CHALMERS
UNIVERSITY OF TECHNOLOGY

Johannebergsrallyt

Development of a realistic 3D rally game using open-source tools and libraries

Bachelor of Science Thesis in Computer Science and Engineering

Sofia Edström, Erik Fägerlind, Erik Lundholm,
Victor Sandell, Joel Severin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

The Authors grants to Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Authors warrants that they are the authors to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law. The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors has signed a copyright agreement with a third party regarding the Work, the Authors warrants hereby that they have obtained any necessary permission from this third party to let Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Johannebergsrallyt
Development of a realistic 3D rally game using open-source tools and libraries

Sofia Edström,
Erik Fägerlind,
Erik Lundholm,
Victor Sandell,
Joel Severin.

© Sofia Edström, June 2015.
© Erik Fägerlind, June 2015.
© Erik Lundholm, June 2015.
© Victor Sandell, June 2015.
© Joel Severin, June 2015.

Examiner: Arne Linde

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

Cover:
Shows a rendered frame depicting a Ford GT40 parked on the roof of the Matematiska Vetenskaper-building at Campus Johanneberg, Chalmers University of Technology.
Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Abstract

Independent developers seek success by developing near AAA-grade game using limited time and development resources. In this thesis, we explore some of the different aspects these developers face. In a small team, with a short time frame, we developed a networked rally game in 3D, focusing on real-time computer graphics. We investigate how in-house developed features and already available open-source software could be used in achieving this goal, aiming for real-time performance on the mid-end consumer hardware of today (2015).

It became clear that, given the limited development resources, only a handful of the features and graphical effects an AAA-grade game features could be supported. The effects that were successfully implemented include car enamel reflections, skid marks, particle systems, bloom, motion blur, shadows and screen-space ambient occlusion (SSAO). Additionally, a basic platform for rally car physics and networked multi-player support was developed.

Sammanfattning

Oberoende utvecklare söker framgång genom att utveckla närmast AAA-kvalitativa spel med begränsad tid och utvecklingsresurser. I denna kandidatrapport utforskar vi några av de olika problem dessa utvecklare möter. Med ett litet team och en kort tidsram utvecklade vi ett nätverksbaserat rallyspel i 3D, med fokus på realtidsdatorgrafik. Vi undersöker hur egenutvecklade funktioner och redan existerande mjukvara baserad på öppen källkod kan användas för att uppnå målet, hela tiden med strävan efter att uppnå realtidsprestanda på den genomsnittliga konsumenthårdvaran idag (2015).

På grund av de begränsade utvecklingsresurserna kunde bara en handfull av de funktioner och grafiska effekter som AAA-kvalitativa spel tillhandahåller stödjas. Effekterna som framgångsrikt implementerades inkluderar bland andra reflektioner i billack, bromsspår, partikelsystem, bloom, rörelseoskärpa, skuggor och screen-space ambient occlusion (SSAO). Dessutom utvecklades en grundläggande plattform för rallybilsfysik och nätverksbaserat flerspelarstöd.

Contents

1 Introduction **1**

1.1 Purpose 1
1.2 Problem Statement 2
1.3 Limitations 2
1.4 Outline 3

2 The Real-Time Computer Graphics Techniques **4**

2.1 Meshes 5
2.2 Graphics Pipeline 6
2.2.1 Application Stage 6
2.2.2 Geometry Stage 6
2.2.3 Rasterization Stage 7
2.2.4 Shaders 7
2.3 Shading 8
2.4 Textures 8
2.4.1 UV-Mapping 9
2.4.2 Normal-Mapping 9
2.4.3 Transparency 10
2.4.4 Pixel Format 10
2.4.5 Render-to-Texture 10
2.4.6 Results and Discussion 10
2.5 Skid Marks 11
2.5.1 Generating Skid Marks 11
2.5.2 Result 12
2.5.3 Discussion 12
2.6 Particles 13
2.6.1 Particle Systems 13
2.6.2 Result 13
2.6.3 Discussion 14
2.7 Reflections 14
2.7.1 Simulating Reflections 14
2.7.2 Environment Mapping 15

2.7.3 Environment Mapping Techniques . . . 15
2.7.4 Result and Discussion . . . 15
2.8 Shadows . . . 16
2.8.1 Simulating Shadows . . . 16
2.8.2 Shadow Mapping . . . 17
2.8.3 Shadow Volumes . . . 17
2.8.4 Choosing a Shadow Technique . . . 18
2.8.5 Implementing Shadow Mapping . . . 18
2.8.6 Results . . . 19
2.8.7 Discussion . . . 19
2.9 Geometry Buffer . . . 20
2.9.1 Previous Work . . . 20
2.9.2 Results . . . 21
2.9.3 Discussion . . . 22
2.10 Bloom . . . 22
2.10.1 Method . . . 23
2.10.2 Gaussian Blur . . . 24
2.10.3 Result . . . 25
2.10.4 Discussion . . . 26
2.11 Screen-Space Ambient Occlusion . . . 27
2.11.1 Previous Work . . . 27
2.11.2 Result . . . 30
2.11.3 Discussion . . . 33
2.12 Motion Blur . . . 34
2.12.1 Previous Work . . . 34
2.12.2 Result . . . 35
2.12.3 Discussion . . . 37

3 The Physics and Networking **38**
3.1 Existing Open-Source Real-Time Dynamic Body Physics Libraries . . . 39
3.2 Networking . . . 39
3.3 Combining Networking With Physics . . . 39
3.3.1 Results . . . 40
3.3.2 Future Work . . . 42

4 General Results **43**

5 Conclusion **44**

Chapter 1

Introduction

Historically, 3D game creation has to a great extent been conducted using commercial development tools, which are often subjects to licensing fees and platform limitations. However, in recent years, the computer gaming market has been flooded by an 'indie wave' of independently developed games. Independent developers successfully sell millions of copies of their games, utilizing digital distribution systems. The time restrictions of the independent developers have increased usage of already existing tools and components. With the narrow financial resources of small game developers, it is preferable if the tools used are free, without future financial commitments such as a percentage of the game's net profit. Open-source software is one solution to the financial problem, motivating our choice for further investigation of how it can be used in a game.

1.1 Purpose

The purpose of this bachelor's thesis is to explore the possibilities and restrictions that arise during development of a modern 3D rally game, using limited time and developer resources. Our first priority is achieving high graphical fidelity, targeting real-time performance on the mid-end consumer hardware of today (2015). Enhancing the graphical user experience, we also focus on gameplay, implementing real-time car physics and networked multi-player support. Considering research about the limited development resources, we narrow our purpose to consistently use open-source software tools and libraries.

1.2 Problem Statement

Subject to the limitations of

- real-time performance,
- limited development resources, and
- limited budget,

mentioned in Purpose, this thesis seek to answer the following research questions:

- Which techniques provide suitable visual fidelity, taking the aforementioned limitations into account, considering implementation of the following computer graphics effects in a 3D rally game: bump mapping, transparent materials, car enamel reflections, skid marks, particles (smoke), shadows, lighting (shading), bloom, screen-space ambient occlusion (SSAO), and motion blur?
- Enhancing the computer graphics effects mentioned above, what is a suitable solution when physics and networking support is implemented in a 3D rally game, taking user experience and the aforementioned limitations into account?
- Supporting the above questions, which free and open-source software can be used for 3D modelling, real-time graphics rendering, and physics simulation support?

1.3 Limitations

To be able to focus on creating a racing game that is as graphically pleasing as possible, we have restricted the project to the development of a smaller, yet well designed game. This limitation is the motivation behind modelling a closed track and not an open world scene. We are also making restrictions regarding the amount of graphical features to implement, in order to manage within the short time frame. Another common feature in racing games we chose not to support is computer controlled opponents. Such a feature would require resources that were better spent implementing graphical features and effects.

A racing game requires fairly few animations, which leads us to the omission of such features in our game. A contributing factor to this decision is that realistic animations, despite not being technically advanced to implement, is a very time consuming task in terms of creative processes. Thus, we also omit people and animals from the 3D world, and therefore we automatically avoid any eventual ethical dilemmas if racing cars accidentally, or on purpose, would harm living creatures.

From the large number of real-time rendering techniques and effects available, we chose to support bump mapping, transparent materials, car enamel reflections, skid marks, particles (smoke), shadows, bloom, SSAO, and motion blur. These were the ones we

thought would best enhance the aesthetics of the game. Pre-computed static global illumination (baked lighting), used in most modern AAA-games, is one notable technique the project time constraints did not allow implementation of.

