does not guarantee delivery, or in-order delivery. The protocol is based on sending packages, and each package has a header that is 20 bytes large (IPv4). The body (user data) can be at most 65,507 bytes[37].

TCP (Transmission Control Protocol) on the other hand is a connection-oriented protocol which provides reliability, both ensured delivery and in order delivery. It provides these features at the cost of speed and latency. In actionbased, low latency games, UDP is most often used simply due to it is superior speed (and hence inherently lower average latency).

6.2 Game state

The game state is a collection of information that describes the game at a certain instance in time. For example in a racing game it is the players and their cars, and the location and orientation of those. In a first person shooter with physics it can be all the players as well as all the objects that can interact in the physics engine. Now, the problem is, how do we share this information between all the connected users. The two largest approaches to at all connecting systems is client/server and peer to peer (P2P).

Peer to peer assumes no central authority, and hence everybody has a copy of the game state and is responsible for updating everybody else on changes in the game state.

In a Client/Server architecture the server is the part responsible to keep the game state intact and the players only need to communicate with the server.

Modern games almost always uses a Client/Server approach since it greatly simplifies security and game state coherency. Though there are examples of attempts using P2P to speed up networking[38].

Since the bandwidth for any connection is limited so has the amount of data that is transferred by the game have to be limited too. A solution is to only send data at a fixed time step, say 30 times per second, which is a very common solution. The problem with this solution is that the client often under samples the game state. Say for example the video updates 60 times per second but the network only updates the game state 30 times per second which means the perceived frame rate is only 30 updates per second. There is however several solutions to this;

Interpolation We can use the previous two game state and interpolate between them to create a smoother effect. The downside with this is that it uses old data, and hence the information on the screen is always one or two frames old.

$$state_i = state_{i-2} * (1 - a) + state_{i-1} * a$$
$$a = dt/avg.frametime$$

Dead reckoning A technique that tries to solve the interpolation problem is dead reckoning[39]. In this technique we send the derivative to the variables too, which enables the client of doing a crude prediction as to where objects will be dt sec after the package has been received (as there is always a short time between the package being received and the information display, and as previously stated it may

even be over two frames).

$$state_i = state_{i-1} + state'_{i-1} * dt$$

Prediction This technique does the same thing as extrapolation, though instead of just using the derivatives to calculate the state it runs a full simulation on the objects involved. For example in a game with a physics engine this would mean running the actual physics engine on the client as well. This gives even better result, but at the cost of cpu power as running simulations often is very cpu intensive.

$$state_i = simulate(state_{i-1}, dt)$$

Grouping The grouping technique is usable when the game state is very large, an thus unsuitable to be sent everywhere all the time. The idea is to group players in smaller game state chunks, for example grouping players by geographical location. The technique is described in greater detail in "Using Groupings for Networked Gaming"[40].

```
if (newState.sequence <
    lastState.sequence)
  discard packet
else if (newState.sequence >
  lastState.sequence)
{
   lastState =
     deltaUncompress(
       lastState,
       newState);
   ackServer(lastState.sequence);
}
```

Figure 23: *Client network loop in Quake 3*

```
deltaCompressState(
  client.lastAckState,
  newState,
  &compressedState);

sendToClient(client, compressedState);
```

Figure 24: *Server network loop in Quake 3*

6.3 Quake 3 networking

The Quake 3 networking model solves a lot of the UDP reliability problems. There is only one type of server to client package in the Quake 3 networking model[41], the game state package. When the client receives a package it sends back an ack with the sequence number in to, bundled in the first package that is on its way over to the server. This means the server always knows the latest package which was received by the client. The use of this is that the server does not send the whole game state, but rather only the changed game state from the last acked package it knows. If a package is lost the same information is sent over and over until the client sends back an ack. The advantage is that the information is guaranteed to be delivered, at a very low (or none) latency cost. The server can just feed the same information over and over again till the client acks it, and the delta compression ensures the packages are relatively small. The client and server code turns out very simple as can be seen in Figure 23 and Figure 24.

6.4 Crankshaft

The networking solution in Crankshaft is based on a Client/Server architecture with interpolation at the client side. Both client and server is state based and thereby react differently depending on the state, for example if they are in the lobby state where players can join the game. The networking before the actual game start is not very critical in any aspect so the description that follows is limited to the actual game running state. There is however a crude layer above this that looks at the packages received and use them to detect disconnections (i.e. timeouts), which applies to all the states.

In the race state the client sends packages to the server at a fixed frame rate of 30 frames per second, and the server send information back in the same frame rate. In this state there is only type of message, for the client to server it is a package with input information, for the server to client package it is the game state. This type of packages are called a non-blocking remote procedure call (rpc)[42]. A non-blocking rpc can be seen as a command, issued on the client and run on the server. It is non-blocking in

the sense that the client does not wait for the server to complete execution.

Each client package consist of a header with a sequence number to prevent out of order packages and a body with input information to control the car. This package is a total of 18 bytes. With the 20 bytes UDP header it sums up 38 bytes, resulting in 1140 bps out from the client.

