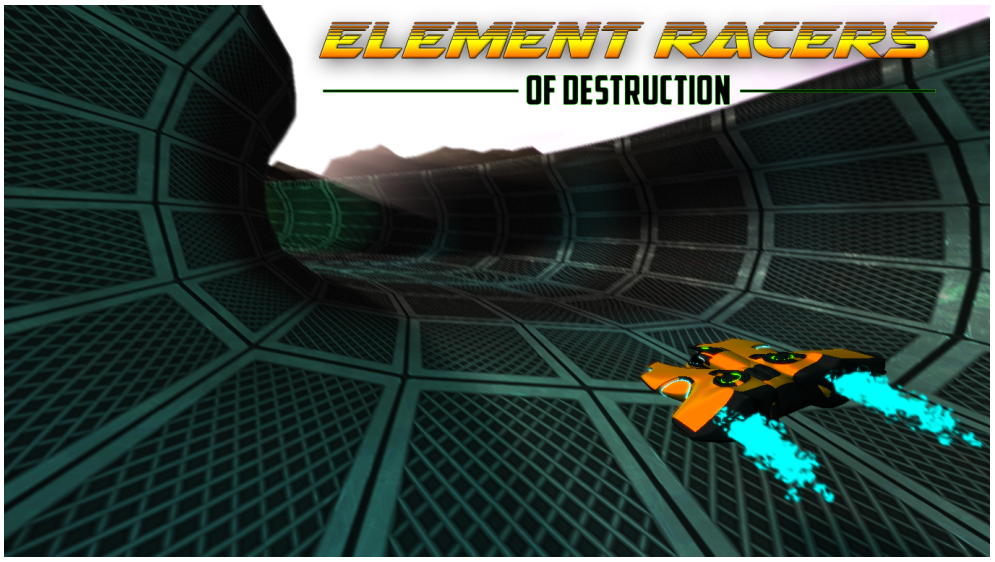
CHALMERS



Element Racers of Destruction

Development of a multiplayer 3D racing game with focus on graphical effects

Alexander Hederstaf
Anton Bergman
Sebastian Odbjer
Johan Bowald

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sverige 2014

Element Racers of Destruction
Paper on development of a multiplayer 3D racing game with focus on graphical effects

Alexander Hederstaf
Anton Bergman
Sebastian Odbjer
Johan Bowald

Examiner: 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

Cover: Ingame Screenshot with the game logotype in top right corner. Image used as menu background in the implementation.

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Abstract

This bachelor thesis details the various graphical effects used when implementing a multiplayer racing game. The aim was to implement suitable graphical effects for a large scale environment and high-speed racing game, with regard to the limited timeframe and resources of the project. The game should also contain enough gameplay to be deemed playable and support multiplayer. Each graphical effect implemented was evaluated and discussed based on its suitability and the final results. The effects implemented, and their corresponding techniques, list names such as deferred shading, shadow mapping, screen space ambient occlusion, bloom, volumetric lighting and particle systems. The other aspects of the game, such as content and gameplay, are somewhat lacking due to the high focus on graphics. We do however consider the final results to be of relatively high visual quality when the various constraints of the project are taken into account.

Abstract

Denna rapport beskriver de olika grafiska effekter som används vid implementation av ett racingspel med flerspelarläge. Målet var att implementera passande grafiska effekter för storskalig terräng och hög hastighet samtidigt som vi håller oss inom projektets begränsade tidsram och resurser. Spelet skulle även innehålla nog med spelmekanik för att vara spelbart och för att stödja flerspelarläget. Varje implementerad grafisk effekt var utvärderad och diskuterad baserat på dess lämplighet och dess slutliga resultat. Bland implmenterade effekter och deras korresponderande tekniker hittar man exempelvis deferred shading, shadow mapping, screen space ambient occlusion, volumetric lighting och ett partikelsystem. Andra aspekter av spelet, så som spelmekanik, är något nedprioriterade på grund av fokus på grafik. Vi anser att den slutgiltiga produkten håller relativt hög visuell kvalité, när projektets begränsingar tas i beräkning.

Contents

Chapter 1

Introduction

This thesis is the result of a bachelor project carried out by four students from Chalmers University of Technology. The project spanned a total of 17 Weeks in the spring of 2014.

1.1 Purpose

The main focus of the thesis is to research and evaluate different graphical effects and the algorithms behind them, by implementation in a 3D game application. All effects and the final results are to be evaluated and put into perspective to the limited number of developers and development time.

1.2 Background

A major portion of the progress that has been made in the computer graphics field can be attributed to the gaming industry, and the demand gaming creates for new and better visual experiences (Porcino, 2004). All the members of the group are interested in computer graphics and advanced algorithms, and share the goal of exploring and learning more about the subject.

In an attempt to pursue our goal to learn more about computer graphics we have undertaken this project of creating a game: as creating a game often involves exploring and understanding a large variety of different graphical algorithms to find the ones best suited for the game in question. In an effort to increase the time spent on researching and implementing graphical algorithms, we chose to develop a multiplayer racing game. Racing games typically requires less time to be spent on things such as AI, logic and animation when compared to other game types.

In today's industry, every game needs some unique feature as a "selling point" to differ from the huge selection of games available on the market. The typical "start to goal" racing game might not attract sufficient interest. We aim to include additional dimensions to the gameplay, thereby enhancing the player's experience. One example could be a power-up system where the players build their own power-ups using different basic elements, instead of picking up power-ups that only do one thing.

Inspiration for the project has been collected from titles such as Star Wars Episode I Racer, Wipeout and other fast-paced space themed racing games. Inspiration was also taken from element based games such as Magicka and the Warcraft III custom game Warlock.

1.3 Problem Statement

The goal of the project is to create a racing game with focus on achieving visually pleasing graphics. Furthermore, as the main setting of the game is large open environments, the problem statement can be defined as follows:

- "What are suitable graphical effects for creating a visually pleasing racing game staged in large virtual environments, given the timeframe of the project?"

This problem statement is handled in all the individual graphical algorithms and effects, Chapter 3 and 4, in their respective discussions. It is also handled with regard to the final product as a whole in the discussion, which can be found in Chapter 5.
Whether a graphical effect is suitable is in no way an arbitrary question. Therefore, we have devised a number of sub statements to define what suitable means in this context.

- "Are the visually disturbing artifacts, assuming there are any, acceptable when compared to the visual gains of using this effect?"
- "Are the visual gains in reasonable ratio to the performance cost generated by the effect?"

As a racing game typically involves high speed motion, we have to consider what graphical effects are suitable for such a setting.

- "What are suitable graphical algorithms when considering the high speed motion of the game?"

As many graphical algorithms work well independently of how the camera moves this is only answered for those graphical effects where it is relevant. To achieve a complete game, other aspects than graphical effects are needed. Therefore, we need to make sure we deal with these when creating the game.

- "Which systems and what elements of gameplay are needed in order for the game to be considered playable?"

A game can be much more enjoyable if one can compete with others, and it is something we want to enable in the game. With regard to the limited timeframe of the project and the heavy focus on graphics, the aim was to implement something that gives a good experience for multiple players with a low investment of development time.

- "What is a suitable solution for multiple players to compete in the game with regard to the gameplay and the time frame of the project?"

The last two problem statements above are handled in a separate part of the paper from the graphical effects. The answers to these problem statements can be found in Chapter 2, which details the design of the game.

1.4 Method

In order to be able to develop a game with high quality graphics, there are many different effects that have to be studied and implemented to find a satisfactory choice for this particular game. To sift through all possible algorithms for every effect efficiently, a general method for finding suitable algorithms is required.

We pose two questions for every algorithm in order to determine its suitability with regard to our game implementation:

- How does the algorithm contribute to the final product?
- Can the same effect be achieved with another algorithm?

Many resources were used to find our various graphical effects, such as textbooks (For example Real-Time Rendering (Akenine-Möller et al., 2011)) and Internet search engines.

When a paper or article was found that contained an algorithm suitable for the game, it was researched further. If a tutorial or the source code related to the paper was available, it was used to speed up the learning process. When we had sufficient knowledge, and had discussed how it would contribute to the game, the algorithm was implemented. The implementation was done in a similar

fashion to the solution found in the paper, with adaptations to better suit the game.

When a new feature was considered complete and as close to bug-free as possible, it was added to the final product and then discussed with the rest of the group to make sure everyone was up to date. This gave us another chance to discuss potential problems that needed to be fixed and to evaluate the choice of algorithm.

All test of performance and visual quality are done on one computer with a specified set of hardware and two display options. The graphics card is an Nvidia GTX 780, and the CPU is an Intel i7 4770K running Windows 7. The two available displays are a 24" computer screen or a 48" TV both running at 1080p resolution.

1.5 Constraints

The main focus of the project is graphics, and techniques related to computer graphics. Therefore, some elements that would normally be prioritized first in a game received less attention: elements such as sound, game logic and 3D models. Already finished frameworks were used as much as possible when anything related to these aspects of the game was implemented. For example, a physics engine was used for collision detection and the physical representation of the world.

Revision control was handled via Bitbucket using Git. Bitbucket has a built-in issue tracker that was used in order for group members to be able to work independently, and in order to ensure fast communication within the group.

The game was programmed in C# using XNA: a DirectX game development toolkit for Xbox and PC developed by Microsoft. The reason behind our choice was that most of the previous bachelor thesis projects that had produced high quality products used XNA. The design choices behind the movement of the racer, button layout and menu layout were made for Xbox 360 gamepads as the main controller option.

1.6 Report Outline

As a computer game does not only consist of graphical algorithms, we present and discuss the relevant parts of the game that are not graphical algorithms or effects in Chapter 2.

The main focus of this paper is the graphical algorithms. In Chapters 3 and 4, the algorithms are presented, explained and discussed. Since we wanted the paper to reflect what the project group's work has resulted in, and provide an approximation for how difficult implementing the various algorithms was, a more informal structure was adapted for discussions and conclusions. The presentation of the graphical algorithms was performed in a more formal manner.

A discussion regarding all the graphical effects and their algorithms is presented in Chapter 5. The effects are compared by their relative contributions to the final product and by how they contribute with regard to the large environment and high-speed setting of the game. The value of the different effects is discussed in contrast to the time required for implementing them. Some thoughts regarding what effects might have been worth spending more focus and time on, when considering the visual results of each effect, are also presented. The conclusion of the work and research is presented in Chapter 6.

Chapter 2

Game Design

This chapter covers all the aspects not related to graphics that contribute to the user experience of the game. They are not the main focus of this paper, but still vital to a game if it is to be considered playable.

2.1 Gameplay

In a game some elements of gameplay are required. The gameplay does not have to offer much for the game to be considered playable, but for an engaging user experience more content is required. The only required gameplay feature in a racing game is the ability to race from start to goal. With an added system of extra abilities for the player to use, such as a speed boost, the gameplay becomes more engaging.

2.1.1 Start-to-Finish Racing

The basic requirement for racing is that the participants start in one place and must somehow get to their goal, typically with the aid of some vehicle. The map used in the game is circular, meaning that the start and goal lines are at the same place. Several checkpoints are used to ensure the player maneuvers the course as it was intended. The checkpoints must be passed in the correct order, which ensures that the player does not take any unintended shortcuts. When a player has passed all checkpoints and the finish line the race is over, with that player proclaimed as winner.

2.1.2 Extra Abilities

To add variation to the gameplay, support for extra abilities were added. The only ability currently available in the game is a speed boost. The speed boost temporarily increases the speed of a racer. This can be used to overtake other players in some parts of the map. The usage of extra abilities makes gameplay more divers and interesting. The speed boost ability charges up whenever it is not used, however, a system of power-ups placed on the racetrack could be used to charge various extra abilities.

2.1.3 Results & Discussion

The racing part of the game can be considered the least amount of gameplay necessary for the game to be considered playable. However, games are often supposed to be entertaining. The extra ability system was added to improve on the gameplay and playability of the game, but it is not considered a requirement for a playable game.

As the focus of the project was on the graphical effects, gameplay is lacking. If more time was spent on the gameplay, features such as extra abilities that would allow a player to sabotage the vehicle of another player could have been implemented.

2.2 Game Physics

All objects in the real world are affected by physics. An approximation of the real world is required to create a believable game world. Several physical systems and properties have to be constructed in the game in order to simulate reality, and effects such as gravity. However, these phenomenon can be both complex and time consuming to model correctly. To aid in the calculations of such systems and properties a physics engine is used. A physics engine simulates objects in a space. It handles motion of objects and also tests for collisions. A correct physical behavior is a strong foundation for the game. However, to get the best gameplay experience the ability to deviate from the laws of physics is also desired.

Developing a physics engine is a time consuming and complex process. Therefore, an open source implementation was used to save time. There are a lot of open source options available, but as the game uses the XNA framework the physic engine had to have support for it. As physics engines are complex and requires time to get used to, good documentation was important. The physics engine that was chosen for the game is the BEPUPhysics engine, as it passed the previously stated requirements.

2.2.1 Racer Physics

The intention was to have the racer hovering over the ground but still be able to fall down from heights. The racer should also align with the surface of the ground. When a racer collides with a wall, the racer should lose speed and rotate in a pitch-direction away from the wall. This is done to get the player back on the track as fast as possible. The acceleration model for the racer was created to lower the penalty for crashes and other decreases in speed, the goal is to have quick acceleration for the first few seconds from start, and slower acceleration when the speed is closer to the maximum.

2.2.2 Racer Acceleration

The current velocity is used to calculate the current acceleration, this achieves an arbitrary acceleration time between zero speed and maximum speed. There are three phases of acceleration for the racers. The first phase lies in the interval of 0% - 60 % of maximum speed, and it has the fastest acceleration. The second phase lies between 61% - 80% and has a lower acceleration. The last interval of 80% - 100% has the slowest acceleration.

2.2.3 Racer Hovering

The racer's hovering is calculated using two vectors between the racer and the ground, as shown in Figure 2.2.1. A ray-cast method supplied in BEPUPhysics is used.The ray-cast projects a vector of a given lenght from a given point in a given direction, and returns information on whether the ray hit anything, and if it did, the distance to the target.. The two vectors from the ray-casting are used to determine the height from the ground, and if the racer needs to be rotated in a yaw-direction. If the difference of the front and back vector is larger than a given threshold, the racer is rotated in a yaw-axis.