1.4 Outline

This thesis is organized into four main parts: the introduction, the computer graphics chapter, the physics and networking chapter, and the final chapters about general results and conclusions. As described in the purpose section of this thesis, the computer graphics chapter is by far the largest and most detailed one. The first part of it introduces relevant computer graphics concepts, after which some more advanced rendering techniques used in our game are presented. A brief discussion on the physics and networking implementation follows. Finally, the thesis is completed with general results and conclusions.

Chapter 2

The Real-Time Computer Graphics Techniques

In order to simplify the development process, we elected to model a real-world location we had immediate access to, and chose Campus Johanneberg of Chalmers University of Technology. Creating all of the models and textures as a part of the project also allowed for everything to fit into the same aesthetic theme. Given the thesis' purpose, models should appear as realistic as possible, while still being able to be rendered in real-time. Relevant techniques were compared, evaluated and, in cases where our timeframe allowed, implemented.

In order to focus on the actual graphical effects, we chose to use an existing rendering engine, with support for features such as resource loading and scene management. The most notable actively developed open-source alternatives are Irrlicht, OpenSceneGraph, and Ogre3D, all of which contain the basic helper features we deemed necessary. However, Ogre3D was chosen, due to previous experience with it within the group. It also has a large user base and active community, which is an important aspect when choosing an open-source project. Ogre3D is an object-oriented cross-platform 3D graphics engine written in C++, which provided a suitable level of abstraction and flexibility [1]. Version 1.9 of Ogre3D was chosen, as it was the latest stable release at the time of implementation.

A unique feature in our game is that some of the graphical effects implemented are modelled in proportion to the speed of the racing car controlled by the player. In practice, this means that some of the effects will be more visible or more active the faster the player is driving. Based on the speed, an effect factor between 0.0 and 1.0 is calculated, and then used to compute the intensity of a certain graphics effect. The features affected by this factor are the reflections, bloom, SSAO, and motion blur. The motivation behind the implementation of this feature is that, while still maintaining a realistic-looking racing game, we want to make the graphical effects as striking and

noticeable as possible. This way, the gaming experience is visually accentuated based on the player's driving style during a race.

This chapter explores techniques used for implementation of shading, textures, skid marks, particles, reflections, and shadows. Finally, the post-processing pipeline is discussed, enabling the effects bloom, SSAO and motion blur.

2.1 Meshes

In this section, the concept of polygon meshes used in modern 3D games is introduced. Different stages in constructing these meshes will be described.

A polygon mesh is a list of vertices describing an object in three-dimensional space [2]. In modern 3D graphics applications, this is performed by listing a set of triangular faces, each containing three connected vertices. Using triangles exploits the fact that computation of these primitive shapes are accelerated in modern graphics hardware.

Many programs for generating polygon meshes exists today, among them are Maya, 3ds Max, Blender, SketchUp and MeshLab. Most of them, such as Maya, 3ds Max and Blender, are multi-purpose programs. In addition to creating meshes, they are used for lighting, animation and game logic. Most of them are also neither open source nor free to use, which are properties that were specifically sought. Blender is one of the more notable exceptions, with roughly the same features as the most complex commercial products, but fully open source. This complexity has the disadvantage of a steep learning curve for the user. Therefore, we chose to combine it with the very accessible, but simple, SketchUp, which is free to use.

At first, relatively accurate measurements were acquired for the objects that were going to be rendered. This was done using maps, photos and models. Using SketchUp, the objects could be drawn efficiently. Figure 2.1 show a car modelled in SketchUp. The drawn meshes were then imported into Blender for further work, such as adding textures and material properties. This method was proven to be effective, and a car could be relatively accurately modelled in a few hours. The use of materials is visualized in Figure 2.2.

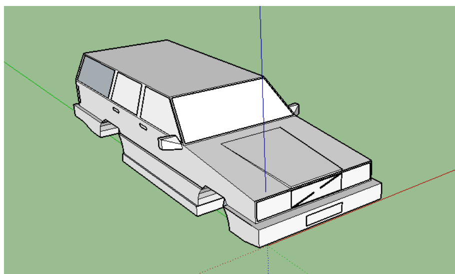

Figure 2.1: Car model drawn in SketchUp.

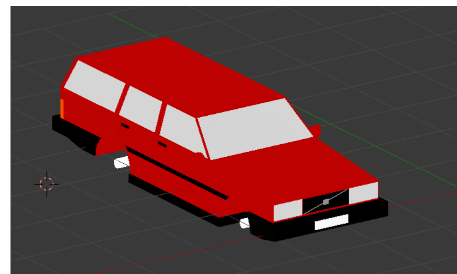

Figure 2.2: Car model with applied materials in Blender.

2.2 Graphics Pipeline

The graphics pipeline, also referred to as the rendering pipeline, is a concept with the purpose of rendering a 2D image of a 3D scene onto the screen [3]. This means capturing the various geometric objects and textures in the scene, including light sources and other effects that affect them, using a virtual camera, and then drawing the resulting image onto the screen. This task is performed in three different stages: the application stage, the geometry stage, and the rasterization stage. The idea of going through these stages sequentially is called forward rendering. This section will describe these stages, visualized by Figure 2.3.

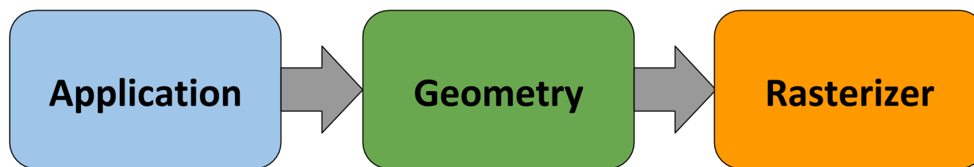

Figure 2.3: The three stages of the graphics pipeline.

2.2.1 Application Stage

The first step is called the application stage. This stage is executed on the CPU (central processing unit) and not on the GPU (graphical processing unit). The graphics hardware is more locked-down in terms of what developers can program it to do. Thus, since this stage is executed on the CPU, this stage in the pipeline is where the developers are given the most freedom. Different parts of a game, such as animation, collision detection and physics, are aspects of the part of the pipeline that are implemented and calculated during the application stage. The freedom of implementation in the application stage allows developers to use algorithms that yield the best performance and desired result for their applications.

2.2.2 Geometry Stage

The next step in the graphics pipeline is the geometry stage, in which all of the per vertex operations are performed on the model in the scene and their meshes. What, how and where objects are supposed to be drawn is also computed in the geometry stage. One of the aspects that differ from the application stage is that all calculations are run on the GPU. The GPU is designed to compute a lot of simple calculations (such as moving a game object using matrix transformations) in a small amount of time. GPUs are far more efficient than CPUs for the process described, which is the reason dedicated graphics hardware exists. There are methods to alter and adjust how these calculations are performed with the use of shaders, which will be discussed further down in the shader section.

During the geometry stage, several transformations - where objects are moved to the correct space - are performed. Initially, all objects are located in model space. An object, such as a car, is transformed from model space into world space, then into view space and finally projected onto the screen. World space is the space with all of the models transformed into it, and represents the game world. This is where rotations, scaling and translations are performed. View space is the scene in relation to the virtual camera's perspective. The last transformation needed is the projection from view space onto the screen. The transformations described are calculated using a model-view-projection-matrix, which is a multiplicative combination of the model, view, and projection matrix, transforming coordinates between the aforementioned spaces.

The next step of the geometry stage is screen mapping, which is the process of adjusting the scene to be rendered to fit the window that it is to be displayed on. When rendering the scene, it is important that objects outside the scene are not rendered, in order to avoid unnecessary computation, improving performance. Clipping is a method for excluding objects that are not in the viewing frame from being rendered.

2.2.3 Rasterization Stage

The third and final stage is the rasterization stage, where the color of each pixel on the screen is decided. When all geometry calculations are complete, the result needs to be converted into pixels with the correct color on the screen. A process called rasterization begins. All the triangles in the scene are traversed in order to determine which pixels they are covering. Per-pixel shading computations, if any, are executed in this stage, using the interpolated color data from the vertices. The results of the per-pixel shading are stored in the color buffer, which stores the colors of each pixel as a combination of red, green and blue. Another aspect that affects how the scene will look is visibility. The depth buffer is used to determine if an object is to be rendered or not by comparing the distance to the virtual camera. If the distance to a pixel that is about to be rendered is smaller than what is in the depth buffer, the color and current depth at that position is changed. The color buffer is in the end what is displayed on the screen.

2.2.4 Shaders

A way to alter and make the process of rendering to the screen more flexible is with the use of shaders. A shader is a small computer program that is executed on the graphics hardware. There are two basic types of shaders: the vertex shader and the pixel shader (also known as the fragment shader).

