CHALMERS

Speed Junkies

The development process of a graphically intense racing game

Alexander Lissenko
William Axhav Bratt
Johan Ekdahl

Department of Computer Science
Chalmers University of Technology
Gothenburg, Sweden 2014

Speed Junkies
The development process of a graphically intense racing game

ALEXANDER W. G. M. LISSENKO
G. WILLIAM AXHAV BRATT
JOHAN R. K. R. EKDAHL

Examiner: B. ARNE LINDE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

In this thesis we present a number of different components, that are parts of developing a graphically intense racing game. The thesis will discuss the development process, with respect to graphics, gameplay and multiplayer feature. The project is developed in a C#/OpenGL environment, where all graphical implementations are custom developed. The development resulted in a deferred renderer with several post processing effects, as well as a multiplayer feature with network compatibility. Ultimately, we discovered that proper research is essential in the development of a graphical application. The development process and the final result is discussed and evaluated thoroughly.

Acknowledgements

We would like to thank our supervisor Ulf Assarsson for his invaluable support and input. We would also thank John Chapman for his intuitive tutorials and in depth discussions on common post processing effects, that helped us understand and implement many of them. We would also send special thanks to all the students who attended the Advanced Computer Graphics course 2014, who gave us so many great ideas and the motivation to see the project through. Furthermore, we would like to thank two former group members. Lastly, we would like to thank *Café Bulten* for providing the best coffee on campus.

The Authors, Gothenburg 2014-06-03

Contents

1

Introduction

Video games, alongside blockbuster movies and commercial advertising, have been one of the most powerful accelerators in computer graphics technology. However, while a movie uses offline rendering, thus having no time constraints per rendered frame, a video game has strict limitations in rendering performance. A video game needs to be rendered in real time in order to run smoothly, which also implies that the rendering needs to be done on the consumer's machine, rather than a powerful computer built for rendering. Consequently, video games have been the main motivator for optimization and alternative techniques within the field [1]. The video game industry have greatly influenced the need for more powerful machines in the homes of consumers.

Developing a video game is however not just something a small team of programmers construct over a short period of time; a typical AAA game [2] normally involves hundreds, sometimes even thousands, of not just programmers, but also artists, story creators and many other employees. A typical AAA game takes several years to develop and together with the huge number of people involved, it's implied that there is a lot of money within this business. AAA games have a dominant place in the video game business, but lately the market for so called Indie games has increased noticeably. Unlike the AAA game developers, an Indie game is generally developed by a much smaller team of developers, with a lot less money involved. Typically, an Indie game starts as a hobby project and if the game is successful enough, it then can reach the market and generate income to the developers. Minecraft is probably the best possible example of what started as a hobby project by the game developer Markus Persson (a.k.a. Notch), which turned out as a major success[3].

As a game developer, you face a lot of different challenges. The most basic challenge could be the game idea - since games are primarily meant for the purpose of entertainment, the game needs to be satisfying to play. Depending on in which genre the game is intended, the requirements might differ. If the game is meant as a Role Playing Game (RPG), the story and the characters might be the most crucial elements to focus on, while in for instance a car racing game, the car models and the feeling of speed might be the main focus. There are a lot of different elements to take into account in order to give the game an entertaining gameplay, and these can differ a lot between the genres.

Another challenge, which has become especially in focus lately, is the visual appearance of the game, mostly referred to as the *graphics*. Graphics in video games has evolved fast, resulting in games that look more close to the real world than ever before. Since the real world is complex, it naturally follows that realistic computer graphics is complex to implement. However, as mentioned above, the hardware in computers and game consoles has improved a lot, and keeps improving. With these improvements, graphics algorithms that used to be too complex to compute in real-time are gradually implemented and optimized for real-time.

1.1 Purpose

The purpose of this project is to perform a case study on developing a graphically intense racing game, with a small development team over, a four and a half month period. However, the game should not just be considered as a demonstration of different graphics algorithms. The game should also contain several elements to increase the users experience of the game; the game should be entertaining to play. Such elements include a natural sense of speed and realistic steering of the car, as well as a multiplayer feature.

1.2 Aims

The project aims to gain understanding of different graphical algorithms, by implementing them. Another aim is to gain insight into the development process of a game, as this includes more than just visual effects. Other important aspects include gameplay and user experience. Also, gaining experience in working in a collaborative environment.

1.3 Problem Statement

The problem consists of developing a game. However, the game can be broken down into parts, i.e. Graphics, game logic and multiplayer; there are separate problems for each part. Each of these parts contains sub-problems, a selection of the most essential issues is as follows:

Graphics

- Which type of rendering algorithm is desired in a racing game?
- Modelling is a time consuming process. Can modelling your own models be justified compared to make use of third-party models?
- Which effects should be implemented, in order to obtain significant visual results?

Game logic

- Which of the major graphics libraries are best suited for this project?
- What are the required additional features to add entertainment value to the game?

Multiplayer

- What are the options in network architecture, for creating a network multiplayer game?
- What can be done to reduce lag?

1.4 Method

In the very beginning of this project, three main framework alternatives were considered: OpenGL combined with C++, XNA framework combined with C#, or an OpenGL wrapper (OpenTK) combined with C#. The OpenGL/C++ approach has the advantage of being platform independent as well as, since this approach has been used for a long time, being well documented. However, since we already had some experience of this approach, in addition to that we consider this approach somewhat complex, we had the ambition to use another approach. The XNA/C# approach was, during the approach evaluation process considered the most effortless approach. The XNA framework is widely supporting several different libraries, making physics integration in the game engine possibly simple. However, the XNA framework is exclusively targeted for Microsoft platforms, such as PC and Xbox, as well as not being developed any further[4]. Thus, the XNA/C# approach was rejected.

The chosen approach was instead an OpenGL/C# based, with the open source toolkit and OpenGL wrapper OpenTK. We had prior experience of C++, sometimes being somewhat complex. Thus, developing the game engine in C#, together with the already familiar OpenGL library was considered the best approach. With this approach, the simplicity acquired from the C# programming language is taken advantage of, as well as the final product could possibly be platform independent. Also, since OpenTK is developed as a replacement to XNA, adding libraries developed for XNA is simple.

To develop the game, *Agile Development* was considered and adapted as a development method. The *Agile Development* method chosen was an own, somewhat informal interpretation of *Scrum*, as it is considered to be one of the standard methods in the industry. However, *Scrum* was rejected in a later stage of the project, as the group size decreased to only three.

1.5 Limitations

With respect to the somewhat short development time, the game is implemented exclusively for a PC environment, as a relatively small demo application. However, since the game engine is developed with the open source and platform independent toolkit OpenTK, adding support for other environments would still be possible in a later stage of the project. Also, since this project aims to evaluate different techniques within computer graphics, the primary focus will be on implementing different graphics algorithms. Thus, rather than implementing an own physics simulation and collision detection, already existing implementations will be used. Choosing between different algorithms to achieve a graphical effect will be done with respect to the algorithm's simplicity and compatibility with other implemented algorithms.

Furthermore, optimizing the implemented algorithms will not be considered as long as the game runs somewhat smoothly. Since the game is considered a demo, only one car model and one racing level will be implemented, enhancing the focus on the different implementations of the graphics algorithms.

1.6 Outline

This thesis will describe the development process of the game engine, called the *Turtle Engine*, developed for the game *Speed Junkies*. The thesis will first discuss aspects in the non-graphical parts of the game, in the *Game Engine* chapter. Concepts such as multiplayer and power ups will be discussed, and the implementation choices will be motivated, together with a description of the result.

Following is an intensive chapter, the *Graphics* chapter, covering the graphical aspects of the game. Here, the theory behind the rendering algorithms, together with the projects approach to *Post Processing effects* are described. An overview of the modelling process is presented, as well as the *Turtle Engine* custom developed particle system.

Lastly, the overall results will be discussed in the *Discussion* chapter, together with some remarks for future work.

2

Game Engine

PRIOR TO the discussion of graphics, another important aspect of game development must be discussed, namely the Game Engine. The Game Engine is a larger abstraction, containing everything from gameplay rules and graphics to physics and networking. The following chapter will cover the design of the Game Engine, excluding the graphics implementations.

2.1 Gameplay

Video games, as well as board games, are primarily developed with the purpose of entertaining the user. In video games, there are several approaches to enhance the users perceived entertainment, approaches that tend to vary from different game genres. For instance, in a *First Person Shooter* (FPS) game, simulating a real weapon can significantly increase the overall experience, since the shooting is a crucial feature of the game.

In a racing game, there are a number of ways to enhance the experience. Examples of such features are: an interesting visual style and acceptable simulation of driving mechanics. However, features that are not present in a real race can also increase the user experience; an example of this kind of feature is the concept power ups. Power ups gives the user a temporarily advantage, adding more variation to the game.

Adding a multiplayer feature was considered crucial. Traditionally, there are two main solutions to implement multiplayer functionality: network based multiplayer or local multiplayer, also referred to as split screen multiplayer. Network based multiplayer can be implemented either to communicate over the local network, or to a global server. Traditionally, network multiplayer has been mostly present on computer games, while split screen has been the main method on consoles. In the *Turtle Engine*, which is developed for a PC environment, the multiplayer is implemented over the network, as described in the Network chapter.