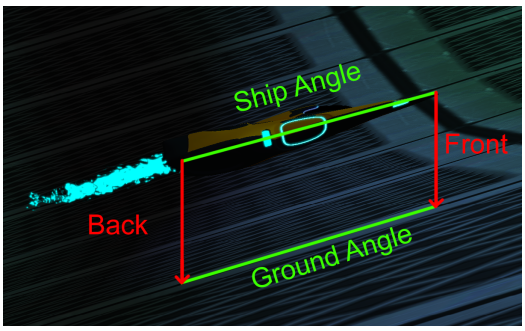
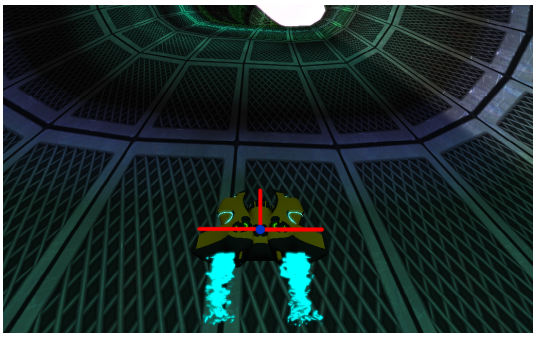


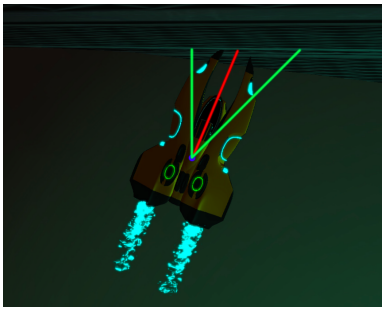
Figure 2.2.1: The racer aligns to the ground using ray-casting. The alignment angle is calculated using a vector in the front and in the back of the racer.

2.2.4 Racer Collision

The collision of the racer detection is done by ray-casting from the front and from the sides of the racer with vectors of the same length as the racer, as shown in Figure 2.2.2a. These rays are tested against all objects in the game environment that the racer can collide with. If one of the rays hit an object in the environment, a collision is detected. When a collision is detected a new ray-cast is done using two vectors, as shown in Figure 2.2.2b. The vectors are set at an angle to the original collision detecting vector, the two angled vectors then check the angle at which the racer collided. The racer is then rotated in direction of the longest of the two vectors to simulate sliding off the surface in the crash.



(a) First phase of the collision detection, checking for objects in front and to the side of the racer.



(b) Second phase of the collision, the green vectors are used to determine the angle of the collision.

Figure 2.2.2: The collision detection for the racer is done in two steps. In (a) the ray-casting finds out if the racer has collided, in (b) the angle of the collision is found using additional vectors.

2.2.5 Collision For Other Objects

BEPUPhysics provides a complete system for handling collision between objects. Each object in the world space is represented by an entity. In BEPUPhysics these entities are split up in different collision groups, the groups have rules for what should happen when collision occurs between entities of the groups. Two collision groups are used in the game: one for racers and one for lap checkpoints. Each time an entity of a specific kind is loaded in the game, it is added to the correct collision group. For example collision between a racer and a checkpoint make the racer pass the checkpoint, while collision between two racers can be ignored.

2.2.6 Result & Discussion

The ability to play the game is dependent on a working physics implementation. Without the physics engine the racers would not follow the ground or be able to drive on the bridge. It would neither be possible to drive through checkpoints to complete a race. If we had tried implementing a physics engine ourselves we would not only lose a lot of time, but we might also experience bugs that caused the racer to go off the map. With the BEPUPhysics engine the gameplay features such as driving around and subsequently racing were possible, without annoying or game-breaking bugs.

The initial choice of physics engine was Jitter Physics due to good reviews, a promising demo and a “start up”-tutorial. Some alternative options were

Bullet or JitlibX, all with support for XNA. During early stages of development some functions of Jitter Physics were hard to implement due to lack of documentation, and the choice was made to migrate to another engine, BEPUphysics. BEPUPhysics has high quality documentation and forum support, something that was very helpful.

The physics engine also improved aspects of implementation that were not considered from the start. For example, the ray-casting was useful for the camera to avoid clipping with the terrain and other objects. The engine also has great support for collision with heightmap terrain that was later used in the game.

2.3 Multiplayer Support

To make the game playable by more than one player at time a multiplayer implementation is required. The multiplayer part of the game is a local split-screen implementation. The style of the game allows for heated competition and a promoted element was the destruction of the other players, in such a setting local multiplayer can give a more engaging experience

2.3.1 Local Split-Screen Multiplayer

In a local split-screen implementation every player needs their own viewport. The viewports are created for optimal layout without wasting any screen space with black boxes, as can be seen in Figure 2.3.1. In the multiplayer mode controller input is handled for every active player for each of the racers. The game's logic is executed the same way as for one player, though it updates a different number of racers. The rest of the map is updated the same way independently of the number of players.

The rendering is done once for every viewport in the same way it would be done for one viewport, but with different screen sizes. Every view has a separate post processing manager allowing for the usage of different post processes for different players. This can be useful for enhancing various gameplay aspects.

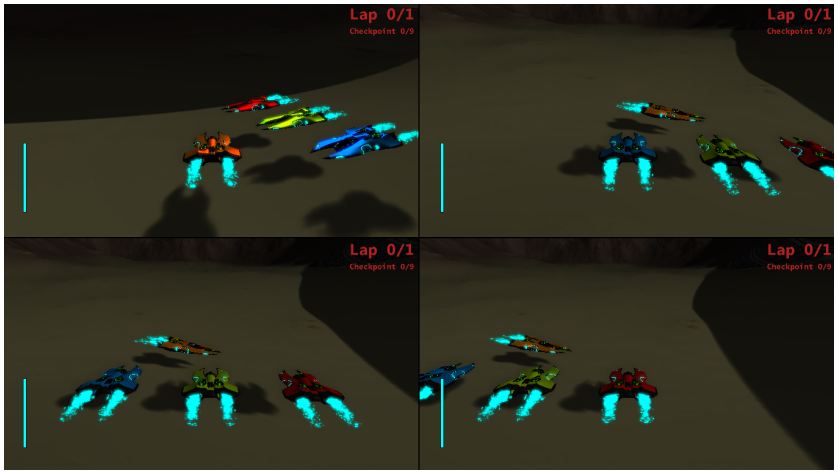


Figure 2.3.1: Four player split-screen using four separate viewports, every player has a view of their own racer and a HUD displaying their status.

2.3.2 Results & Discussion

We chose a local multiplayer experience for the game instead of a network or Internet implementation. A negative attribute of networked multiplayer is that it introduces a lot of possible things that may go wrong. With local multiplayer we did not have to worry about disconnects or synchronization errors. Local multiplayer also enabled us to focus on the most important aspect of the project: graphics. The negative technical aspects of online multiplayer together with the time saved by a split-screen implementation made local multiplayer a suitable solution for the game.

Chapter 3

Rendering Algorithms

This chapter covers all the graphical effects that create and light the scene. The effects covered create an image of the scene with all models and terrain with complete lighting. The result of these effects can then be changed further using some of the post processing algorithms presented in Chapter 4.

3.1 Lighting

A fundamental aspect of providing a virtual environment with a sense of depth lies with the lighting. Geometry that is hit by light should be illuminated, and its shape should be discernible by observing the absorption and reflection of light.

3.1.1 Reflection model

The Phong reflection model is common in real-time 3D applications for lighting scenes. The Phong reflection model simulates light using a non physical method. The light computation is divided in three parts: Ambient light, Diffuse light and Specular light (Phong, 1975). An adaptation to the Phong model is the Blinn-Phong model. The Blinn-Phong model calculates normals in a different way for the specular component, producing a smoother specular highlight. The Blinn-Phong model proposes the usage of a half-vector for specular calculation (Blinn, 1977).

The first component of the Phong reflection model is the ambient lighting, as seen in Figure 3.1.1. Ambient light is an approximation of indirect light. When rendering the scene, ambient light is added as a constant amount to every object and does not depend on the geometry.

Figure 3.1.1: With ambient light, the outline of the racer is visible, although the shape of the racer can not be grasped by the viewer.

The second component is the diffuse lighting, as seen in Figure 3.1.2. Diffuse lighting makes the shape of objects visible, as it depends on the normal of the surface and the direction to the light.

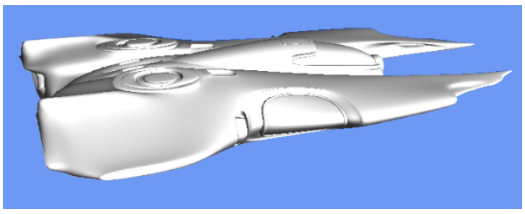


Figure 3.1.2: Diffuse light makes the racer shape discernible.

The third component of the Phong shading model is the specular component, as seen in Figure 3.1.3. Specular highlights simulate bright spots on shiny objects, which helps to clarify the object's material type and how the object is oriented with regard to the light source.

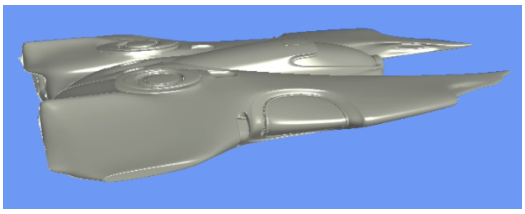


Figure 3.1.3: Ambient, Diffuse and Specular light combined with tweaked light parameters. The specular component is visible as white reflections, for example near the cockpit and at the edge of the wing.

3.1.2 Results & Discussion

The result from both the Phong and Blinn-Phong reflection models gives the racer a metallic look. The difference between them is visible in the smoothness of the specular part. The Phong model gives a more bright and concentrated specular highlight while the Blinn-Phong is smooth. We liked the high contrast look from the Phong reflection model and kept it. The color texture of the racer contains several small lights which are not handled by the Phong reflection model. To make these parts emit light an emissive component is included in the light calculation. The emissive component is calculated by using a glow-mask on the color texture with 100% intensity (James, 2004). The result of the lighting step is shown in Figure 3.1.4.

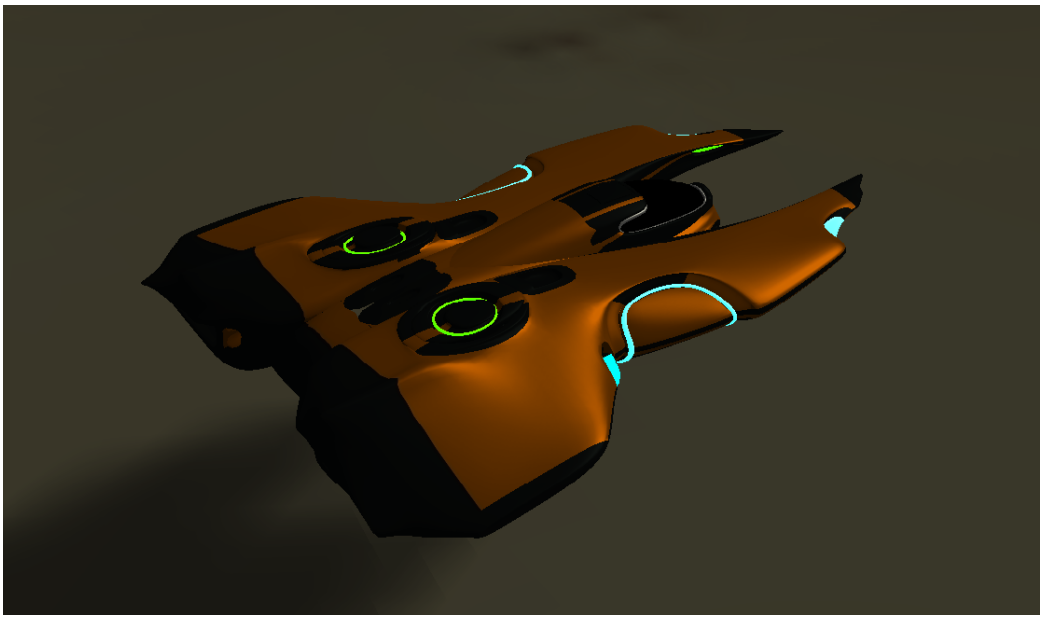


Figure 3.1.4: Result of using Phong reflection model with an emissive component. The emissive component can be observed on the green and blue lights. The specular light is most visible as light orange on the racer's right wing. The diffuse component makes the racer's shape discernible.

3.2 Bump Mapping

Surfaces of real materials are often rough or uneven. Even the smoothest surfaces have imperfections that can not be seen by the naked eye. These imperfections play a large role in how light is reflected. In order to simulate the behavior of real world surfaces for the 3D objects in the game we implemented Bump Mapping.

Bump Mapping is a technique that can give more detail to objects in a 3D environment. The technique modifies the surface normals, and thereby the way that light reflects from the object. This change in light reflection can contribute to the perceived shape of an object in 3D. Bump mapping was first introduced by Blinn in 1978 (Blinn, 1978).

Bump mapping is a technique that aims to counteract a common problem with 3D models, which is the fact that they often appear too smooth. Surfaces in the real world often have imperfections, and those can be modeled with bump maps. With bump maps one can achieve the same result as a higher detail model, but to a lower performance cost.

A commonly occurring Bump mapping technique is Normal mapping. In Normal mapping, a texture containing three dimensional normal vectors is used. The texture can then either be used to replace or offset the existing normals. The new normals are then used for lighting calculations. Bump mapping is a computationally cheap way to add more detail to the world.

A more recent technique uses the Geometry Shaders to actually displace the geometry rather than just changing the normals and the lighting. This technique is referred to as Tessellation with Displacement mapping. This is more expensive than normal mapping, but it gives better results and is still cheaper than a higher detail model (Nvidia Tesselation, N.D).