The vertex shader is run once for every vertex. It is possible to adjust the position, lighting, texture and texture coordinates with the use of the vertex shader. A vertex shader receives the attributes of a vertex as input and outputs new vertex data that has been modified by the shader.

The pixel shader is run for each pixel. It is in the pixel shader that the color of the pixel is determined and output. It is possible to change how a pixel should look in relation to other pixels, by sampling surrounding pixels using screen coordinates, if the entire scene is sent into the shader. Thus, it can be used to create post-processing effects. A pixel shader can also be used for achieving other effects, such as filtering for blurring or similar.

2.3 Shading

Shading is the process of computing what color each pixel on the screen should have. This is performed by calculating how light interacts with objects in the scene, which translates to solving a rendering equation. Kajiya's rendering equation is an integral equation, describing the amount of light emitted from a surface in a particular direction, given a function of incoming light and a bidirectional reflectance distribution function (capturing how light is reflected off the surface) [4]. It is the prevalent physical model which all realistic rendering algorithms solve. There exists a multiple of different shading algorithms for solving this equation, or in the case of real-time rendering, approximating it. The most noteworthy approximation algorithms with real-time performance are Blinn-Phong, Cook-Torrance, Phong and Lambert. We briefly explain these methods, and then present our choice, followed by a suggestion of future work in our game in this area.

The Phong model was proposed by Bui Tuong Phong in 1973 as an efficient approximation of the rendering equation, splitting the solution into three parts: an ambient part (not hit by direct lightning), a diffuse part (large highlights on dull surfaces) and a specular part (full reflectance of light) [5]. The Lambert model can be used for modelling the diffuse part, and Cook-Torrence was proposed as a way of taking into account that different materials reflects wavelengths of the color spectrum in different ways [6]. Blinn-Phong is a variant of the Phong model used for a more computationally efficient rendering [7].

We choose to use the Phong model since it is simple, efficient and supported in Ogre3D. In the future, we suggest that some research is devoted to how realistic lighting of asphalt can be achieved.

2.4 Textures

Different texturing methods are used for adding detail and realism to the 3D models in our game. Texturing is an umbrella term, covering various techniques of taking a surface and changing its appearance with color, images, and functions. A simple image or color attached to a 3D model will not add as much detail as preferred, because there will be no surface irregularities or specific material characteristics, which would have been the case if the model was a real world object. Texture mapping is a collection

of techniques which can be utilised in combination to achieve such a result. It is also possible to manipulate the look of surfaces in real-time, by off-screen rendering to a texture for usage in image-filters and post-processing effects. The following section will briefly describe the different methods used for texturing, ending in a section discussing the results of them.

2.4.1 UV-Mapping

To be able to apply an image to a 3D model, it is necessary to first make a UV-map of the model [8]. A UV-map, where UV denotes the axes of a 2D-texture, is created by unfolding the faces of the mesh at its seams onto a flat surface. A texture can be seen in Figure 2.4. The map is then used to fit a suitable texture, and to apply the color from it to the model. The texture shown in Figure 2.4 applied onto a model is visualized in Figure 2.5. The unfolding step is also called unwrapping, and can be performed either manually or automatically, or with a combination of both, in 3D modelling software such as the open-source tool Blender.

Figure 2.4: Image containing a texture.

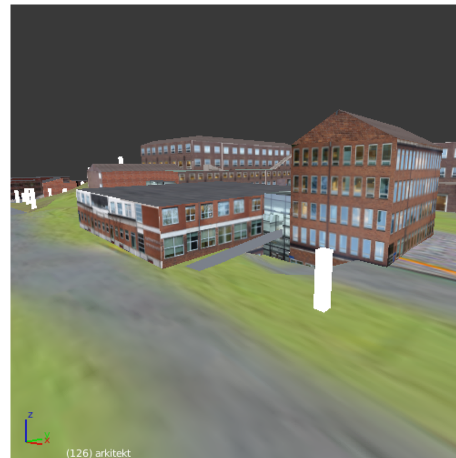

Figure 2.5: Texture applied to house using UV-mapping.

2.4.2 Normal-Mapping

Texture maps are flat and cannot show any realistic surface details. Therefore, another method is needed for creating realistic surface geometry, such as cracks and height differences. Instead of modelling all the irregularities with polygons, which would be very computationally expensive in real-time, there are ways to mimic uneven surfaces [3]. Bump mapping is used as a method for simulating the effects of detailed geometry in meshes without adding actual geometry. A common way to implement bump mapping is with normal mapping. Normal mapping rearranges the surface normals of the 3D models face's, with the normals of a more elaborate model. The new normals are then used when calculating light and shadows, which makes the model look as if its structure is more complex than it actually is.

2.4.3 Transparency

The main techniques for achieving transparency in 3D-rendering is image masking, key mapping and alpha mapping. These are all very similar. A value is assigned to each pixel, stored in either its color values (image mapping and key mapping), or in a separate channel (alpha mapping). The value denotes how much of the pixel's color values will be blended with what is behind the transparent material.

2.4.4 Pixel Format

Textures are a way to store pixel data as an image. The pixels can be stored in various formats depending on how the data is supposed to be interpreted by the program using the texture. An example of how pixel data can be stored is with a 32-bit array of integers, where 24 bits represent a pixel's RGB-values (8 bits per color channel), and the last 8 bits store the alpha channel [9]. Sometimes, the alpha channel is left out, and a 24-bit array is used instead. In cases when higher precision and larger range is needed (as in the geometry buffer, discussed in its own section), for example 16-bit floating-point numbers per channel can be used.

2.4.5 Render-to-Texture

Not only the frame buffer, which stores color data on its way to the display, can be a target when rendering all, or part, of a 3D scene [10]. Other possible render targets are textures, which can be off-screen rendered in real-time, and used in later frames, just like any offline-rendered texture, in the program. This method is called Render-to-texture, and is a very important feature in real-time rendering, because it enables implementation of effects like dynamic real-time reflections, shadows, bloom, motion blur and SSAO.

In addition to supporting render-to-texture, Ogre3D also supports multiple render targets (MRTs), enabling rendering to several targets at the same time, depending on how many targets the GPU supports.

2.4.6 Results and Discussion

Based on the level of realism and detail in relation to the low cost, we decided to use high resolution photographs as textures for all the buildings in the game. In some cases, such as textures for rounded meshes, asphalt and other terrain-related surfaces, photographs could not be utilised for visually pleasing results, and instead artificial textures were created. Regarding transparency in textures, as seen utilised in Figure 2.6, we chose alpha mapping, since it is the de-facto format used in modern software and hardware.

Figure 2.6: Transparency in windows.

Normal mapping was also implemented, aiming for more accentuated surface irregularities. The alternative to said combination, highly detailed meshes, would not have been applicable in real-time due to its computational cost. However, for future work, we propose further polishing of the normal-mapping technique.

2.5 Skid Marks

Skid marks are a key feature in any game where a car is driven, since they provide visual feedback regarding the traction regarding the player's car. Visual feedback is an essential feature in any game. The user needs to be able to identify a certain behavior in the game with visual or other types of feedback. When a car in our game start sliding out of control, it produces a trail on the ground. Apart from providing visual feedback, skid marks also add realism to the game. The trail left under the tires of the car, gives a clear indication to the player that the car, or rather each specific tire, has started to lose its traction.

2.5.1 Generating Skid Marks

Billboards can be used to represent the trail that is left behind the car. Billboards are 2D objects that always face the viewer [11]. All of the billboards are attributed the same material and properties when created. However, these properties, such as transparency, width and height can be altered in order to obtain the desired result, which in this case is skid marks.

Another method for simulating skid marks in a game is using decals. A decal is an additional layer, or texture, that is created on top of existing textures and meshes, changing the color. These decals are mapped on a surface of a polygon or shape. The alpha value of the decal texture is modified to match the surface shape of the object that it is to be attached to. Alpha mapping the image causes it to be blended onto the

surface. This can be used for effects such as blood spatter or a poster on a wall, but also to create skid marks on the ground.

2.5.2 Result

In this project, billboards were chosen as the method to use for the implementation of skid marks. This method was chosen since billboards in Ogre3D are well documented and developed, and implementing skid marks this way is less time consuming than using decals. The use of billboards yielded a satisfactory result. One modification that had to be made is that the billboards that are created in this game are not always facing the viewer. The direction of the billboards is adjusted to the normal of the tire which it is created under to fit the terrain. Billboards are generated using a billboard set in Ogre3D [12]. The created billboards' positions are also determined by what the billboard set is attached to, which in this case is a wheel.