In most well known racing games, there are typically several different race tracks and other cars that are unlocked as the player advances in the game. This was not considered achievable in this project, as the development time is limited to less than five months. Inspired by Nintendos successful racing game series *Mario Kart*, the idea of *power ups* was considered. *Power ups* are different advantages the player can obtain during the race, involving effects such as speed boost and weapons to fire at an opponent.

2.1.1 Results

By introducing power ups in the *Turtle Engine*, more variety is added to the races. Power ups give a more dynamic element to an otherwise static and predictable race. In the *Turtle Engine*, four power ups are implemented in the *Turtle Engine*: *Speed boost*, a *Missile*, *Lights out* and *Smokescreen.* The power ups are obtained as the player drives through a box on the racing track, triggering an event that summons a power up, the type of power up obtained is determined randomly. The power up is stored until the player chooses to activate the power up, by pressing the P key.

The *Speed boost*, when activated, increases the cars maximum speed for a limited period of time. Thus, the player can overtake the opponent, but in addition the car is more difficult to maneuver.

The *Missile* works by spawning a high velocity body in front of the car, that will travel in a straight line, hopefully pushing other cars off the road; as its inertia is significantly greater than the cars.

Lights out turns off the directional light for a short period of time, forcing the players to solely rely on the spotlights of the car for navigation. The *Smokescreen* causes the exhaust to increase the smoke generation for a short period of time, creating a visual obstacle for other players to pass through.

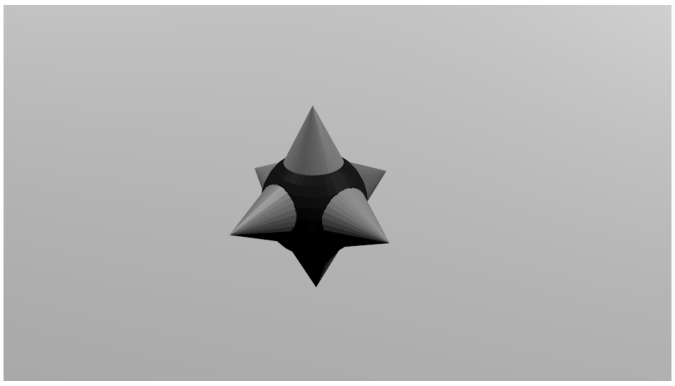


Figure 2.1: Image displaying the *missile Power up* rendered in *Blender*

2.2 States and Menus

An important factor of a game is to have a point of origin. This is in most cases a start menu. From here, the user will have the ability to choose if he wants to play a single- or multiplayer game as well as the ability to change the settings. The PC platform varies from machine to machine in terms of hardware. Thus, the user should have the ability to customize the graphics settings.

An approach using Windows Forms and GLControl was considered, but was rejected in favor of a state based solution. GLControl was considered unnecessarily complex, since only a fairly simple text based menu was desired. Another advantage of a state based approach, is the possibility of using the *Turtle Engine's* graphical features in the menu. For instance, a scene rendered in real time could be used as a background, in the menu, as well as including several graphical effects to enhance the users experience of the menu.

The design is based around two separate states. One for the game, which only accounts for gameplay and rendering of gameplay. The other state is the menu state, it manages the main menu and submenus.The application is either in the game state, or the menu state. However, since modularity was kept in mind during the development of the state based system, adding additional states is considered a simple task.

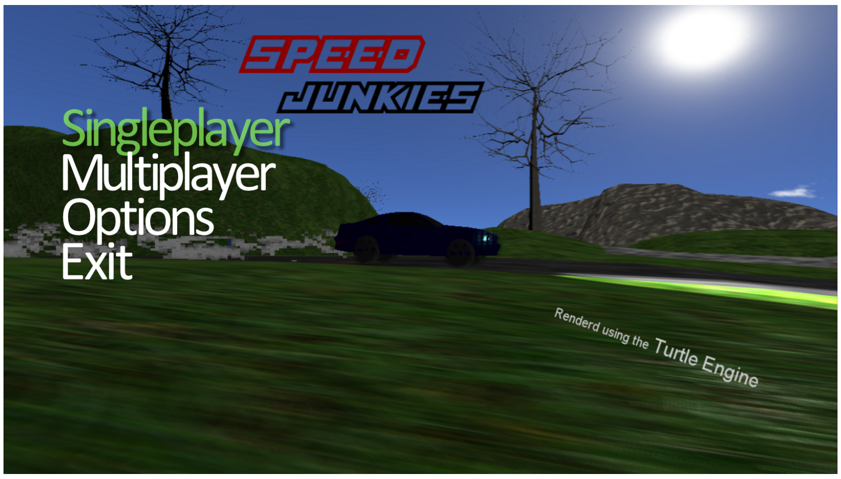


Figure 2.2: Image displaying the final result of the menu of the game

2.3 Physics & Collision Detection

A common application of a robust collision detection system is physics simulation. For a car racing game this is not strictly necessary, but it can make the gameplay significantly more interesting and enjoyable. With physics simulations, desirable effects such as cars bouncing off each other in the event of a collision can be implemented. When the bounding volumes of two objects overlap, realistic behaviour can be calculated taking the respective mass and velocity of each object into account. This is achieved with the help of the collision detection system, that will be discussed in the following segment.

A vital component of any simulated game is collision detection. Collision detection can be very difficult to implement properly and efficiently. However, there are numerous existing implementations. Since this project primarily focuses on graphics, the third party library *BEPUphysics*[5] was chosen. Other libraries were considered, such as *Bullet*[6], but were ultimately rejected due to poor reputation, lack of documentation or lack of the support for OpenTK. Another important part is physics, which BEPU also handles.

2.3.1 Implementation

In the game engine, BEPUs automatic handling of bounding volumes, collisions and physics is utilized. Bepu also manages phase collisions, that allows for events to be fired when two specific entities collide. For instance there is an extra force applied to the car, to make it go faster than the wheels can provide on their own. However, if no wheel touches the terrain, there will be no extra force applied to the car hull.

In this project the cars are simulated as dynamic objects, with weight, that reacts to each other and the ground. However, the ground is static. Thus, the ground will simply push away any dynamic entity, regardless of its weight.

The camera is also handled by BEPU. With the help of the built-in raycast methods, creates a possibility to shoot rays back from the camera to make sure it is not behind any terrain.

2.4 Network

As proposed in the gameplay section, allowing network multiplayer adds replayability to the game since when competing against others every race might, and probably will look entirely different. Thus, providing the option for multiplayer over the network in the *Turtle Engine* is desired in order to improve the perceived gameplay.

Traditionally, network multiplayer games are played over peer-to-peer networks, where the user is connected directly to the opponents IP-address, alternatively having a dedicated server managing all clients. The *Turtle Engine's* netcode, however, uses a twist on peer-to-peer clients, a multicast network. On program startup, the *Turtle Engine* connects to a predefined multicast network, thus foregoing the struggles with typing in IP-addresses and dedicated server management. If the user wants to play a multiplayer session with others on a local network, he simply clicks *Join Multiplayer Game* and is then forwarded to the current session on the local network.

So how does this work? During the loading of graphical content, the multicast sockets are also initialized. Currently predefined for the *Turtle Engine* is the multicast address 233.234.234.234, the ad-hoc multicast block[7], on port 11245. To not interfere with current applications on this address, a fairly low Time To Live (TTL) value is set. The TTL value is set to 1 or 2, depending on the local network structure. When joining a network session, the program scans the open socket for two seconds. If any data was received within that timeframe that can be parsed to game data, a connection attempt will be done. The current leader of the network will send a network ID for the client and the client will start broadcasting its data. For every new ID that is received by the client, a new player model will be loaded into the session. In the event of a disconnect, this data will be unloaded.

Despite the network is running on a client to client basis, the network makes use of a leader to manage new connections to the network. The leader election works in the following way: if the client did not connect to a network on startup, the client will start a new one and choose itself as a leader. When the leader disconnects, the successor is chosen, depending on the users connected and leadership will be transferred.

There are drawbacks using a non server based network implementation, as it requires more of the individual clients; each client has to do local calculations to reach a synchronous state. In certain game genres this works by default, for instance in Real Time Strategy games, where units are following set paths. However, in the *Turtle Engine* there is an element that brings a non consistency, BEPU. A physical calculation always yields the same result, given that the same parameters are put into the calculation. Because of network restrains, this will require too much data to be transferred between clients. However, just ignoring this problem will cause the clients to de-synchronize. Thus, resulting in a bad user experience, especially for a racing game. Sending a limited amount of information, for instance only current position and orientation, will give origin to what is commonly known as lag. A way to counter this is by introducing prediction. In a prediction based model, the clients are given enough data to run local simulations. Thus, all the clients will do local calculations with that data, until the next broadcast is sent.

3

Graphics

As stated in the introduction chapter, the main focus of this project is the graphical aspects. In this chapter, the theory behind the basics in computer graphics is briefly discussed together with the alternative solutions that are available, followed by more extensive theory and results of the main methods, implemented in the *Turtle Engine*.

All surfaces in a 3D scene in a graphics application are made up of a large set of primitives, such as triangles. The more triangles present, the more detailed the scene is. *The Graphics Rendering Pipeline*, hereby referred to as the *pipeline*, is considered to be the core component in the real time computer graphics field, and is a representation of the different stages that are present in real time rendering. The pipeline consists of three main stages: the *Application Stage*, the *Geometry Stage* and the *Rasterizer Stage*.