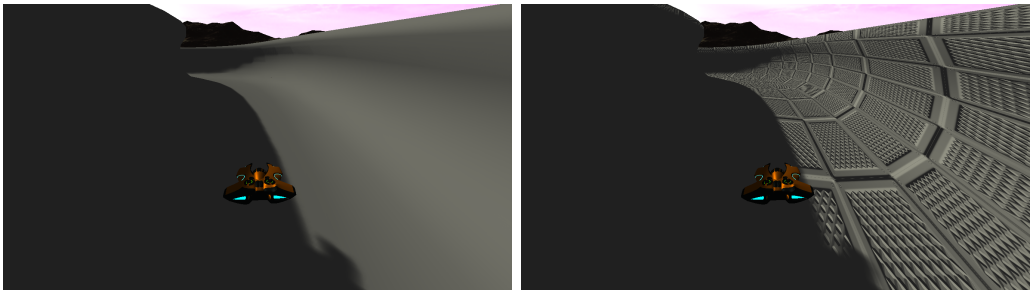
3.2.1 Implementation of Bump Mapping

Tessellation is a feature available in DirectX11 and higher (Nvidia Tesselation, N.D). Therefore due to the constraints of using the game development framework XNA, which only supports DirectX9, implementing Tessellation was not an option. The Bump mapping technique used in the game is Normal mapping.

The bump mapping is done in the first pass of the renderer when rendering the models or the terrain (see Section 3.3 Deferred Shading). In the case where the normals are overwritten by the normal map, the output normal value is simply the value stored in the texture's RGB channel. The offset variant must be used for objects that can be scaled or rotated since such modifications would change the original normal. In the offset method, the tangent of the original normal is required in addition to the normal. The bi-tangent is constructed by taking the cross product of the normal and the tangent. The three vectors make up a linearly independent system, a coordinate basis. With this basis, the data from the normal map can be applied to change the final normal.

3.2.2 Results & Discussion

We use normal mapping in both the suggested ways. The terrain applies the normals from the normal map without modification, while all other objects use the normal maps as an offset. The usage of normal maps provides a lot of additional detail to the models in the game, as can be seen on the bridge model in Figure 3.2.1.



(a) Bump mapping disabled (b) Bump mapping enabled

Figure 3.2.1: The algorithm is only applied to the bridge model for comparison. In (a) there is no bump mapping and the model lack detail and is too smooth, in (b) bump mapping is applied and the model looks more detailed. The part of the bridge that is in shadow is unchanged as the normal map only changes the lighting and not the actual geometry.

One problem with normal maps that we encountered is the data format of the normal map file. Depending on where and how the normal map was created it can use a different coordinate system for the normals than what is used in the game. This could lead to lighting bugs that can be hard to notice for smaller objects. However, if the object is large enough the lighting caused by incorrect normals is often easily noticeable and fixed. For example the normals for the terrain were on a different format than what is used in XNA. As we do not, in most cases, create the content and models for the game ourselves, we do not have full control over the format of the normal maps. The bump mapping was not used to full effect in the game as we only had one model with available normal maps. If we had more control over the art assets in the game, we would have used bump maps for most objects as it makes them look more realistic.

Given that the source file does not have misaligned or incorrect normals, no noticeable artifacts were introduced in the game by the Normal Mapping technique. With any type of correct normal map the objects in the game showed higher detail, even if the normal maps had a low resolution.

3.3 Deferred Shading

There are many objects in the world beside the sun that emit light. To simulate the complexity of the real world, a large amount of light sources are often required. In order to efficiently support the use of multiple light sources an appropriate rendering technique was required.

The two techniques that were considered are Forward Shading and Deferred Shading. Both techniques create an image where all objects have been lit using the Phong Reflection Model (see Section 3.1 Lighting). The difference lies in how the techniques apply the light to the objects. The Forward Shading technique process all lights for all objects in one pass. The disadvantage of the Forward Shading technique is that the performance cost of lighting the scene is bound to the number of objects, in addition to the number of lights. To enable more lights and for the game we use the alternative technique Deferred Shading. In Deferred Shading the performance cost of lighting the scene only scales with the number of lights, independently of how complex the scene geometry is.

The concept of Deferred Shading was first introduced in 1988 (Deering et al., 1988). The paper did not mention the word deferred but it mentioned the most important concept of Deferred Shading: Every pixel in the final image is only shaded once per light source. The usage of geometry buffers (G-Buffers) and Deferred Shading similar to what is in use today was introduced in the paper Comprehensible Rendering of 3D Shapes (Saito & Takahashi, 1990). The first game that used Deferred Shading was Shrek for the original Xbox. The Shrek game was released in 2001 and was developed at DICE by some of the people behind the SIGGRAPH talk (Geldreich, N.D). Deferred Shading was rarely used prior to a talk at the annual SIGGRAPH conference in 2004 ”Deferred Lighting and Shading”, which promoted the technique. Deferred Shading was still uncommon until the development of S.T.A.L.K.E.R Shadows of Chernobyl, where many additional techniques were covered (Shishkovtsov, 2005).

3.3.1 Implementation of Deferred Shading

Rendering the Geometry Buffers The first step in modern Deferred Shading is creating geometry buffers using the 3D models that can be seen by the camera. In DirectX 9 and higher, a feature referred to as multiple render targets is available, which is necessary for deferred shading (Duluk Jr et al., 2001). The depth, color and normal buffers and a buffer containing specular and glow masks are set as render targets, and then, all geometry is drawn to these buffers using a shader. The different buffers are covered more in depth below.

The depth is stored as the view space depth from the camera, giving a linear distance from the camera to the far plane. Storage of the depth in linear space

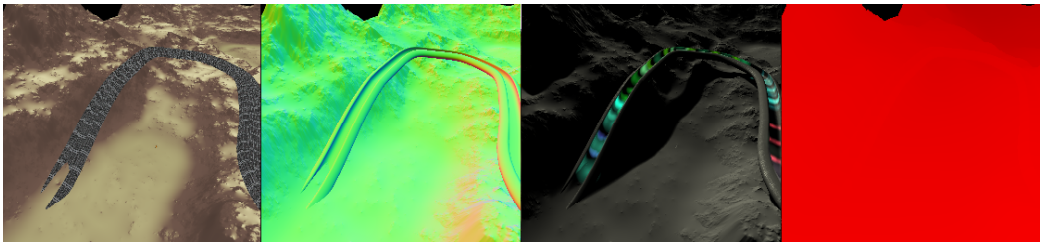


Figure 3.3.1: The render targets, not including final image. From left to right: color buffer, normal buffer, light map and depth buffer

works well with several post processing effects. Linear view space depth also preserves equal precision for all distances between the near and far plane. When using non-linear perspective-space depth, the precision is very high near the near plane and much lower close to the far plane. Non-linear depth is useful for applications with very large distances and detail in the foreground so that the depth precision is used where it matters. When the distances used are small in general, either works well as even a reduced precision of a 32-bit buffer is very high (Kemen, 2012).

The color is either sampled from a texture, or the base diffuse color of the object is used. The normals are the objects' base normals or normals changed with a normal map (see Section 3.2 Bump Mapping).

In this process, glow and specular masks are also generated in a separate specular and glow render target. The specular mask is used to modify the amount of specular light on different surfaces. The glow mask is used together with the color buffer to create emissive light sources.

All render targets must be the same size when using multiple render targets (Duluk Jr et al., 2001). In our case, we only have 16 bits left in the specular and glow render target in case we want to render additional information using only 4 render targets of 32 bit depths.

Calculating the Lighting There are two main types of lights used in the game: directional global lights and point lights. The techniques to render these are somewhat different. The light from all lights is added with blending in a lightmap render target. For directional lights the entire resolution of the G-Buffer is used and light is calculated for all pixels, by using a quad that covers the entire screen. For point lights, a sphere of the same size as the light's area is used. The vertices of the sphere are sent to the shader to calculate light only for the area of the screen covered by the sphere. This makes the lighting much more efficient, and as a result, several small light sources have the same cost as one large light source. In the shader, the vertex position on the sphere would be used for light

calculations if not corrected. Instead, the position is reconstructed using the depth buffer and screen space position of the pixel. The light calculation is a Phong calculation with diffuse, specular and glow (see Section 3.1 Lighting). The ambient light is added for the directional light to avoid additional shader passes, although the ambient light could be added separately. Shadows are also applied in the light shaders to change the amount of light that is added to the light map (see Section 3.4 Shadows).

Combining the buffers and adding a Skybox effect The lightmap is blended with the color buffer in a last step with color correction. This is done in a shader over a full-screen quad. In this step, the skybox is drawn. It is drawn only where the depth value is equal to the farplane value. The blend step produces the final backbuffer that can be either displayed to the screen or used for post processing. The Deferred Shading algorithm implemented in the game does not handle transparent objects, as only one depth value can be stored in the depth map. Instead transparent objects can be drawn after the blend using an additional pass with Forward Shading.

3.3.2 Results & Discussion

The main advantage of Deferred Shading over Forward Shading is that the geometry is decoupled from the shading. In Forward Shading, all lights are calculated for each object. This gives high complexity when there are many light sources in the game. Compared to if the game used Forward Shading, with Deferred Shading it is possible to use many more light sources without decreasing performance. Another advantage that is useful for many post processes is the G-Buffers that are available in a Deferred Shader. The depth buffer and normal buffers were easily accessed, which made implementing post processing less complicated.

The amount of lights possible with Deferred Shading is well suited for the kind of racing game we developed. As we wanted a feeling of speed, the possibility of having many smaller lights that can go by at high speed adds to the effect. Having access to the G-Buffers has helped us a great deal during the development process of other techniques and made them much easier to understand. Deferred Shading saved us more time than it took to learn and implement.

The major issue we had with Deferred Shading is that we could not use transparency. Instead, we solved that through drawing transparent objects such as particles after the deferred lighting pass using a Forward shading method. The resulting image quality is not much different from using a Forward Shader,

but we had the ability to use many more light sources. This allowed us to create a more vivid environment, as can be seen in Figure 3.3.2.

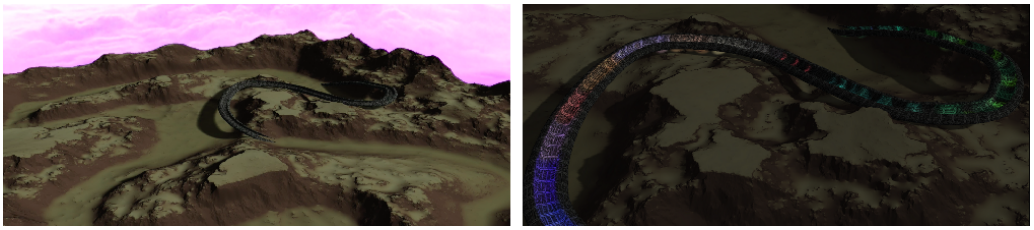


Figure 3.3.2: The left side image displays an overlook of the terrain rendered with deferred rendering and only a directional light source. The right side image illustrates the use of one directional light source and many smaller light sources along the bridge.

3.4 Shadows

Shading is an important visual aspect to consider when creating graphics for a game. Shadows play a large role when humans interpret shapes. Furthermore, we use shadows to orient ourselves, as they affect how we perceive the objects around us. Therefore, it is important that they are of high visual quality. The virtual shadows also have to possess a certain degree of realism, based on what you might expect from everyday life. In this section a number of techniques for creating shadows in 3D graphics will be introduced and discussed. The techniques chosen for this project will also be evaluated further, while considering the results they generated in the game.

3.4.1 Simulating shadows in 3D graphics

Shadow is in essence the absence of light. Therefore, light distribution plays a large part in what ends up in shadow. Light is made up of clusters of electromagnetic energy called photons. The physically correct way to represent shadows is to simulate the paths of photons from where they are emitted, through their reflections and refractions to where they are finally absorbed. There are several techniques with varying abstractions that try to simulate light and shadows through the behavior of photons or light rays. Some examples are photon mapping and ray tracing. This generates stunning graphical results. The only problem is that they are very computationally expensive. Therefore, we are forced to look for other solutions when simulating shadows in real-time, where performance is key.

Shadow mapping was introduced by Williams in 1978, and it is a process by which shadows are created in computer 3D graphics (Williams, 1978). Shadows are created by testing which pixels are visible by the light source: those not visible are in shadow. It is one of the most widely employed techniques for creating shadows in real-time applications and has been used in a large amount of PC and console games. The main contender to the shadow mapping technique was introduced by Crow in 1977 and is referred to as shadow volumes (Crow, 1977). A shadow volume describes the 3D area that is occluded from the light source by the geometry. Shadow volumes, just like shadow maps, have been used in a large number of games. The technique made a noteworthy impression as a good way to create shadows through its appearance in the game Doom 3. In order to clarify the reasons behind our choice of shadow simulation technique, it is beneficial to go over the principles behind the main contenders.

Shadow mapping The basic principle of applying shadows with the use of shadow maps consists of two steps. The first is to generate a depth map, which stores information about the distance to the visible geometry in the scene from the light's point of view. The second step is to use this information to apply shadows to the scene during the rendering process.

When creating the depth map, the scene is rendered from the light's point of view as if it was a camera. Depending on the type of the light source, either perspective projection or orthographic projection is used. Perspective projection can be used for light sources such as point lights, where the area of effect varies. Orthographic projection is useful for directional lights, such as the sun, where the light source is far away.

The depth buffer is extracted from our light perspective rendering, since the depth is the only relevant information. The depth is stored as color values in a texture. The depth map must be updated any time the light source or some geometry in the scene moves. If multiple light sources are to be used, one depth map will be needed for each light source. The next part involves shading the scene during rendering by using the depth map. The scene is rendered from the camera's viewport, one pixel at a time. While this is happening, the position of the area visible in the pixel is compared to the depth map, in order to find out if it is visible from the light's point of view. In order to perform the shading calculation on the object of interest, i.e. the area represented in the pixel, it has to be transferred from camera space to light space. The transfer is done using matrix multiplications. Once it has been transferred, the comparison is straightforward. The depth or Z-value stored in the area of interest is compared to that stored in the depth map. If it is greater than that stored in the depth map, i.e. farther away from the light, it is considered occluded by some geometry and

therefore in shadow. Depending on the result of this test, the pixel shader draws this particular area as either in shadow or in light.

