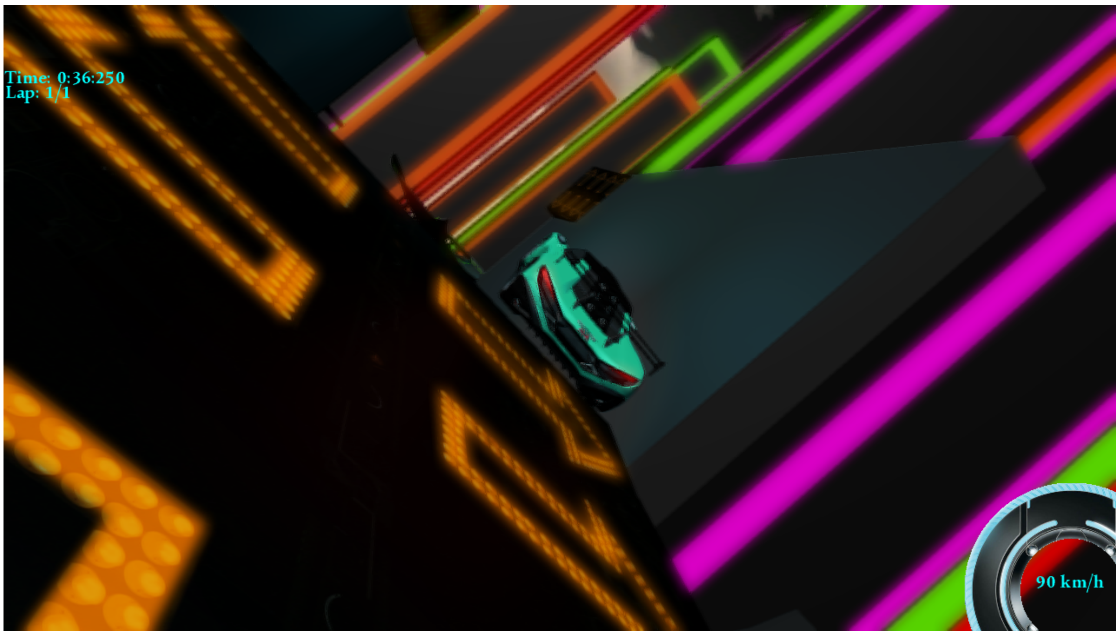


Gravitron

A study of game development and its graphical effects

Bachelor Thesis, group 80

Daniel Andersson
Patrik Ingmarsson
John Martinsson
Viktor Runemalm
Adam Scott

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

Gravitron
A study of game development and its graphical effects

Daniel Andersson,
Patrik Ingmarsson,
John Martinsson,
Viktor Runemalm,
Adam Scott

Examiner: Arne Linde
Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Cover:
Shows one player in mid-air after entering a gravitational junction in Gravitron.
Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

This bachelor thesis explores the possibilities of creating a visually pleasing game within a limited timeframe of three months, and which graphical effects that may be crucial. The game is influenced by the neon light city environments in TRON, and uses gravitational shifts to incite excitement, hence the name Gravitron.

Many of the techniques sought after during this project are included in the final result. Among these techniques, bloom, god rays, and screen space ambient occlusion are considered the most beneficial for the desired visual setting. The combination of a deferred renderer and these graphical effects makes the luminous city environment in Gravitron possible.

The project use XNA Game Studio 4.0, alongside C# with .NET, and the IDE used is Visual Studio 2010/2013. An agile approach is adopted and the group deploys an iterative development model.

Using the XNA framework gave the group some knowledge about graphics and its underlying structures. Unfortunately, this slowed down the development process. Tool-kits, like Unity and Unreal Engine, could be used, if the progression of the game is considered more important than the learning process.

Sammanfattning

Detta kandidatarbete undersöker möjligheterna att skapa ett visuellt tilltalande spel inom en tidsram om tre månader, samt vilka grafiska effekter som kan vara centrala. Spelet influeras av den neonljusfyllda stadsmiljön i TRON och använder gravitationsskiftningar för att skapa spänning, därav namnet Gravitron.

Många av de tekniker som efterfrågas i detta projektet är inkluderade i slutresultatet. Bland dessa tekniker återfinns: bloom, god rays och screen space ambient occlusion. Dessa anses även vara de mest givande för den miljön som efterlystes. Kombinationen av en deferred renderer och dessa effekter gör den lysande miljön i Gravitron möjlig.

Projektetgruppen använder XNA Game Studio 4.0, tillsammans med C# med .NET, och den IDE som används är Visual Studio 2010/2013. Gruppen använder en agil utvecklingsprocess tillsammans med en iterativ utvecklingsmodell.

Användningen av ramverket XNA gav gruppen kunskap om grafikutveckling och dess underliggande strukturer. Dessvärre bromsade detta utvecklingsprocessen. Verktygssatser, som Unity och Unreal Engine, kan användas, om utvecklingen av spelet anses vara viktigare än inlärningsprocessen.

Acknowledgements

We would like to thank Ulf Assarsson for being our mentor during this bachelor thesis, and for his great supporting role during the writing process of this report. We would also like to thank Erik Alveflo for consultation during the early development phase of the game engine, and Uno Ullvén for his help with creating illustrations and figures. We would like to show our gratitude to Magnus Gustafsson as he helped us with our report.

A special thanks goes out to the guys that kept telling us that *everything is awesome*, you know who you are. It has been very cool to be a part of a team.

Contents

1

Introduction

The focus of this thesis will be on the graphical techniques used in this project. Therefore, game logic and other underlying functionality of the game will not be thoroughly explained and discussed. However, a brief presentation of the resulting gameplay will be included in Chapter 3.

The thesis is carried out by students that have completed the computer graphics course TDA361 at Chalmers University of Technology. It is assumed that the reader has the same, or equivalent, amount of knowledge in the field of computer graphics.

1.1 Background

The video game industry today is already huge, however, it is constantly expanding and growing rapidly. Games are everywhere and they are an important part of our daily life. One prominent game market that has grown rapidly the last years is the indie scene. Indie games are developed independently from any publisher, typically by small teams testing new ideas and concepts.

To render a realistic picture, computer graphics simplifies and approximates reality. The techniques that depict reality best requires a lot of computing power, which makes them slow, and are therefore not suited for real-time applications. Because of this, the video game industry uses even rougher techniques to achieve fast rendering at the expense of adherence to reality.

The computational power of personal computers has improved tremendously in the last couple of years, which has allowed a possibility of higher graphical authenticity. The constant advances in the research area of graphics continuously raise the bar of what is considered as good-looking games. The competitive environment in the game industry drives the research in graphics forward. These two factors push each other in a self-catalyzing way.

Since the computing power of the graphical processing units is evolving, the need to optimize algorithms for smaller games becomes dispensable. Time can then be directed towards increasing the amount of implemented visual effects as an alternative to micro-managing every algorithm.

1.2 Purpose

The purpose of this project is to explore the possibility, with a team of five, to create a game within the timeframe of three months, which is graphically pleasing and exciting. The project will focus on achieving a visually exciting gameplay through a multitude of graphical effects. By implementation and analysis of different graphical effects, the group members intend to learn more about the underlying graphical techniques used to render 3D graphics.

The goal is to take artistic influences from TRON: Uprising [1], which takes place in a dark city environment with glowing light sources outlining most objects in its world, and explore which graphical effects are suited for this visual setting. The group also wants to achieve a unique gameplay element by introducing gravitational shifts. Therefore, the name Gravitron will be used throughout the report when referencing to the game.

1.3 Problem Statement

Questions connected to the purpose of this project include:

- How far can a group of five reach with a time limit of three months in order to create a TRON inspired racing game?
- Which graphical effects are suitable when creating a racing game set in a dark city environment, with many glowing lights?

Suitability is an ambiguous term, and is in this project considered with regards to visual appearance, efficiency and ease of implementation. In order to specify the latter question of the above, the following sub-questions have been constructed:

- Which graphical effects are required to achieve the glowing lights?
- Which graphical effects are suited for use in a dark environment?
- Which graphical effects contribute most towards the visual goal set in the project?
- Which graphical effects can be used to enhance the sense of speed in a racing game?

1.4 Limitations

This project focuses on graphics and will use already existing and implemented effects wherever possible. Therefore, the group will resort to existing frameworks when possible, i.e., any relevant libraries, features or models that can be acquired will be used to save time.

The project has a timeframe of roughly three months, which is a lot less than the regular timeframe for game development. The largest asset for the group is the estimated 400 hours of work per individual, but it is also the largest limitation of the project. Therefore, it is vital that this resource is managed with care.

A common problem in computer graphics is optimization; not all users have a high-end graphics card, which makes it important that algorithms work for average cards, and therefore, avoids any kind of redundancy. This project, however, will not focus on optimizations, and, thus, target high-end systems with a lot of computing power.

Since the group wants a graphically aligned game, a racing game is chosen. A racing game does not require many, if any, animations which can be time consuming to realize. Instead the implementation and study of graphical effects is prioritized.

1.5 Method

One of the first important decisions a group has to make is how the group will be composed. Two different constellations were considered, one with a fully democratic group, and one with a group leader. This group has chosen a project leader that can keep the project under supervision, as well as solve minor conflicts. Different responsibilities are distributed between group members. Being responsible does not mean doing the task alone, but rather making sure the task gets done.

The group adopts an agile approach and uses iteration cycles during the product development (see Figure 1.1). At the end of each iteration the group strives to have a working implementation of the product with increasing quality and complexion. This method reduces early planning overhead, and enables planning in an iterative manner at the end of each cycle. Taking on a whole project directly would be extremely difficult, which leads to dividing the project into smaller parts. The subprojects are graphics, game logics, and models. The process of modeling is quite time consuming, and pre made models are used when possible. The models are modified to suit the needs of the project.

The project uses XNA Game Studio 4.0, alongside C# with .NET, and the IDE used is Visual Studio 2010/2013. JigLibX is chosen as the physics engine of the game. Functionalities and effects are added as the project progresses, and therefore, a component based design is chosen.

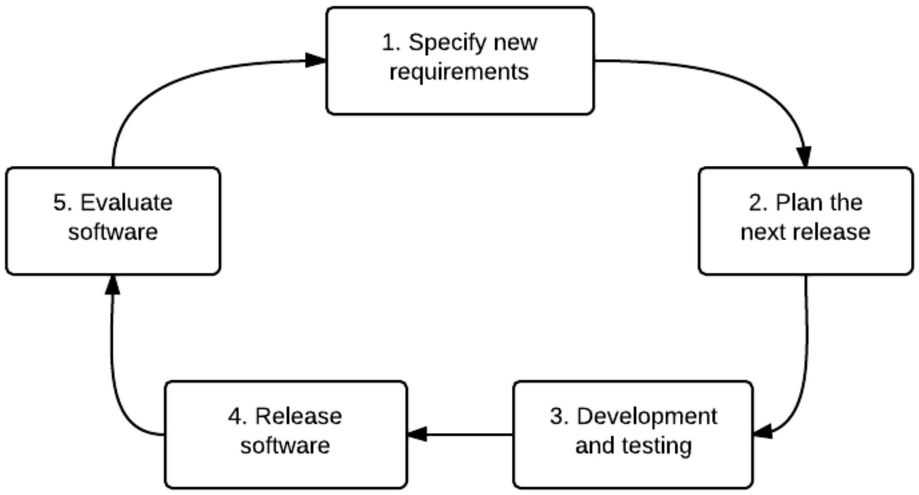


Figure 1.1: An illustration of the iteration cycles used during the development of Gravitron.

A git repository is used, which allows the subprojects to be developed independently. The group is split over the subprojects in teams of one to three persons, depending on the work needed to reach the goals of the current iteration. To make everybody equally incorporated in the whole project, the members rotate between the different subprojects.

2

Graphics

This is the scientific main theory part of the report. Each section will start with an explanation of the techniques that have been considered in the respective field, and then, proceed with a description of how these techniques are used in Gravitron, followed by results and a brief discussion.

2.1 Deferred Rendering

The standard rendering technique for most engines is forward rendering. Each object is rendered once for each light to the back buffer. Lighting may also be calculated for points not visible on the screen. In other words, if there are many dynamic light sources, classic forward rendering should not be used [2].

An alternative technique is deferred rendering. As the name implies, lighting will be applied after the geometry is drawn, similar to a post-process (see Section 2.2). This way each object and light will only be drawn once, and since the lighting is drawn in screen space, unnecessary light calculations for points that do not show up on screen are avoided.

Since Gravitron is TRON-inspired, and uses dark environments with lots of lights creating contrasts, a deferred renderer is preferable.

2.1.1 Previous Work

Deferred rendering was introduced by Michel Deering et al. at SIGGRAPH 1988, although they did not use the term deferred [3]. The modern form of deferred shading that is used today was introduced by Saito and Takahashi in 1990, also without using the term deferred [4]. The first game that used deferred shading was Shrek for Xbox, shipped in 2001 [5].

The technique consists of mainly three steps (see Figure 2.1). The first step is to draw the objects to a geometry buffer (G-buffer). The G-buffer must, as a minimum, consist of the diffuse color, surface normal, and world coordinates for each pixel on the screen [6]. Further possible information may be emissive color, specular power, and specular intensity.

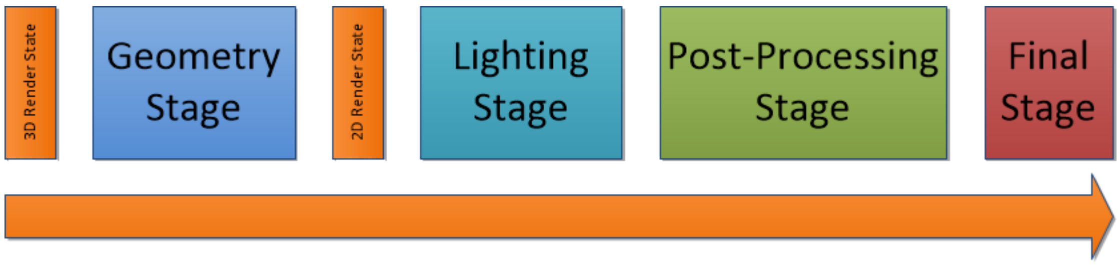


Figure 2.1: The stages of the deferred rendering pipeline. The geometry stage is the only stage working with the 3D scene and all the subsequent stages operate with the information derived from the 2D G-buffer.

In the second step, the contribution of the light sources is accumulated and blended additively. The lights cannot have infinite reach so they are rendered as volumes: spot lights as cones, point lights as spheres, and directional lights as quads or boxes. To calculate lighting, the surface normal, position, and color of the pixel is read from the G-buffer.

After the lighting stage, it is time to process the image before it is shown on the screen. Since deferred rendering cannot use hardware anti-aliasing, a preferable post-processing effect may be anti-aliasing (see Section 2.2.2). The last step is to write the image to the back buffer.