The *Application stage* is, as the name implies, driven by the application itself. Thus, the developer has full control of what happens in the Application stage. In the *Application stage*; processes such as collision detection and optimization algorithms are implemented, then forwards geometrical primitives to the *Geometry stage*.

In the *Geometry stage*; methods such as transformation of the primitives, from the Application stage, are implemented. Lighting computations for each vertex are also performed in this stage, in a vertex shader. The primitives are then, after the transformation and lighting computations, mapped to the screen.

The *Rasterizer stage* converts the primitives, from the *Geometry stage*, into visible pixels on the screen. The color of each pixel is calculated using a fragment shader. Visibility aspects are also solved here, by sorting the primitives in the direction of the depth.

In the following chapter the rendering algorithm will be described, together with the results generated by the engine.

3.1 Modelling

Models are a crucial part of any graphics application, as without models there are nothing to view on the screen. However, the modelling process can be considered complex and time consuming. In this section, the choices regarding the modelling process will be discussed, as well as the method of loading models into the application.

3.1.1 The Modelling Process

When creating 3D models, there are several different softwares available. Softwares that were considered in this project were *Blender*, *3D Studio Max* and *Maya*. *Blender*, unlike *3D Studio Max* and *Maya*, is open source, thus being a suitable alternative for a non-professional modeller. However, *3D Studio Max* and *Maya* are possible to use under a student license, thus also being suitable for this project.

The modelling of the car was done in *3D Studio Max*, using tracing with respect to reference images. The game environment was modelled in *Blender*, by sculpting a plane. However, the end result looked flat. Thus, adding vegetation was considered, comparing the time for modelling new models against the visual quality they would provide. It was decided that the environment was unpassable in its unmodified state, so the models where made. A simple cactus model as well as a script generated tree model was introduced, improving the visual appearance of the game environment.

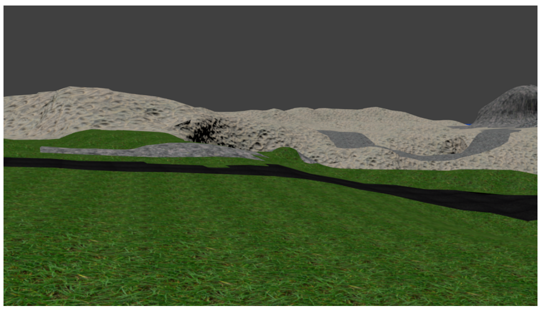


Figure 3.1: Terrain without vegetation

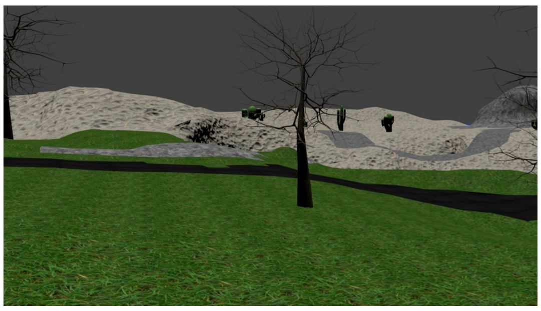


Figure 3.2: Terrain with vegetation

3.1.2 Model loading

When the models are finished, they need to be imported into the application. There are several ways of storing 3D object information, notable examples for games are: *MD3*[8], ID Software's *Quake 3* model file format and *MDX* developed and used by Blizzard. These formats are complex and stores detailed information about the models, even animation properties. For static models there are different standards depending on the modeling software that is used, as well as a few universal standards, such as the *OBJ* format. During the development process of the *Turtle Engine*, several file formats were considered. However, due to the simplicity together with respect to the fact that two different modelling softwares were used, the *OBJ* format was decided upon. Since the only components in need of animation are the wheels, only limited animation is necessary. Fortunately, the physics library BEPU has support for simple animations, such as animating the wheels. Thus, no other animation implementation is needed.

The *OBJ* format is based on plain ASCII text, thus easily read with a plain text editor. The data is stored with specific tags, followed by floating point or integer values[9]. The tags are as follows:

- *v* is the vertex tag, followed by three floating point values, describing the position of the vertex.
- *vt* is the texture tag, followed by two floating point values, describing the UV mapping.
- *vn* is the normal tag, followed by three floating point values, describing the direction of the normal.
- *f* is the face tag, followed by three sets of three integer values, combining the vertex, texture and normal tags together. Thus, a face is created, which in OpenGL is the triangle primitive.

However, OpenGL does not have the capabilities to handle the triangles likewise as the *OBJ* format. The *OBJ* format indexes the vertices, texture coordinates and normals with three separate pointers, where OpenGL only has the ability to use one[10]. Thus, without unfolding some or all of the data, the *OBJ* format can not directly be imported to OpenGL. A drawback of this, is that the vertexbuffer array size will be increased compared to the *OBJ* file size. However, with the current implementation of the game engine, the models are not large enough for this to be an major issue, and therefore it is acceptable.

3.2 Bump Mapping

In a graphically intense application, well detailed geometry can significantly increase the perceived realism. However, adding more detail to the geometry also implies adding more triangles to render as well as making the modelling process more complex and time consuming. Thus, adding more detail is a major drawback in terms of performance drop and overall development time.

Bump mapping is a technique commonly used in computer graphics in order to easily obtain more detail at an already rendered surface, i.e. by adding bumps and other deformations to the surface. Bump mapping, unlike the more traditional method displacement mapping[11], does not deform the actual model. Instead, only the normals of the model are modified, creating an illusion that the model is deformed. By modifying the normals rather than deforming the model, bump mapping is a much faster technique than displacement mapping and therefore a suitable technique in real time applications, such as games. Despite being a simplification, the final result from bump mapping is visually equal as displacement mapping.

Within the bump mapping technique, there are two main methods. The first method uses a height map to simulate the displacement at the surface of the model, yielding the modified and desired normal. This method was invented by Blinn[12] and works as follows:

1. Find the height map that corresponds to the surface's position
2. Calculate the surface normal to the height map
3. Combine the surface normal from the previous step with the actual (geometric) surface normal, yielding a combined normal that points in another direction.
4. Calculate the interaction between the intended bumpy surface and the lights in the scene, using a reflection model.

The result is a surface that appears to have bumps and other deformations, but actually just has a texture that tricks the eye.

The other method, also the method that is used within this project, is called normal mapping[13]. The normal map is a texture map that contain all the modified normals from each points of the surface. By specifying the normals rather than deriving them generally creates more predictable results, which is desirable since the normals are easier to work with. Thus, this method is considered the most commonly used within bump mapping.

The normal map method can as well be implemented in several different ways. The most correct and most referred to is called tangent spaced normal mapping, and does all necessary calculations in the so called tangent space. This method creates a tangent and a bitangent for each normal, thus spanning a space; the tangent space. From the vectors defining the space, it is possible to set up a matrix, a TBN-matrix, for normal transformations between this tangent space to the desired model space, yielding a physically correct result. Another method is called object spaced normal mapping, and is an approximative method that, despite not being physically correct still yields as pleasing results as with the correct tangent spaced method. Object spaced normal mapping is so simple that all necessary calculations can be done in relatively small fragment and vertex shaders. The main difference between these two methods is that the object spaced method does not support deformation of the objects, which is not considered an issue in this project.

3.2.1 Results

Tangent spaced normal mapping was, despite the method's complexity, chosen for this particular project. Tangent spaced normal maps are considered to be the most common implementation within the industry. When generating the environment normal map texture in Blender, exporting the texture in tangent space is the default option. Even though object spaced normal mapping would yield approximative results pleasing enough for this project, tangent spaced normal mapping was found to be more well documented than the object spaced method.

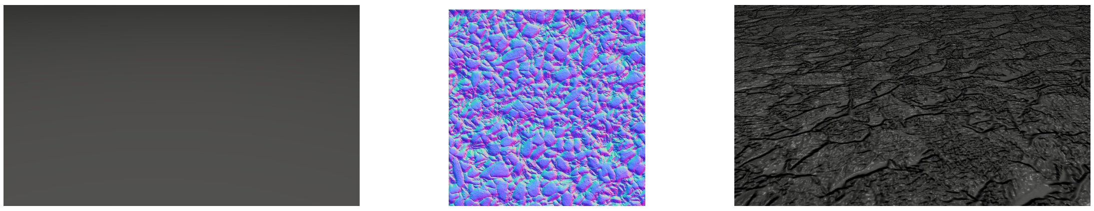
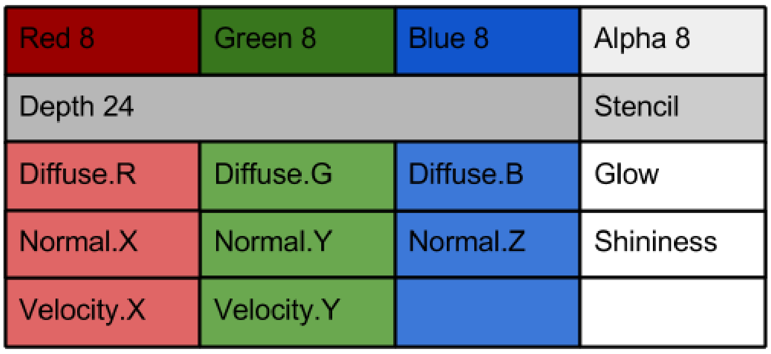

(a) Plane without bumpmap **(b)** Bumpmap **(c)** Plane with bumpmap

Figure 3.3: Figures a to c compares a plane with and without bumpmap, b is the bumpmap that is applied

3.3 Deferred Rendering

Deferred rendering, which is an optional technique to the standard forward rendering technique, was first propounded in a paper about optimizations of *Very-Large Scale Integrators* by a research group from *Schlumberger Palo Alto Research, CA*[14]. In a standard forward rendering engine, one would have to calculate all lighting for every object. The deferred approach, however, renders every object to a *G-Buffer*[15], through a geometry pass. The *G-Buffer* is created by rendering the geometry to several textures. These textures will be used to store essential information about the scene. The *Turtle Engine* stores diffuse(Figure 3.7a), normals (Figure 3.7b), depth (Figure 3.7c) and velocity (Figure 3.7d) during its first render pass. These variables are stored as colors in a texture, for instance: normal x-,y-,z-positions are stored as the red, green and blue components in the texture, as can be seen in figure 3.4. One thing to note is that textures can only hold values between zero and one. To counter this, a rescaling is done to get the components into *texture space*.