Shadow volumes Shadow volumes did not become relevant for real-time applications until 1991 when Tim Heidmann introduced a way to render shadow volumes fast enough for real-time using the stencil buffer (Heidmann, 1991). The basic concept for using shadow volumes can be described as follows: The first step is to construct the shadow volumes. To do this all silhouette edges visible to the light source needs to be found. The silhouette edges are then extended in the direction of the light, i.e. away from the light source, to form the shadow volumes. Depending on the implementation a front and back cap might have to be added to each surface to form a closed volume. These shadow volumes now contain the depth information needed to define what is in shadow as the scene is rendered. The process of rendering the scene consists of a number of steps. First, the scene is rendered as if it were completely in shadow. Then, for each light source, the depth information from the scene is used to construct a mask in the stencil buffer. The stencil buffer is used to mask the visible geometry that is in shadow. The scene is then rendered again as if light is hitting all surfaces, but the mask is used to prevent the shadowed areas from being lit. Additive blending is used to add the rendering from each light to the scene. There are a number of variations to how shadow volumes are commonly implemented. The difference between these methods can mostly be seen in how they generate their masks. Some implementations use one pass and some two, since some require less precision in the stencil buffer.

Choosing a shadow simulation technique When comparing the two techniques and considering which might suit a certain application best, there are some aspects to consider. Shadow maps is often faster, depending on how much time is needed to fill the shadows. This is dependent upon the resolution of the shadow map for shadow mapping and the size of the shadow volumes for shadow volumes. In addition, shadow volumes require the use of an extra buffer, the stencil buffer, which slows it down further. Another thing to consider is that shadow volumes are often considered more accurate as they produce consistent shadow quality. Shadow maps on the other hand are reliant on their resolution for quality and this poses problems for large scenes or when you zoom in. They do however offer a lot more flexibility and a lot of the aliasing problems associated with shadow mapping can be negated or removed completely by implementing variations of the technique such as cascaded or variance shadow maps. With regard to the amount of graphical effects we aimed to incorporate into the game, we considered a future scenario

where we eventually run into performance issues a very real possibility. Thus, we opted for the less performance costly option, shadow mapping.

3.4.2 Additional work on shadow mapping

Shadow mapping in its basic form often fails to produce satisfactory results. In most cases some aliasing artifacts are present and the technique needs to be adapted to the graphical setting of the application. Fortunately, there are a lot of ways to improve or adapt the technique in order to remove artifacts and achieve high quality shadows. The techniques relevant for achieving a satisfying shadow quality given the circumstances of this particular game are introduced below.

Shadow biasing Erroneous self-shadowing or surface acne is a common problem when working with shadow maps. Each texel in the shadow map has one depth value defined for that entire texel, the value at the center of the texel. The problem occurs as the shader compares an actual depth value of some position against the depth value in the shadow map. This position is as likely to be self-shadowed as to have no shadow: depending on the angle at which the light hits the geometry. Another way surface acne occurs is through precision errors. This occurs when the difference in depth value between a texel in the camera's view that has been transferred to light space and the corresponding texel in the depth map is so small that it erroneously fails.

A straightforward and often adequate solution for solving any self shadowing problems is to add a fixed depth offset to the pixel position in light space, as illustrated in Figure 3.4.1. When using a depth offset, one thing to look out for is the so called Peter Panning effect. If to large of a depth offset is used, it can create the illusion of objects hovering by removing the shadows close to their base.

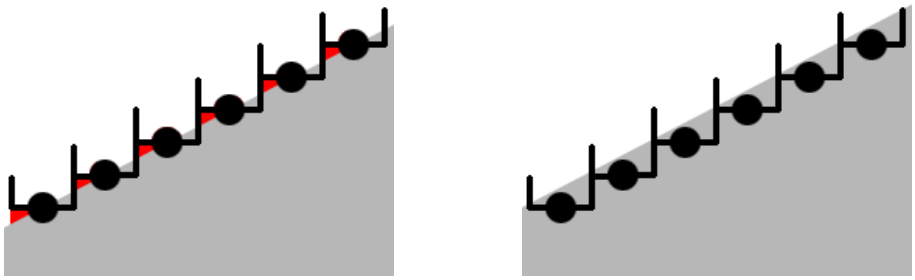


Figure 3.4.1: This image shows the depth values representative for these areas as the black dots. The red areas are where the self-shadowing occurs. As can be seen in the right hand image, this can be solved by offsetting the depth values.

Another solution is to use a slope scaled depth-bias, as illustrated in Figure 3.4.2. This technique aims to take care of surface acne while avoiding the Peter Panning effect. Areas with steep slopes, compared to the light's point of view, tend to need a larger offset than flat areas. Therefore, the offset is adjusted accordingly.

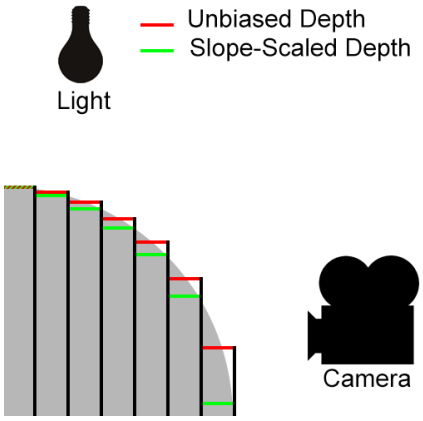


Figure 3.4.2: The image shows the principle of a slope scaled depth-bias. As the slope gets steeper from the light's point of view, the offset represented by the green lines gets larger

Percentage closer filtering One common problem with shadow mapping is the jagged blocky artifacts that appear on shadow edges under certain circumstances. These artifacts occur when the projection of the shadow map onto the geometry causes the shadow map to become magnified (Bunnell &

Pellacini, 2004). These aliasing artifacts can be negated through the use of rendering software with programmable shaders, which allows for the use of a technique called Percentage Closer Filtering which was introduced in 1987 by Reeves (Reeves et al., 1987). To solve the aliasing issue the shadow map is sampled multiple times for each pixel and the samples are then averaged together.

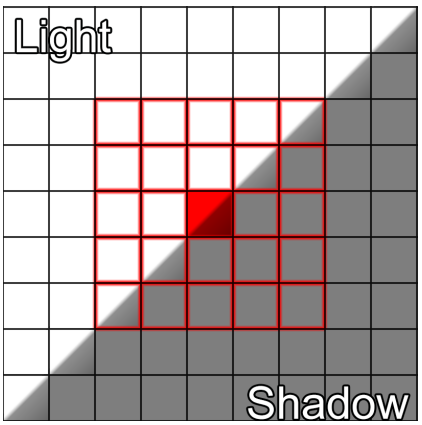


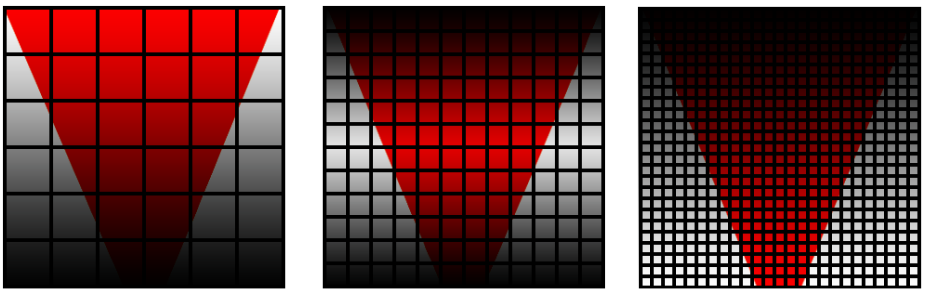
Figure 3.4.3: The red outlined area surrounding the bright red center pixel represents its sample region.

The amount of shadow in the center pixel in Figure 3.4.3 is a result of the average shadow per sample in the surrounding area outlined in red. The sample region is not pixel oriented, i.e. each sample does not correspond to one pixel. Instead, the size of the sample region is defined by the number of samples and the shadow map size. Increasing the number of samples creates smoother edges and removes more artifacts, but it is also more expensive. There are some variations to how the sampling is done in different implementations, some use random sampling while some sample in a fixed grid.

Cascaded shadow maps When working with large environments in games and a far away light source such as the sun, there is one problem that often occurs. If the camera is closer to the area that is being rendered in shadow than the light source, the size of the area being rendered will be larger on the camera's projection plane than the light's projection plane. The size disparity means that multiple pixels in the rendered output will be mapped back to a single pixel in the shadow map. This is referred to as perspective aliasing. It is one of the most prevalent problems with shadowing and can sometimes be difficult to overcome. There are several ways to solve this, but cascading shadow maps (CSM) tend to be the best solution and are commonly employed in most modern games (Microsoft, 2013).

The basic concept of CSM is fairly straight forward. The frustum is partitioned into multiple frusta and a shadow map with varying resolution is rendered for each subfrusta. The pixel shader then samples from the map that is closest to the required resolution of the current pixel being rendered.

Shadows are of the highest quality when there is a 1:1 relation between the mapping of texels in the shadow map and pixels transferred from camera space to light space (Microsoft, 2013). This phenomena and its effect on shadow quality is illustrated in Figure 3.4.4. Optimally something like that seen in Figure 3.4.5 should be achieved.



(a) Low resolution shadow map (b) Medium resolution shadow map (c) High resolution shadow map

Figure 3.4.4: The white texels represented in the red view frustum of the images are the areas with highest quality shadows, where there is a 1:1 ratio. The darker areas have decreasing shadow quality as they grow darker. The images display how the white area moves through the view frustum with regard to changes in the resolution of the shadow map.

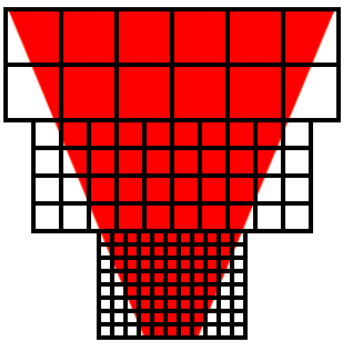
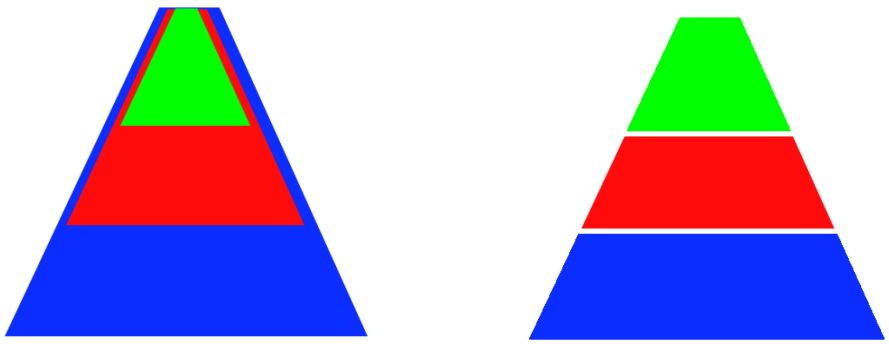


Figure 3.4.5: Different shadow maps of varying resolution illustrated in a single frustum.

To achieve optimal shadow quality throughout the view frustum, the frustum will have to be divided into subfrusta. Partitioning into multiple frusta basically means creating several near and far planes along the Z-axis of the view frustum. Whether you chose an exponential or linear increase in the size of the subfrustums is not so important, it is merely a preference. There are two typical ways of doing this, they are referred to as "fit to scene" and "fit to cascade" in a paper by Microsoft which discusses the subject (Microsoft, 2013). In the fit to scene method new far planes are created along the Z-axis for each new subfrusta's far plane while letting them all have the same near plane. This basically means that the previous subfrusta is included in the next one. When using the other alternative, fit to cascade, the near plane of the next subfrusta is placed by the previous subfrusta's far plane, thereby wasting no shadow map resolution. The basic concepts of the two methods are illustrated in Figure 3.4.6.



(a) Fit to scene.

(b) Fit to cascade.

Figure 3.4.6: The left side image illustrates the fit to scene method, while the right side one illustrates the fit to cascade method. The green, red and blue areas each represent a subfrusta.

Orthographic view and projection matrices are then created for each subfrusta. These are used to render a shadow map for each subfrusta. One way of doing this is by rendering to the same texture or render target several times, one for each shadow map, but with different viewports for each. When applying the shading, the pixel shader samples from the shadow map that contains the correct Z-value for the current pixel being rendered.

Due to the fact that percentage closer filtering had already been implemented there were visible seams between the shadow map layers, as can be observed in Figure 3.4.7. To remove the sharp edges visible along the borders between shadow maps, we needed to blend between the shadow map layers. This led

to us opting for the fit to scene method, as it made it a lot easier to seamlessly sample and blend between the different shadow maps.

(a) Morph between levels disabled. (b) Morph between levels enabled.

Figure 3.4.7: Without the morph, as can be seen in the left side image, an edge is visible between the different shadow maps.

3.4.3 Results & Discussion

After implementing basic shadow mapping, and viewing the results, we noted that we had some aliasing artifacts. We considered these artifacts visually disturbing and agreed that the current shadows could not be considered satisfactory. We then proceeded by handling the most obvious aliasing problems. The first step in handling them consisted of understanding the cause of the problem. The cause of erroneous self-shadowing became evident early on whilst researching how to improve shadow mapping. The solution also did not prove that difficult to implement. Other problems such as perspective aliasing and their causes did not become evident until quite a bit of research had been conducted. Their solutions, such as cascaded shadow maps, were also rather complex and required more time to implement.

After implementing the shadow mapping variations or improvements (see Section 3.4.2 Additional work on shadow mapping) we considered the quality of the shadows to be sufficient for our needs. All the major artifacts relevant to our graphical setting had been handled at this point. The shadows looked similar to what one might expect from real life shadows. We had also managed to maintain a relatively low performance cost due to the nature of shadow mapping and its emphasis on speed. There are still some minor shimmering artifacts along the edge of the shadows at some angles, but these are barely noticeable unless you are standing still. With regard to the high speed motion of the game and the rarity of occasions where a player would be standing still, we deemed it

unnecessary to spend more time on shadows by trying to solve this. The final results for our shadows are illustrated in Figure 3.4.8 and 3.4.7b.

Figure 3.4.8: The image displays the shadow of the racer on a flat surface.