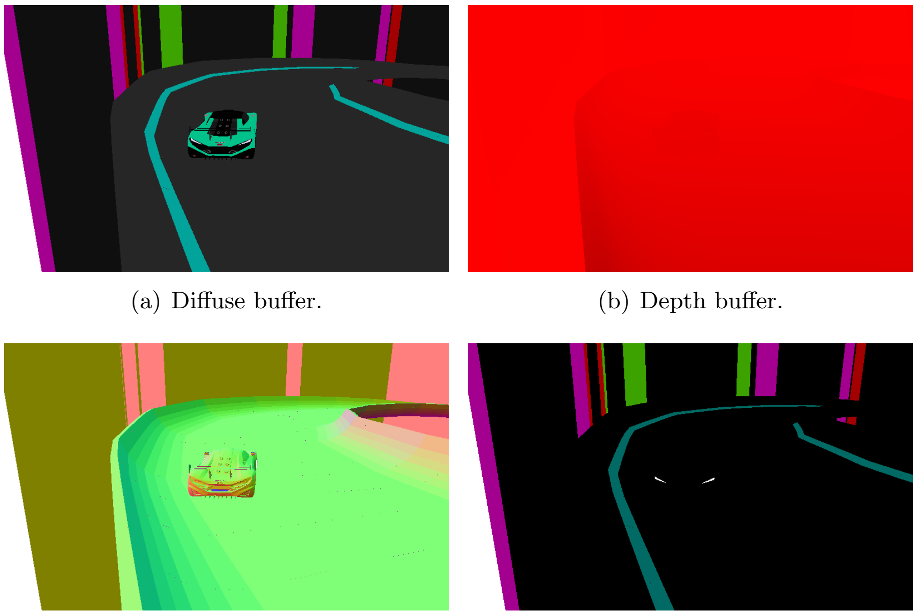
To reduce some overhead, in terms of texture accesses when computing the lighting, in conventional deferred shading, deferred lighting can be applied [7]. When using deferred lighting, the diffuse and specular components are separated and stored in different light maps. These light maps are combined with the color buffer in the G-buffer in a third pass.

As further optimization, the depth can be stored instead of the world coordinates, which can be recreated when they are needed. It is also possible to only store the x- and y-coordinates, as the z-coordinate can also be recreated [8].

2.1.2 Results

The first stage of the deferred renderer is to create the G-buffer. The G-buffer of Gravitron consists of four render targets (see Figure 2.2): the diffuse buffer with the color of the scene, the depth buffer for reconstruction of world coordinates, the normal buffer where the surface normal for each pixel of the screen is stored, and an emissive buffer which contains the emissive properties of the scene.

The diffuse, normal, and emissive buffers use a 32-bit ARGB pixel format, using 8 bits per channel. However, a 32-bit float format, with 32 bits for the red channel, is used for the depth buffer. In the diffuse buffer, the RGB-channels are the RGB-values of the diffuse color of the pixel, and the alpha channel is used for specular intensity. Similarly, the normal buffer contains the 3D-normal and specular power of the pixel.

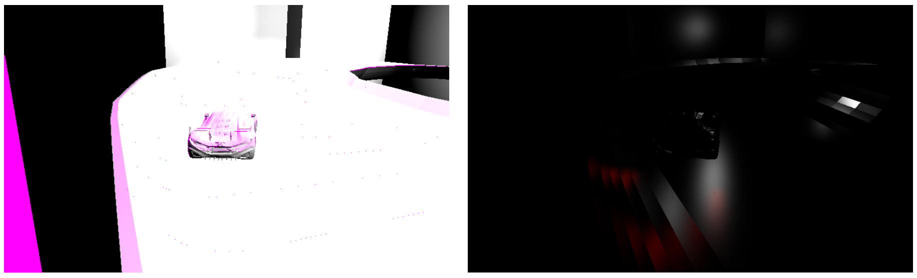


(a) Diffuse buffer. (b) Depth buffer.

(c) Normal buffer. (d) Emissive buffer.

Figure 2.2: The G-buffer used in Gravitron. (a) The diffuse buffer contains the colors of the scene. (b) In the depth buffer, the depth of the scene is stored. (c) The normal buffer consists of the world-space normals, and (d) the emissive buffer contains the emissive materials of the scene.

In the lighting stage, Gravitron makes use of deferred lighting. The lighting is calculated according to the Blinn-Phong model and stored in light maps (see Figure 2.3), one for diffuse lighting (see Figure 2.3(a)), and one for specular lighting (see Figure 2.3(b)).



(a) Diffuse Light Map. (b) Specular Light Map.

Figure 2.3: The light maps created using the information stored in the G-buffer (see Figure 2.2), the depth for recreating the world-space position of the pixels and, the normal for lighting computations. These maps only contain the lighting of the scene.

Lastly, the light maps and the diffuse buffer are combined (see Figure 2.4). The value of the diffuse light map is multiplied with the value of the diffuse buffer in the G-buffer, and then the value of the specular light map is added.

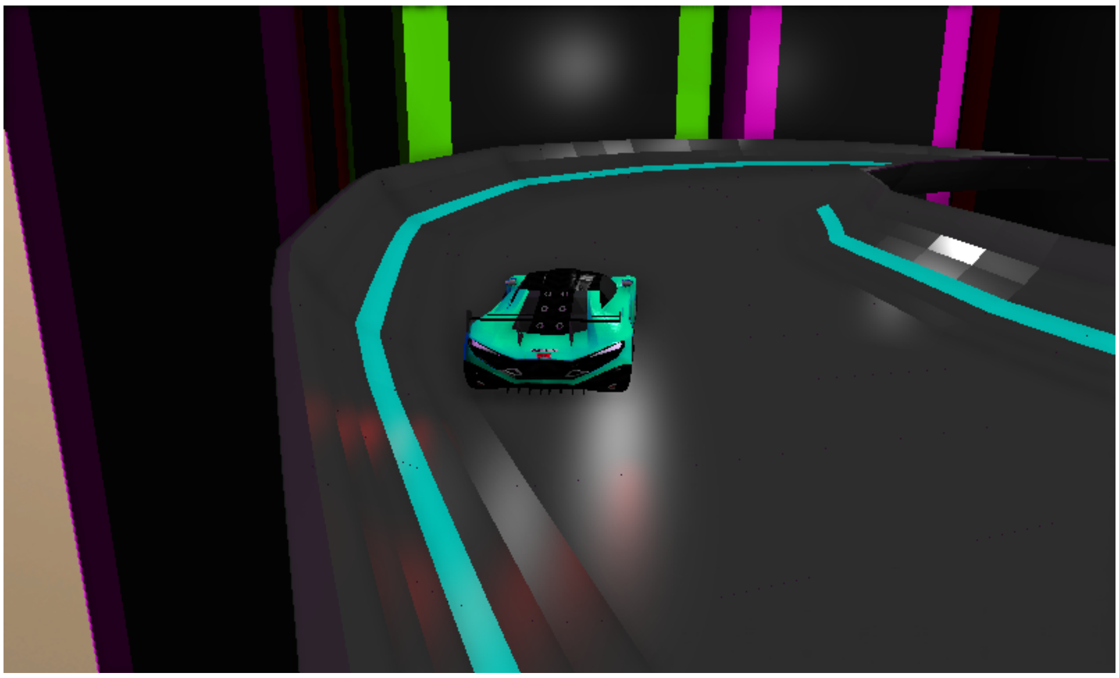


Figure 2.4: The final image after combining the light maps (see Figure 2.3), with the color of the scene contained in the diffuse buffer (see Figure 2.2(a)).

In order to simplify the implementation of the bloom post-process (see Section 2.2.5), the emissive color is rendered to a separate buffer, the emissive buffer. To do this, the specular component is simplified to not contain material specific color, but only a specular intensity and specular power.

2.1.3 Discussion

Deferred rendering is important to our game since it creates the opportunity to use a vast amount of light sources. These light sources can contribute to achieving a TRON-like environment, because of the contrast they create when put in an otherwise dark scene. However, it may have been possible to solve the problem of many light sources when using forward rendering too. One solution could be to cull away light sources at a certain distance to reduce them to a manageable amount. This would probably also allow a high count of light sources, but the users may experience the lights turn on and off, which most likely would annoy them. To resolve this, we could have used heavy fog to conceal the effect of unlit light sources, but that could have created an undesirable atmosphere in the scene.

One disadvantage of using deferred rendering in the development of Gravitron is the time it took to implement, which is one of our most valued resources. However, when using forward rendering, part of the G-buffer would need to be rendered anyway for the post-processing stage, which is half the work of implementing deferred rendering. Therefore, we believe that deferred rendering is preferable in Gravitron.

2.2 Post-Processing

Post-processing is the technique of applying a set of desired effects to a scene after it has been rendered into a texture. It allows for artistic alteration and modification of the image, which may enhance the perceived quality, or create a certain setting.

Each subsection in this section starts with a very brief explanation of why the effect is important. It then proceeds with an overview of the available algorithms and techniques, and ultimately ends with motivations of the projects implementation choices.

2.2.1 Edge Detection

In this project, edge detection is used to allow the ability of anti-aliasing (AA). The standard AA solution, Multisample Anti-Aliasing (MSAA), is not suitable since a deferred renderer is used [9]. The reader is referred to Section 2.2.2 for an explanation of what anti-aliasing is and how it is performed.

Edge detection is a method used to find discontinuities in the discrete space of a digital image. These discontinuities usually occur at edges. They are detected by approximating the intensity gradient magnitude (g) of a pixel, and check if it is large enough to indicate an edge. The approximation of g is achieved by analyzing variance in color, depth, normals, or light intensity in the image.

Previous Work

There are two main approaches used for edge detection: the first one being the template matching (TM) approach, and the second one being the differential gradient (DG) approach [10]. The difference between the TM and DG methods is mainly how they approximate g.

Both TM and DG use convolution masks to approximate the local intensity gradients of an image [10]. The DG approach is more accurate, but is computationally expensive. However, it only requires two convolution masks to be used, while TM usually employs eight to twelve masks. For use in Gravitron, accuracy was chosen over performance.

Basically, the masks are used as templates for a pixel neighborhood. The observant reader will notice that the sum of all coefficients in a mask is zero (see Figure 2.5). This means that when the mask is applied over a homogeneous pixel neighborhood the output will be zero, resulting in no edges being detected. However, if the neighborhood is heterogeneous the output pixel will not be zero and an edge could be present. Whether or not the result is considered an edge depends on the magnitude of g and what thresholds are set in the edge detector.

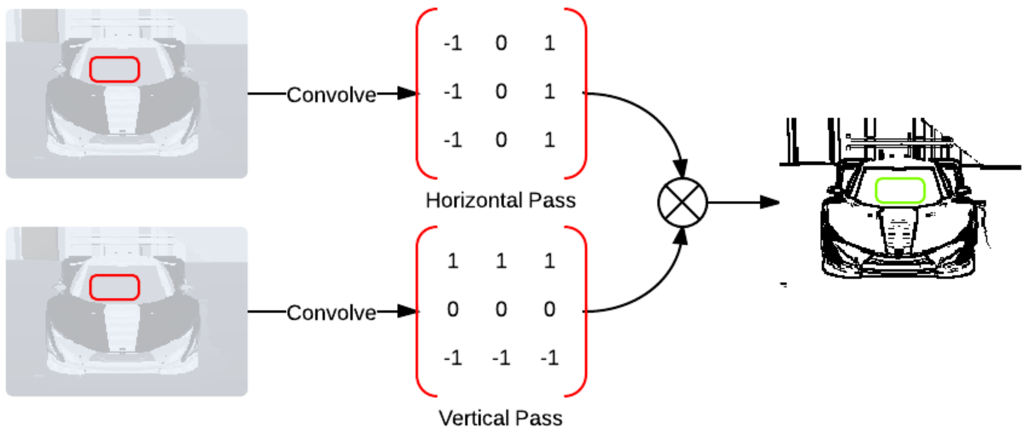


Figure 2.5: An illustration of a two-dimensional convolution between an input image (left) and the Prewitt masks (middle). The resulting edge detected image (right). The red rectangles (left) is an example of where the masks could be placed during a convolution and the green rectangle (right) is where the output pixel would be placed.

In this thesis, four of the most frequently used DG convolution masks have been considered. They are: the Roberts mask, the Sobel mask, the Prewitt mask and the Laplacian [11, 12]. These masks can operate on the least amount of information available in an image.

$$G_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} G_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

(a) The Roberts Masks.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

(b) The Sobel Masks.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

(c) The Prewitt Masks.

$$G_x = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} G_y = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

(d) The Laplacian Masks.

Figure 2.6: Four convolution masks frequently used for edge detection.

The most appealing aspect of the Roberts masks is their simplicity, that is, they are quite small (see Figure 2.6(a)). They perform a simple 2D spatial gradient measurement for each pixel in a given image [11]. However, they are very susceptible to noise [13], and was thus discarded for use in Gravitron.

The Sobel, Prewitt and Laplacian methods work in much the same way. They approximate a 2D spatial image gradient using 3x3 masks (see Figure 2.6). According to Acharjya et al [12], edges detected by the Laplacian are irregular and thick, whereas edges detected by Prewitt and Sobel give much clearer results. They concluded that the Sobel masks, on average, performs better than the Prewitt and Laplacian.

The initial edge detection shader of Gravitron took heavy influence from a shader proposed by Agnius Vasiliauskas [14]. The appealing aspects of this shader were its ease of implementation due to very accessible code and its low computational cost. It was therefore implemented.

Since the only thing that differs in terms of implementation for the Sobel, Prewitt and Laplacian edge detection techniques are the masks used, the choice was made to implement a general edge detection shader that could handle generic 3x3 masks. The implementation was as easy as adding a convolution method to the initial shader and then use this instead of the approximation technique used before.

Results

The results, produced by the four edge detection techniques, proved to vary in quality (see Figure 2.7). It is clear that the initial technique, as proposed by Vasiliauskas, is not performing as well as the other three.

Taking a closer look at the Sobel, Prewitt, and Laplacian, one can see that the Sobel seems to over detect the edges a little (look at the thick edges around the car and also in the background of the car), while the Laplacian, on the other hand, seems to be under detecting the edges, if only a little. The best performing method for our purpose, which is to allow anti-aliasing, is the Prewitt masks. The goal is clear edges, which can be used to calculate weights in the anti-aliasing pass.