Red 8	Green 8	Blue 8	Alpha 8
Depth 24			Stencil
Diffuse.R	Diffuse.G	Diffuse.B	Glow
Normal.X	Normal.Y	Normal.Z	Shininess
Velocity.X	Velocity.Y		

Figure 3.4: The projects' approach to the *G-Buffers*, image inspired by *Guerilla Games Deferred Renderer* presentation[16]

The second pass is the lighting pass. Now that all the information about the geometry is stored, the *G-Buffers* are accessed to perform lighting calculations. This is done by drawing a fullscreen quad over the screen. By accessing the coordinates, one can do per pixel lookups in the *G-Buffers*. The stored buffers are used to emulate three types of lights. A directional light can be compared to sunlight as it is cast over the entire scene. Point lights are another example of light that can be emulated, these lights can be described as orbs of illumination as they illuminate 360 degrees around its location. These lights are also described by an attenuation, so the further away an object actually is from the light source this specific light source will contribute less and less light. The third and final type of light, that the engine supports, are spotlights. These contain the same properties as point lights but with a limited frustum.

Attaining the world space position can be achieved in two ways. The first option is to use an extra *G-Buffer* to store the position. However, by using a depth texture, which is needed for several postprocessing effects, the screenspace position can be reconstructed using the texture coordinates of the quad and the depth. The code required for the reconstruction, figure 3.5, will generate the positions as seen in figure 3.6.

```
vec3 getPosition(vec2 pos, float depth)
{
    vec4 screen = vec4((pos.xy*2)-1, (depth.x*2)-1, 1.0);
    vec4 worldPosUnHo = projectionMatrix * screen;
    return worldPosUnHo.xyz/worldPosUnHo.w;
}
```

Figure 3.5: The reconstruction code of a coordinate using only depth and texture coordinates.

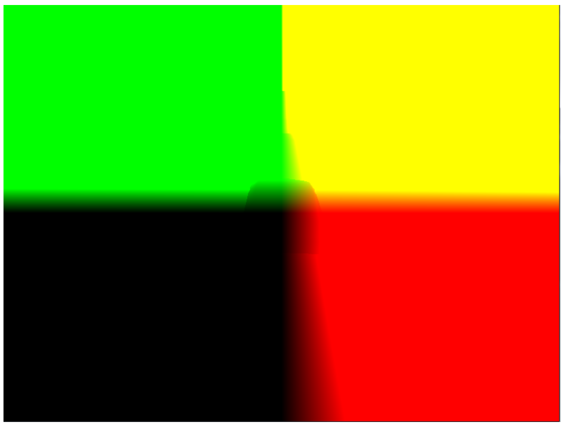
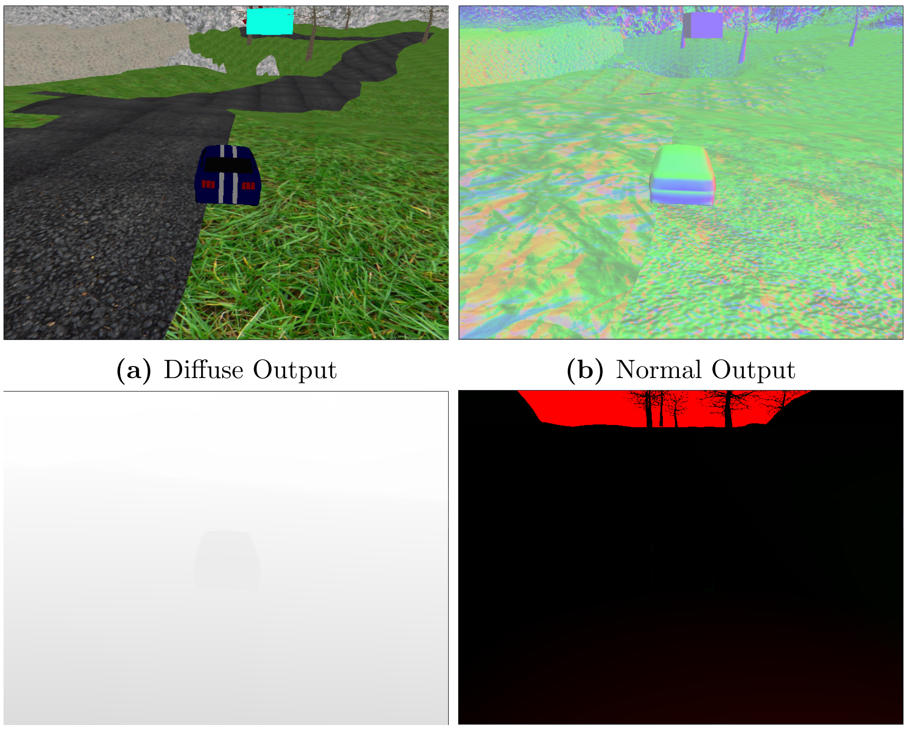


Figure 3.6: Result of the depth reconstruction.

The deferred rendering process is, however, not the best technique and does contain some disadvantages. For instance, the *G-buffers* are a large flaw with this approach. The more information that has to be accessed in later rendering steps, the more textures will be needed. Also, not compressing the information as tightly as possible will also increase the total buffer size. An example of this is the shininess factor. A single float is needed to store this information. Therefore, acquiring a stand-alone buffer for this data is a waste of memory. Thus, storing this float in the normal buffer as the already unused alpha channel saves a separate buffer, preserving the ability to access the shininess in the lighting stage. Bandwith issues on more constrained systems can be solved using other rendering techniques, for instance *Clusterd deferred forward rendering*[17].

Another drawback of using a deferred renderer, as opposed to a forward render, is the inability to render transparent objects.[18] This is because when the models are projected to the rendered buffers, the information about objects that are behind the currently visible objects is lost. There is no way to restore this lost data in later stages, without using other techniques. Likewise, the information about edges is lost as well. Hence, the ability to do standard anti-aliasing is lost. Fortunately, there are solutions to these problems; one possible solution is the implementation of *deferred lighting*. With this technique, the rendering process is separated to three parts. First, a *G-Buffer* pass where necessary information for the following lighting pass is gathered. Secondly, lighting using this data is computed and stored into a diffuse and specular texture. Next a third pass is done, where all the objects are re-rendered to the screen and the lighting from the previous stage is added. Since all object information is restored in the last step, anti-aliasing and material specific calculations are now possible. The transparency issue can be handled by rendering the transparent object with a forward rendering algorithm after the deferred rendering stage is completed, or as *Michael Ferko of Comenius University* proposed, where an *A-Buffer*[19] is used to render n-layers of transparent objects[20].


(a) Diffuse Output **(b)** Normal Output

(c) Depth Output **(d)** Velocity Output

Figure 3.7: Outputted *G-Buffers* from the first rendering stage.

3.3.1 Results

The Turtle Engine is built as a deferred renderer, with capabilities for multiple materials. The abstraction was made to just forward the textures themselves as diffuse color. In the lighting pass this diffuse texture is dampened dependent if it shall be used as an ambient or diffuse factor. This is not an entirely legitimate way to perform lighting but it still managed to perform, with pleasing results. As for specular factors, it is forwarded with the G-Buffers so materials will look to be lighted differently, depending on the material factors. The light sources are created with mathematical functions representing shapes. Thus, all lights will be rendered with a full screen quad instead of a geometrical object.

3.4 Gaussian Filter

To enhance the final appearance of various effects, a blur algorithm is desired. There are several different blurring methods, including *Box*-, *Tent*-[21] or *Gaussian Filter*[22]. Gaussian Filtering is a blurring method, is used to reduce the detail, as well as the noise of the image. It is also very useful when upscaling textures to hide artefacts or just to add a soft feel to an effect. The blurring is done by weighing samples in a circle around each pixel according to a bell curve normal distribution, as seen figure 3.8.