One future improvement that could be implemented, if given more time, is to modify the rendering of the least detailed shadow maps. The least detailed shadow maps could be static unless the light source moves, ignoring the movement of objects such as the racers. This would be acceptable, as the shadow cast by a racer at distances where the least detailed shadow maps are used would barely be noticeable. This improvement would reduce the number of shadowmap re-renderings in the game.

3.5 Heightmap Terrain

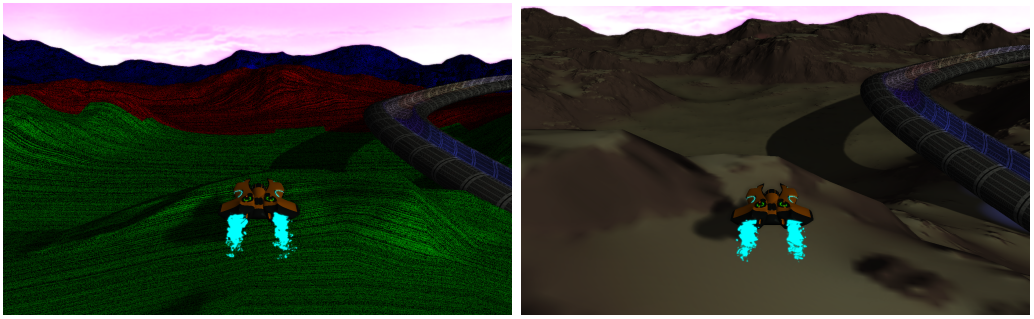
Due to the high speed nature of racing games the player often travels over large distances. Thus, large environments are needed to embody the race tracks. If the game takes place in an open environment, there will be a high amount of visible terrain at some points. To make the game run smoothly independently of the amount of terrain that is visible, the amount of triangles and visual quality must be controlled. In the following section some of the techniques used for creating a terrain that can be efficiently rendered are introduced and discussed.

There are many possible solutions when creating some kind of environment and terrain for the game. The terrain for the racing can be created using modeling software and be exported to a format that can be used in the game. The terrain can also be stored as a two dimensional height field that is used to generate vertices to render. In the case of using a model for the terrain the number of triangles required is too large and would have a negative impact on

performance. However, 3D models store data in a 3D format, therefore multiple levels of height can be used for terrain such as caves and tunnels. The height field approach is limited to a more basic representation of terrain.

In earlier implementations the height field is stored in a vertex buffer that is sent to the GPU for rendering (Losasso & Hoppe, 2004). At the time of the paper the GPU could not modify vertex buffers, and all processing had to be done on the CPU. To be able to modify the terrain on the GPU the height field could instead be stored as data in a 2D vertex texture that can be read by the GPU (Asirvatham & Hoppe, 2005). The data is used to change the height of the vertices of a plane.

If the terrain is sent to the GPU with full detail for the entire heightmap there will be performance issues as a vertex is required for every pixel in the heightmap. There are many proposed solutions to the performance problem that does not decrease the visible quality of the terrain, most are some kind of level-of-detail (LOD) system. Losasso and Hoppe used a clip map based strategy run on the CPU (Losasso & Hoppe, 2004). Later Asirvatham and Hoppe used a version running on the GPU, also using clip maps (Asirvatham & Hoppe, 2005). Continuous Distance-Dependant Level of Detail (CDLOD) suggests using a quad-tree based model to provide a better distribution of level-of-detail (Strugar, 2009). The CDLOD algorithm is based on the three dimensional distance between the observer and the mesh, visible in Figure 3.5.1a, which is a large improvement for terrain with large height differences. All the above papers also handle the problem of transitions between the different levels-of-detail to avoid gaps.



(a) Levels illustrated with color.

(b) Reference image.

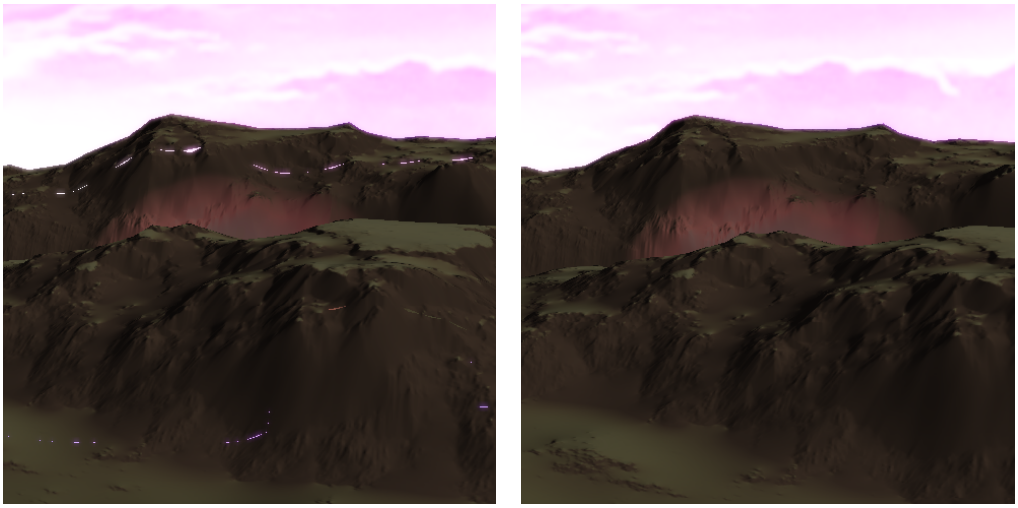
Figure 3.5.1: Figure (a) shows the different level of details where the closest (green) has the highest detail. Figure (b) shows the same scene without the color shader. The bridge to the right is not affected by CDLOD as it is a model and not part of the height map.

3.5.1 Implementation of Level-of-Detail Terrain Rendering

The data that is needed to create and display the terrain is stored in three separate image files. The files contain the height values, the normals for the terrain and the color texture for the terrain. The terrain is created by sending a set of vertices with no height values to the GPU, where they are processed using the height value image. The texture coordinates used for the images are generated using the entire vertex buffer which contains a plane of different size quads. When the final vertices have been generated using the height values, the color, normals and depth are drawn to their respective buffers (see Section 3.3 Deferred Shading).

The set of vertices that are sent to the GPU are generated by the CDLOD algorithm. Every time the camera moves, the level-of-detail information needs to be updated. The selection is done using a quad-tree containing all levels of detail as bounding boxes for the terrain data at that location. A set of distances from the camera are also created, to identify where the different detail levels should start. The selection starts at the highest level in the quad-tree, which contains the entire terrain in one bounding box. At every level the bounding box is checked against the distance levels and when the quality to distance ratio is found to be sufficient, or when the highest possible quality level has been achieved the bounding box is added to the selection. The result of the selection is a set of nodes with bounding boxes of different sizes. For each node a quad scaled to the size of the bounding box is added to a vertex buffer giving many vertices for high quality nearby terrain and fewer vertices for terrain farther away.

When the height values are modified on the GPU, they take the vertices that are passed from the CDLOD algorithm and displace them using the heightmap image. This introduces an error known as gaps between different levels of detail. In Figure 3.5.2a the gaps can be seen as the background skybox leaks through. The solution to this error is to morph between the different levels of detail and sizes of the quads (Strugar, 2009). The morphing is done for the last 50% of every level so that it transitions smoothly to the lower levels of detail. For the GPU to know which level-of-detail the current vertex is in, metadata is sent to the vertex shader. In Figure 3.5.2b the morphing has taken care of the gaps and no artifacts are seen where the transitioning between the levels occur.



(a) With gaps in the terrain. (b) Gaps in the terrain fixed.

Figure 3.5.2: (a) The scene with visible gaps between different level of details (b) The same scene with gaps fixed.

3.5.2 Results & Discussion

In the first versions of the game, we used a 3D model for the racetrack and terrain. If we had managed to use a 3D model effectively, it would have saved a lot of time since the code required for rendering models was already implemented. We soon realized that it would require a lot of work and experience to produce a model of high enough quality for the game while still keeping the number of polygons low. We tried to look online for free models to solve the problem of creating a good on ourselves, but there was not much to find. The usage of heightmaps started as a way to solve the content creation problems we encountered. To create the terrain, we used the program World Machine 2.0 which has a vast array of tools to generate a realistic terrain.

We first used the heightmap to create a vertexbuffer on the CPU but with this approach the quality was very low with visible triangles. Keeping the quality low was needed to keep the polygon count for the entire terrain within reasonable levels. The solution was to use the CDLOD algorithm and to generate the terrain on the GPU.

With the CDLOD algorithm we had a high quality terrain without visible triangles. With the LOD system the performance cost was low even with much higher quality close to the camera than previous solutions. Two visible artifacts were introduced with the heightmap and CDLOD terrain. One is that the 4096*4096 texture (maximum possible size in XNA) used for the color of the ground has too low resolution to adequately represent the entire terrain. This can

be fixed by using a system of tiled texturing, level-of-detail for the color texture or by increasing the size of the texture. We found that while the color artifacts are quite visible when standing still they do not cause any major disturbance when going in higher speeds, as such there was no need to handle the problem. The second artifact was the gaps produced between the detail levels. This problem was fixed using the morphing technique described in the CDLOD paper. Level-of-detail is a subject with much research available and if given more time the implementation we used could be improved. With regard to the timeframe we found that the implementation was satisfying enough to not improve it further.

As the game is played in high speed and the camera moves relatively far each frame we encountered a problem with the change in detail of different formations in the terrain. If the second level of detail started too close to the camera or if the morphing was done over a too small part of each level the transition was visible to the player. We fixed this by having the second level-of-detail start relatively far away from the camera and by starting the morphing when 50% was left of the current level. With the fix the change in the terrain from low to high detail take enough time that it is not noticeable. The extra triangles used by having more terrain covered by the most detailed level and the extra morphing calculations compared to morphing over less of the level did not cause a noticeable performance cost.

The usage of height-maps and the CDLOD algorithm in particular gave a good final result. The height-maps made it possible to create a more detailed and realistic outdoor environment than what would have been possible with only 3D models. The usage of the level-of-detail algorithm gave us good detail near the camera while keeping the detail lower farther away. In racing games there are often scenes with a large open terrain and long roads with highly varying distance to objects, to achieve a high amount of detail close to the viewer without the CDLOD algorithm a lot of unnecessary triangles would have to be rendered.

3.6 Particle System

The term particle system was introduced by Reeves as a collection of small graphical objects that are used to simulate fuzzy phenomenon that are otherwise hard to simulate (Reeves, 1983). These phenomenon are often natural phenomenon or chemical reactions such as rain, snow, fire and explosions. As some of these phenomenon were to be simulated in the game, a particle system was implemented.

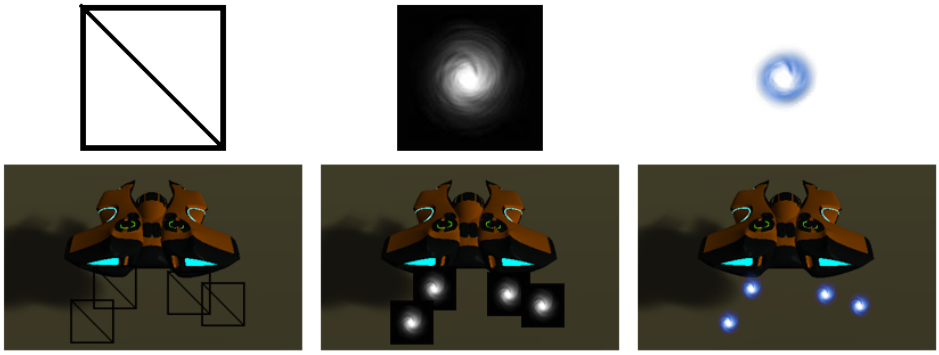
3.6.1 A basic particle system

The fundamentals of a particle system typically consist of two things, particles and emitters. The particle consists of data describing its attributes. Each particle acts autonomously but share common attributes. A particle typically contains attributes such as velocity, color, position, size, lifespan and acceleration. The emitter is what controls the attributes of the particles. It is the source of the particles and determines their behavior. It also handles parameters such as emission rate and number of particles, while being the source from which the particles are emitted. Depending on the desired complexity of a particle system any kind and number of attributes can be added to provide new options and behavioral parameters for the emitters. This basically sets imagination as the limit when creating new particle effects.

The typical update loop for a particle system, which is performed once every frame, usually consists of two distinct parts: the parameter update stage and the rendering stage. During the update stage all attributes of the particles are updated: such as each particle's position based on its velocity and its new velocity based on its acceleration. New particles are also spawned based on the emission rate and all particles that has exceeded their life spans are removed.

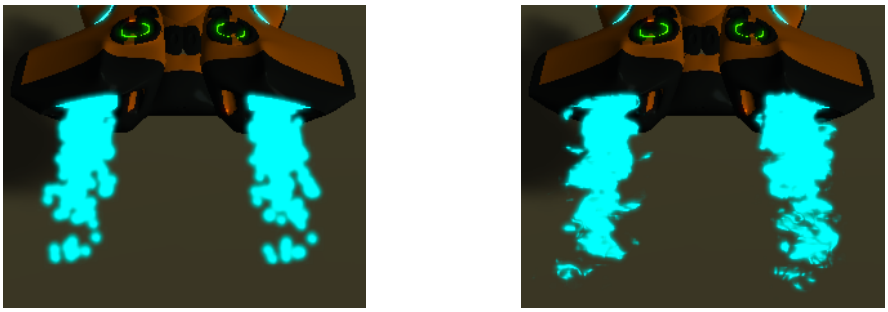
Once the update is done, each particle is rendered. A particle is modeled and rendered using a 2D quad to which a texture is attached using the alpha channel. The quad always faces the camera so as to create the illusion of it being 3D. This way of rendering objects is often referred to as Billboarding and is illustrated in Figure 3.6.1. The particle is then rendered using the shape of the texture, with its color and size assigned to it by the emitter.

As specific post processing effects, such as heat haze, should be applied exclusively to the particles, the particles are rendered to an off-screen render target. The particles are then blended back into the scene after the effect has been applied on the particle render target. The result can be observed in Figure 3.6.2



(a) The Particle as a 2D quad. (b) Texture applied to the quad. (c) Color and alpha added to the particle.