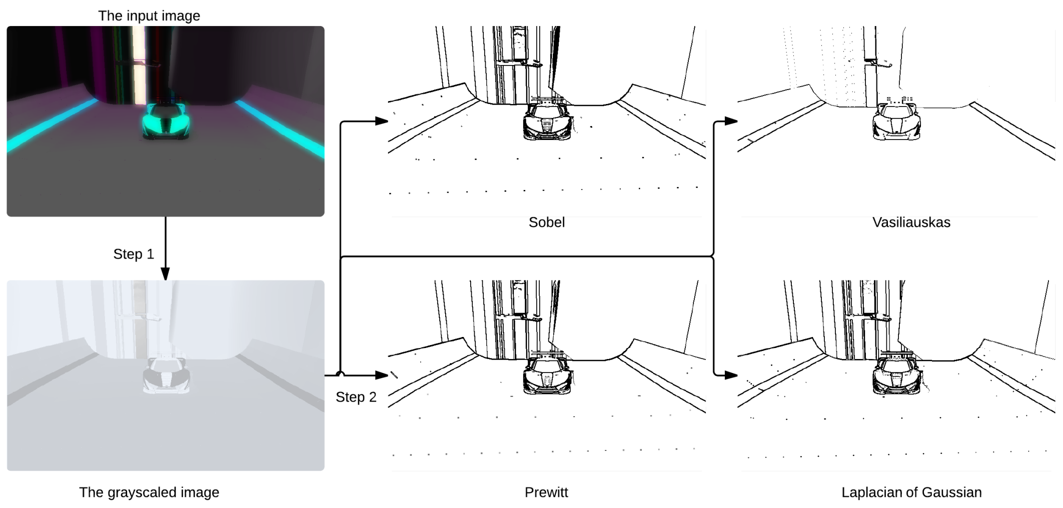
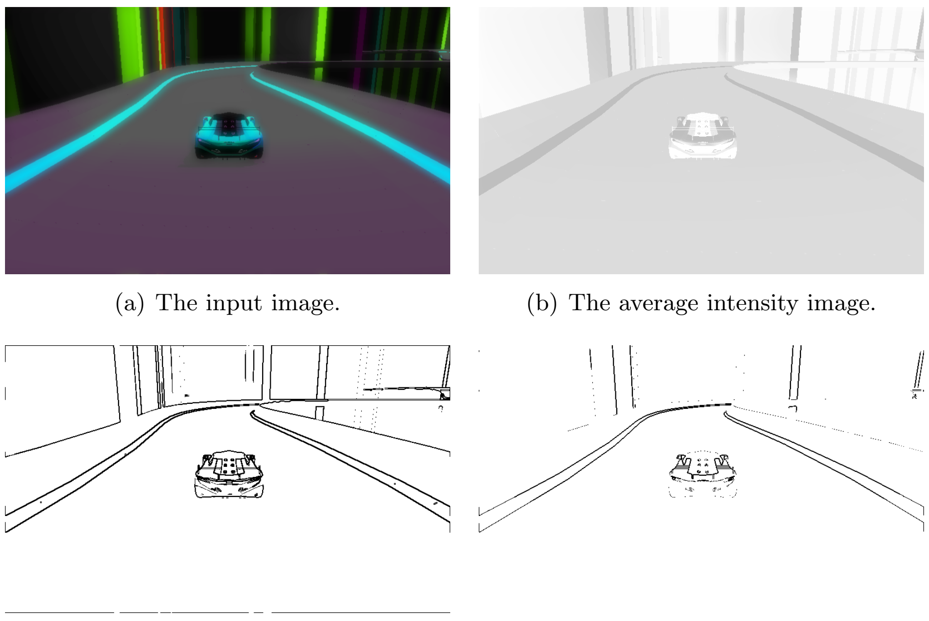


Figure 2.7: The edge detection results using Sobel, Prewitt, the Laplacian and the simpler shader written by Vasiliauskas. This is done in two main steps: in step 1, take the input image and make a gray scale version of it, in step 2, perform the convolution between the gray scale image and the masks, square the results, add them together, and then take the square root of that sum which results in the final images seen at the end of their respective arrows in this flowchart.

Using different threshold settings had a huge effect on the edge detection results (see Figure 2.8). The thresholds are used to decide when a pixel should be marked as an edge (black) in the output image. It can be observed that by increasing the span between the lower threshold and the upper threshold, the amount of detected edges either increases (see Figure 2.8(c)) or decreases (see Figure 2.8(d)). A span that is too large will result in over detection of edges, and a span that is too small will result in under detection; which thresholds that give the best edge detection quality are not known beforehand.

An empirical analysis with different thresholds had to be done in order to see which settings that gave the best results. Threshold setting 1 proved to give the best quality (see Figure 2.8).


(a) The input image. (b) The average intensity image.

(c) Threshold setting 1. (d) Threshold setting 2.

Figure 2.8: The results of different threshold settings. (a) The input image. (b) The average intensity image. (c) The result of a lower threshold set to 0.1 and an upper threshold set to 0.7. (d) The result of a lower threshold set to 0.25 and an upper threshold set to 0.4.

Discussion

The edge detector by itself does not really contribute to our visual goals, but it is used in our anti-aliasing technique (see Section 2.2.2). One property that would have improved our final implementation of the edge detector turned out to be the ability to detect edges other than horizontal and vertical ones. We could have implemented a template matching edge detector, as they use eight to twelve convolution masks, which could have been used to further our capabilities of differentiating between edges. However, the edges detected using our approach should be more accurate than those detected with a template matching edge detector.

2.2.2 Anti-Aliasing

A frequently occurring problem when rendering a scene in computer graphics is undersampling. The scene must be sampled in order to get its discrete version to be displayed on the screen. This often results in jagged contours of objects and is more formally known as aliasing [15]. The method used to reduce aliasing is called anti-aliasing.

Previous Work

For more than a decade, the standard solution to anti-aliasing has been Multisample Anti-Aliasing (MSAA) and Supersample Anti-Aliasing (SSAA) [16]. However, a deferred renderer can not directly take advantage of the multisampled framebuffers used by MSAA, which makes MSAA unsuitable for this project [9]. Supersampling is not suited for real-time rendering because of its slow computational speed. Therefore, anti-aliasing as a post-processing step must be considered.

Research of anti-aliasing as a post-processing technique has recently become popular in both academia and the industry [9]. The recent Morphological Anti-Aliasing (MLAA) method, proposed by Reshetov [17], inspired a burst of new real-time anti-aliasing techniques that rival even MSAA in terms of quality and performance [9, 16].

The sought after attributes for anti-aliasing in this thesis are: the ability to perform in real-time and preferably quality on par with MSAA. The MLAA technique previously mentioned is designed to run on the CPU, and therefore, not suitable for real-time graphics rendering [17]. However, many of the techniques that evolved from MLAA are designed to run in real-time, and are thus, viable options in this project.

A quite modern MLAA based technique considered is Enhanced Subpixel Morphological Anti-Aliasing (SMAA), and builds upon Jimenez's MLAA. Jimenez's MLAA in turn is an evolution of the original MLAA technique proposed by Reshetov [17]. SMAA is designed to run in real-time as a post-processing step and is believed to rival the anti-aliasing quality of MSAA [16]. Source code and good documentation for SMAA is also available, which is great for fast adaptation of the method.

Another technique considered was Directionally Localized Anti-Aliasing (DLAA), which is a simplification of MLAA, basically removing the weight calculation step to make it perform in real-time [9]. The simplicity of this method is intriguing and, thus, a viable option for use in Gravitron. However, the method does produce blurrier results than MSAA [16], which is not desired.

Other methods considered were Hybrid MLAA and Fast Approximate Anti-Aliasing (FXAA). Hybrid MLAA turned out to be for use on the Xbox360, which is not the target platform for this project, while FXAA, a cutting edge method being developed by Nvidia, is yet to be released [9]. Because of this, neither of them is applicable in this thesis.

Neither FXAA nor Hybrid MLAA was viable candidates, and will therefore not be implemented. DLAA is a good option with the worst drawback being that it does not produce high quality results. Finally, SMAA is the only technique of those considered that performs in real-time and is believed to produce results of same quality as MSAA, therefore it will be implemented.

Since all the considered techniques are some form of evolution of the MLAA method, an explanation, albeit brief, is in order. The MLAA technique consist of three main steps, in step 1, detection of edges, in step 2, calculation of weights, in step 3, blending of neighboring colors [17].

Edge detection is performed in order to find the edges at which anti-aliasing is necessary, which is accomplished by finding discontinuities between pixels in the image. The reader is referred to Section 2.2.1 for a more thorough explanation of edge detection. The weights are then calculated by identifying predefined shapes around these edges [17]. Finally, the weights are used to blend neighboring pixels in an intelligent way (see Figure 2.9).

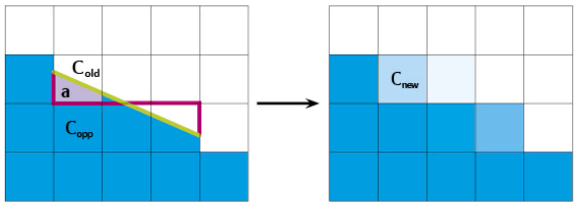


Figure 2.9: The image shows how the pixel C_{opp} contributes to the pixel C_{old} using the weight a, as calculated by MLAA. First, edges are detected (magenta lines). Predefined shapes are then recognized using the detected edges. The shapes are used to try and reconstruct the actual edge (the green line), which in turn allows calculation of the coverage area for the considered pixels. The complete formula in this case is: $c_{new} := (1-a)*c_{old}+a*c_{opp}$, where c_{new} is the anti-aliased version of c_{old}.

The SMAA technique is quite complex and, therefore, a detailed explanation is considered out of scope for this thesis. It is, as previously stated, a method based on Jimenez's MLAA, which in turn is an evolution of MLAA.

Results

The adaptation of SMAA was harder than expected, even with the great source of premade code and documentation. After an effort to port the SMAA technique into the post-processing pipeline used in Gravitron, no positive anti-aliasing results were produced. However, anti-aliasing is not the most visually contributing effect in a graphical application and, thus, not a priority for this project. Instead of putting numerous of hours into debugging, the choice was made to fall back on a simple DLAA inspired algorithm.

The DLAA inspired method used in Gravitron utilizes an edge detector that detects horizontal and vertical edges using the Prewitt convolution masks, and marks them with a unique color (see Figure 2.10). The anti-aliasing is achieved by sampling the detected edges, and then perform a vertical or horizontal blend depending on the value sampled (see Figure 2.10). This method does not produce results of high quality, but edges in the final scene does look less jagged (see Figure 2.11). The horizontal blend is performed by simply sampling the three neighboring pixels to the left and right of the pixel considered for anti-aliasing in the scene. By averaging the sampled pixels, the final anti-aliased pixel color can be achieved. The same approach is done for a vertical blend, with the difference being that the pixels sampled are the three above and beneath the pixel.

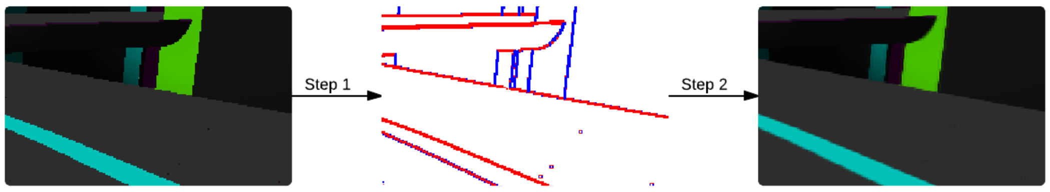
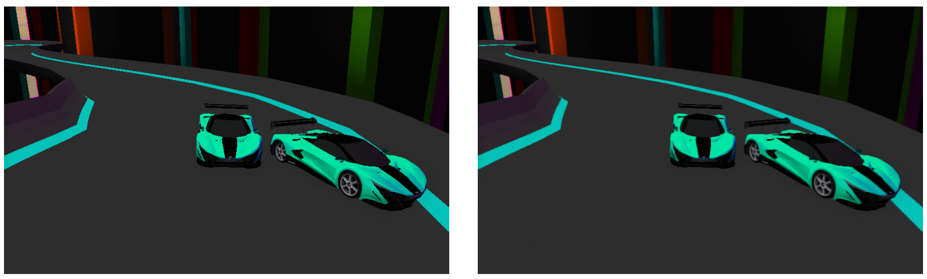


Figure 2.10: This figure illustrates the DLAA inspired anti-aliasing method used in this thesis. It consists of two main steps: in step 1, vertical (blue) and horizontal (red) edges are detected using Prewitt convolution masks, in step 2, a vertical or horizontal blend is performed depending on the detected edge. The scene before anti-aliasing is shown to the left and the resulting anti-aliased scene is shown to the right.



(a) The scene without anti-aliasing. (b) The scene with anti-aliasing.

Figure 2.11: Examples of anti-aliasing from Gravitron: without anti-aliasing (left) and with anti-aliasing (right). Look at the edges of the car and the track. They are not as jagged in (b) as they are in (a).

The edges in the anti-aliased scene (see Figure 2.11(b)) is not as jagged as those in the original scene (see Figure 2.11(a)). However, a problem observed is that the edge detector marks some edges as both horizontal (red) and vertical (blue) (see bottom left edge in Figure 2.10), which can result in edges that look dithered instead of smooth. Nevertheless, the overall impression of the scene is that edges look smoother, which is the main purpose of the anti-aliasing technique.

Discussion

Our implementation of anti-aliasing ended up being quite basic, but its contribution to the final product is visible and edges are perceived as less jagged. If we had managed to adapt a more sophisticated anti-aliasing method, such as SMAA, we probably would have been able to reduce the aliasing even further.

One thing we could have improved upon is the number of directions in which edges can be detected in our edge detection algorithm. Our current edge detector only marks horizontal and vertical edges. It would have been preferable to also be able to detect diagonal edges, giving our anti-aliasing algorithm more blending freedom, which in turn, most likely would have produced more convincing results. That being said, the implementation of our final anti-aliasing method ended up taking time that might have been better spent on other, more visually prominent, features.

2.2.3 Gaussian Blur

Image smoothing is not a vital part of the game by itself, but it is an essential part of achieving bloom (see Section 2.2.5). This can be done in several ways, and one effective way of blurring an image is Gaussian blur, also known as Gaussian filtering or Gaussian smoothing [18]. Gaussian blur is mostly used to reduce noise and remove details [19]. Generally in games, Gaussian blur (see Figure 2.12) is used to create heat haze, bloom, or depth-of-field [20].

Previous Work

Major image blurring filters are mean, median, Gaussian and bilateral filtering. Bilateral filtering preserves edges and is therefore not suitable for use in this project, since the bloom effect requires blurred edges [21]. The median filter takes the median of the values around the center pixel and replaces the center pixel with that value. The median technique, however, is expensive to compute compared to mean and Gaussian filtering [22].