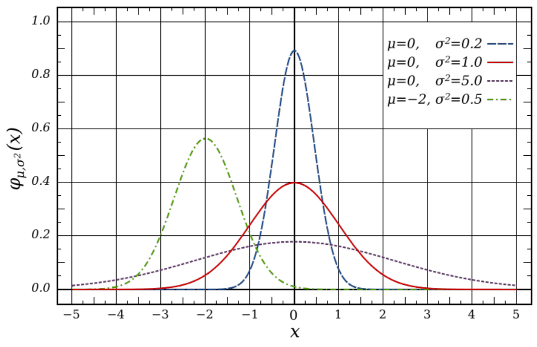


Figure 3.8: Image describing a normal distribution[23]

The projects implementation uses three samples with a large offset. This results in a strong blur, without requiring a lot of samples. Since texture lookups are time consuming, this is a desired effect. The Gaussian Filter was chosen since the end result is smoother, as well as not contributing with a significant decrease of performance.

3.5 Perlin Noise

Many graphical algorithms need some kind of noise, to provide a seemingly random element as input. There are a lot of choices to consider when generating random noise, such as regular white noise, Worley noise [24] or Perlin noise[25].

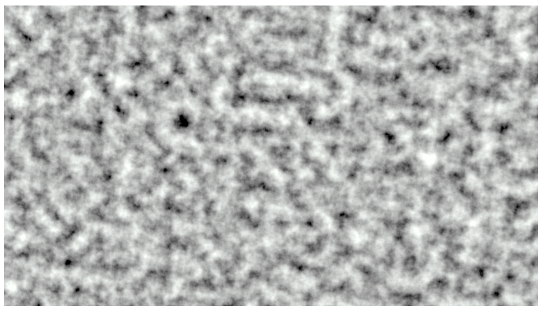


Figure 3.9: Image showing Perlin noise generated with *Blender*

The Perlin noise is a type of noise that maintains a smooth pattern while still being random. This is suitable for a number of effects where smoother noise is desired, for instance cloud generation, or as in this case, smoke. The project uses a pre-constructed Perlin noise algorithm[26], with slight modifications and updates to be compatible with the current version of GLSL.

3.6 Particle System

Various effects, such as fireworks and smoke, are built up as a large set of smaller objects, *particles*. In order to generate these effects in a graphics application, some kind of dedicated structure is needed.

A particle system is a data structure that tracks and creates particles, where a particle is represented as an arbitrary object. The particle system is responsible for the movement and manipulation of the particles. Normally, the particles are spawned from a point called an *emitter*. However, particles can also be emitters themselves, thus constructing a recursive system. The emitters can also be represented as areas and volumes, rather than points.

In the *Turtle Engine's* specific implementation; a particle system is an entity that creates and moves abstract particles. These particles are points with properties such as velocities and positions. The system then renders an object at each particle position, as the particle system is built with the ability of render many instances of the same objects, rather than just one.

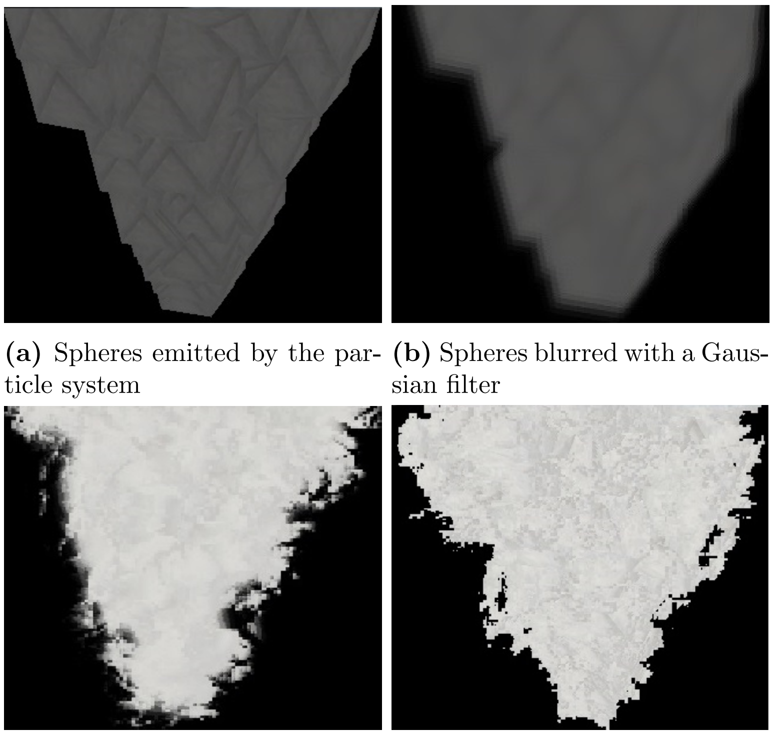
3.7 Mega Particles

With the current implementation of the deferred renderer, there is no possibility to render transparent or semi-transparent objects. Thus, standard billboarding is not possible as a smoke implementation.

The choice of technique, for rendering smoke, is based on the desire to have volumetric smoke which can be lighted and shaded, so that it looks like a true part of the environment. This requirement limits the selection, as these attributes corresponds to newer techniques. Two techniques were considered as the base of our implementation, either a ray marching algorithm like the one mentioned in the paper written by *Jos Stam*[27] and in the one written by *Keenan Crane*[28], or Mega particles that in mentioned in the paper written by *Robert Larsson*[29] and the central subject in the presentation given by *Homam Bhanassi*[30]. Both techniques have have their advantages. When using ray marching, physically based formulas are used for moving the smoke in space. Thus, giving a very accurate movement of the smoke. Mega particles on the otherhand are spheres with some filters applied to them, and therefore will not behave as natural, as simulated smoke, but since they are rendered as geometry, it is easier to generate shadows for them. Mega Particles seemed simpler to implement on a basic level, and as there is no need for accurate behavior of the smoke. Thus, this method was chosen as the basis in the projects implementation.

In short Mega particles produces volumetric clouds without using too much performance power, however there are a few drawbacks, such as the difficulty in controlling the final look and from far away the smoke can appear as viewed through frosted glass. However, we found a way to minimize the artifact, with minor overall quality loss. The implementation is described below.

The way the algorithm works is by rendering spheres to an offscreen render target. The spheres are in this case controlled by the particle system. It is of course possible to render any object instead of the spheres. However, spheres are considered to be the best shape as they are uniform. After rendering the spheres to a texture, that texture is blurred and distorted to make it look like smoke. The distortion is an offset of each pixel, where a perlin noise texture decides how much each pixel should be displaced. As the Perlin noise is cloud shaped by nature, the result looks like clouds. The blur smoothes out any roughness that might otherwise appear, as seen in figure 3.10c compared to 3.10d. The same filters are applied to the depth texture as it needs to follow the shape of the actual texture. Lastly, the smoke is blended into the scene using the depth texture, the entire process is seen in pictures 3.10a, 3.10b and 3.10c.



(a) Spheres emitted by the particle system **(b)** Spheres blurred with a Gaussian filter

(c) Spheres blured and distorted **(d)** Spheres only distorted

Figure 3.10: Figures a to c displays the mega particle process, d displays an artifact when b is not performed

What has been described up to this point looks good in still images, but for an fully animated smoke, there needs to be more movement in the smoke. Thus, the Perlin noise is changed for every render pass, therefore the smoke appears animated.

A problem was discovered when viewing a very thin smoke streak from a distance, the smoke appeared to jump up and down, not behaving like smoke. Therefore, the depth was needed to be considered in the distortion. Thus, the further away a pixel is, the less the pixel is distorted. The resulting implementation created a minimal reduction in the overall quality. However, the smoke still looks better for larger clouds rather than small streaks.

3.8 Post Processing Effects

The classical method of constructing a scene in a graphics application, is by rendering each polygon of a predefined model. Polygon rendering, as described as the *pipeline* in the preface, is done in object space, with geometry as input and projects the 3D scene onto the 2D screen space. However, polygon rendering is not always the most convenient approach: when rendering a scene with many advanced graphical algorithms in 3D space, the rendering time for the conventional polygon rendering method significantly increases. In addition, the modelling process becomes more tedious in order to increase the fidelity of the models.

To overcome this issue, *post processing effects*, also referred to as *image-based effects*, are introduced[31]. As the name implies, the post processing effects are introduced after the polygon rendering process and are done entirely in screen space, with a 2D image as input. As screen space only needs to take account to two dimensions, the post processing effects are considered a cheap method to enhance the visual effects of the application, although not being physically correct. A common signifier among post processing effects are that they are to simulate the optical artefacts of a camera lens. Some common post processing effects are *Motion Blur, Depth of Field, Screen Spaced Ambient Occlusion* and *Bloom*, and a more extensive description of the effects implemented, in this project, follows.

3.8.1 Motion Blur

As can be implied from the term racing game, the key feature is speed. Since a racing game could be considered a simulation of a real race, the sense of speed is important. As can be seen in feature films and broadcasts of real races, when the camera pans to follow the car, the background is significantly blurred. The blurring effect occurs, because the camera shutter speed is significantly slower than the cameras movement speed. This will cause a stretching and blurring of objects that does not move in the same direction as the camera pans. This phenomena is called motion blur.

To implement the effect of motion blur as a post processing effect can be done in several ways. The most basic implementation is to use an *Accumulation Buffer*[32], which is used to average a series of images. This technique yields pleasing results, but has the drawback of requiring two renderings per frame.