The server to client package also consist of a header with a sequence number, but the body consist of the game state with the orientation of all the cars in the game. The size of this package is $9 + 52n$ bytes where n is the number of players. This gives an output of $600 + 270 + 1560n$ bps (the UDP header is 600 bps), for example in the case of 4 players this is about 7 kbps.

The sequence number in the packages ensure that out-of order packages is not used, preventing snapping when the game first runs in state i, then in i-1 and finally in i+1.

Bytes	Field
1	Message type
4	Local id number
4	Sequence number
1	Break
4	Steering
4	Throttle

Table 1: *Crankshaft client to server package*

Bytes	Field
1	Message type
4	Sequence number
4	Number of players
For each player:	
4	Id number
12	Position
12	Direction
12	Speed
12	Up

Table 2: *Crankshaft server to client package*

6.5 Results

The network in Crankshaft is lightweight and fast, but unreliable and unsafe. The sequence numbering ultimately proved not very useful in non stress situations. From our own measurements (simply outputting the number of skipped packages per second) we concluded that out of order packages was almost never a problem in a LAN environment. The interpolation on the other hand did very good for the visual impression. Overall, the networking solution works very good with the task at hand, but would preform inadequately in a more stressful situation with a larger game state.

6.6 Discussion

In the end we had the choice of doing client side prediction or not, but choose not to. As noted in "Networked Physics"[42], the improving bandwidths of networking today decreases the need for prediction, as the information can be sent with higher intervals and more precision. Another argument is that prediction merely reduces the latency, but as the human reaction time is somewhere around 180-200ms[43], it is often better to add a small latency in favor of frame rate.

We also chose not to do anything to reduce the data flow, as it was already very low. Delta compression does not improve bandwidth usage at all when the game state is constantly changing (the cars are always moving). Grouping might save bandwidth if the number of players was high, or if the physics simulation incorporated more objects like barrels and destructible fences.

The security aspects of the game is basically nonexisting, but was considered out of scope for the task and timeframe.

7 Discussion

Our goals with the project was to develop a racing game which would be comparable to other modern racing games. This goal was deliberately high and we knew from the start that we were not going to achieve it, but we also knew that parts of the goal could be completed. And they were. Partly the graphics, although not the content, has a high standard, with the drawback that it is not as optimized as required by a modern game. The same goes with physics, which in its basics is a pretty rigid and true physics simulator, but proved hard to tweak to the desired behavior. Similarly the network worked, but we did not have the resources to test it outside a small lan environment or under stressful situations and thus we only knew it worked in the most perfect of worlds.

Our method of development and responsibility distribution worked very well for us. Our group had a low experience level among the majority of participants but quickly learned and master a large number of areas.

The biggest problems was optimizing the game, and tweaking the input/physics. The first problem meant the game was running at a very low frame rate, sometimes as low as 16 frames per second. The second problem meant that it was very hard driving through the map, which in turn meant it was not a very fun game to play, which of course is a major drawback for any game.

Overall we were happy with the result given the tight time frame, but at the same time we would very much have liked to work more with it and removing some of mentioned problems.

References

[1] Dataspelsbranschen, "Spelförsäljningen under år 2007," 2007. http://www.dataspelsbranschen.se/statistics.aspx.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston, MA: Addison-Wesley, January 1995.

[3] R. Fine, "Enginuity," *GameDev.net*, 2003. http://www.gamedev.net/reference/programming/features/enginuity1/.

[4] J. Tulip, J. Bekkema, and K. Nesbitt, "Multi-threaded game engine design," in *IE '06: Proceedings of the 3rd Australasian conference on Interactive entertainment*, (Murdoch University, Australia, Australia), Murdoch University, 2006.

[5] "Model class," *Microsoft MSDN*. http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.model.aspx.

[6] K. Channa, "Light mapping - theory and implementation," *flipcode.com*, 2003. http://www.flipcode.com/archives/Light_Mapping_Theory_and_Implementation.shtml.

[7] L. Hodorowicz, "Advanced lightmapping," *flipcode.com*, 2001. http://www.flipcode.com/archives/Advanced_Lightmapping.shtml.

[8] H. Gouraud, "Continuous shading of curved surfaces," *IEEE Transactions on Computers*, 1971.

[9] B. T. Phong, "Illumination for computer generated images," *University of Utah, UTEC-CSs-73-129*, 1973.

[10] "Phong shading," *wikipedia.org*. http://en.wikipedia.org/wiki/Phong_shading.

[11] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *University of Utah*, 1977.

[12] A. F. Andrew Woo, Pierre Poulin, "A survey of shadow algorithms," *University of Toronto, Ontario, Canada M5S 1A4*, 1990.

[13] A. V. Nealen, "Shadow mapping and shadow volumes," *devmaster.net*, 2005.

[14] F. C. Crow, "Shadow algorithms for computer graphics," *University of Texas at Austin*, 1977.