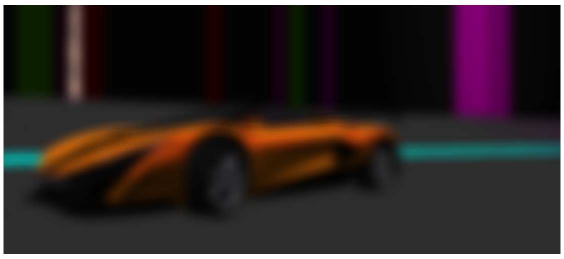


Figure 2.12: An illustration of the Gaussian blur effect.

Gaussian filtering and mean filtering are similar, since both methods average pixels in order to distort the image [23]. The Gaussian filter uses larger weights for the central pixels when compared to the mean filter, which has uniform weighting. This means that the weights of the Gaussian function is similar to a normal distribution curve (see Figure 2.13) [19]. Both effects are generated by a convolution between an image and a convolution mask (see Section 2.2.1 for more information about convolution masks).

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-x^2}{2\sigma^2}} \tag{2.1}$$

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{2.2}$$

In Gaussian blur, the values of the convolution mask are calculated via Equations 2.1 and 2.2 (see above) [18]. In equation 2.1 and 2.2 the parameters x, and y is the distances from the origin along their respective axis, and σ is the standard deviation for the Gaussian blur function. For efficiency, it is profitable to utilize the separable property of Gaussian blur, and divide the process into two separate passes [19], one horizontal and one vertical, where the order in which they are performed is interchangeable. The end result is the same as using the two-dimensional convolution masks to convolve with, but the two-dimensional operation requires more calculations.

Mean-filtering does not have this attribute, which makes it less appealing. The technique considered for implementation in Gravitron is therefore Gaussian blur, due to its inexpensive computational cost.

The discretization of Gaussian blur is achieved by sampling the function at discrete points, often at corresponding positions to the midpoints of each pixel. Point sampling the function reduces the computational cost, but can give large errors for small filter convolution masks [18]. For smaller convolution masks, accuracy can be maintained via integration of the Gaussian function over the area of each pixel [19].

After the discretization of the continuous Gaussian values into the discrete values of the convolution mask, the sum of the convolution mask will differ from one [19]. This will result in the image becoming darker or brighter, but the distortion can be avoided by normalization, that is, dividing each term by the sum of the terms of the convolution mask.

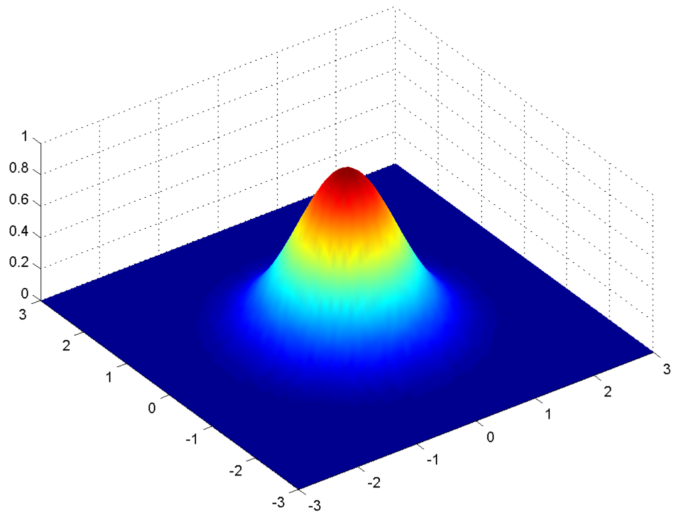


Figure 2.13: An example of a Gaussian function with its center around the origin. The values of the convolution mask are represented by the height and color, and show that the coordinates closer to the origin are of greater value.

Gravitron does not include any effects solely using Gaussian blur. However, the results of effects including Gaussian blur are presented in Section 2.2.5 and 2.2.6.

2.2.4 Motion Blur

One essential part when developing a racing game, making it distinct from other game genres, is achieving a sense of speed. A great way to accomplish this is by using motion blur. Motion blur is a technique used to simulate the naturally occurring phenomenon of objects being perceived as blurry at high speeds.

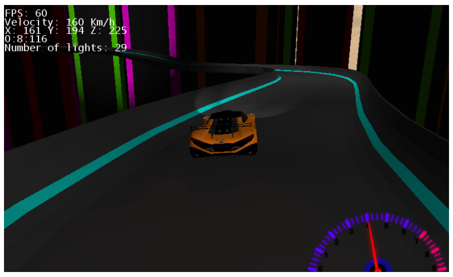
Previous Work

There are two major and popular motion blur techniques: image space motion blur [24] and post-process motion blur [25]. When using image space motion blur, the result tends to look realistic and the implementation is simple in comparison to the post-process alternative. However, this process can be expensive when rendering in real-time. Rendering motion blur as a post-processing effect is the more efficient option, and therefore, used in this thesis.

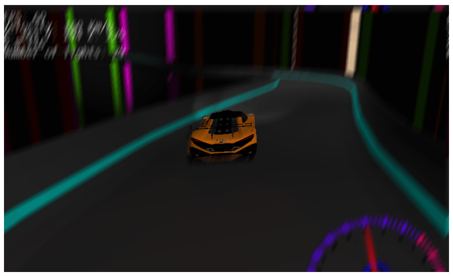
The post-processing technique that was chosen for this thesis was introduced in 2007 by Rosado [25]. The technique was selected partly because the article, in which it was displayed, provided great example code. The pixel shader uses the value stored in the depth buffer to retrieve the 3D-position for each pixel on screen. This 3D-position is transformed with the previous and current view-projection matrix, in order to obtain a measure of how each pixel has moved between the last two frames. The measure corresponds to the pixels velocity, and will determine the direction in which that pixel will be blurred (see Figure 2.14).

Results

When comparing the results without motion blur (see Figure 2.14(a)), with the results with motion blur (see Figure 2.14(b)), it is clear that the motion blur effect contributes to achieving a sense of speed. Since the car always will be moving at a constant speed, relative to the camera, the surface of the car should never be blurred. A mask can be used to make sure that no blur is performed in that area of the screen (see Figure 2.15).



(a) The scene without motion blur.



(b) The scene with motion blur.

Figure 2.14: (a) The figure shows the scene without motion blur, (b) and how it changes with motion blur.

Figure 2.15: Here, the texture used to mask the scene is shown. The black area informs the motion blur shader where no blurring is necessary.

Discussion

Motion blur aids in achieving a sense of high speed in our game. We consider the effect to be worth the time we put into the implementation. Motion blur might not make the game achieve the desired TRON-look, but we do believe it does help make it a better racing game, by increasing the sense of speed. Proper motion blur reflects how we experience high speed movement in reality, and by not incorporating motion blur, the movement of the car may be harder to percieve.

The method we chose is not the most accurate, but we regard it as the best option for our game. However, when using the motion blur in Gravitron constantly, the scene became over blurred and somewhat distorted. Instead, we believe that it is best used in combination with speed boosts.

2.2.5 Bloom

As previously stated, the intention of the project is to create a game with influences from TRON, which involves dark environments with many light sources. The standard lighting model is not capable of capturing the phenomenon that the human eye produces when looking directly into an intense light source, known as glare. In order to solve this, several techniques exists, which are able to approximate this phenomenon and allow glare to be reproduced in a real-time renderer.

Previous Work

Glare produces scattered light around bright regions, and can be reproduced with two techniques, flare and bloom [26]. Flare can be described as the spikes and the halo surrounding the light, while bloom refers to the loss of contrast in the proximity of the light. The latter suits this project, since a vast amount of emissive materials are used in Gravitron, and bloom is great for the highlighting of them.

Bloom for small, point-like objects can be produced by attaching a billboard with a ”glowy” texture [27]. However, bloom for larger objects with more complex shapes will most likely become unnecessarily complicated if this method is used. The fact that the project do not make use of any point-like objects, which requires bloom, is why billboards will not be used to achieve this effect in Gravitron.

The method used to create bloom in Gravitron, is a post-process of the scene [27], which suit our implementation of deferred rendering well. The post-process can be achieved in three main steps (see Figure 2.16): in step 1, the emissive components of the original image are rendered into a separate texture; in step 2, the texture is blurred, preferably with help of Gaussian blur; in step 3, the blurred emissive texture is combined with the original scene. The reader is referred to Section 2.2.3 for a more detailed explanation of Gaussian blur.

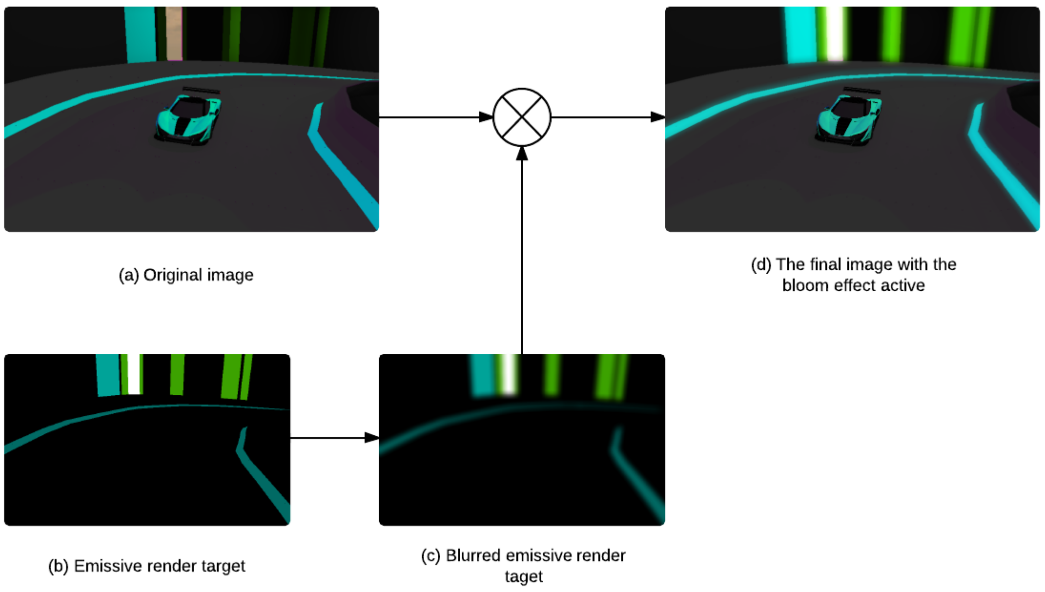
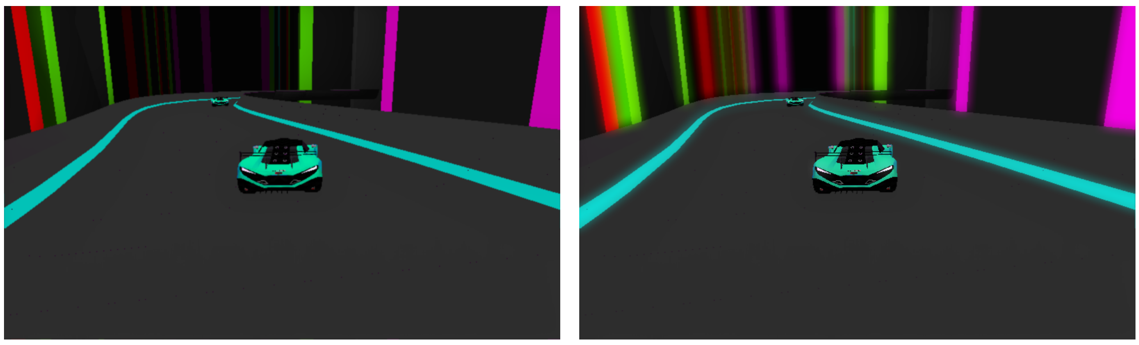

(a) Original image
(d) The final image with the bloom effect active
(b) Emissive render target
(c) Blurred emissive render taget

Figure 2.16: Rendering steps for adding bloom to the scene. (a) Contains the original image, (b) shows solely the emissive parts of the scene. (c) Is the emissive parts after the blur and (d) contains the final image after the blurred emissive image have been additively blended into the original image.

Results

When comparing the image without bloom (see Figure 2.17(a)), alongside the image with bloom (see Figure 2.17(b)), the visual contribution of the bloom effect becomes apparent. Since our models make use of an extensive amount of emissive materials, the effect makes a huge impact on the appearance of Gravitron, without it, the emissive materials appears to be flat and dull.



(a) An image without bloom. (b) An image with bloom.

Figure 2.17: A comparison between (a) an image without bloom, and (b) an image with bloom.

Discussion

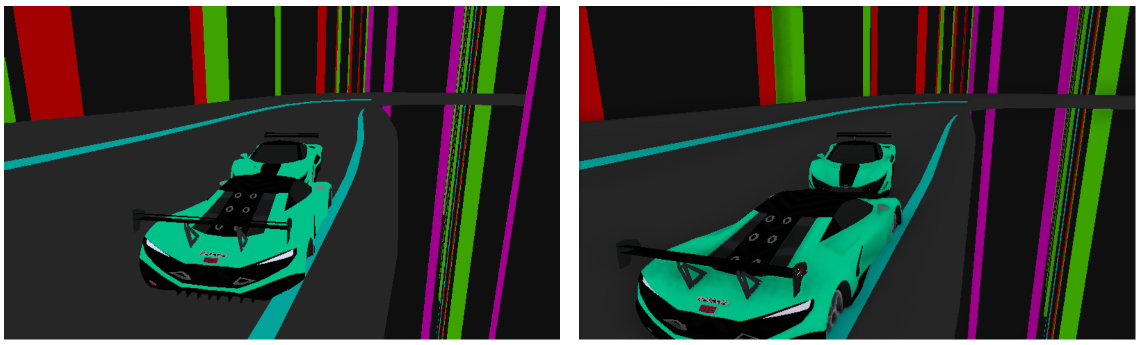
One of the objectives of Gravitron was to create a dark city environment with many bright light sources outlining the buildings. The bloom effect makes the emissive colors in the scene look like neon lights, which greatly aids in achieving the artistic goal set in this project.

We could probably have improved upon the ”neon light effect” by adding white emissive colors to it. A neon light is basically a white light source in a colored transparent container. However, the team did not have any experienced 3D-modelers, and therefore these changes were not prioritized.

Since the Gaussian blur method is not only used in our implementation for bloom, but also for our screen space ambient occlusion (SSAO) effect (see Section 2.2.6), the effort put into implementing the effects was significantly reduced. In addition, the emissive render target was already pre-computed by our deferred shading implementation (see Section 2.1).

2.2.6 Screen Space Ambient Occulsion

Ambient occlusion (AO) is a measure of how much of the hemisphere of a point is occluded, and roughly corresponds to the amount of indirect illumination [28]. Even though it is not a realistic phenomenon, since reflected light from the sky or other objects is rarely homogenous, the scene feel more realistic by enhancing the perceived depth in the image (see Figure 2.18).



(a) An image without SSAO. (b) An image with SSAO.