Figure 3.6.1: The principles of rendering a particle. The top images show a particle during the conceptual stages. The bottom images are screenshots from the different stages.



(a) Heat haze disabled (b) Heat haze enabled

Figure 3.6.2: The plasma emitted from the thrusters of the racer is a particle effect. In image (a) the heat haze effect is not enabled and individual particles are clearly visible. In image (b) the heat haze effect is enabled, which provides a more vivid result where the effect does not give away that it consists of a number of particles.

3.6.2 Results & Discussion

Most of the particle effects we wanted in the game were somehow related to various gameplay aspects. Due to the relatively small amount of gameplay implemented in the final product, our need for particle effects were fairly limited. There are few events and such that could trigger explosions and similar effects. The primary need for a particle system was to simulate the plasma emitted by the thrusters of the vehicles. This phenomenon did not call for a very complex

particle system and could be created using a basic implementation. The result of the effect simulating the thrusters adds a lot to the visual impression of the game and we are satisfied with how they turned out. The effect also adds a sense of movement and speed to the game. As previously mentioned, we render the particles to an off-screen render target in order to apply a heat haze effect to them. Due to the blurring associated with this effect, we could use a small amount of particles and still achieve great results. The visual impact it adds to the game is remarkable when considering its basic implementation. It causes no visible artifacts and also maintains a low performance cost despite almost no optimization. This is largely due to the low number of particles in use. If more particle effects were to be added, some optimization would have to be done to maintain acceptable frame rates, especially if some of them contained a large amount of particles.

Future work would mostly involve creating various particle effects. Some of these effects would likely require adding additional complexity to the system and optimizing it. Soft particles as described by Lorac would also have to be implemented for particles that risk intersecting scene geometry (Lorach, 2007). Possible optimizations would aim to minimize overdraw, and prevent fluctuations in frame rate due to a sudden increase in visible particles. Therefore, something similar to the technique presented in GPU gems 3 on "High Speed, Off-Screen Particles" might serve us well if we were to add dust or smoke effects to the game (Cantlay, 2007).

Chapter 4

Post Processing Effects

A post process effect is an effect that is applied to an image of a scene after it has been rendered. Post processing can be used for effects that add a certain aesthetic to the game, or to visually present information to the player.

4.1 Structuring of the Post Processing effects

To produce a game of high graphical quality, that can be run in real time, several different post process effects can be useful. To implement post-processing in an effective way, that can handle many different types of post processing, a manager is needed. One implementation includes having advanced and basic post processes. Advanced post process effects contribute to the final image and are built using several basic post process effects added together. Basic effects are used in many advanced effects and some of the most used are blurs and masks.

4.1.1 Implementation of a Post Processing Manager

Data input from the scene is required to render a post process. The buffers that are used in most post process algorithms are depth, normal and color buffers. These buffers can easily be acquired from the G-Buffers available in the Deferred Renderer (see Section 3.3 Deferred Shading).

The game supports one to four players playing at the same time in a split-screen environment and there are four different viewports that all require post-processing. The ability to display different effects for each player is important to the visual impact of the game. The game has one post process manager for every active player and every manager can have a set of post processes independently of the other managers. The manager can add and remove effects in runtime, and effects can also be disabled without having to be

removed.

The post processes are split up in two variations depending on their usage and implementation, there are basic effects and advanced effects. A basic post-processing effect has its own set of parameters and is executed in a single pass using a shader over a part of the screen. An advanced post process does not have its own shaders, instead it is composed of one or many basic post-processes.

To produce the final image that contains the result of all the post processes, the manager loops over all its advanced effects executing them in order using the result of the previous process as the input for the next. This adds another level of sequencing that can be used for advanced effects. An example of where this is taken advantage of in the game is when the Bloom effect is applied after the Heat-Haze effect for the plasma thrusters. This is done in order to enhance the glow of the plasma.

4.1.2 Results & Discussion

A major advantage to using the structure of basic and advanced post processes is the reuse of code. Instead of rewriting the blur function for SSAO and Bloom and all other effects using it we can just use the same basic effect with different parameters. By using a manager the advanced effects can also be sorted in any way, and as a result new artifacts can be avoided.

4.2 Blur

There are several post processing effects that require the removal of noise or artifacts. The ability to smear color to nearby pixels is also useful for some post processing effects. There are a lot of different blur-techniques used in modern 3D-games and we introduce and discuss those relevant for the game.

There are several blur algorithms that can be implemented and used. The difference is the amount neighboring pixels contribute to the center pixel's new color. The two main algorithms that are interesting for usage with SSAO and Bloom and other effects implemented in the game are Gaussian Blur and Bilateral Blur.

Gaussian Blur Gaussian Blur is a weighted blur. The weights are distributed using a gaussian function, which is based on the distance from the center. The function used for the blur is shown in Figure 4.2.1 and a curve of the function is shown in Figure 4.2.2.

$$\frac{1.0}{\sqrt{2 * \pi * blurAmount}} * \exp \frac{-n^2}{2 * blurAmount^2} \tag{4.2.1}$$

Figure 4.2.1: The blurAmount variable sets the strength of the blur, while n is the offset from the center pixel.

The weight is used to calculate the amount of color the pixel receives from the neighboring pixel. The farther away from the center pixel the less effect it has.

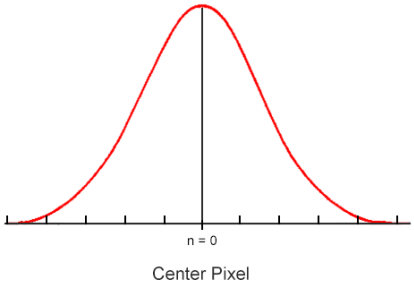


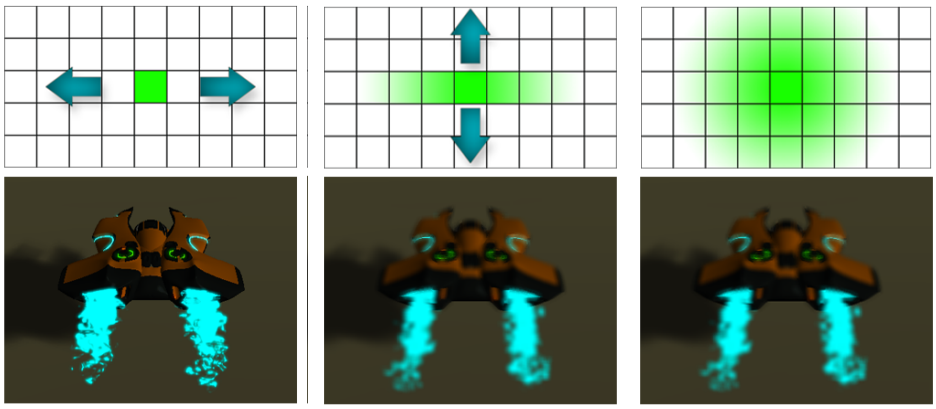
Figure 4.2.2: The red area represents the sample region for the center pixel. All the pixels in the area contribute to the final color of the center pixel. The variable, n, is the offset from the center pixel.

Bilateral Filter Bilateral Filter uses the same base algorithm as the Gaussian Blur but adds an edge detection algorithm to avoid edge based artifacts. The Bilateral filter samples the normal and depth map in addition to the scene image sampled by the Gaussian Blur. Edges can be detected in two cases. When the depth difference between the center pixel and the sample is greater than a certain value there is an edge. The other case is when the angle between the center normal and the sample normal is over a threshold there is an edge as the sample's geometry faces another direction. Depending on the application everything outside the edges can be ignored when blurring a pixel. For example, when using Bilateral Filtering the ground color does not bleed to the racer.

4.2.1 Optimizing the Blur Algorithms

To optimize the gaussian blur the game engine applies the blur in two steps: as done in (James, 2004). Instead of calculating all values in one pass it is broken up

in two passes. First, the horizontal blur is done 4.2.3a, then the vertical blurring is applied to the result of the horizontal blur 4.2.3b. This optimization gives a linear complexity instead of a quadratic complexity as the blur is done for n + n pixels instead of n*n pixels.



(a) First pass. (b) Second pass. (c) Final image.

Figure 4.2.3: These figures display the two passes of the gaussian blur and the final result. The top images describe how one pixel blurs to neighboring pixels. The bottom images show the blur steps for an ingame image.

4.2.2 Results & Discussion

Blur is often used to reduce noise in an image. The usage of blur in the game is in most cases by applying it as a last pass in some of the post process effects to give a smoother result. One such example is its use in the last pass of the SSAO-effect (see Section 4.5 Ambient Occlusion), where its task is to get rid of the noise introduced by the random sampling. Blurring is also used as a standalone effect for the game-over screen. This causes the player to focus on the scoreboard at the end of the race. The result of the different blur implementations can be seen in Figure 4.2.4

(a) No Blur (b) Gaussian Blur (c) Bilateral Filter

Figure 4.2.4: Comparison between different blur algorithms. In (b) using Gaussian Blur the whole image is blurred. With a Bilateral Filter (c) the edges are still sharp while the other parts of the image are blurred.

4.3 Bloom

The emissive light in the game does not look like real light sources. To achieve a more realistic look, without having to add all emissive lights as real light sources, the post process Bloom can be used.

Bloom is a post process effect that modifies the lighting of the scene. Bloom is an effect that simulates when bright light bleeds on to the darker areas of a scene. The Bloom effect produces more realistic light sources and cooler special effects. The Bloom effect makes bright colors in the scene bleed to surrounding areas (James, 2004).

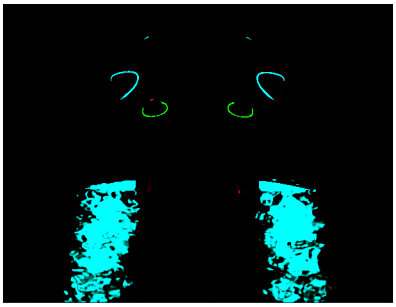
4.3.1 Implementation of Real-Time Glow

The algorithm is executed in three steps. The first step locates the brightest locations in the scene, by sample and only the pixels with a certain brightness value are saved to a buffer. In Figure 4.3.1a the original screen is shown, the lights on the racer are bright but do not affect the body of the racer. The first step is to mask out the lights of the original image. In Figure 4.3.1b the first step of the algorithm is shown.

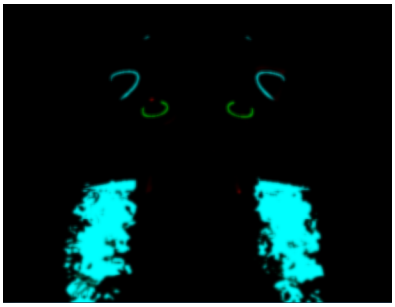
The second step of the algorithm is to blur the lights that were extracted to a separate buffer, shown in Figure 4.3.1c. The blur is done using a two step gaussian blur (see Section 4.2 Blur). Figure 4.3.1c does not have as bright colors, but the colors are smoother and more spread out giving an even spread of the light.

The last step of the algorithm is the blending of the blurred light sources to the original image. The strong bright lights in the original picture are preserved and the blurred light is added on top of the original light. A saturation function is also applied to make the resulting image have a similar overall brightness level.

(a) No Bloom



(b) Light mask



(c) Mask with blur

(d) Final result

Figure 4.3.1: (a) Source image with no Bloom. In (b), light sources have extracted from the source image. All colors over 0.5 (black = 0 and white = 1) are used in this image. In (c), the light sources have been blurred using Gaussian blur. In (d), the original image (a) has been blended with the blurred lights (c) to create the final result.

4.3.2 Results & Discussion

The resulting shown in Figure 4.3.1d has lights that spread to darker areas around the light and thereby represent more realistic light sources, look at the blue rings of light on the side of the racer for the most visible result.

Bloom can be a useful technique to achieve a sci-fi setting in a game with many glowing lights. For a racing game such as ours there are many light sources in a scene: different special effects that act as light sources, the lights placed in the environment, as well as the lights on the players' racer. We implemented Bloom to make all the light sources feel more glowing and to have them stand out in the scene. The version we use does not have a depth test but that is acceptable as we do not have a setting that would cause incorrect light bleed from the background. One example of what kind of scene could produce such errors is a house wall that clips a street light in a suburban setting, there the light

would bleed to the wall in a realistically incorrect way.

The straightforward implementation we used is simple to implement and it gives good results making it a good choice for development within the limited time frame we had. As the algorithm uses a two-step Gaussian blur (see Section 4.2 Blur) the performance cost is low in contrast to the resulting image quality we gained. We have not observed any disturbing problems or artifacts using this technique.

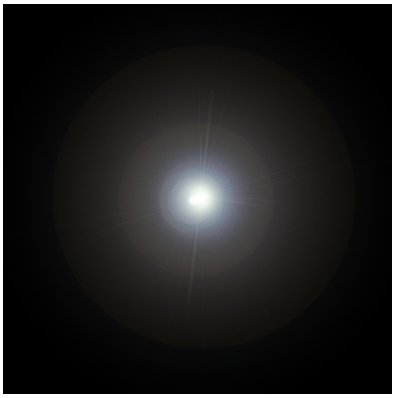
4.4 Volumetric Lighting

Small objects such as smoke or dust-particles can be illuminated by a light source. This creates an effect known as shaft of light, god rays or volumetric light. The Volumetric Lighting effect approximates the effect caused by these obstructing objects and gives a sense of volume to the light in the scene. The effect also a creates a feeling of depth in the scene.

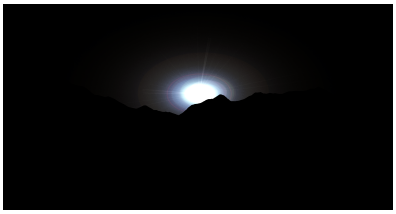
The first non-real-time implementation of Volumetric Light in 3D used a modified shadow volume algorithm (Max, 1986). Real-time implementations include a slice-based rendering technique (Dobashi et al., 2002) and hardware shadow maps (J. Mitchell, 2004). All of these techniques suffer from various sampling artifacts.
In 2007 K. Mitchell proposed a new solution to the problem of creating a low performance cost and realistic volumetric lighting. (K. Mitchell, 2007).