Billboards are created in short intervals and some optimizations and improvements are needed to increase the performance and improve how the trail looks. The car's speed affects the length of each billboard, in order to optimize the number of billboards that needs to be created and rendered into the scene. Another factor that affects how the skid marks look is the traction of each wheel, in regard to how visible a skid mark billboard appears. Figure 2.7 show how lower traction results in more visible and darker skid marks. Higher traction on the other hand, as seen in Figure 2.8, means that the car is sliding less, which in return will make the skid marks less visible. The alpha value of the material used for the skid marks is altered to achieve the desired result.

Figure 2.7: Dark skid marks, a result of the car's tires having low traction.

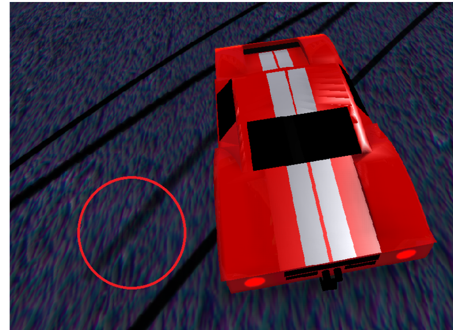

Figure 2.8: A lighter, more transparent skid mark, which is caused by more traction from the tire.

2.5.3 Discussion

The current skid marks do not take the terrain into account. Different terrain properties does not affect how the skid marks look. Neither does the terrain contribute to if skid marks should be created or not. We suggest improvement of these areas as future work.

2.6 Particles

Particles can be used for a wide range of graphical effects. These effects include dust or dirt that flies off a car's tires when it is braking or losing traction. Particles can also be used to simulate other effects, such as fire, rain, and explosions. By combining these various effects, a more realistic graphical outcome can be achieved. This section covers how particles can be used in a game.

2.6.1 Particle Systems

Particles are created using particle systems, in which all particles are controlled in order to create a desired effect. All particles created by a particle system are bound to that system, which means their lifetime is managed entirely within that system [13].

Particles are created by an emitter belonging to a particle system. The emitters' positions of a system define the positions where the particles are created. Certain attributes are given to emitters, such as the emission rate and the size of particles. Initial force can also be added to particles, in order to make them move away from the source. Gravity can also be added to make the particles behave more physically accurate. Particles that are generated from a particle system can sometimes look too similar and some variety in the emitted particles is often good. By randomizing the properties of particles' when they are created, this effect can be achieved. The time that the particles are alive, or the angle at which the particles are launched at, are some of the aspects that can be altered by randomizing the particle system state management.

Particles are often created with the use of billboards. Only two-dimensional textures are needed to visualise the appearance of particles. This method is very efficient in terms of performance. Another method of simulating particles is rendering them as points or lines on the screen. Particles can also be implemented using vertex shaders [14].

2.6.2 Result

Particles in our game are created using the Ogre3D particle system. The existing particle system within Ogre3D provides all of the needed functionality. It was an easy decision to chose the existing system, considering implementing a particle system can be difficult and take a long time. The use of particles was only required for one effect, dust. The car, in combination with the traction of the tires, creates dust or dirt. Thus, the amount of particles emitted per second is adjusted in proportion to the traction of the tire that the particles are originating from. The texture used for the dust particles is an image with different shades of grey with added transparency.

The dust is created when the car loses traction and starts sliding. Dust is produced under the center of the tires' positions at varying angles and launched upwards with gravity affecting them. Figure 2.9 shows added transparency to the particles. Created

particles disappear after a short amount of time. The gravitational force is slightly reduced in order to emulate the effect of air resistance.

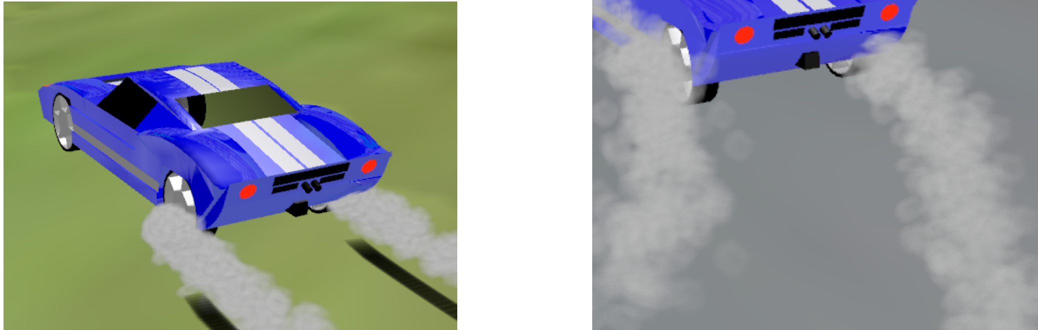

Figure 2.9: Dust created beneath the car's tires that gradually fades away.

2.6.3 Discussion

The use of particles was limited only to dirt and dust. There are other effects, besides dust, that requires the use of particles, but no such effects were chosen to be added to the game. Particles that depend on terrain is a feature we suggest as a candidate for future work. Taking that into account would create a more realistic feeling when collisions happen. The materials that the car collides with would create metal or sparks flying off the car.

2.7 Reflections

Reflections are an important component in making a 3D scene looking realistic, and can contribute substantially to the visual gaming experience. The following section covers a brief comparison of two different techniques for accomplishing reflections, and a more detailed description of the chosen method, environment mapping. The decisions taken are motivated and the result of reflections in our game is presented.

2.7.1 Simulating Reflections

Reflections in 3D games can be achieved with different methods. For reflections on curved surfaces, two common techniques are environment mapping, or reflection mapping, and ray tracing. Environment mapping is a technique that projects the render target's distant environment onto its surface using a precomputed texture, sometimes created in real-time. The environment mapping technique is more efficient than the more physically correct ray tracing approach, since tracing the path of every light source to calculate an image can be very computationally expensive, as the level of detail in the environment increases. Environment mapping on the other hand, is not as accurate as ray tracing, but provides nice substitution with pleasing visual results. For calculating reflections on planar surfaces the method differs. Often the reflection on flat objects is rendered as a copy of an object mirrored in the plane.

2.7.2 Environment Mapping

Blinn and Newell first introduced environment maps in 1976, and they have since then been improved and further developed as progress in graphics hardware techniques has been made [3]. An environment map can be used when the reflection target does not reflect itself, and when the environment being reflected is far away. The map can be precomputed with 3D graphics software, such as Blender, or computed in real-time using the render-to-texture technique. Environment map reflections are generated by loading or creating a 2D texture, representing the reflecting object's surroundings [15]. The normals of the pixels contained within the object's surfaces is then calculated, and used with the view vector to compute the reflection vectors. The reflection vectors are used to calculate an environment map index, which in turn is used to color the corresponding texture pixel.

2.7.3 Environment Mapping Techniques

The environment maps can be represented with various methods [3]. The two most common of them is cube mapping and sphere mapping. The first introduced with support in commercial graphics hardware was sphere mapping. A sphere map is created using the texture based on the image of the environment as reflected in a perfect sphere. Since the reflected environment is basically represented by a picture of a sphere, the reflection is only valid from one single viewpoint. A new map can be calculated when changing view direction, but can cause visual errors.

The other common approach to environment mapping, cube mapping, was first introduced by N. Greene in 1986, and is the most frequently used method in real-time rendering. Cube mapping, as opposed to sphere mapping, is independent of the viewpoint and does not cause visual artifacts. A cube map is generated by creating textures of how the environment is projected onto a cube, when placed in the center of a scene [16]. The reflection is then rendered one time for each side of the cube, with the camera centered in the cube and a view angle of 90 degrees from the surface.

2.7.4 Result and Discussion

The environment map solution with its superiority in computational efficiency in comparison to ray tracing, stored as a cube map, was the approach used in the game to simulate reflection. The motivation behind the decision of using cube maps was that it is the most commonly used solution for modern real-time applications. When creating reflections with an environment map, it is possible to make them dynamic by constantly recalculating the map as the target of reflection is moving. This can be a computationally expensive feature compared to having a static pre-computed map, but since the racing car is a moving object, the reflections can not look realistic if they are not changing with the car movement. Based on this, and the fact that the car is the only moving object in the game subject to reflection, we chose the dynamic alternative.

The result of the final implementation is presented in Figure 2.10. Additionally, only one side of the cube is updated per frame, spreading the workload over six frames in order to increase performance. For future work, we suggest rotating the cube so that only three sides are visible at all times, possibly doubling performance.

Figure 2.10: Tree reflections in car enamel.