Figure 2.18: An illustration of the difference with and without Screen Space Ambient Occlusion (SSAO). The geometry of the car seems flat in the image without SSAO (a), but in the image with SSAO (b) you can see the actual shape of the car.

Previous Work

There are several existing methods to compute AO, and the difference between them is in the extent to which accuracy is traded for speed. The methods using ray tracing are the most accurate, but they are also very slow, and are therefore not suitable for real-time rendering. However, the results from such calculations can be used in real-time, although it adds the limitation that the scene must be static [29].

A class of real-time methods, collectively known as screen space ambient occlusion (SSAO), trades accuracy for a significant increase in performance. SSAO, compared to other AO methods, works on all types of scenes, is exceedingly faster, and simpler to implement and integrate to existing rendering pipelines [30]. However, it is far from accurate, and suffers from numerous quality issues. Despite that fact, due to its simplicity and speed, SSAO has become popular in 3D games and other interactive applications.

The first version of SSAO was the ”CryTek Screen Space Ambient Occlusion” [31]. After came improvements and variations of it, such as ”Approximating dynamic global illumination in image space” [32] and ”Multi-layer dual-resolution screen space ambient occlusion” [33], which sought to increase quality and speed. A quite different approach in screen space is Ambient Occlusion Volumes [34], which is analytical and produces smooth and near ground truth results at impressing frame rates, but is still too slow for usage in games.

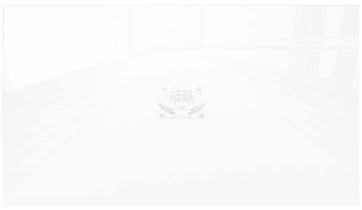
The technique used in Gravitron is based on ”A Simple and Practical Approach to SSAO”, presented by José María Méndez [35], and is desirable due to its simplicity and accessible code. It samples randomly within a sphere surrounding the pixel, but all contributions sampled behind the point, for which the occlusion is computed, will be discarded, i.e., only the samples that are sampled within the hemisphere will contribute to the ambient occlusion term. The occlusion is calculated as seen in Equation 2.3, where n is the normal of the occluded point and v is the vector between the occludee and the occluder. Blurring is used to avoid the noise caused by sampling randomly.

$$Occlusion = \frac{max(0, \hat{n} \cdot \hat{v})}{1 + ||v||} \tag{2.3}$$

Results

There are three artistic control parameters, and their impacts of the AO were studied empirically (see Figure 2.19). As can be seen in the top row, when the sample radius parameter grows, the influence of an object increases, i.e., the car creates a larger contact shadow. With a low value of the sample radius, corners in the scene becomes more distinguished, which produces a more detailed AO (see Figure 2.19(a)). However, if the value is too low, it will begin self-occluding, resulting in an overall darker image.

The middle row shows the effect of the intensity parameter, which determines the strength of the occlusion. The impact of the last parameter, scale, is shown in the third row, and it scales the length of the vector between the sample and the point for which the occlusion is calculated. Values higher than one only results in a slightly fainter AO (see Figure 2.19(i)). When scaled with a low value, the image begin to show ”dark halos” (see Figure 2.19(g)), which are prominent at the edges of the track. The images in the middle column have the same parameter values, and are also thought of as the result with the highest natural fidelity.

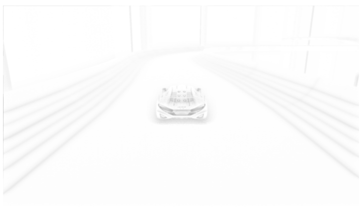

(a) Sample radius = 1

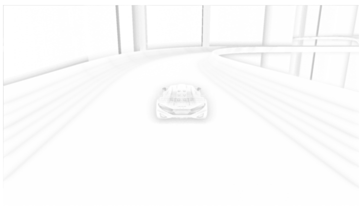
(b) Sample radius = 10

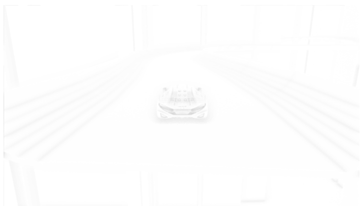
(c) Sample radius = 20

(d) Intensity = 0.5

(e) Intensity = 1


(f) Intensity = 2


(g) Scale = 0.3


(h) Scale = 1

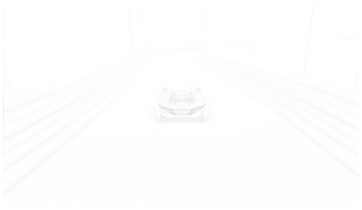
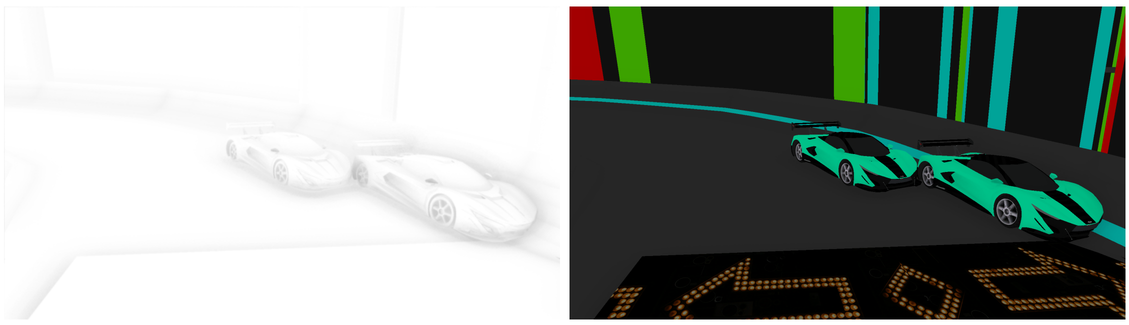
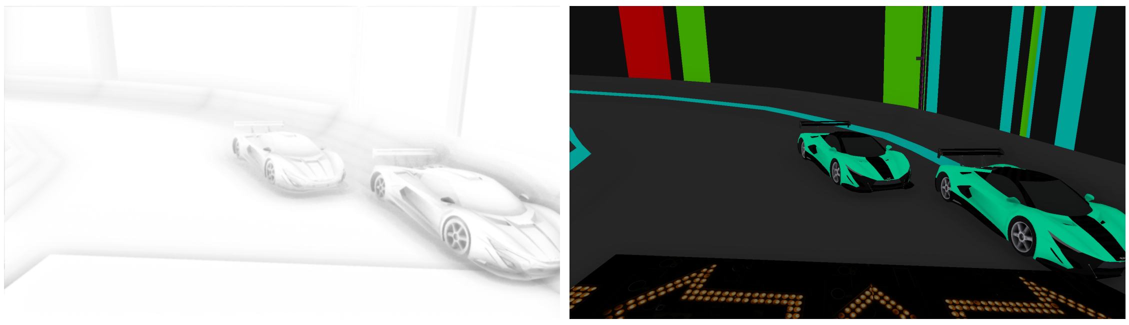

(i) Scale = 2

Figure 2.19: Comparing different values of sample radius, intensity, and scale. The images in the middle column have the same parameter values (sample radius = 10, intensity = 1, scale = 1), and work as a reference. Each row shows the effect of individually changing one of the parameters from its default value. These results are best viewed on a monitor with high contrast, or a printed copy of the report.

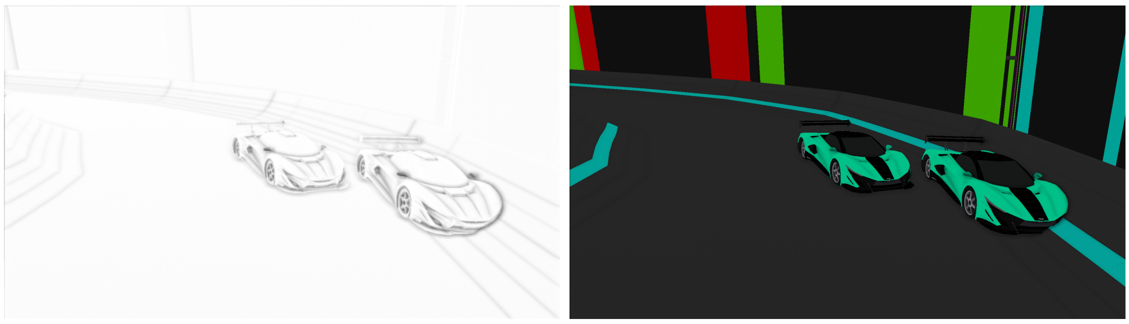
When the SSAO, with the parameter values that produced the most accurate occlusion, was applied to the final image, it produced no significant increase in perceived depth (see Figure 2.20(a)). The tweaking of the parameters begun anew (see Figure 2.20). The intensity was first increased, and the AO became more pronounced, but was still thought of as vague (see Figure 2.20(b)). The sample radius was lowered and the corners of the body of the car became visible (see Figure 2.20(c)), which brought out its shape.



(a) Sample radius = 10, Intensity = 1



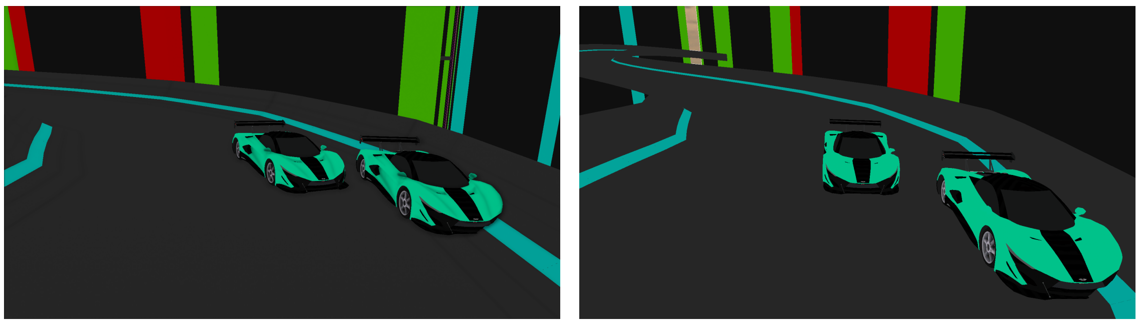
(b) Sample radius = 10, Intensity = 2



(c) Sample radius = 2, Intensity = 2

Figure 2.20: The first pair of images shows the resulting AO with the default parameter values. The desired increase in perceived depth is not achieved, which lead to the second pair of images where the intensity parameter was increased. To emphasize the shape of the car, the sample radius was reduced, which resulted in the final pair of images. The perspective of these images slightly varies, but that is not an effect of SSAO.

The configuration of sample radius = 2, intensity = 2, and scale = 1 produced the best results and was therefore chosen as the SSAO configuration in this thesis. The effect creates depth in the scene by darkening corners, and nooks, but the contact shadows is nearly gone (see Figure 2.20(c)). However, Gravitron is desired to be fast paced and the contact shadows between the car and the road should not be very noticeable anyway, because the car will be in focus all the time, and therefore the visuals of the car was prioritized. The geometry of the car looks more refined with our SSAO configuration than without (see Figure 2.21).



(a) The scene with SSAO. (b) The scene without SSAO.

Figure 2.21: This figure illustrates the visual difference between a scene with SSAO (a) and one without (b). It is clear that the geometric shape of the cars is more defined with SSAO enabled.

The impact of the effect on the frame rate can be seen in Table 2.1. The time spent calculating the AO is 0.32 ms when using the resolution 800x480, and 1.44 ms with 1920x1080, which implies that the time it takes for the AO to be computed increases linearly with the resolution.

	Without SSAO (fps)	**With SSAO (fps)**	**Time (ms)**
800x480	465	405	0.32
1920x1080	218	166	1.44

Table 2.1: Showing the frame rates for the resolutions 800x480 and 1920x1080 with and without SSAO enabled, and the time in milliseconds spent computing the effect, using a GeForce GTX 780.

Discussion

AO is important to create more pleasing graphics and is considered as one of the core post-processes in Gravitron. The perceived depth is enhanced, which makes surfaces seem less flat. However, in our current scene, we do not have a lot of complex geometry that benefits of AO. Nevertheless, the shape of the car is greatly enhanced, and if we were to update the scene with more complex geometry, such as detailed building structures, the result of AO would be of greater significance.

Our implementation of SSAO required very little time, and has quite the impact on the visual appearance of the game. Therefore, implementing the effect was well worth the effort. The accuracy may be low, but the speed is not, and spending more computing time for more accurate AO cannot be justified.

2.2.7 God Rays

When an object partly occludes the sun or another intense light source, there is a possibility that a phenomenon called light shafts will occur (see Figure 2.22). However, this requires that the space, through which the light is transported, consists of a sufficient amount of light scattering media, such as gas molecules or aerosols [36]. The god ray effect approximates the phenomenon, and helps distinguish light sources with high intensity in a scene - usually the sun.

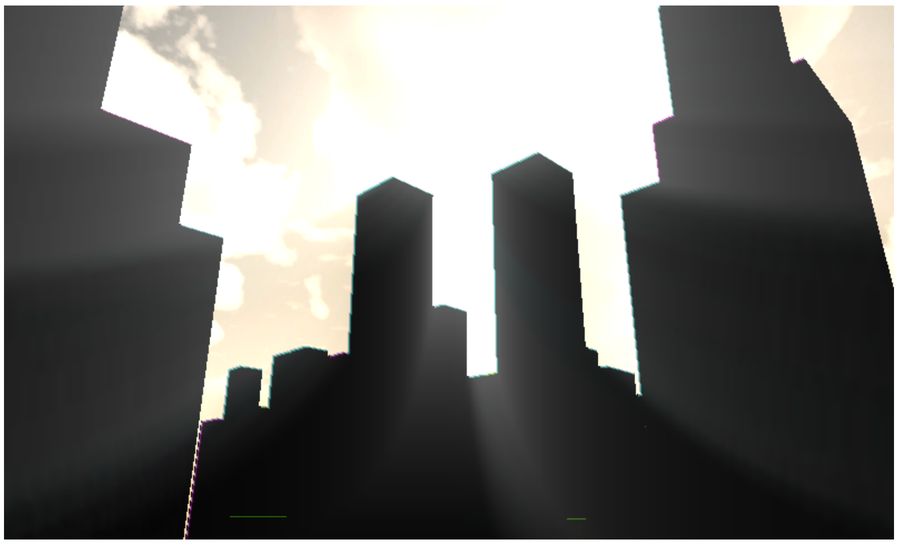


Figure 2.22: Light shafts, here referred to as god rays, emerging between the buildings.

Previous Work

The very first occurrence of the effect, in computer graphics, was a modified shadow volume algorithm [37]. However, the technique is not rendered in real-time, and is therefore not considered for implementation in Gravitron.