A more efficient technique is using a *Velocity Buffer*[33], which also is the most adopted technique and therefore the chosen technique in the *Turtle Engine*. This buffer is created by interpolating the velocity in screen space at each vertex of the model. The velocity can be calculated using the difference of two model-view-projection matrices; one for the current frame and one for the previous. The vertex shader performs this calculation. Consequently, the vector, obtained from the difference, is transformed to coordinates in screen space. Thus, the objects speed at each pixel is now known. As the unblurred image is rendered, the speed and direction, received from the velocity buffer, at each pixel is used for sampling the image, in order to obtain the blur.

Results

In the *Turtle Engine*, the blur consists of a maximum of 20 samples, depending on the length of the velocity vector, obtained from the velocity buffer. In order to preserve the colors, the result of the sampling is divided with the amount of samples. The result of the motion blur implementation can be seen in figure 3.11.

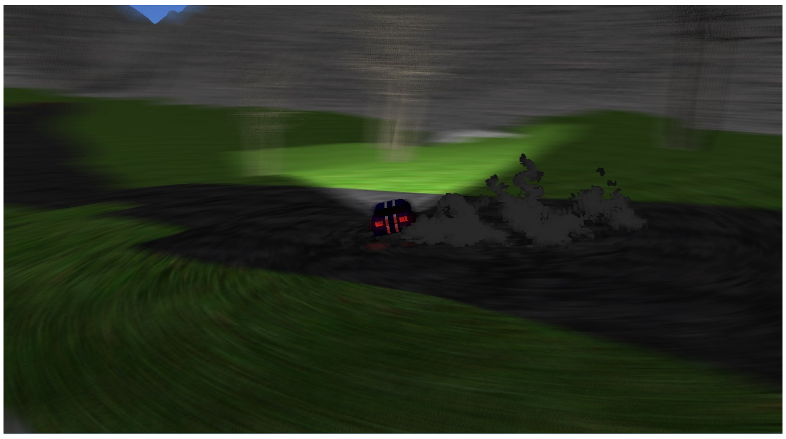


Figure 3.11: A showcase of the motionblur

3.8.2 Bloom

The bloom effect, similar to motion blur is an attempt to simulate a camera instead of an eye. Because the lens can not be truly perfect when pointing it directly at a light source, the light rays will convolute with an Airy disc[34], creating a glow or halo like effect. The bloom effect emphasizes the light sources, making it easier to distinguish light sources in the world, adding artistic mood to the scene. There are two standard methods of implementing bloom, either by using a bright-pass filter, that replaces darker pixels with black, and retains the brighter pixels[35]. The other alternative, also the method implemented in the *Turtle Engine*, is based on a technique proposed by Greg James (Nvidia) and John O'Rorke (Monolith Productions)[36].

Results

Within the material file for an object, a new type of parameter is added: the glow factor. This parameter will, after the first pass of the deferred renderer, be stored in the alpha channel of the diffuse texture. Following, a new render pass is performed where the alpha channel, will be multiplied with the RGB-channels. A material with a zero glow factor will result in a black color, while a non-zero color will be translated to the new render target. This glow texture is also multiplied with a factor to make the glow seem a bit brighter than the diffuse texture itself.

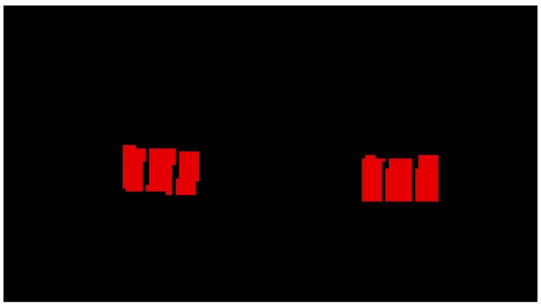


Figure 3.12: Resulting texture of the glow pass

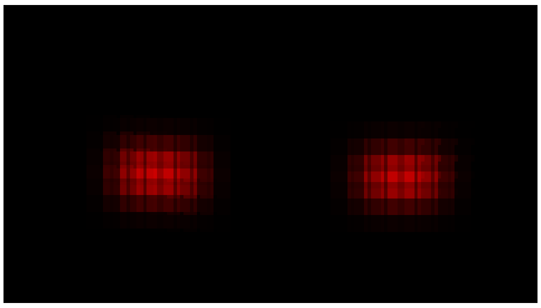


Figure 3.13: Result of the glow pass after gaussian blur is applied

At this stage a texture containing only the glow sources has been created, figure 3.12. By applying gaussian blur to this texture, the glow/halo effect will appear, as seen in figure 3.13. In the post processing stage, the texture is blended to the final image and the glow sources does now appear to be glowing.

3.8.3 Screen Spaced Ambient Occlusion

All shadows are not cast by directional lighting, some shadows are instead caused by the shape of an object; this is called ambient occlusion. These small details are often to minute to be distinguished by the shadow map. Fortunately, these small shadows do not depend on the direction of the incoming light. Thus, these shadows can be computed using another type of algorithm, without using too much computing power; such an algorithm is explained below.

Screen Spaced Ambient Occlusion (SSAO) is a technique used to simulate the effect of self shadowing surfaces. The basic idea of the technique is to sample points around every pixel, to calculate the amount of incoming light hitting the pixel. When implementing ambient occlusion, there are several options to choose among, the first concept was introduced by Crytek, used in the game Crysis[37]. Newer concepts in the same theoretical line follows, with for instance NVidias *Horizon Based Ambient Occlusion* (HBAO)[38] and *Screen Spaced Directional Occlusion* (SSDO)[39]; this technique also accounts for directional light sources, with minimal performance impact.

Following is a general description of the SSAO algorithm, in order to put the projects implementation into perspective. For every shaded pixel, a number of samples are taken within a hemisphere, which is facing in the direction of the normal. There are two general approaches to choose the pixels sample points, in order to make an unbiased estimate of the occlusion. One approach is to choose these points is from a pre-constructed noise texture. However, that implies, for every sample a texture lookup is needed, in order to get the positioning of the sample point. The other way, is to use a predefined *Poisson-disc distribution*. However, calculating a *Poisson-disc distribution* in real time, by dart throwing, requires a lot of computation[40]. Recent work has been made to compute these distributions faster and more suited for real time. Daniel Dunbar and Greg Humphreys at University of Virginia have developed an algorithm that can create a two dimensional *Poisson-disc distribution* with time complexity O(N). Robert Bridson at University of British Columbia has further developed this method, to make the distribution n-dimensional with the same time complexity[41].

Results

The *Turtle Engine* uses a pre-calculated *Poisson-disc distribution* for all samples. Since the *Poisson-disc distribution* is almost uniform, noise patterns that follows with few and equally positioned samples, should be harder to notice.

Currently the SSAO is performed during the lighting stage of the deferred render and added directly to the ambient term. Thus, blurring the ambient occlusion factor, in order to minimize the noise effects, is currently not possible. But rendering the SSAO to a secondary render target, then reapplying the SSAO as a post processing effect, will generate an occlusion to highlighted areas, which may make the lighting seem unrealistic. But due the current state of the models used in the engine, the results are still visually pleasing, which can be seen in figure 3.14 and in figure 3.15.

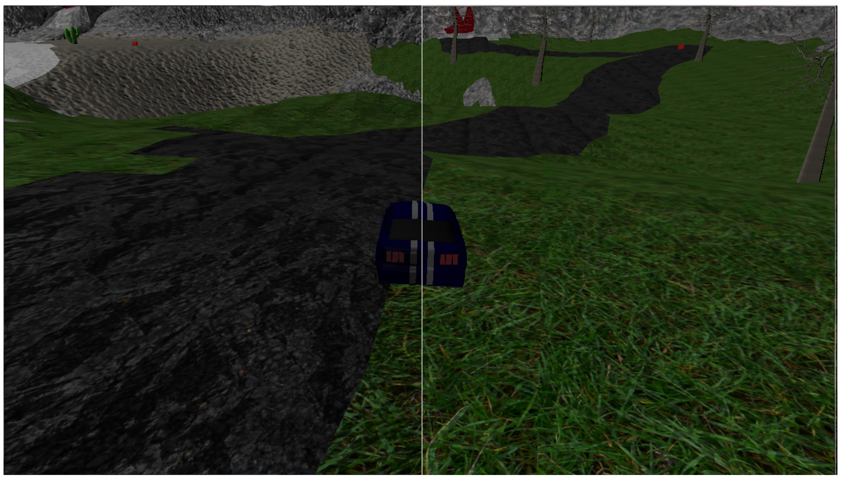


Figure 3.14: Image with SSAO on the left side and without SSAO on the left side

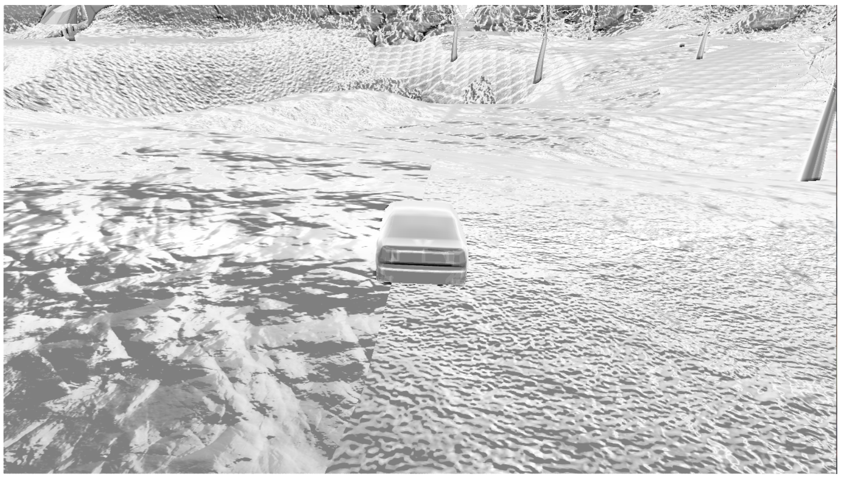


Figure 3.15: Displaying the SSAO outputted from the lighting stage

3.8.4 Skydome as a Post Processing Effect

The skydome is one of several techniques to render a sky, where the most common technique is considered to be the skybox. While implementing the skydome effect in the first pass of the deferred renderer, all the geometry information, for instance normals, were passed along to the second stage. Consequently, spotlights and directional lights where impacting the lighting of the skydome. This was considered an undesirable side effect and lowered the overall quality of the rendered image. Thus, it was decided to implement the skydome as an postprocessing effect, in order to counter this issue.