2.8 Shadows

Shadows are one of the most important aspects in creating realistic looking images in games. They provide information regarding the whereabouts of the objects in a scene and help the viewer or player get a good visual overview. Shadows helps onlookers perceive the world around them [17]. They provide a sense of depth that helps with interpretation of shapes. Shadows also adds another layer of realism by making the game world appear more like the real world, as it becomes more lifelike. This section will describe various ways shadows are implemented in computer games, and specifically, how they are used in this game. Different techniques for simulating shadows will be described, such as shadow mapping and shadow volumes, followed by a discussion comparing the two. The implementation of the technique chosen to be used for this project, shadow mapping, will be further described.

2.8.1 Simulating Shadows

In order for shadows to improve the visual experience, they often need to be of high quality and look as realistic as possible. Two techniques for creating highly realistic shadows are photon mapping and ray-tracing. The techniques generate shadows by

simulating individual photons and light rays. The more rays simulated, the better and more lifelike the shadow will look. However, the downside of these methods is that they are not feasible in a real-time context since they require an unreasonably large amount of computational power, and thus time - both of which are crucial components in real time computing. Therefore, when dealing with real-time rendering, other methods for creating shadows are used that provide better performance, but with the trade-off that they appear less realistic. Almost all shadow techniques are based on the most common techniques, shadow mapping and shadow volumes. This section will look at these two techniques, and specifically techniques based on shadow mapping.

2.8.2 Shadow Mapping

The concept of shadow mapping was introduced in 1978 by Williams [18]. It is a widely used method for simulating shadows in real time computer games. Creating shadows with this method is performed in two steps. The first step is to create a depth map from the light source. This depth map is created by rendering the scene from the light source's point of view. To store the shadow map, the depth buffer is used. The next step is to check if a pixel is in shadow or not. This is done by comparing depth values from the shadow map to the scene being rendered from the viewer's perspective. The position of each pixel in the shadow map is compared to the corresponding position in the image to be rendered from the viewer. If the distance to the light source is greater than the depth value stored in the shadow map at that position, that particular point is occluded from the light source, and is thereby in shadow.

The shadows created when using the standard shadow mapping can look somewhat crude, but various add-on techniques (which are discussed later in this section) can improve the appearance. It is the simplicity of the process (and result) that makes shadow mapping a very attractive method with regards to performance. Modern hardware can run shadow mapping algorithms very fast.

One of the downsides of using shadow mapping is that the resolution of the shadow map determines the quality of the shadow. A shadow map with too low resolution can result in aliasing artifacts, which are distortions in the pattern. Increasing the resolution of the generated shadow map decreases the aliasing artifacts, but at the cost of performance. Another disadvantage of using shadow maps is that they create hard edges. Both of these issues, as well as solutions to them, are discussed further later in this section.

2.8.3 Shadow Volumes

Shadow volumes is another method for simulating shadows which was introduced as a viable option by Heidmann in 1991 [19]. Silhouette edges are the edges created by a 3D object that is visible from the light source projected onto a 2D-plane. The silhouette edges are extended to infinity, forming a volume that extends across the entire scene.

Some implementations require that a back or front cap is added to create a closed volume. The first step in using the shadow volumes when rendering the scene is to first render the scene as if fully covered in shadows. Then for each light source, using the stencil buffer, a mask is created with holes in it using the depth information stored in the shadow volumes. These holes represent the locations at which the scene is lit and thereby not in shadow. The last step is to render the scene once again, but this time it is rendered fully lit. With help of the stored mask in the stencil buffer, the portions of the scene to be covered in shadows are not lit. There are several methods that are used for implementing shadow volumes. The different methods for creation of the volumes are fairly similar. Although, the step that varies the most regarding the implementation is the creation of the masks.

2.8.4 Choosing a Shadow Technique

There are multiple factors at play when deciding what shadow technique to select. The most prevalent are: appearance, performance, and implementation complexity. In terms of appearance, shadow volumes yields the best results. In comparison, shadows generated using shadow maps can look relatively simple, harder, and less detailed than shadows generated using shadow volumes. It is the difference in appearance that makes the use of shadow maps faster in terms of performance. Shadows using shadow volumes look better, at the cost of computational power and time. The third aspect, implementation complexity, is a bit different than the previous two. Appearance and performance are running time related, while implementation complexity is related to time as a project resource instead of a computer resource. Shadow mapping is easier and faster to implement than shadow volumes due to how the previously described algorithms behave. This became the most decisive aspect when choosing what technique to use. Shadow mapping is more suitable for our game, as performance is prioritized over visual fidelity, along with implementation time being a precious resource.

2.8.5 Implementing Shadow Mapping

The tradeoff with using shadow mapping is the lower visual fidelity, which can be somewhat mitigated using various build-on techniques, which will be described here. Another shortcoming is that shadows created using shadow mapping can suffer from bias issues.

Depending on the angle between the occluding object and the light source, there might be a risk for encountering surface acne, that is, some parts are shadowed and some are not. This effect looks bad, but can be solved by adding an offset, a bias, to the compared depth in the shadow map, thus removing the uncertainty in the comparison [20].

Another problem that can occur with shadow mapping is that if the camera is closer to the object casting a shadow than the light source that is causing the shadow, the

shadow generated will have very rough edges and be of very poor quality. This is caused by the limited resolution of the shadow map itself. One way to mitigate this problem, is implementing a technique called cascaded shadow mapping, which uses multiple shadow maps [21].

Cascaded shadow mapping yields much better looking shadows, without increasing the performance cost significantly. The basic idea is to use multiple shadow maps with different size (and therefore quality of shadows). On top of that, the frustum, which is the region of the screen that is visible to the camera, is divided into the multiple parts, depending on the distance to the camera. Objects close to the camera cast a shadow generated using the the shadow map with the highest resolution and shadows for objects further away are generated using shadow maps with lower resolutions. This results in good looking shadows close to the camera and lower quality shadows far away, degrading with the distance from the camera.

2.8.6 Results

Shadow mapping was chosen to simulate shadows in our game, the result can be seen in Figure 2.11. The reason behind choosing shadow mapping was made in regards to it being easier to implement, while retaining the satisfactory quality needed, at a lower performance cost compared to shadow volumes. An implementation of cascaded shadow mapping was undertaken, but later suspended due to the project time constraints.

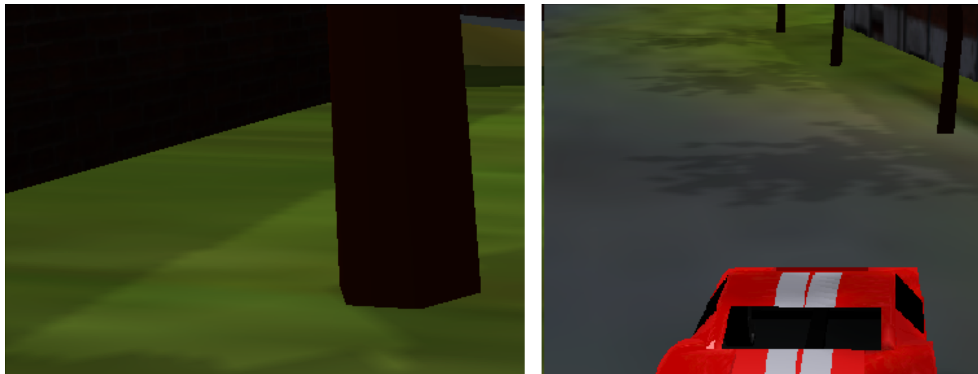

Figure 2.11: A wall and trees casting shadows onto the ground.

2.8.7 Discussion

Even though cascaded shadow mapping was never implemented, the shadows currently in the game are of satisfactory quality. Implementing cascaded shadow mapping would not only make the shadows look better, but also likely help with performance. However, the current shadow maps is not an issue in terms of performance. Thus, the current shadow implementation is considered viable for the current world size in the game.

2.9 Geometry Buffer

Using a geometry buffer (also known as a G-buffer) might improve rendering performance and is essential for many post-processing effects that our game use, most importantly SSAO, which is discussed in its own chapter. Firstly, the previous work in the area is presented, and then our implementation is discussed. Finally, thoughts for improvement of our implementation are expressed. The geometry buffer will be the basis for the next following chapters that explain various post-processing effects that are applied to the (almost) final render of the original scene.

2.9.1 Previous Work

When using forward rendering, all fragments are shaded as they are drawn, which essentially leads to a mapping of their final color into a texture. Drawing the scene multiple times, each time accounting for one light source, the scene colors are additively accumulated one light at a time. Instead, the fragment's properties may be drawn into a texture [22][23]. By associating each drawn pixel by the occupying fragment's material properties, normal and other relevant information, and then applying the shading calculations in a second pass, only fragment parts that are visible in the final render need to be shaded. The technique is known as deferred shading. Figure 2.12 shows an example texture with normals to the geometry in our game.