4.4.1 Implementation of Volumetric Lighting

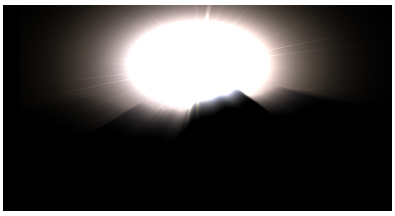
The Volumetric Lighting algorithm consists of four passes. The first pass renders a sun texture to a render target, as seen in Figure 4.4.1a. The second pass masks out objects, including the terrain, from the scene using the depth buffer(see Section 3.3 Deferred Shading), the mask can be seen in Figure 4.4.1b. The third pass is the main pass of the effect. For each pixel a vector is projected to the sun. Along the vector a number of evenly distributed samples are taken. The color value of these samples contributes to the final pixel. The values are given weights, based on distance to the center of the sun and an artistic decay variable. Samples closer to the source pixel are given a greater weight. These sample colors are multiplied by the weight and added to the source pixel, creating rays of light as shown in Figure 4.4.1c. The final pass blends the result of the third pass and the original image together. The result can be seen in Figure 4.4.1d



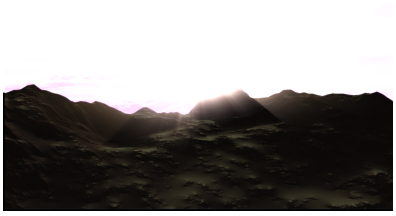
(a) First pass, sun texture.



(b) Second pass, scene mask.



(c) Third pass, rays added.

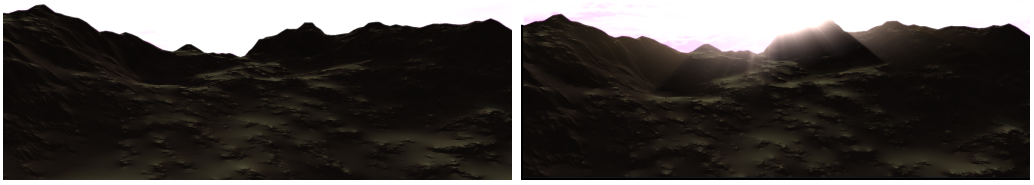


(d) Final pass, blending with source image.

Figure 4.4.1: (a) The texture used to draw the in-game light source. In (b) a scene mask has been added on top of the sun texture. In (c) the result of the third passis shown. In (d) the original image has been blended with the sun rays (c) to create the final result.

4.4.2 Results & Discussion

The implementation of Volumetric Lighting gives a sufficient result, considering the short implementation time. The effect could have been more visible if the environment was designed with volumetric light in mind. For example, an environment with trees or other objects in front of the light source would have utilized the volumetric light effect to a greater extent. An artifact caused by this method appears when the light texture is not in sight, which causes the rays to not appear. However, with the high-speed setting of the game, this artifact is barely noticeable. The final result is shown in Figure 4.4.2. We find that the visual quality of the game was improved by the volumetric light effect.



(a) Volumetric Lighting disabled. (b) Volumetric Lighting enabled.

Figure 4.4.2: (a) Displays the source image with no volumetric light for reference. (b) Displays the same scene with the effect added.

4.5 Ambient Occlusion

In real-time computer graphics the only light visible to the camera is the light reflected from a light source via some surface. In reality the light can bounce any number of times before it hits the eye, this is known as indirect lighting. To simulate some aspects of indirect lighting and to make the game look more realistic we implemented a technique called Ambient Occlusion.

Today there are many different techniques and algorithms to simulate indirect lighting (also referred to as global illumination) in real time. Ambient occlusion is a technique that simulates the amount of indirect light a pixel receives, how much of the ambient light is occluded. Ambient occlusion techniques are only required when doing real-time rendering without ray tracing. In the case of ray tracing the amount of indirect light is calculated by following the rays from the light.

Ambient occlusion has been used before as a technique in non real-time applications to approximate global illumination but before 2007 there were no real-time algorithms available. The first paper with real-time ambient occlusion compares the depth and normal of a pixel with surrounding pixels in image space, while previous algorithms calculated occlusion amount in 3D space (Shanmugam & Arikan, 2007). With the release of the game Crysis by Crytek in 2007 further progress was made on ambient occlusion in real time (Mittring, 2007) and the technique was given the name Screen Space Ambient Occlusion (SSAO). With the development of Starcraft II several improvements were introduced (Filion & McNaughton, 2008). Instead of sampling in a sphere around the pixel as had been done before (Filion & McNaughton, 2008) and instead sample in a surface aligned hemisphere. The result of the new sampling is that a flat surface has no self occlusion. In the same paper each occluder was

also given a weight based on the distance to the pixel that is to be occluded, this makes nearby occluders have more impact than those farther away. The Ambient occlusion algorithm implemented in the game is (Méndez, 2010), which is a simplification of Shanmugam and Arikan algorithm.

4.5.1 Implementation in the game

The SSAO effect consists of two passes. The first pass where the amount of occlusion is calculated and the seconded pass that adds two different kinds of blur.

To calculate the ambient occlusion term, every pixel on the screen is treated as a small sphere. For each sphere a number of neighboring spheres are sampled, as shown in Figure 4.5.1. The amount a neighboring sphere will occlude the origin sphere depends of two factors:

- Density:
  Distance “d” to the other sphere.
- Weight:
  Angle between the origin spheres normal “N” and the vector “V” between the two spheres.

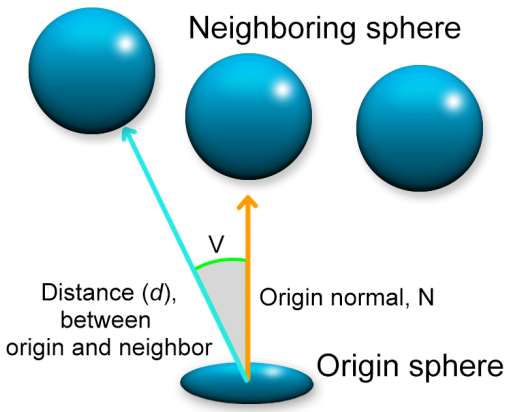


Figure 4.5.1: Image showing the data gathered from one neighboring occluder sphere. A vector from the origin to the occluder (top left) is created. The distance to the occluder is used together with the angle between the normal of the surface and the vector to calculate the amount of occlusion contributed by the neighboring sphere.

A formula that can be used to calculate the occlusion bearing these two factors in mind can be seen in Figure 4.5.2 (Méndez, 2010).

$$max(0, N \cdot V) * \frac{1}{1+d} \tag{4.5.1}$$

Figure 4.5.2: A formula that can be used for ambient occlusion.

The first term takes the angle to the normal in account: the smaller the angle, the greater the occlusion. The second term is to attenuate the effect linearly with the distance: the farther away a neighboring point is the less it contributes to the occlusion.

For each point in screenspace, a few samples of neighboring point's occlusion are calculated using the same occlusion formula above. The accumulated result is divided by number of samples to get the average occlusion for the screenspace point.

To get a more detailed result from the ambient occlusion each point needs to have random samples. Without random samples there will be banding artifacts. The result over all will be of higher quality, but as a trade off noise is introduced to the image, as can be seen in Figure 4.5.3. To achieve this we need to randomize the sample points. This is done by reflecting each sample point's position using a unique normal for every pixel. The random normals are stored in a pre-rendered texture.

Figure 4.5.3: Comparison between ambient step(left) and final blend(right) without blur step. The noise of random sampling is clearly visible in the dark parts in the top of the image.

To get rid of the noise introduced by the random samples, a second step is required. The implementation in the game uses two different kinds of blur in this step. A Bilateral blur and a Gaussian Blur. The bilateral blur gets rid of most of the noise and keeps the edges of the ambient occlusion intact. The Gaussian pass is to smooth out the remaining noise artifacts. The result of after the blur step can be seen in Figure 4.5.4

Figure 4.5.4: Comparison of ambient step(left) and final blend(right) after the blur step. The amount of noise is greatly reduced, but some small bright smudge artifacts appear in the darker parts of the image. The artifacts are visible by looking closely at the top left of the image.

4.5.2 Results & Discussion

The implementation of SSAO that we chose for the game gives a decent final result with some artifacts. The most notable contribution from the SSAO can be seen on the racers and on the terrain where they enhance the 3D feeling. The most disturbing artifacts are the dark blurry spots that occur randomly on the screen, the artifacts can be seen in Figure 4.5.4. To avoid most of the dark spots it is possible to add a second pass of Gaussian Blur with a higher blur amount, but then the occlusion will not be as sharp as desired. In our opinion the artifacts are not major enough to not use the technique though we recognize that further improvements can be made given more time. The SSAO used in the game is not a very expensive technique, when using SSAO performance was not lost on the machines we used for testing. The technique is a good fit for most environments though it could be argued it gives even more effect in detailed indoor settings. A comparison of the game with and without SSAO can be seen in Figure 4.5.5, the effect is barely visible but still contributes to a higher quality final image.

During the implementation of ambient occlusion, we encountered several problems that delayed the process. The algorithm that is used in the final product is an adaptation of the one used in Méndez's article and the one used in the Starcraft II article. We were not pleased with the initial result of our ambient occlusion using Méndez's algorithm. Therefore, we chose to implement the Starcraft II version to get more artistic control of the effect. We did not succeed in fully implementing it, however, and because of the time constraint we switched back to Méndez's version. In the end we ended up using the Méndez algorithm while adding a Gaussian Blur with a low blur amount after the bilateral blur, which gave us an acceptable result.

(a) SSAO disabled (b) SSAO enabled

Figure 4.5.5: The result of the SSAO implementation. The effect gives most effect on the ridges on the ground in the middle of the image (b). A small artifact in form as a black halo is present around the racer.

4.6 HUD

To make information available to the user, such as available power-ups and which lap the player is currently on, an in-game user interface is a sufficient solution. Some alternative techniques for displaying information to the user are discussed in the following section.

Information can be displayed in a console as text for some games, but as graphical games with a requirement for focus on other parts of the screen than the console became popular, a new way to display relevant information was necessary. The idea of displaying information on top of the visual world was inspired from its usage in fighter jets, where relevant data is displayed on a screen in from of the pilot. HUD is a technique of displaying information to the user that has been used in games for a long time. In the oxford dictionary a HUD or Head-Up Display is defined as:

```
"A display of instrument readings in an aircraft or vehicle
that can be seen without lowering the eyes, typically
through being projected on to the windscreen or visor."
```

HUDs are very common in games today and the style and design of the HUD is often an identifying factor of many games. HUDs can cover large areas of the screen with information, or they can be minimalistic and only display the most important information. In some games there is no overlay HUD and the information is instead displayed directly in the 3D world, for example the dashboard of a car or a digital display on a weapon indicating how much ammunition is left.

4.6.1 Adding a User Interface

The user interface in the game is rather simple. It is an overlay HUD implemented using 2D images and text with different fonts and colors. The HUD for each player tracks and displays statistics such as current lap and available power-ups. The updates to the game logic are sent to the HUD every frame where the displayed information is updated.

4.6.2 Results & Discussion

Implementing an in-game HUD instead of the overlay HUD we used could add some detail to the game world. However, if we were to create an in-game HUD we would need to model a cockpit for the racer, and that would take a lot of time as we had limited experience with modeling. Additional we prefer that a racing game is played in a third person perspective which does not make the dashboard a viable option to display information even if we could manage implementing it within a reasonable time frame.

The overlay HUD is comparatively simple to implement and it can display the information required. As the HUD is drawn over the final image after the rendering using 2D images, there are no artifacts reducing the quality of the image. The one thing that could cause problems is if there are color interactions between the rendered image and the HUD elements, but this can be avoided with good design.

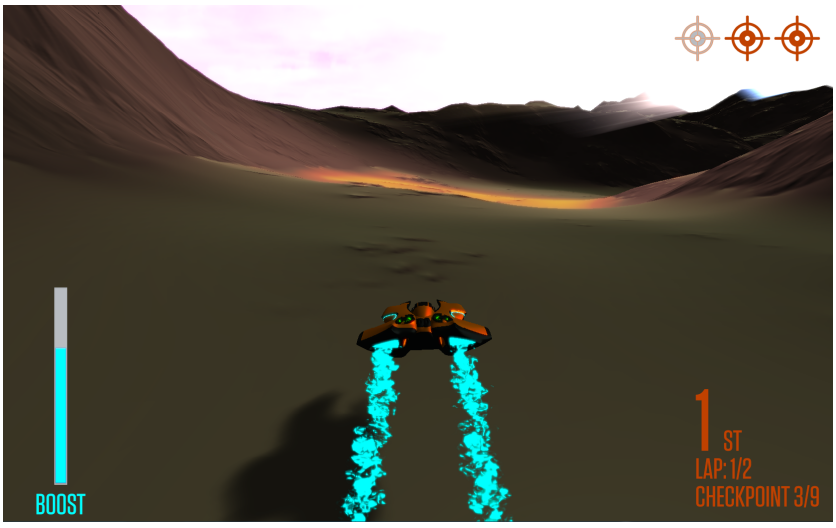


Figure 4.6.1: This is a concept image of the in-game HUD. In the top right corner the weapon status is shown. The full orange icon indicates a ready projectile. In the bottom left of the viewport the BOOST-bar is shown, and on the right competition info is shown.