Following is a description of the skydome implementation. The skydome is rendered on an offscreen render target, positioned and scaled to the correspondent size of the map. A mild blooming effect is also applied, in order to give the skydome a slightly brighter finish. Then in the post processing pass, the skydome is depth blended on the farplane of the view frustum. Due to the fact that the particle system already uses the depth, no additional texture lookups are required, except for the skydome rendertarget.

4

Conclusion

4.1 Results and Discussion

The result of the four month development time, is a racing game that is both playable and visually appealing. There are many features to this result; all contributing to the final look and feel of the game. The most impactful features are: Motion blur, BEPU-physics and a varied game environment. As these features provide a sense of speed, logical movement and something to look at while tackling the challenge of the race. The other implementations are as important, but are hard to distinguish from each other in terms of effect, as they together contribute to the visual quality of the rendered image.

In hindsight, the choice of development framework played out as we foresaw. Because the framework is not supported by a community as large as XNA or standard OpenGL, finding support and suggestions took a lot longer either to find, or to implement. Also, because OpenTK is a merge between OpenGL and XNA, their incompatibilities between each other was noticeable in the development. An example of this it the way that OpenTK stores its matrices, XNA and OpenTK stores them as row major, while OpenGL and GLSL expects them to be column major.

The implementation of BEPU was probably the largest time sink in the project, as it had to be redesigned several times, before a working system could be built. Problems that are typically simple, such as the re-scaling of entities managed by BEPU, were difficult to debug. The team thought that it would be easy to add the physics- and collision library later in the development process, but it would have been more efficient to build the engine around the physics library instead.

A deferred renderer with support for directional-, point- and spotlights was successfully implemented. However, there was only shadow support for the directional lights, utilizing a simple shadow map algorithm. This implementation was poorly suited for the race track, as it was too large for a simple shadow map solution, and therefore ultimately turned off. In the future, we would like to implement a cascaded shadow map, as this would make shadows crisp enough to utilize in the large rendered environment. We would also like to change the current mathematical representations of lights, to a method using geometrical shapes. This would increase performance, since view frustum culling would be possible, currently not supported in our implementation.

The racing track was lit by a single directional light, as well as the pair of spotlights from each car. However, we would like to add additional light sources in addition to the current setup. this would result in a more dramatic lighting. Thus, more advantage would be taken of the deferred rendering algorithm.

The models were seen as visually pleasing by the development team, which took us a combined work of approximately 80 hours. This is thought, by the development team, to equal the time needed to finalize the development of a cascaded shadow map algorithm, with the addition some sort of soft shadows. The alternative would have been to use pre existing models. However, creating our own models made it easier to customize individual elements during the development process, as well as giving us more artistic freedom.

The current way, in which the *OBJ* files are converted to the vertex array objects, that OpenGL uses, is inefficient in terms of memory usage. With very large files, crashes will occur on low end systems. We would have liked to optimize the excess amount of data, making it closer to a one to one size correspondence.

The smoke rendering algorithm that we implemented was a basic version of the mega particle algorithm, which provided a remarkable result considering how simple the final implementation was. Another feature with our implementation is that the algorithm takes about the same amount of time to execute, no matter how much smoke is rendered, as the only increase in computation is rendering more spheres. Improvements to the smoke rendering would be to improve transitions between the noise textures, blending intermediate noise texture, i.e. making the noise animation look smoother. Improving computation time of the blur would be the key optimization, as the distortion is two texture lookups, and therefore, it can not be optimized any further. The decrease in quality when viewing small streaks is also left for future optimization.

The implemented motion blur had bleeding issues, as is apparent in picture 4.1. However, this issue was not considered a priority, as it is hard to see from the players view-port, as is demonstrated in picture 3.11 in the motion blur section. However, this issue could be distracting when two cars are constantly overtaking each other, or driving side by side for extended periods of time. Therefore, we would like to correct this issue in a future stage of the project.

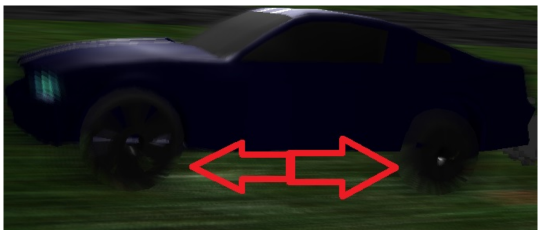


Figure 4.1: The wheels are so blurred background is visible

Our SSAO implementation provided good results, even though it is based on one of the less visually correct algorithms. However, the results are not as visually prevalent due to the rather dark color scheme of our textures. We would like to, in a future stage, supplement the algorithm with noise and blur. Alternatively, we would use the Poisson disc generation algorithm, as mention in the SSAO section.

4.2 Conclusion

The resulting game is in our opinion well realized, in perspective to our set goals. Of course we would like to have added more content and features, but due to time constraints and team size we had to prioritize essential components that we thought would contribute the most to the game. Therefore, game development in a small and rather inexperienced team, over a short period of time is nothing we would recommend. However, if the game idea is clear, and if the development is done with respect to the physics library, rather than the opposite; there is still a chance of creating a visually pleasing, as well as an entertaining game.

Probably the greatest insight of the development team, is that proper research is crucial, with respect to time, as well as the final result. For instance, by taking the time to analyze the various rendering algorithms, we might have ended up with a deferred renderer in an earlier stage of the project. Thus, adding more visual effects might have been possible. However, doing proper time estimations for the implementations is still considered difficult, due to lack of experience.

4.3 Future Work

In addition to the correction of the algorithms mentioned in the discussion, there are additional features we would like to implement, to both improve performance and the visual quality of the game. Examples of features that could be added are, a height mapped terrain, as this would decrease the time it takes to load the map. Another example is adding a cube map to the car, as as reflections would make the car look more like a part of the environment.

These are just some of the features that would make a positive impact to the game and can hopefully be implemented in the near future.

Bibliography

[1] T. Tamasi, The Evolution of Computer Graphics, in: NVision 08, SVP, Content & Technology, NVidia, 2008.

[2] S. Kent, The Ultimate History of Video Games: From Pong to Pokémon and Beyond- The Story That Touched Our Lives and Changed the World, 1st Edition, Prima Publishing, 2001.

[3] Minecraft, A longer history.
URL `https://minecraft.net/game`

[4] Computer and Video Games, Microsoft email confirms plan to cease XNA support.
URL `http://www.computerandvideogames.com/389018/microsoft-email-confirms-plan-to-cease-xna-support/`

[5] BEPUphysics.
URL `http://bepuphysics.codeplex.com/`

[6] Bullet Physics.
URL `http://bulletphysics.org`

[7] IPv4 Multicast Address Space Registry.
URL `http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml`

[8] The MD3 Format.
URL `http://www.xbdev.net/3dformats/md3/`

[9] The OBJ Format Specification.
URL `http://www.martinreddy.net/gfx/3d/OBJ.spec`

[10] glDrawArrays Specification.
URL `http://www.opengl.org/sdk/docs/man/html/glDrawArrays.xhtml`

[11] X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, S. H.-Y, Generalized Displacement Maps, Eurographics Symposium on Rendering (2004) 227–233.

[12] J. F. Blinn, Simulation of Wrinkled Surfaces, SIGGRAPH '78 Proceedings (1978) 286–292.

[13] J. D. Cohen, M. Olano, D. Manocha, Appearance-Preserving Simplification, SIGGRAPH '98 Proceedings (1998) 115–122.

[14] M. Deering, S. Winner, B. Schediwy, C. Duffy, N. Hunt, The triangle processor and normal vector shader: a VLSI system for high performance graphics, SIGGRAPH '88 Proceedings (1988) 21–30.

[15] T. Saito, T. Takahashi, Comprehensible Rendering of 3-D Shapes, SIGGRAPH '90 Proceedings (1990) 197–206.

[16] M. Valient, Deferred Rendering in Killzone 2, in: Guerilla Develop Conference, Guerilla Games, 2007.

[17] O. Olsson, M. Billeter, U. Assarsson, Clustered Deferred and Forward Shading, High Performance Graphics (2012) 1–10.

[18] R. Koonce, Deferred Shading in Tabula Rasa, GPU Gems 3 (2007) 429–457.