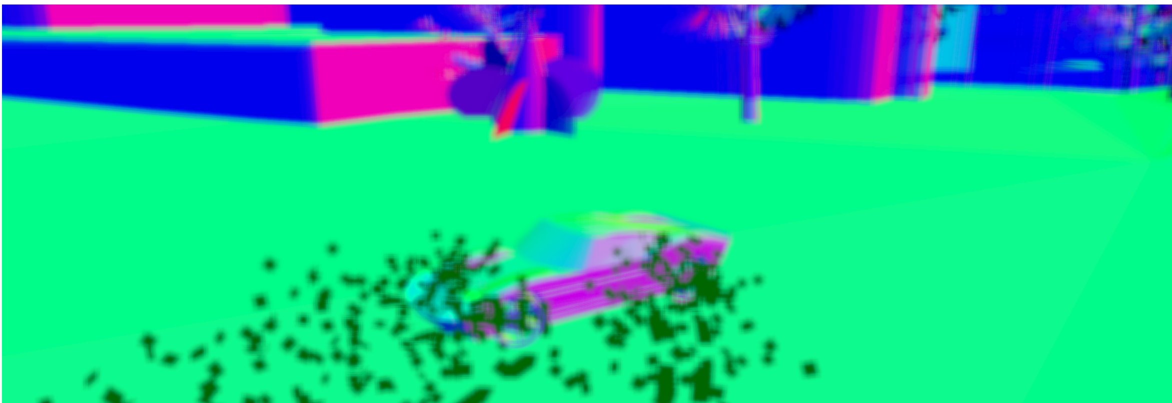

Figure 2.12: A normal texture of the scene, used for deferred rendering. The colors used for representation have no actual meaning as colors, instead they encode normal coordinates.

While deferred shading might improve performance, there are some implementation issues. Some of these are addressed by split techniques such as inferred lighting and light pre-pass rendering [24]. The main systematic problem with deferred rendering and light pre-pass rendering lies in their inability to handle transparency gracefully; when drawing transparent objects in the scene, rendering has to resort to forward rendering as a post-pass after drawing and shading all opaque objects in the scene. Also, with deferred shading, the in modern hardware commonly used anti-aliasing technique MSAA does not work properly. The problem is circumvented by the split techniques,

as with them, the scene is drawn three times, where the last pass makes use of such anti-aliasing in a correct manner. With inferred lighting, the lighting draw pass may also be performed at a lower resolution, yielding better performance but more artifacts.

Which rendering technique is the best one is a controversial topic and also depends on the type of scene rendered. Two factors that are often mentioned are bandwidth and number of light sources. As deferred shading produces more data than what can be stored in a single texture on modern hardware, a MRT is used. The MRT allows multiple textures as targets for a single draw command. On earlier hardware, MRT support was not always available, and if it were, storing multiple textures for a G-buffer required high memory usage, compared to what was available for that hardware generation [24]. Van de Hoef and Zalmstra made some actual performance tests on modern hardware, which in all test cases show that deferred shading is faster than light pre-pass [25]. For few light sources and few vertices, forward rendering is significantly faster than the other algorithms. For more vertices and/or more lights, deferred shading is twice as fast. The only case in which inferred lighting is faster than deferred shading, is in scenes with many vertices and many light sources (300+). It should be noted that the difference measured is marginal at even than many number of lights, and that these lights cannot affect shadow casting in the scene.

2.9.2 Results

As the game is set in an outdoor environment without many more lights than the sun, the player's car lights and possibly other players' car lights, deferred shading seems to be the best fit when considering performance. However, an implementation proved to be non-straight forward in Ogre3D 1.9, as it is designed to support the fixed-function graphics pipeline for maximal compatibility with old hardware. Some effort was put into investigating how lighting for deferred shading could be implemented, but as shadows (otherwise handled by Ogre3D) had to be accounted for in the implementation too, the progress in this direction was abandoned, given the time constraints we had.

In order to support other post-processing techniques, such as SSAO and to some degree bloom, rendering of fragment information other than color still needs to be performed. The current solution uses forward rendering for lighting calculations, as managed by Ogre3D. The scene is then drawn in a separate pass, where geometry information is output into an MRT with two textures. Table 2.1 lists the attributes stored in our game. The first texture stores the world-space normal represented on the pixel, while the second texture stores world-space position and the emissive intensity of the object. The format and interpretation of these components are discussed where they are used, in the sections about SSAO and bloom, respectively.

Notably, the textures used in an MRT must have the same pixel format on most hardware [26]. In our current implementation, we use the 16-bit floating-point 4-vector format FLOAT16_RGBA for both textures. Lower resolutions produce significant ar-

Table 2.1: Geometry buffer storage allocation in our game.

	r	**g**	**b**	**a**
MRT 1	*position.x*	*position.y*	*position.z*	*emissive*
MRT 2	*normal.x*	*normal.y*	*normal.z*	(empty)

tifacts all over the screen with the current SSAO implementation. Another limitation is that Ogre3D 1.9 hide the depth buffer as an abstraction. Therefore, it is not possible to access a texture of it on hardware that support such depth-buffer textures, without manipulation of the Ogre3D source code. As the other attributes would still need to be draw into an MRT, we chose to include the position there too. We motivate this choice by the project time constraints and hardware compatibility.

2.9.3 Discussion

We think the next logical step in the development of the game is implementing deferred shading properly. Given some more time, it could greatly improve performance and open up the opportunity of having many light sources in the scene, for example on the car. Also, if new post-processing effects that need more data stored in the geometry buffer are developed, lowering the pixel-format resolution/range in order to reduce memory-bandwidth usage is an important performance improvement worth further investigation.

2.10 Bloom

In the human eye, as well as in photographs taken by a camera, intense light sources seems to be surrounded by a blurry glow known as bloom [27]. There is also a sparkling effect known as flare. The umbrella term for these phenomena is glare. Even though glare is mostly visible at night, it can be seen around really bright objects in daylight too. Car lights are an excellent example of that, which motivates proper handling of bloom in our rally game. Here, we present our findings about how the bloom effect could be included in the rendering process, in order to enhance the perceived realism and thus improve the gaming experience. There is no rational decision behind only supporting bloom, but not flare, other than the project's time constraints. Our subjective judgement decided, and bloom was slightly preferred visually.

The color buffers on modern hardware have a limited range in what brightness values can be represented, which in turn is correlated to how bright pixels modern monitor technology is capable of producing. Not even the brightest white (full brightness for the red, green and blue components that make up a pixel) is able to induce noticeable bloom in the human eye. When lighting calculations produce pixel color values brighter than what can be possibly stored in the color buffer, the value is clamped to the maximum value that can be represented.

Adding bloom with software for an already rendered scene with saturated brightnesses, as a post-processing effect, is a procedure in which the brightest parts of the scene are identified and extracted, in order to add an artificial bloom halo [27]. The naïve technique is to assume that the brightest parts in the scene were clamped as a result of being too bright, and then blindly apply a uniform amount of bloom to them. An improvement to the aforementioned technique, is masking out the parts that are assumed to be glowing beforehand, and then only applying bloom to them [28]. Masked bloom is also commonly referred to as glow, as it gives the game artist more control to specify exactly which parts should be glowing at all times. However, the pre-determined nature of this technique may also be a drawback, as the bloom no longer is truly dynamic.

A solution to the problem of saturated brightness, is sacrificing memory and memory bandwidth, storing brightness values with a higher range, still maintaining the same precision [29]. The range of brightness values that can be represented is known as the dynamic range of the image format. Rendering pipelines which can handle the larger-than-usual brightness range are commonly referred to as having a high dynamic range (HDR). The benefit of implementing such a pipeline would be that a technique known as tone mapping could be applied post-rendering. Tone mapping greatly enhances how a scene is quantized to the low dynamic range format displayed on the monitor, mimicking the human eye in its process [30]. The quantization process involves an analysis of local and global brightness levels of the whole scene when it has finished rendering, implying that the scene must have been rendered to a HDR pixel format first.

2.10.1 Method

Regardless of method used for bloom identification and extraction, the limited brightness range modern monitors are afflicted with imply that in order to show bloom to the user it has to be simulated. It should be noted that users might also prefer simulated bloom on saturated brightness, as glare produced by partially over-bright monitors might cause discomfort [31]. This section covers a description of how simulated bloom might be synthesized for display on a medium with only a limited brightness range, such as today's monitors.