Chapter 5

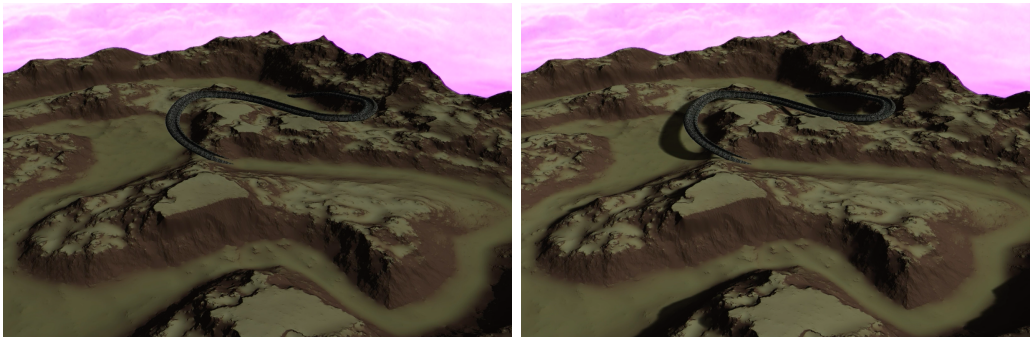
Results & Discussion

The goal for this project was to implement a graphically pleasing multiplayer racing game. By doing this we aimed to learn more about computer graphics and the graphical effects involved in creating a game. With regard to the goals that we set out with we consider the project successful. The game is in no way a finished product. It does, however, support multiplayer and contains sufficient gameplay to be considered playable. The focus of the project was graphics and during the course of the project we examined and worked with a large amount of graphical algorithms, while learning a lot in the process. As with all projects, there are some parts and solutions that we are especially satisfied with. There are also certain parts that we are less enthusiastic about, in this case some graphical effects. This was mainly because they were not used to their full potential or because they were not suitable for the environment of the game.

A decision that we are quite pleased with is our choice of framework. XNA was well documented and thus easy to work with. It provided us with the basic framework for handling the game's update loop as well as mathematical operations and resource management. It also provided useful tools for handling the controller input. This saved us a lot of time and was easy to work with once you got familiar with the code. There are some libraries with more extensive functionality, that would have let us achieve greater quality graphics. Some examples are the Unreal Engine and Unity. However, using these would have meant less of the code being implemented, and thereby understood, by us. Since learning about computer graphics was central to the groups goals we opted for XNA.

Two of the algorithms that we are especially pleased with are Heightmap Terrain using level-of-detail, shown in Figure 5.0.1a, and Cascaded shadow mapping, shown in Figure 5.0.1b. The results garnered by using these graphical algorithms are well suited to the setting of the game, which consists mainly of large open environments. These solutions allowed us to achieve reasonably

detailed models, textures and shadows close to the camera while maintaining performance by decreasing detail for parts of the scene farther away from the camera.



(a) Heightmap terrain with models (b) Shadows enabled.

Figure 5.0.1: ((a) Heightmap terrain using level-of-detail with a bridge model in the center of the environment. (b) Shadows are rendered using cascaded shadow maps, mostly visible on the left side of the bridge.

The effect that demanded the most effort to achieve satisfactory results with was the Screen Space Ambient Occlusion. Ambient Occlusion was something that we genuinely wanted to implement in order to see how it would appear in the game. The results, however, were not that great, as it is more suited for areas cluttered with objects that create a lot of self shadowing. One typical example is indoor environments. The game consists mainly of large open environments in which the ambient occlusion does not add much visual impact to the scene. Considering the amount of effort and time spent on implementing it the results it generated were quite lacking.

The solution we set up for creating particle effects were implemented so as to allow specific designs for each effect. All the components needed for creating new particle effects in the game are completed. However, most of the planned effects were tied to gameplay and few particle effects were actually implemented as the final product's gameplay was limited. Because of this the potential of the particle system was not fully utilized. As a result the total impact on the game's visuals from the particle system could have been in better ratio to the time spent.

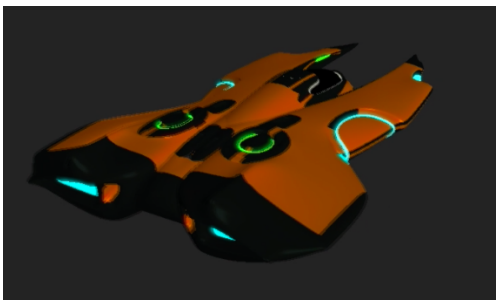
The Volumetric Lighting effect, shown in Figure 4.4.2b, could have been utilized to a greater extent if the layout of the environment had been different. Volumetric Lighting is most clearly visible when there are multiple objects obstructing parts of the light source. With the current map design this does not occur often enough to fully utilize the visual impact of the effect. If the

environment of the game had been designed with the behavior of the effect in mind, it would have produced more visual impact.

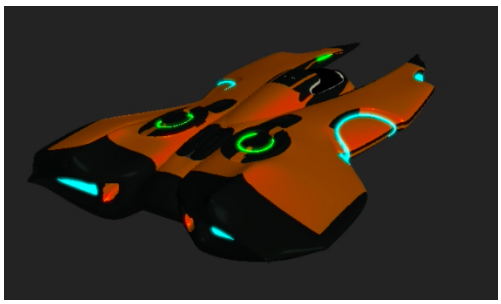
The visual impact of the game as a whole would have benefited a lot if more time could have been spent on modeling. Due to the limited timeframe and our lack of experience with creating content, we decided that trying to learn about advanced modeling would have been an inefficient use of time.

Towards the end of the project, it became apparent that the non-graphical elements of the game would be somewhat lacking in quality in comparison to the graphical effects. We felt that this was rather unfortunate since the goal of the group had always been to create a comprehensive game experience. This was something we had to accept, however, as it was stated that the graphics would be the main focus. If the focus had been on producing a complete game, rather than focusing on graphical effects, we would have prioritized game design and graphical effects equally.

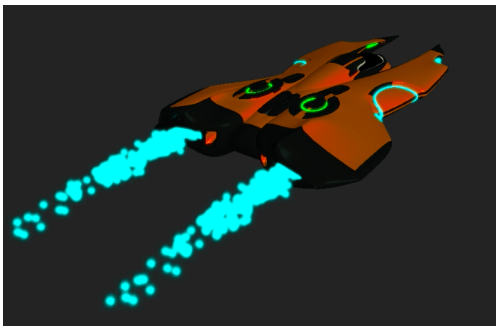
Despite the somewhat lacking content of the game, we consider the final product to be of high visual quality. Some of the visual results of the game are illustrated in Figure 5.0.2, where various effects are added to the image while observing the racer.



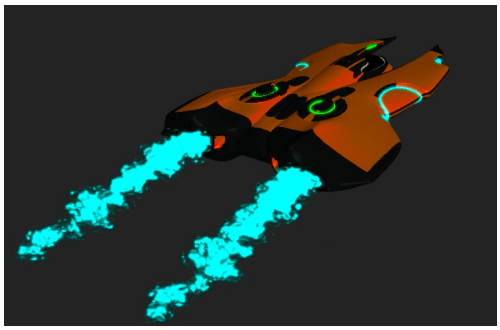
(a) Racer model after the light calculation.



(b) Bloom effect added.



(c) Plasma Particles Added.



(d) Heat Haze effect added to the particles.

Figure 5.0.2: These figures display the combination of multiple effects. Each one is used to further improve the visual impact of the game. Image (a) shows the racer after the light calculation. In (b) the Bloom post process effect is added, the result of this effect is most noticeable around the blue and green lights of the racer. In (c) Plasma Particles have been added using the particle system. In (d) the Heat Haze post process effect is used to distort the plasma particles, simulating the distortion of the air due to the high temperatures of the plasma.

Chapter 6

Conclusion

The purpose of this project was to examine and implement suitable graphical effects for a multiplayer racing game. The setting of the game, such as large environments and high speed motion, defined what was suitable. We used the game engine XNA and the functionality it offers to lessen the workload, thereby increasing the quality of the final product.

The method through which we examined and chose graphical effects roughly consisted of three stages. We first researched suitable effects given the setting of the game. Once a suitable effect had been found we proceeded with implementing it. When an effect was deemed finished, we evaluated it and decided whether the effect should be included in the final product. We also considered if additional improvements were needed.

The results were satisfactory given the goals of the group and the limited timeframe. The main focus of the project was graphics, which is also reflected in the final product. The graphics of the game vastly outshines other aspects such as gameplay, content and level design. As with all projects we had some parts that we deemed especially successful and some which were less successful. The graphical techniques that generated results which we are especially satisfied with, considering the large environment setting of the game, are Cascading shadow maps and Heightmap Terrain. The solution for the handling of post-processes, where we combine some basic effects into more complex effects, was also something that helped produce a higher quality final product. The techniques that we felt were not used to their full potential are Volumetric Lighting, Screen Space Ambient Occlusion and the Particle System. This was mainly due to a lack of qualitative environment design and gameplay. These were in turn lacking because of the limited timeframe of the project.

References

Akenine-Möller, T., Haines, E., & Hoffman, N. (2011). *Real-time rendering*. CRC Press.

Asirvatham, A., & Hoppe, H. (2005). Terrain rendering using gpu-based geometry clipmaps. *GPU gems*, *2*(2), 27–46.

Blinn, J. F. (1977, July). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, *11*(2), 192–198. Retrieved from http://doi.acm.org/10.1145/965141.563893 doi: 10.1145/965141.563893

Blinn, J. F. (1978, August). Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, *12*(3), 286–292. Retrieved from http://doi.acm.org/10.1145/965139.507101 doi: 10.1145/965139.507101

Bunnell, M., & Pellacini, F. (2004). Shadow map antialiasing. *GPU Gems: Programming Tech Tricks for Real-Time Graphics*.

Cantlay, I. (2007). High-speed, off-screen particles. *GPU Gems*, *3*, 513–528.

Crow, F. C. (1977). Shadow algorithms for computer graphics. In *Acm siggraph computer graphics* (Vol. 11, pp. 242–248).

Deering, M., Winner, S., Schediwy, B., Duffy, C., & Hunt, N. (1988). The triangle processor and normal vector shader: A vlsi system for high performance graphics. In *Proceedings of the 15th annual conference on computer graphics and interactive techniques* (pp. 21–30). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/54852.378468 doi: 10.1145/54852.378468

Dobashi, Y., Yamamoto, T., & Nishita, T. (2002). Interactive rendering of atmospheric scattering effects using graphics hardware. In *Proceedings of the acm siggraph/eurographics conference on graphics hardware* (pp. 99–107). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association. Retrieved from http://dl.acm.org/citation.cfm?id=569046.569060

Duluk Jr, J. F., Hessel, R. E., Arnold, V. T., Benkual, J., Bratt, J. P., Cuan, G., ... others (2001, May 8). *Deferred shading graphics pipeline processor*. Google Patents. (US Patent 6,229,553)

Filion, D., & McNaughton, R. (2008). Effects & techniques. In *Acm siggraph 2008 games* (pp. 133–164). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1404435.1404441 doi: 10.1145/1404435.1404441

Geldreich, R. (N.D). *Gdc 2004.* Retrieved May 19, 2014, from Rich Geldreich's website: https://sites.google.com/site/richgel99/home.

Heidmann, T. (1991). Real shadows, real time. *Iris Universe*, *18*, 28–31.

James, G. (2004). *Gpu gems: Programming techniques, tips and tricks for real-time graphics: Chapter 21, real time glow.* Pearson Higher Education.

Kemen, B. (2012). *Maximizing depth buffer range and precision.* Retrieved May 19, 2014, from the Outerra 3D engine blog: http://outerra.blogspot.com.ar/2012/11/maximizing-depth-buffer-range-and.html.

Lorach, T. (2007). Soft particles. *NVIDIA DirectX*, *10*.

Losasso, F., & Hoppe, H. (2004). Geometry clipmaps: Terrain rendering using nested regular grids. In *Acm siggraph 2004 papers* (pp. 769–776). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1186562.1015799 doi: 10.1145/1186562.1015799

Max, N. L. (1986, August). Atmospheric illumination and shadows. *SIGGRAPH Comput. Graph.*, *20*(4), 117–124. Retrieved from http://doi.acm.org/10.1145/15886.15899 doi: 10.1145/15886.15899

Microsoft. (2013). *Cascaded shadow maps.* Retrieved May 19, 2014, from the Microsoft dev center: http://msdn.microsoft.com/en-us/library/windows/desktop/ee416307(v=vs.85).aspx.

Mitchell, J. (2004). Light shafts: Rendering shadows in participating media. Presentation at Game Developers Conference 2004.

Mitchell, K. (2007). *Gpu gems 3: Chapter 12, volumetric light scattering as a post process* (First ed.). Addison-Wesley Professional.

Mittring, M. (2007). Finding next gen: Cryengine 2. In *Acm siggraph 2007 courses* (pp. 97–121). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1281500.1281671 doi: 10.1145/1281500.1281671

Méndez, J. M. (2010, May). A simple and practical approach to ssao. *GameDev.* Retrieved May 15, 2014, from http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753

Nvidia. (N.D). *Directx 11 tessellation — what it is and why it matters.* Retrieved May 19, 2014, from Nvidia: http://www.nvidia.com/object/tessellation.html.

Phong, B. T. (1975, June). Illumination for computer generated pictures. *Commun. ACM*, *18*(6), 311–317. Retrieved from http://doi.acm.org/10.1145/360825.360839 doi: 10.1145/360825.360839

Porcino, N. (2004, April). Gaming graphics: The road to revolution. *Queue - Search Engines*, *2*(2), 62. doi: 10.1145/988392.988409

Reeves, W. T. (1983). Particle systems—a technique for modeling a class of fuzzy objects. In *Proceedings of the 10th annual conference on computer graphics and interactive techniques* (pp. 359–375). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/800059.801167 doi: 10.1145/800059.801167

Reeves, W. T., Salesin, D. H., & Cook, R. L. (1987). Rendering antialiased shadows with depth maps. In *Acm siggraph computer graphics* (Vol. 21, pp. 283–291).

Saito, T., & Takahashi, T. (1990). Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th annual conference on computer graphics and interactive techniques* (pp. 197–206). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/97879.97901 doi: 10.1145/97879.97901

Shanmugam, P., & Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on gpus. In *Proceedings of the 2007 symposium on interactive 3d graphics and games* (pp. 73–80). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1230100.1230113 doi: 10.1145/1230100.1230113

Shishkovtsov, O. (2005). Deferred shading in stalker. *GPU Gems 2: Programming Technique Performance Graphics and General-Purpose Computation*.

Strugar, F. (2009). Continuous distance-dependent level of detail for rendering heightmaps. *Journal of graphics, GPU, and game tools*, *14*(4), 57–74.

Williams, L. (1978). Casting curved shadows on curved surfaces. In *Acm siggraph computer graphics* (Vol. 12, pp. 270–274).