[15] T. A. Möller and U. Assarsson, "Approximate soft shadows on arbitrary surfaces using penumbrawedges," *Department of Computer Engineering, Chalmers University of Technology, Sweden*, 2002.

[16] A. F. Andrew Woo, Pierre Poulin, "Casting curved shadows on curved surfaces," *Computer Graphics Lab New York Institute of Technology Old Westbury, New York 11568*, 1978.

[17] F. Zhang, H. Sun, L. Xu, and L. K. Lun, "Parallel-split shadow maps for large-scale virtual environments," *Department of Computer Science and Engineering The Chinese University of Hong Kong*, 2006.

[18] W. Donnelly and A. Lauritzen, "Variance shadow maps," *Computer Graphics Lab, School of Computer Science, University of Waterloo*.

[19] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering antialiased shadows with depth maps," *Pixar San Rafael, CA*, p. 284, 1987.

[20] W. T. Reeves, "Particle systems - a technique for modeling a class of fuzzy objects," *Computer Graphics 17:3 pp. 359-376*, 1983. http://portal.acm.org/citation.cfm?id=800059.801167&coll=portal&dl=ACM&CFID=65087275&CFTOKEN=20160208.

[21] S. Drone, "Real-time particle systems on the gpu in dynamic environments," *International Conference on Computer Graphics and Interactive Techniques held in San Diego, California. SESSION: Course 28: Advanced real-time rendering in 3D graphics and games pp.80-96*, 2007. http://portal.acm.org/citation.cfm?id=1281500.1281670&coll=portal&dl=ACM&CFID=65087275&CFTOKEN=20160208.

[22] N. Glasser, "Texture splatting in direct3d," *Gamedev*, 2005. http://www.gamedev.net/reference/articles/article2238.asp.

[23] I. Ernst, H. Rüsseler, H. Schulz, and O. Wittig, "Gouraud bump mapping," *Siggraph*, 1998. http://delivery.acm.org/10.1145/290000/285311/p47-ernst.pdf?key1=285311&key2=6431788021&coll=GUIDE&dl=GUIDE&CFID=25111382&CFTOKEN=92099845.

[24] N. Tatarchuk, "Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, (New York, NY, USA), pp. 81–112, ACM, 2006.

[25] V. Young, "Programming a multiplayer fps in directx: Culling," *"Gamasutra"*, 2005. http://www.gamasutra.com/features/20050411/young_01.shtml.

[26] D. Picco, "Frustum culling," *flipcode.com*, 2003. http://www.flipcode.com/archives/Frustum_Culling.shtml.

[27] T. Akenine-Möller and E. Haines, *Realtime Rendering*. A K Peters, 2002.

[28] K. Hoff, "A faster overlap test for a plane and a bounding box," 1996. http://www.cs.unc.edu/~hoff/research/vfculler/boxplane.html.

[29] "State changes," *Circlesoft.org*, 2006. http://www.circlesoft.org/pages.php?pg=kbasepage&id=12.

[30] P. Scott, "Shader model 3.0, best practices," *nvidia.developer.com*, pp. 9–10. ftp://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_SM3_Best_Practices.pdf.

[31] G. Fiedler, "Integration basics," 2006. http://www.gaffer.org/game-physics/integration-basics.

[32] G. Fiedler, "Fix your timestep!," 2006. http://www.gaffer.org/game-physics/fix-your-timestep.

[33] S. Hadap, D. Eberle, P. Volino, M. C. Lin, S. Redon, and C. Ericson, "Collision detection and proximity queries," *Siggraph*, 2004. http://delivery.acm.org/10.1145/1110000/1103915/cs14.pdf?key1=1103915&key2=2964509021&coll=GUIDE&dl=ACM&CFID=65266420&CFTOKEN=94163159.

[34] G. Fiedler, "Spring physics," 2006. http://www.gaffer.org/game-physics/spring-physics.

[35]

[36] J. Watte, "Gamedev.net multiplayer and network programming forum faq," *GameDev.net*, 2005. http://www.gamedev.net/community/forums/showfaq.asp?forum_id=15.

[37] J. Postel, "RFC 768: User datagram protocol," Aug. 1980.

[38] O. A. Abdelwahed, "Distributed gaming," *GameDev.net*, 2003. http://www.gamedev.net/reference/articles/article1948.asp.

[39] J. Aronson, “Dead reckoning: Latency hiding for networked games,” *Gamasutra*, 1997. `http://www.gamasutra.com/features/19970919/aronson_01.htm`.

[40] J. Aronson, “Using groupings for networked gaming,” *Gamasutra*, 2000. `http://www.gamasutra.com/view/feature/3158/using_groupings_for_networked_.php`.

[41] “The quake3 networking model,” `http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking`.

[42] G. Fiedler, “Networked physics,” 2006. `http://www.gaffer.org/game-physics/networked-physics`.

[43] “Reaction time,” *Wikipedia*. `http://en.wikipedia.org/wiki/Reaction_time`.