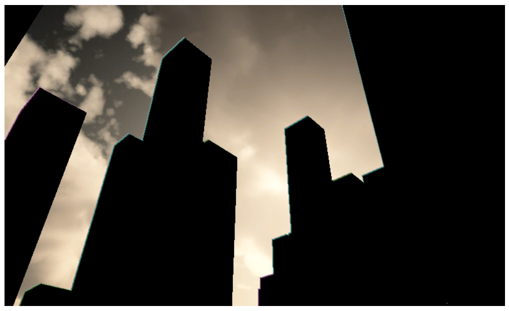
Two methods that are efficient enough for real-time rendering are slice-based volume rendering [38] and hardware shadow maps [39]. Due to the fact that both methods are forward rendering techniques, none of them are suitable for this project.

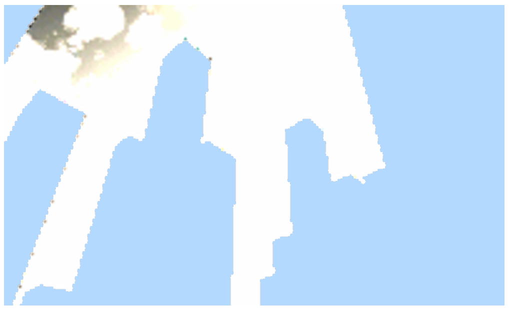
Recently, an implementation of god rays as a post-process was proposed [36]. Since a deferred renderer is used in this project, which provides a functional post-processing pipeline, this is an interesting option. The post processing method can be described in three steps (see Figure 2.23): first, the foreground objects are detected and masked out by using the depth buffer; secondly, rays are generated and stored in a separate render target; lastly, the rays are additively blended with the original image.

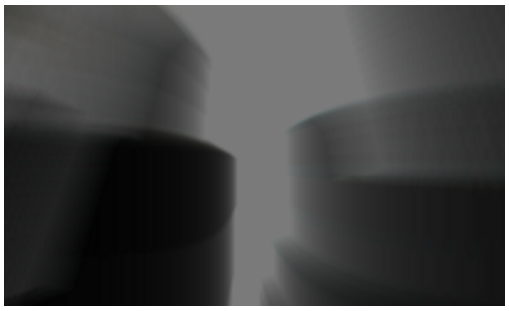
To generate the rays, the color value of each pixel is calculated by taking a number of samples on a vector, from the screen space light position to the pixel. The combined color of the samples on such a vector represents the color of the corresponding pixel. However, no samples that are occluded will contribute to the final pixel color. The sampling is described in Equation 2.4.

$$L(s, \theta, \phi) = exposure \times \sum_{i=0}^{n} decay^i \times weight \times \frac{L(s_i, \theta_i)}{n}, \tag{2.4}$$

where s is the distance traveled through the media, θ is the angle between the ray and the light source, ϕ is the view location, n is the number of samples and $L(s_i, \theta_i)$ [36] is the daylight scattering model. The *exposure* factor describes the overall intensity of the rays, *weight* controls the intensity of each sample and *decay* smoothly attenuates the light contribution of each sample as the distance to the light source increases.


(a) The original image.


(b) Mask of the occluding object, stored in the alpha channel.


(c) The god rays after sampling.

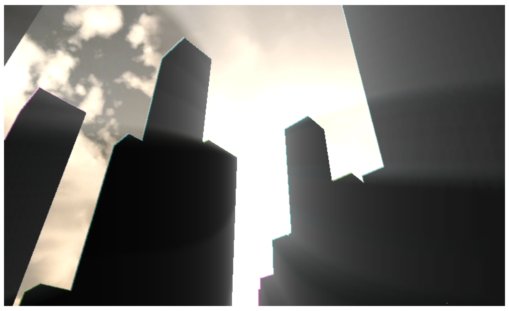
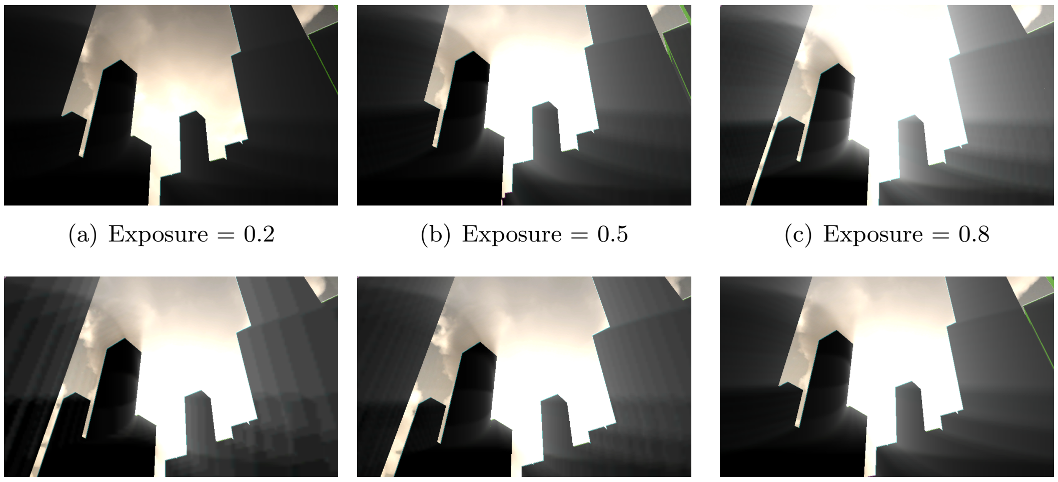

(d) The final image with the god ray post-process applied.

Figure 2.23: The pictures illustrate the procedure of creating the god ray effect as a post-process. (a) Contains the original image, (b) shows an image with the foreground objects masked out, (c) displays the original image after the sampling, and (d) is the sampled image additively blended into the original image.

Results

There are a number of parameters (see Equation 2.4), that can alter the appearance of the rays (see Figure 2.24). When comparing the different exposure values, one can clearly see the impact of the parameter. With the low exposure values (see Figure 2.24(a)), the god rays are barely seen, while the high exposure image (see Figure 2.24(c)), indicates overexposure of the sky when looking straight into the light. When choosing an exposure value in between (see Figure 2.24(b)), the result shows a balanced highlighting of the light shafts, as well as a moderate exposure of the sky.

When analyzing the two images with a low number of samples per pixel (see Figure 2.24(d) and Figure 2.24(e)), the undersampling of the rays becomes apparent. This makes the light shafts appear as segmented boxes, rather than smoothly faded rays. When comparing the impact from a higher number of samples, the one with 80 samples (see Figure 2.24(f)), indicates slightly less discrete rays, compared to the one with 40 samples (see Figure 2.24(b)). However, since the gain in quality is modest, the trade-off in performance is not justified for Gravitron.



(a) Exposure = 0.2 (b) Exposure = 0.5 (c) Exposure = 0.8

(d) 10 Samples per pixel (e) 20 Samples per pixel (f) 80 Samples per pixel

Figure 2.24: A comparison of properties of the God Rays. The top row shows the results of different exposure values, with a constant number of samples (40), while the bottom row shows variation in number of samples, with a constant exposure value (0.5).

Discussion

The god ray effect gives the player the perception of a scene that is less static, which is essential when it comes to games, and especially racing games. After careful observation, the god rays may not provide any particular contribution to the TRON-feeling in the game, but it does work well in our dark city environment. The light shafts beaming down through the city skyline creates a contrast in the otherwise dark environment.

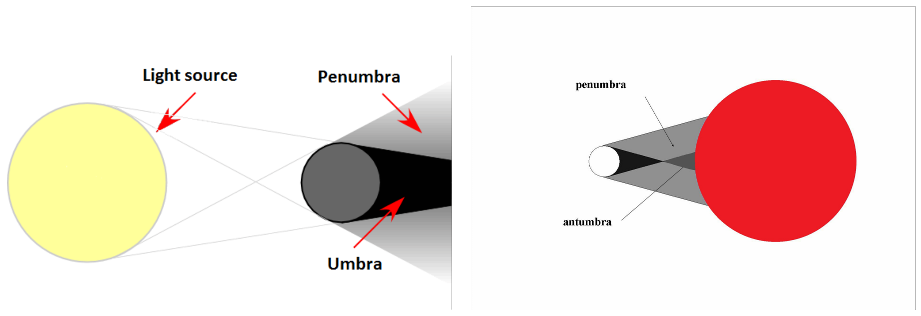
When it comes to visibility of the effect, the god rays are obscured by the large amount of tall buildings that are present in the game, and are therefore not a common sight in Gravitron. Although, this could easily be solved by thinning out the city skyline. We could also have improved the aestethic aspect of the effect by adding more red color to the god rays, as well as the sun, which probably would have created the feeling of a sunset.

The amount of work required to implement the post-process effect was quite low. More effort could have been spent on the implementation, mainly when it comes to the tweaking of different parameters. In spite of this, we regard the overall results of the effect as satisfying.

2.3 Shadows

Determining if an object is mid-air or touching the ground is easy if a corresponding shadow is apparent. Gravitron features jumps between platforms, and shadows aid the player in determining where the car is and how to maneuver.

A shadow is the phenomenon when a space is occluded by an object in between the light source and the space in question. There are three components required for shadows to take form: a light source, an occluder and the shadowed object. The shadow can be divided into three different parts: the umbra, penumbra and the antumbra (see Figure 2.25) [40].



(a) The umbra and the penumbra. (b) The antumbra and the penumbra.

Figure 2.25: The black area (umbra) represents where no light reaches, and the grey area (penumbra) represents where some light reaches (a). Beyond the umbra extends the antumbra (b), which just like the penumbra is partially occluded.

2.3.1 Previous Work

Ray-tracing computes shadows with high accuracy. However, since ray-tracing is not suited for real-time rendering, it is not considered as a viable solution for use in this project.

There are two commonly used groups of techniques for rendering shadows in real-time. One of them, shadow volumes, uses the stencil buffer and the depth buffer, to create shadows by extruding edges and calculating where the shadows are in the view frustum of the camera. The shadows are calculated with either depth-fail or depth-pass [41]. The other group of techniques, shadow mapping, is simpler. By only utilizing a depth buffer, created from the point of view of the light source, and z-testing, shadows are created quite efficiently [42].

Shadow volumes produces more accurate shadows, however, it is expensive in mesh-heavy scenes because of the necessity of an edge detection algorithm [41]. Shadow mapping, on the other hand, scales excellently with large scenes, does not require a lot of computational power, and is easy to implement. Shadow mapping does, however, have problems with artifacts, and is limited to hard shadows without aid from other techniques. Taking all of this into account, shadow mapping is the technique chosen to render shadows in Gravitron.

Shadow mapping was introduced in 1978 [43]. The technique is performed by rendering the depth, as seen from the perspective of a light source, to a depth buffer (see Figure 2.27). The depth buffer is then used when computing the lighting of the scene. Lastly, screen space positions are translated to the view space of the light source. If the z-coordinate is greater than the value in the depth map, the point is in shadow (see Figure 2.26).

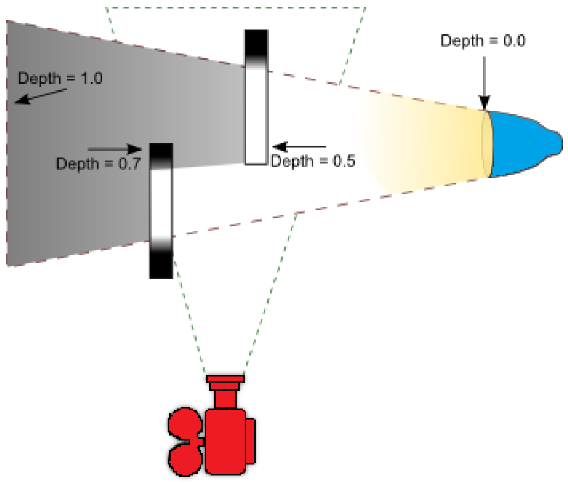


Figure 2.26: This picture presents a light shining on two objects, which in turn casts shadows. The depth values are normalized distances between the objects and the light source.

2.3.2 Results

Gravitron uses one unique shadow map for each car, which follows the car (see Figure 2.28). The shadows are very simple, and due to the use of shadow mapping they have some aliasing problems. However, the aliasing is not very noticeable while playing the game, since the resolution of the shadow map is relatively high in comparison to the area of the object casting the shadow.

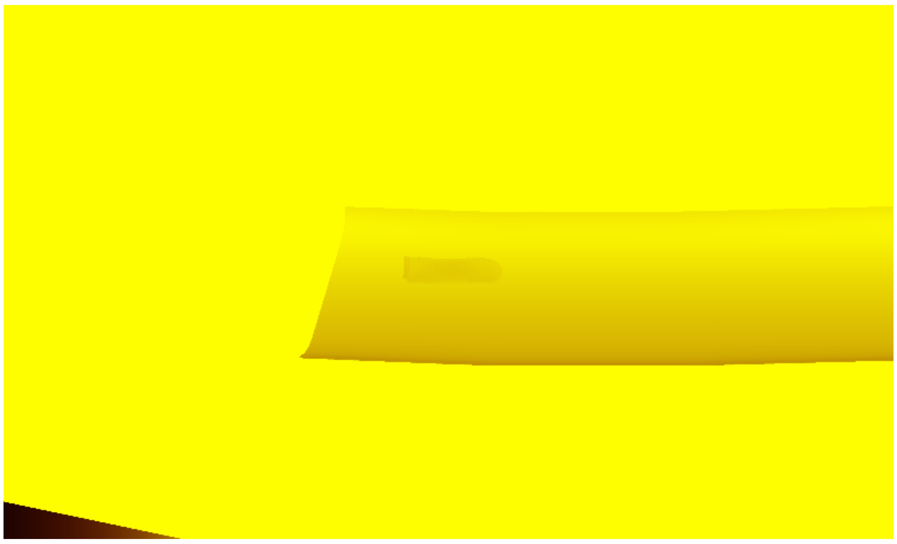


Figure 2.27: The depth buffer used for rendering shadows in Gravitron. Darker colors corresponds to points/objects closer to the light source.

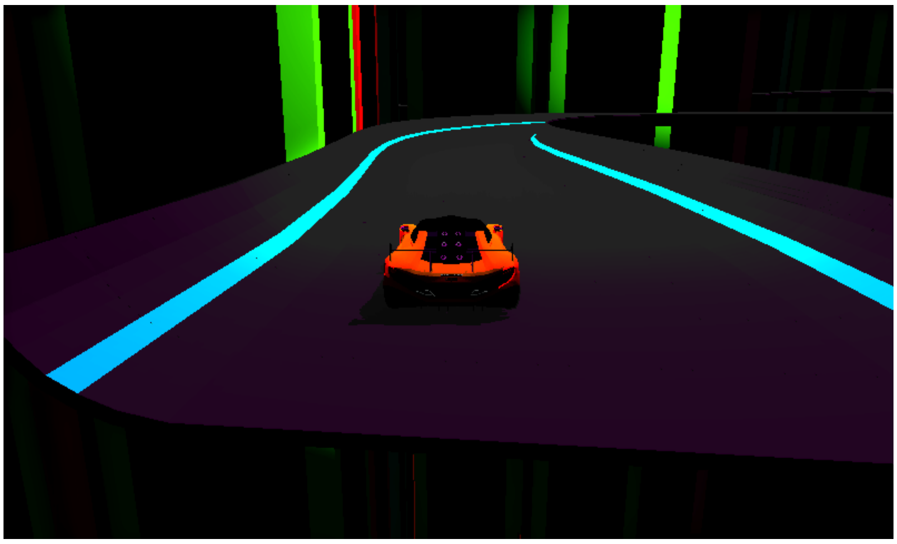


Figure 2.28: The shadow cast by the car.