[19] K. Myers, L. Bravoil, Stencil Routed A-Buffer, SIGGRAPH '07 Proceedings (2007) 21.

[20] M. Ferko, Real-time Lighting Effects using Deferred Shading , Proceedings of CESCG 2012 (2012) 151–158.

[21] W. Jarosz, Fast Image Convolutions, in: SIGGRAPH, Walt Disney, 2001.

[22] E. Elboher, M. Werman, Efficient and Accurate Gaussian Image Filtering Using Running Sums, in: 2012 12th International Conference on Intelligent Systems Design and Applications, IEEE, 2012.

[23] Gholcomb, Normal Distribution PDF.
URL `https://controls.engin.umich.edu/wiki/index.php/File:Normal_DistributionPDF.png`

[24] S. Worley, A Cellular Texture Basis Function, SIGGRAPH '96 Proceedings (1996) 291–294.

[25] K. Perlin, Noise and Turbulence.
URL `http://mrl.nyu.edu/~perlin/doc/oscar.html`

[26] Perlin Noise and GLSL.
URL `http://www.kamend.com/2012/06/perlin-noise-and-glsl/#comments`

[27] J. Stam, Interacting with Smoke and Fire in Real Time, Communications of the ACM (2000) 76–83.

[28] K. Crane, I. Llamas, S. Tariq, Real-Time Simulation and Rendering of 3D Fluids, GPU Gems 3 (2007) 30.

[29] R. Larsson, Interactive Real-Time Smoke Rendering, Master Thesis (2010) 1–61.

[30] H. Bahnassi, W. Bahnassi, Volumetric Clouds and Mega Particles, in: In Framez, Electronic Arts Montreal, Microsoft.

[31] Post-processing NPR effects for video games, VRCAI '13 Proceedings of the 12th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry (2013) 147–156.

[32] P. Haeberli, K. Akeley, The Accumulation Buffer: Hardware Support for High-Quality Rendering, SIGGRAPH '90 Proceedings (1990) 309–318.

[33] S. Green, Stupid OpenGL Shader Tricks, in: Game Developers Conference, NVidia.

[34] J. E. Greivenkamp, Field Guide to Geometrical Optics, 1st Edition, SPIE Press, 2004.

[35] T. Sousa, Adaptive Glare, Shader X3 (2004) 349–355.

[36] G. James, J. O'Rorke, Real-Time Glow, GPU Gems 3 (2004) 21.

[37] M. Mittring, Finding next gen: CryEngine 2, SIGGRAPH '07 Proceedings (2007) 97–121.

[38] L. Bavoil, M. Sainz, Image-Space Horizon-Based Ambient Occlusion, in: SIGGRAPH 2008, NVidia, 2008.

[39] T. Ritschel, T. Gorsch, H.-P. Siedel, Approximating Dynamic Global Illumination in Image Space, I3D '09 Proceedings (2009) 75–82.

[40] D. Dunbar, G. Humphreys, A Spatial Data Structure for Fast Poisson-Disk Sample Generation, SIGGRAPH '06 Papers (2006) 503–508.

[41] R. Bridson, Fast Poisson Disk Sampling in Arbitrary Dimensions , SIGGRAPH '07 Sketches (2007) 22.

A

Appendix

A.1 Definitions

- **Noise texture** - A texture filled with random numbers.
- **Dart throwing** - Iterative process that refines an existing points consisting of a series of random candidate points, the following point that meets certain requirements is chosen.

A.2 Images

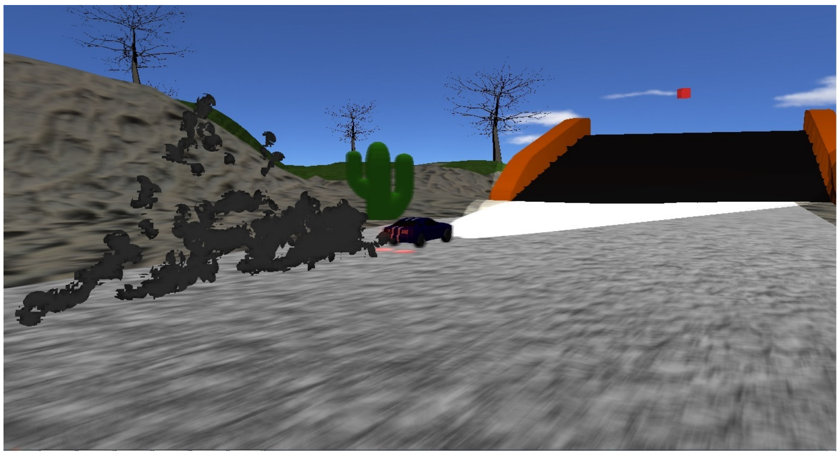


Figure A.1: Screenshot of the game 1

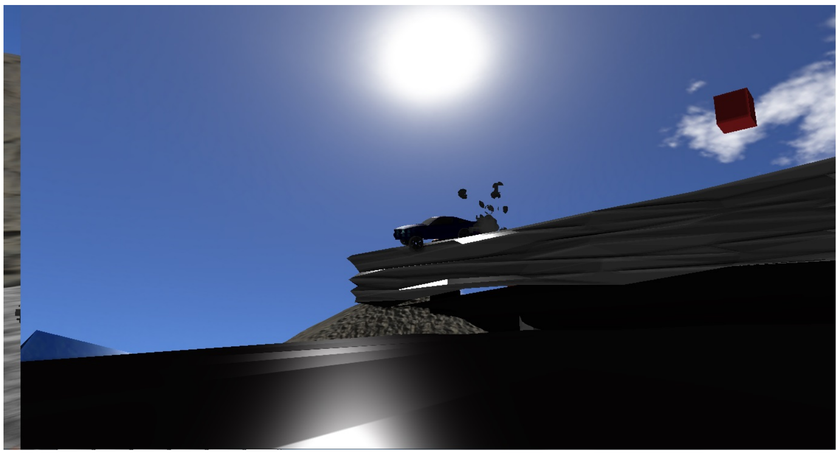


Figure A.2: Screenshot of the game 2

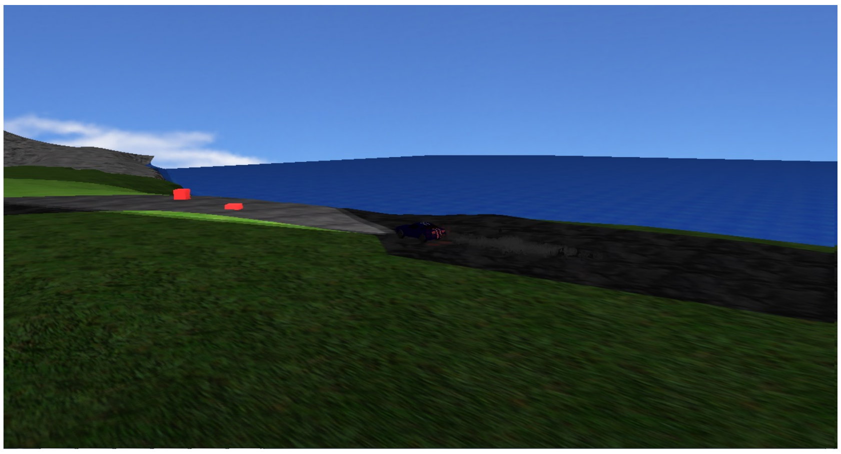


Figure A.3: Screenshot of the game 3

A.3 Contributions

A.3.1 Project

Alexander Lissenko

- Basis of particle system
- Car model
- Parser for multiple materials
- Normal map, later fixed by William
- Implementation of speed boost power up
- Half time presentation with Johan
- Final presentation and opposition with Johan and William
- Administration

Johan Ekdahl

- Particle system
- Mega particle rendering algorithm
- Gaussian Blur
- Perlin noise
- Finding textures
- Modelling and texturing of the race track
- Half time presentation with Alexander
- Final presentation and opposition with William and Alexander

William Axhav Bratt

- Deferred rendering algorithm
- BEPU implementation
- Glow shader
- Motion Blur
- Sound implementation
- Network
- Final presentation and opposition with Johan and Alexander

A.3.2 Problem solving, synthesis and analysis

During the development process, each member of the group got assigned to specific tasks. For major development changes the entire group would be consulted, while minor changes were individual responsibility. When an individual got stuck he would consult other members and consult to scientific papers on the subject at hand.

A.3.3 Report

Final editorial work was done by Alexander Lissenko, with additional help with proof-reading from William Axhav Bratt.

Alexander Lissenko

- 1 Background
- 1.4 Method
- 1.5 Limitations
- 1.6 Outline
- 2.1 Gameplay, with Johan
- 3 Chapter introduction
- 3.2 Bump mapping
- 3.8.1 Motion Blur
- 4 Conclusions chapter, with Johan and William

Johan Ekdahl

- 1.1 Purpose
- 1.2 Aims
- 3.1.1 Modelling process
- 3.4 Gaussian filter
- 3.5 Perlin noise
- 3.6 Particle system
- 3.7 Mega particles
- 4 Conclusions chapter, with William and Alexander

William Axhav Bratt

- 2.3 Physics & collision detection
- 2.4 Network
- 3.1.2 Model loading
- 3.3 Deferred rendering
- 3.8 Post processing effects (except 3.8.1 Motion Blur)
- 4 Conclusion chapter, with Johan and Alexander