To summarize the whole process: after the brightest parts in the image have been identified and extracted, the selection is blurred and then added back to the scene again, brightening it even further. This way, the pixels that otherwise would have been saturated in brightness, also smear some of their light intensity on nearby pixels. This simulates what occurs in the human eye, being similar to the light halo seen around bright objects. In our method, however, we are not considering the relative brightnesses in the scene. The human vision's ability to better differentiate between darker colors than lighter ones [32] alleviates the problem, as our game is set in a uniformly daylight-lit environment.

When the brightest pixels have been selected by rendering them into a render-target texture, they are blurred in software, combining and spreading them out over a larger area. On the rasterized image, blurring is achieved by, for each texel (texture pixel) in the texture, combining its color with the average color of all neighbouring texels within a certain radius. We focus on two suitable algorithms that can be efficiently implemented targeting the GPU, allowing real-time performance: box blur and Gaussian blur.

Box blur combines (interpolates) the color of each texel with its neighbors, weighting each contributing pixel equally with respect to how much it influences the final result. Using box blur with a small radius yields a small and non-natural looking blur, while using it with a larger radius yields an even more non-natural looking result with a very strong blurriness.

Gaussian blur produces a blur more natural than box blur, comparable with what can be found in nature. Applying box blur with a small radius repeatedly, incrementally building the blur, approximates Gaussian blur with high accuracy in just a few iterations [33]. However, the interpolation might be performed in just one pass, exploiting the fact that the interpolation weights for Gaussian blur can be pre-computed independently of the scene's colors.

Having a blur filter was important for other effects than bloom in the game, such as SSAO and motion blur. An important detail that had to be accounted for was that the filter had to adapt dynamically to the current effect factor. In order to meet the time constraints of the project, one filter solution that could be used everywhere was developed. The choice of using Gaussian blur was motivated by its high quality and the easy integration of the effect factor it allows for. A discussion on how the Gaussian blur filter might be implemented follows. In particular, the effect factor requirement as well as the general real-time constraints of the project is considered.

2.10.2 Gaussian Blur

In detail, the first step of Gaussian blur is taking a texel and its neighbours, and passing their coordinates (with the currently active texel as origin) to a Gaussian function [3]. The Gaussian function is the density function of the normal distribution, which is used to determine a weight matrix for the sampled texels. The weights are used to decide how much the current texel is going to be affected by the color of its neighbours. In the next step the texel values in the sample are multiplied by its corresponding weights. Lastly, the average of the neighbour products replaces the current texels value. The result will be one blurred texel, which because of the normal distributed weights will be more affected by texels close to it than the ones farther away. For a completely blurred texture, this method is applied for all the texels in it.

However, the Gaussian function will produce a non-zero result for all the neighbouring texels, independent on how far away from the origin texel they are located. This means that to blur merely one texel, the weighted average of all the texels in the entire texture

is calculated. Since the weights far away from the origin will be very close to zero, this will result in a lot of unnecessary computations considering the visibility of the blurring effect. It is therefore beneficial to decide on a sample radius, and thus leave out all the weights too close to zero. Another aspect of this method is the conversion of the Gaussian continuous values to the discrete ones needed for the weight matrix. When converting the values, the sum of them will not equal 1, and therefore brightness in the texture will be lost during blurring. This is not important when the blur is used for the purpose of creating bloom, but it is important when used in other contexts, such as SSAO and motion blur later on. To account for this loss, the values can be normalized by dividing the values in the weight matrix by the total sum of the weights.

A very useful characteristic of the Gaussian filter is that it is separable [28]. This implies that a two dimensional Gaussian blur will produce equal results as calculating two separate one dimensional blurs. The one dimensional filtering requires less computational power, which is desirable in real-time rendering.

2.10.3 Result

This section present the result of implementation and optimization of bloom in our game, taking the effect factor into account. As was previously mentioned when the method was established, our final implementation uses masked bloom together with a vertical-horizontal separated Gaussian blur kernel.

To begin with, results from the game with the blur identification method based on the assumption that bright pixels probably was calculated as even brighter, but later saturated to fit into the dynamic range of the texture pixel format. Figure 2.13 shows the problem where objects with bright colors that should not cause bloom erroneously have been bloomed. Only the car lights should have been bloomed, not the whole car body or the banner.

Figure 2.13: Naïve bloom identification leads to undesirable bloom progression.

Figure 2.14 shows bloom applied to the scene based on the masked bloom identification method, featuring a correctly applied halo around the lights. Notice how the car body does not cast any bloom on the surrounding pixels. The magnitude of the bloom is

based on the current effect factor. Finally, Figure 2.14 shows how bloom is applied to the sky, causing the brightest clouds to brighten up even further. The sky also bleed some of its color to the surroundings.

Figure 2.14: Only the car's light receive the bloom effect. The magnitude depends on the current effect factor. The left car drives faster and therefore have a higher effect factor, increasing the bloom's spread.

Figure 2.15: Bloom in the sky.

2.10.4 Discussion

Given the time we had to implement and test the bloom, we think the bloom works well. There are no noticable bugs, but the bloom identification could be improved. With more time to invest in development, we propose a full HDR graphics pipeline as the next step, considering the opportunities it unlocks in other areas too, such as tone mapping.

2.11 Screen-Space Ambient Occlusion

The approaches used in order to approximate the rendering equation so far do not provide full global illumination, where light that has illuminated one surface bounces off and illuminates other surfaces as well. The solution used has been to evaluate whether a particular part of an object is lit by a light, and then illuminate it accordingly, disregarding the appearance of neighboring geometry that should have been influenced by light bouncing off the surface in real life. Ambient-occlusion techniques seek to account for nearby occluders, obscuring incoming light, dampening the effect of ambient lighting as modelled by the Phong model. In order to increase the perceived photorealism in the game, ambient occlusion was added to the rendering engine. The following discussion will focus on real-time friendly alternatives that approximate the ambient occlusion. Finally, our choice, combining several of the techniques, is presented and motivated.

2.11.1 Previous Work

Approximating ambient lighting and ambient occlusion is not only used for real-time purposes. It is also used in order to speed up off-line rendering such as ray tracing, path tracing and similar techniques [34]. These techniques still use rays for finding the amount of occlusion but omit color calculations in the occlusion-finding pass. However, even just finding occlusion values for every point in the rendered part of the scene, including shooting rays to geometry outside the camera's view, becomes too computationally heavy for real-time applications. The limits of real-time on today's consumer hardware makes inclusion of geometry that cannot be seen directly by the camera impractical, hampering photorealistic accuracy.

The screen-space variant of ambient occlusion, SSAO, refers to a specific class of algorithms that only use the (almost) final render of the scene, darkening the edges around small holes, creases and objects located in near proximity to each other. One of the first recognized algorithms, Crytek SSAO, uses the depth buffer in its hunt for nearby occluders [35]. This is an improvement of an earlier method, instead using surface normals projected into screen space for crease detection [36]. By comparing the depth of each pixel in the render with the depth of pixels representing nearby points, the amount of ambient occlusion is determined. To keep the number of samples within the limits of acceptable real-time performance, Crytek SSAO samples pixels representing randomly selected points bounded by a sphere, centered around the pixel to be shadowed.

Even though sample-point positions are statically randomized inside the sphere, the low sample count and the fact that the depth buffer is usable only for some of these points creates banding artifacts (stepwise changes in color, rather than smooth color gradients). As sampled points not in direct view of the camera are not rendered into the depth buffer, the number of valid samples might become extremely low in some cases, yielding artifacts. Likewise, extremely sharp viewing angles also produce artifacts, as the sample points (in world space) all correspond to the same pixel in view

space. To mitigate the effect of both the banding and extreme condition pixels, the sample points are moved in a random fashion dependent on the pixel's position in the render. The random movement is done with a small kernel, usually $4x4$ in size, yielding high-frequency ripples in the formed occlusion texture. The noise is then reduced by averaging (blurring) the occlusion values with its neighbors, at least such that the $4x4$ noise is smoothed out. The simple box filter, or even Gaussian blur, might be used for that purpose.

A cross-bilateral filter may also be used in order to reduce the low-frequency noise significantly, while still keeping the crease edges sharp; an example being Gaussian blur with screen-space pixel distance as the first weight parameter, combined with camera-space scene depth as the second weight parameter [37]. Even though not truly separable into horizontal and vertical passes, like Gaussian blur is, approximating the result with two separate blur passes in each direction works well for the aforementioned filter configuration.

Improving upon Crytek SSAO, the sphere may be replaced with a normal-oriented hemisphere, such that the amount of occlusion is calculated purely based on occluders on the front-facing side of the (surface) triangle. Since geometry behind the opaque surface should not contribute to any ambient occlusion, this method more closely captures the ambient occlusion model. However, the surface normal is needed in order orient the hemisphere in the right direction. Storing and reading the surface normal might in turn be taxing on performance. We have not found who the original inventor of this technique is, but it is described in detail by John Chapman [38], and also part of the Ogre3D SSAO Demo [39].