2.3.3 Discussion

The implementation of shadow mapping took more time than expected. First, a problem with the directional light stopped the progress, and later a problem with the view-projection matrix transformation to the shadow view space delayed the implementation. Other than that, the implementation was straight forward and smooth.

The result of the shadows in Gravitron is not very dominant, and prioritizing the implementation can be questioned. However, it does add a sense of perceived depth. Having shadows under the car promotes the feeling of being in mid-air.

2.4 Particle System

Objects with fuzzy and irregular shapes are harder to visualize and are a recurring problem when creating effects such as: fire, dynamic water, smoke, and snow. The movement of these effects are usually perceived as random and chaotic. In games, particle systems are often used to manage these objects. This section will describe the particle effects that have been considered during this project and the integration with the game engine.

2.4.1 Previous work

Particle systems were first introduced in 1983 and they create objects of primitive particles that move over time [44]. These particles have a lifetime, which is defined upon their creation, and at the end of it, they are deleted. Non-deterministic variables are used to create a random behavior, this gives the object a more irregular visual.

Different methods to realize a particle system was considered, either to create an entire particle system from scratch, or to use an existing framework. The latter option was chosen due to the limited timeframe, and that the performance of the particle system would most likely be better.

There are two types of particle systems, one that is static and one that is animated. A static particle system is distinct in the way that it renders the entire lifecycle of a particle at once. This is often seen in the rendering of hair. In that example, a strand of hair is the result of a particle being displayed in all its life stages at once.

An animated particle system goes through a lifecycle of a particle one step at a time. This is useful when creating flowing effects such as water or fire. The animated particle system was therefore deemed more appropriate for this project.

Particles are usually drawn as billboards or as point sprites. Billboards are a common way to display textures in particle systems, and they consist of a quad with four vertices marking the corners. Point sprites, on the other hand, only need one vertex in the center to define its position. Only using one vertex means that less computations are performed on the engine. However, larger particles will not be rendered when the center is off screen. Since Gravitron needs large particles, a billboard based particle system is required.

Two particle systems were considered: Dynamic Particle System Framework (DPSF) and Mercury Particle System. Mercury Particle System is an open source project, and DPSF is not, but DPSF does provide great tutorials and code examples. Mercury Particle system was discarded, since its developers stopped supporting it in 2013. Since the developer of DPSF is still working actively with the particle system, that was chosen. DPSF also seemed simple to integrate with the existing game engine, and it offered great customizability for the particles.

2.4.2 Results

The particles are drawn in real-time as billboards. The different billboards originate from png-files, which gives them a base that can be used for various effects. Upon the initiation of the particles, different attributes are set. These attributes control where the particles spawn, how they move, and how their appearance change over time. Several different particle effects were created for this thesis and are featured below.

The initial idea of the game was to achieve a resemblance to the classic TRON-game, where the vehicles have glowing tracks that appear behind them. The particle system simulated this effect by creating a glowing tail that follows the car (see Figure 2.29(a)).

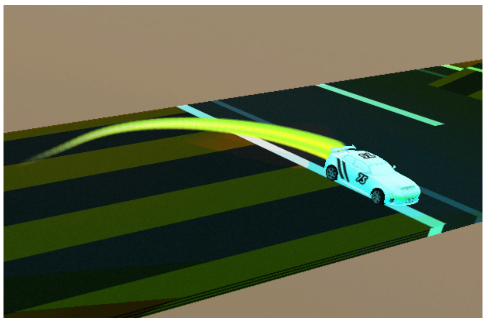
Another prioritized effect is rain. Rain can be costly for the processor, especially since all the particle calculations for this thesis are computed in real-time on the CPU. In order to decrease the computations, multiple raindrops are drawn per raindrop particle, instead of separately (see Figure 2.29(b)).

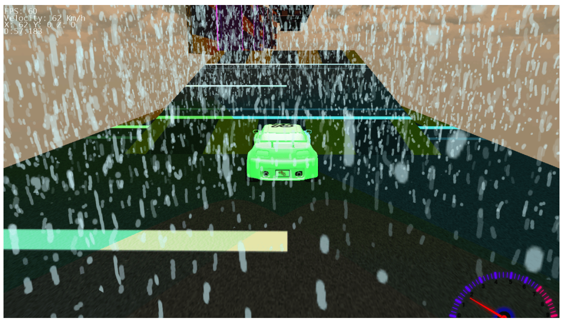
The ability to prepare the player for the gravitational shifts in the game is important. A spinning circle, or the ”gravity circle”, was created for that purpose. Furthermore, its rotational direction indicates which way the gravity will turn. This is the largest particle that the system uses (see Figure 2.29(c)).

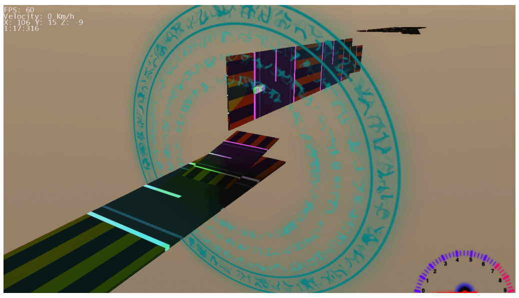
Smoke is a multi-purpose effect, and can be useful in almost every game. In this case, it might be spawned from the engine, the exhaust pipe, and traps. Smoke clouds were created, and can easily be adapted to any of these situations (see Figure 2.29(d)).

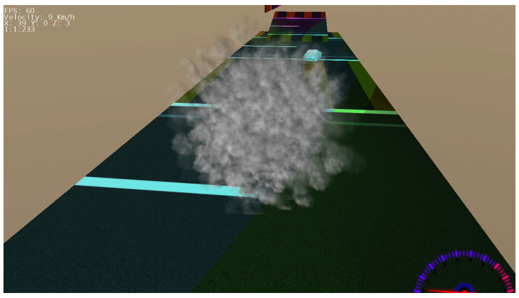
The game aims to have a futuristic appearance and aerial lights can contribute to this (see Figure 2.29(e)). Basically, the effect consists of lights flying by above the track at a random velocity in the opposite direction of the race. This effect also intensifies the sense of speed.

Unfortunately, there are some problems integrating the framework into the final version of the project, since it has trouble coexisting with deferred rendering. Due to some implementation restrictions, the particles cannot be trivially rendered into the G-buffer.


(a) Glowing tracks gives the scene a more TRON-resembling look.


(b) The result of the rain effect.


(c) Gravity circle indicating gravity change.


(d) The result of the smoke effect.

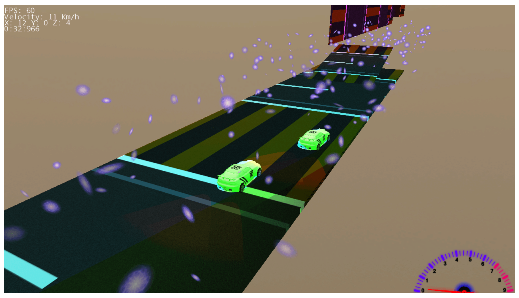

(e) The result of the aerial lights effect.

Figure 2.29: The figure shows the different particle system effects that were created for this thesis.

2.4.3 Discussion

The framework was a great solution when used for controlling and customizing the particles the way we wanted to. At first this seemed like the preferable option, but we realized our mistake when we tried to integrate it with deferred shading. Our results in the final version would most likely have been better, if we had implemented our own particle system. That would have given us full control of how the particles are drawn. However, if we had implemented our own particle system, the amount of different particles would not have rivaled the amount of particles we achieved with DPSF. Although, we probably would have had some that worked in our final version.

3

Results

The reader should now have a basic understanding of the individual contribution to the visuals of each effect used in Gravitron. In this chapter, the combined results of the techniques, explained and analyzed in Chapter 2, will be presented.

Gravitron is close to achieving the visual ambition set in this project (see Figure 3.1). Neon lights, outlining models in the city environment, are enhanced using the bloom effect (see the neon lights outlining the track and the buildings in Figure 3.1). It is also visible that geometric shapes in the scene, mainly the cars, are enhanced by screen space ambient occlusion (see the hood of the cars in Figure 3.1). The overall impression of the scene is that edges look smooth, due to the anti-aliasing technique used in Gravitron. However, from certain angles, edges can be perceived as dithered or over blurred.

In Gravitron, the player drives on a racing track in a gloomy city environment. The track is split into four sections (see Figure 3.2), which are connected by what will be referred to as gravitational junctions. A gravitational junction consists of a jump, which triggers a gravitational shift, and a platform (see Figure 3.3). The gravitational shifts are a part of the core game mechanics in Gravitron, and they bring an interesting element to the game. Textured jumps with emissive arrows give the player an indication of the direction in the upcoming gravitational shift.

The deferred renderer allows some artistic freedom regarding lighting of the game scenes. One application of this is the spotlights that are attached to the car. This enables the car to light up the environment, adding to the feeling that the car is actually part of the scene (see Figure 3.4).

God rays, beaming down from above, in the dark city environment, brings life to the scene (see Figure 3.5). This is an aspect in which Gravitron is lacking, due to the particle system not making it into the final implementation. There are only a few moments in

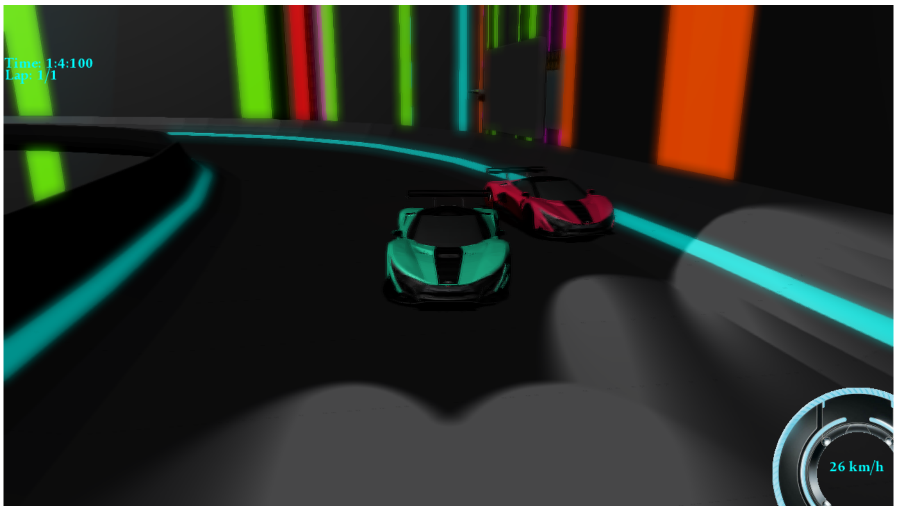


Figure 3.1: A scene from the final implementation of Gravitron. The figure shows the dark environment in Gravitron with glowing outlines on the building and the track.

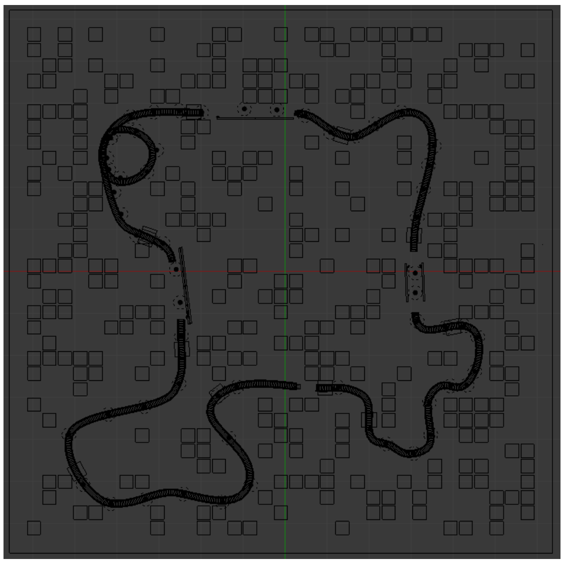
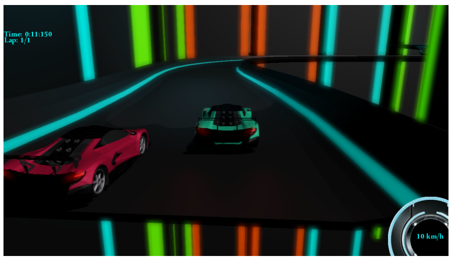


Figure 3.2: An overview of the race track used in Gravitron. The track is split into four sections, one in each quadrant.

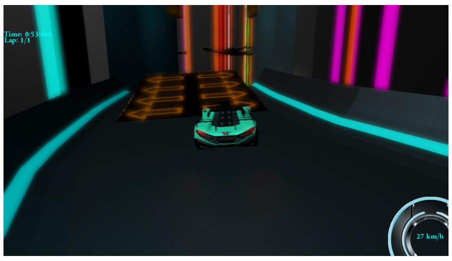
the game, when the god rays are clearly visible. Nevertheless, in those moments, they enable brightness to be a part of the otherwise dark game.

Shadows cast by the car amplify the depth visualization of the scene, which further incorporates the car with the environment. However, there are scenes when the shadows are concealed due to the dark environment.

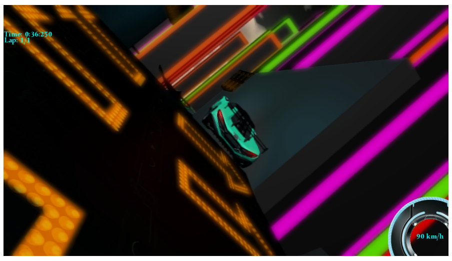
Motion blur intensifies the sense of velocity, in which the vehicle in the scene is moving



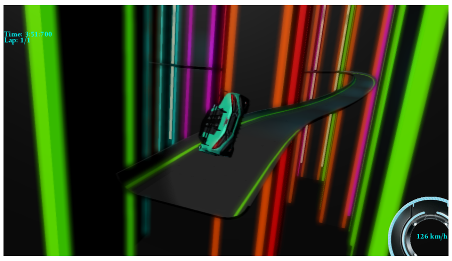
(a) The starting position.



(b) A gravity junction.



(c) A car jumping to a platform.



(d) A car jumping to the main track.

Figure 3.3: The figure shows the gravitational shift mechanics used in Gravitron. (a) The cars at their starting position. (b) A gravity junction, consisting of a jump and a platform. (c) The driver can choose a direction on the jump, guided by the arrows, which will trigger a gravitational shift in that direction. (d) The player trying to jump back onto the main track from a platform.

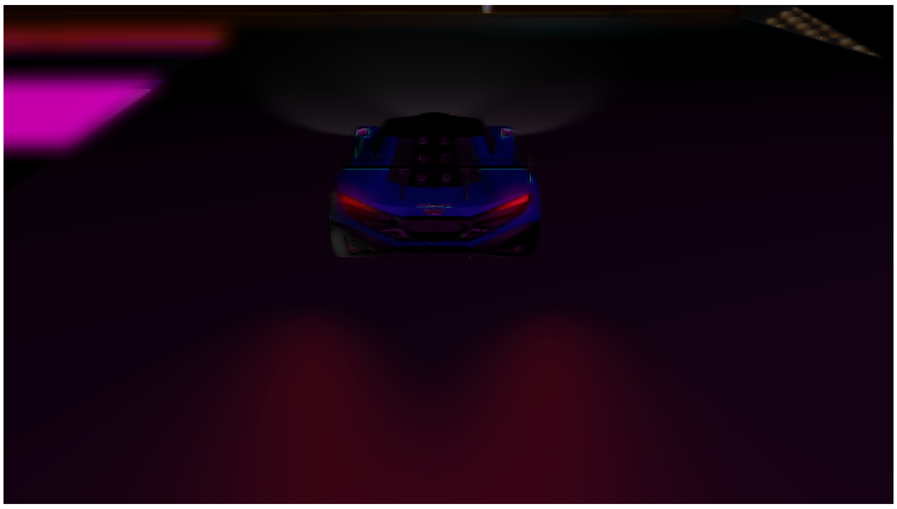


Figure 3.4: The car lights in Gravitron.

(see Figure 3.6). However, when using the motion blur in Gravitron at all times, the blur can be perceived as excessive. Therefore, the final motion blur is preferably only used with speed boosts.

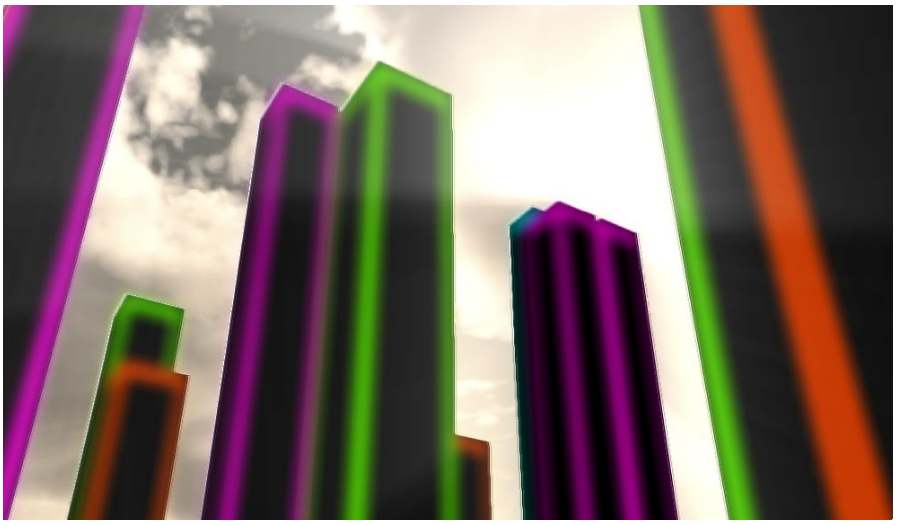


Figure 3.5: The god rays beaming down into an otherwise dark city environment.

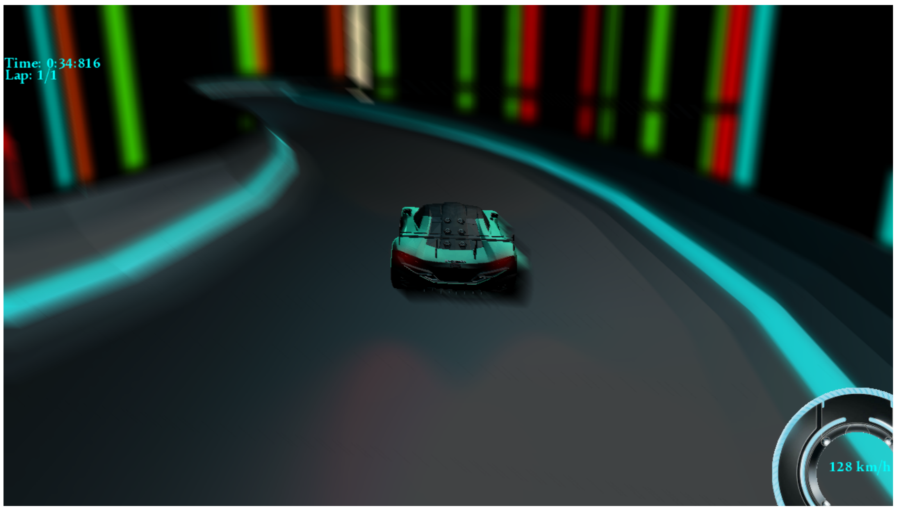


Figure 3.6: The figure shows the motion blur used in Gravitron. Observe how the car gets extruded in the opposite direction of its velocity.

4

Discussion and Conclusion

Our intention with this project was to explore which graphical effects are best suited for a racing game set in a dark environment with a lot of glowing light sources. Of all the post-processing effects implemented, we believe that bloom, screen space ambient occlusion, and god rays, contributes the most towards the visual experience we were trying to achieve. Bloom is the most prominent one, since it creates the impression of emissive materials that are glowing.

A TRON-environment was used as inspiration for the glowing outlines of our models in the game, and we believe that the models we have created aids in reaching our visual goal. The combination of a deferred renderer and the graphical effects researched made the luminous city environment in Gravitron possible.

The screen space ambient occlusion effect required very little time to implement. Nevertheless, it still has an impact on the final visual appearance of the game that we are satisfied with and, thus, an effect well worth its implementation time. Unfortunately, we do not have a lot of complex geometry in the models that benefit from ambient occlusion. If the models are updated with more complex geometry, the effects of ambient occlusion will be even more significant.

Another effect that we believe is important for the game is motion blur, as it brings out a sense of speed, which is vital in a racing game. We consider the final result of the motion blur effect used in Gravitron to be a success, especially if combined with speed boosts.

Shadows were given a lot of priority, but regrettably, the final result of the shadows is not satisfying. Most of the time the shadows are not very noticeable, since the environment that is used in Gravitron is very dark. In afterthought, we probably could have prioritized shadows less and instead made it our priority to create new effects, or improve already existing effects, with a greater visual impact.

The development model that we used allowed us to quickly dive into the development phase, but turned out to be inadequate in some regards. We believe that an additional step in the process, promoting a more thorough research of the techniques would have improved the quality of our work. The method used, without this additional step, could be better suited for a team more experienced in game development.

A question we asked ourselves at the start of this project was: ”How far can a group of five reach with a time limit of three months in order to create a TRON inspired racing game?”. We feel the need to make it clear that this project does not represent how far a group of five will reach, but rather, how far they *can* reach. That being said, in order to reach further with our visual goal, some graphical effects could have received a higher priority. The particle system, as previously stated, would probably have brought more excitement to the game, and should most likely have been given more priority.

Finally, an important part of this thesis was to get an understanding of the underlying techniques used in modern 3D-graphics rendering, therefore, XNA was used. If we had used a development tool-kit, such as Unity or Unreal Engine 4, the game would most likely have progressed further. Nevertheless, we have succeeded in creating a game which contains some of the artistic elements and game features we set out to achieve.

Bibliography

[1] (2012) Tron: Uprising. [Online]. Available: http://www.imdb.com/title/tt1812523

[2] O. Olsson *et al.*, "Tiled shading," *Journal of Graphics, GPU, and Game Tools*, vol. 15, no. 4, pp. 235–251, 2011. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/2151237X.2011.621761

[3] M. Deering *et al.*, "The triangle processor and normal vector shader: a vlsi system for high performance graphics," in *SIGGRAPH '88 Proceedings of the 15th annual conference on Computer graphics and interactive techniques.* New York, USA: ACM, August 1988, pp. 21 – 30. [Online]. Available: http://dl.acm.org/citation.cfm?doid=378456.378468

[4] T. Saito *et al.*, in *SIGGRAPH '90 Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, August 1990.

[5] R. Geldreich. Gdc 2004 presentation on deferred lighting and shading. [Online]. Available: http://sites.google.com/site/richgel99/home

[6] F. Policarpo *et al.* (2005) Deferred shading tutorial. [Online]. Available: http://nume.googlecode.com/svn-history/r12/trunk/monnezza/redsh/ogre1/Deferred_Shading_Tutorial_SBGAMES2005.pdf

[7] A. Lauritzen. (2010) Deferred rendering for current and future rendering pipelines. [Online]. Available: http://bps10.idav.ucdavis.edu/talks/12-lauritzen_DeferredShading_BPS_SIGGRAPH2010_Notes.pdf

[8] S. Hargreaves *et al.* (2004) Deferred shading. [Online]. Available: http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf

[9] J. Jimenez *et al.*, "Filtering approaches for real-time anti-aliasing," in *ACM SIGGRAPH Courses*, 2011.

[10] E. R. Davies, *Machine Vision*, 3rd ed. Morgan Kaufmann, 2005.

[11] N. Senthilkumaran *et al.*, "Edge detection techniques for image segmentation – a survey of soft computing approaches," *International Journal of Recent Trends in Engineering*, vol. 1, no. 2, pp. 250 – 254, May 2009. [Online]. Available: http://ijrte.academypublisher.com/vol01/no02/ijrte0102250254.pdf

[12] P. P. Acharjya *et al.*, "A study on image edge detection using the gradients," *International Journal of Scientific and Research Publications (IJSRP)*, vol. 3, no. 12, December 2012.

[13] L. Davis, "A survey of edge detection techniques," *Computer Graphics and Image Processing*, vol. 4, no. 3, pp. 248 – 270, September 1975.

[14] A. Vasiliauskas. (2010, June) Edge detection pixel shader. [Online]. Available: http://coding-experiments.blogspot.se/2010/06/edge-detection.html

[15] T. Akenine-Möller *et al.*, *Real-Time Rendering*, 3rd ed. Taylor & Francis Group, 2012.

[16] J. Jimenez *et al.*, "Smaa: Enhanced morphological antialiasing," *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, vol. 31, no. 2, 2012.

[17] A. Reshetov, "Morphological antialiasing," in *Proceedings of the 2009 ACM Symposium on High Performance Graphics*, 2009.

[18] M. S. Nixon *et al.*, *Feature Extraction and Image Processing*, 2nd ed. Academic Press, 2008.

[19] A. W. R. Fisher, S. Perkins *et al.* (2003) Gaussian smoothing. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

[20] D. Rákos. (2010) Efficient gaussian blur with linear sampling. [Online]. Available: http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

[21] A. W. R. Fisher, S. Perkins *et al.* (Unknown) Bilateral filtering for gray and color images. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

[22] ——. (2003) Median filter. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm

[23] ——. (2003) Mean filter. [Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/mean.htm

[24] S. Green *et al.*, "Stupid opengl shader tricks," in *Advanced OpenGL Game Programming Course*, 2003.

[25] G. Rosado. (2007) Motion blur as a post-processing effect. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch27.html

[26] T. Ritschel *et al.*, "Temporal glare: Real-time dynamic simulation of the scattering in the human eye," *Computer Graphics Forum*, vol. 28, pp. 183–192, March 2009.

[27] G. James. (2004) Real-time glow. [Online]. Available: http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html

[28] D. Filion, "Principles and practice of screen space ambient occlusion," in *Game Programming Gems 8*, A. Lake, Ed. Cengage Learning, 2010, pp. 12–31.

[29] M. Pharr *et al.*, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[30] F. Liu *et al.*, "Multi-layer screen-space ambient occlusion using hybrid sampling," in *Proceedings of the 12th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, ser. VRCAI '13. New York, NY, USA: ACM, 2013, pp. 71–76. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/2534329.2534335

[31] M. Mittring, "Finding next gen: Cryengine 2," in *ACM SIGGRAPH 2007 Courses*, ser. SIGGRAPH '07. New York, NY, USA: ACM, 2007, pp. 97–121. [Online]. Available: http://doi.acm.org/10.1145/1281500.1281671

[32] T. Ritschel *et al.*, "Approximating dynamic global illumination in image space," in *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, ser. I3D '09. New York, NY, USA: ACM, 2009, pp. 75–82. [Online]. Available: http://doi.acm.org/10.1145/1507149.1507161

[33] L. Bavoil *et al.*, "Multi-layer dual-resolution screen-space ambient occlusion," in *SIGGRAPH 2009: Talks*, ser. SIGGRAPH '09. New York, NY, USA: ACM, 2009, pp. 45:1–45:1. [Online]. Available: http://doi.acm.org/10.1145/1597990.1598035

[34] M. McGuire, "Ambient occlusion volumes," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 12:1–12:1. [Online]. Available: http://doi.acm.org.proxy.lib.chalmers.se/10.1145/1730804.1730984

[35] J. M. Méndez. (2010) A simple and practical approach to ssao. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753

[36] K. Mitchell. (2007) Volumetric light scattering as a post-process. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html

[37] N. L. Max, "Atmospheric illumination and shadows," in *SIGGRAPH '86 Proceedings of the 13th annual conference on Computer graphics and interactive techniques.* New York, NY, USA: ACM, August 1986, pp. 117–124.

[38] J. Mitchell. (2004) Light shafts: Rendering shadows in participating media. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/Mitchell_LightShafts.pdf

[39] Y. Dobashi *et al.*, "Interactive rendering of atmospheric scattering effects using graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 2002, pp. 99–.

[40] Cyberphysics. (2001) Shadows. [Online]. Available: http://www.cyberphysics.co.uk/topics/light/shadow/shadow.htm

[41] H. Yen Kwoon. (2002) The theory of stencil shadow volumes. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/the-theory-of-stencil-shadow-volumes-r1873

[42] Microsoft. (2013) Common techniques to improve shadow depth maps. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ee416324%28v=vs.85%29.aspx

[43] L. Williams, "Casting curved shadows on curved surfaces," *SIGGRAPH Comput. Graph.*, vol. 12, no. 3, pp. 270–274, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/965139.807402

[44] W. T. Reeves, "Particle systems - a technique for modelling a class of fuzzy objects," *ACM Transactions on Graphics (TOG)*, vol. 2, no. 2, April 1983.