Taking the normal-oriented hemisphere method further, Horizon-Based Ambient Occlusion (HBAO) treats the depth buffer as a continuous height-field and ray-marches, along the randomized sample direction, in order to find the most occluding piece of geometry [40]. Each time the ray hits geometry, the trajectory is adjusted to approach the surface normal, in order to establish a highest point of horizon. The angle between the highest point of horizon and the surface plane determines the rate of occlusion. In order to reduce over-occlusion, a cause of banding artefacts, the surface plane is lifted from the true surface by a certain bias distance [37]. Additionally, the individual samples' contribution are weighted by their distance r from the original pixel, for enhanced smoothness. A radial attenuation function

$$a(r) = 1 - r^2 \tag{2.1}$$

is used for the falloff calculation. A similar approach is taken by the volumetric ambient occlusion technique, where the definition of occlusion is changed to better match the assumed geometric volume that occluded parts of a scene often are accessible through, in real-world conditions [41]. The assumptions are based on a fuzzy logic classification scheme.

Finally, ending the discussion on suitable real-time screen-space ambient occlusion techniques, volume-based ambient occlusion and summed-area table ambient occlusion is mentioned. Volume-based ambient occlusion ray-marches in a pre-computed 3D texture for occlusion estimation of a particular area, covering surface detail more precisely [42]. Summed-area table (SAT) ambient occlusion relies on a pre-process stage of the geometry buffer, using the GPU to calculate a SAT representation of the surface [43]. In order to store the accumulated values, a pixel format with high precision and range is needed. The benefit of the method is that only the four corner pixels of a rectangle covering the area need to be sampled. Additionally, no blurring pass is needed, as the summed area table generation not only smoothes out noise, but also ensures no banding artefacts occur as otherwise associated with the low sampling frequency of previous methods.

When the most suitable general method of achieving SSAO have been selected, there are a few additional implementation details that can be changed in order to increase either performance or visual fidelity. Before an introduction of the quality-enhancing techniques available, some methods of reducing memory bandwidth are presented. Whether these methods successfully increase performance is highly dependent on limiting factors in the hardware and scene being rendered. During evaluation, it is worth critically keeping in mind that modern hardware excel at reading memory from consecutive memory locations, utilising cache prefetching. Also, seemingly reading a single attribute from a texture texel might in the background correspond to reading the all attributes, discarding the ones not requested, due to hardware constraints.

Complicating the analysis even further, pixel shaders (threads) run in parallel on the GPU, shading a number of pixels at the same time [44]. But, their parallel nature is entirely dependent on all conditional branches (e.g. if-statements) all taking the same path. Diverging branches lead to suspension of one of the branches while the other is executed, downgrading the execution scheme to a serial one. It is probable that nearby pixels are shaded at the same time in the same warp, where a warp refers to a collection of threads executing simultaneously. Thus, taking decisions in a shader (e.g. exiting early) is probably only beneficial for large areas in the render where the same decision is taken. Different hardware behave differently in this regard.

Instead of storing each pixel's world-space position as a 3-vector in the geometry buffer, it is possible to reconstruct it from the pixel's normalized depth, position relative to the screen boundaries and camera properties [45]. The camera position, orientation, field-of-view, near- and far clipping planes (used to clip the render distance within a far and near interval) of the current frame all need to be accounted for. The technique uses an expensive [44] trigonometric function for determining the position, but may in some implementations save enough bandwidth to be worth the effort of implementation. Instead of reconstructing the position fully to a 3-vector, referencing it to the camera position and orientation, it is possible to keep the obtained value normalized and in screen space. The obtained value represents more than a single position; it represents

the range of camera-space positions the pixel had captured at the time of rasterization. Having a relationship between camera-space and screen-space position admits pixel-by-pixel traversal of the depth texture. If the neighboring samples all correspond to the same pixel, there is not enough information to decide whether the pixel is occluded. The SSAO shader is in this circumstance allowed to exit early, possibly increasing performance.

As an alternative to storing surface normals as 3-vectors, the per-pixel normals may be approximated using camera-space positions and dFdx/dFdy instructions, as described in [37]. The dFdx/dFdy instructions correspond to the partial derivative of any shader variable (in this case camera-space position), with respect to either the horizontal or vertical view-space coordinate [46]. Using the pure face normal, instead of an averaged one for the whole triangle, gives more correct results in many cases, exploiting the fact that ambient occlusion occurs at sharp edges. For correctly modelled geometry, where triangle edges coincide with sharp edges, the pure triangle normal is a better representation than the normal used for interpolation, which is an average of the whole triangle's face normal. However, as the distance from the edge increases, the more incorrect the reconstructed normal might be.

Yet another possibility is exploiting temporal and spatial coherence for recently drawn geometry [47], leading to Temporal SSAO (TSSAO), described by Mattausch et al.[48] By reusing the ambient occlusion map from the previous frame, only re-projecting it with regards to the current camera position and orientation, a dynamic solution where not every part of the scene needs SSAO-recalculations is created.

Being limited to the parts of the scene that is visible, the frame buffer might be extended in order to allow for increased information storage. One way of achieving that is rendering the scene with different depths, a technique known as depth peeling [49]. For each depth layer, the scene is drawn with decreased depth culling. Another possible way to include more information at the edges of the screen, is (maybe not surprisingly) increasing the viewport size. Rendering the SSAO at half resolution, still performing the blur passes at full resolution, can sometimes provide nearly four times shorter frame times. While flat surfaces with no ambient occlusion are not affected, the appearance of high-frequency geometry degrade visually. Applying a second full-resolution pass, targeting only areas of the render which need further refinement, as described by Bavoil and Sainz[49], is a middle-way in terms of performance and quality.

2.11.2 Result

Deciding between Crytek SSAO, normal-oriented hemisphere SSAO, HBAO, volume-based SSAO and SAT-based SSAO, we made our choice primarily based on the performance and aesthetics, as reported in detail by previous research, referred to in the previous section. Volume-based SSAO was discarded as too demanding on memory for the visual fidelity we desired. SAT-based SSAO had the benefit replacing a high num-

ber of samples with a few static ones. However, the number of samples per shadowed pixel we are constrained with on today's hardware (16 in our case) roughly amounts to the static number of samples SAT-based SSAO uses. With a requirement of storing occlusion data in a larger data-type format, there is no gain in performance, and thus, SAT-based SSAO was discarded as a technique of the future. Conversely, while Crytek SSAO is fast, it does not meet our visual quality standards.

With the choice left to HBAO and normal-oriented hemisphere SSAO, we note that both meet our performance and visual standards, where we prefer HBAO slightly. Both techniques can be configured to look very different based on hemisphere radius. The fact that two techniques look different does not necessarily mean that one is better than the other. We resorted to the main project goals and noted that HBAO had a more complicated implementation, while the complexity of a normal-oriented hemisphere SSAO implementation paired well with our prior experience and time constraints. Additionally, we use the distance-falloff weighting from HBAO, in order to smoothly transition between occluded and unoccluded areas.

For noise-reduction filtering, our already developed Gaussian-blur shader solution was chosen. Again, the choice was based on the project time constraints. While a cross-bilateral filter, combining Gaussian blur and screen-space depth, could have produces a smoother result, the results with just Gaussian blur look good enough for us. This concludes the broad method we used for ambient occlusion approximation. A screenshot from our game is presented in Figure 2.16, where the results of SSAO is visible as shadowing around the back tires and along the rear axle. There is also some occlusion between rear lights and spoiler. The crease along the wall to the right is also occluding itself. Figure 2.17 shows how the SSAO affects the bonnet of another of our car types.

Figure 2.16: Our normal-oriented hemisphere SSAO implementation. Occlusion is seen as shadows around the crease created when the car and wall intersects with the ground.

Figure 2.17: Ambient occlusion present on the bonnet of the right car, where it covers three of the sides around the striped middle sheet of metal.

We tried the method where the normal is reconstructed by dFdx/dFdy of the world-space position (again, dFdx/dFdy calculates the partial derivative in screen-space). The dFdx/dFdy instructions exploit how neighboring pixels are calculated in the same warp, making them a fast replacement for reading the normal from a texture. As the normal was stored in its own texture, previously only used by the SSAO shader, the geometry buffer could be reduced to one texture. In practice, no measurable performance gain resulted from this change. Even though only the position texture was read from in the shader, compared to the previous shader that read both the position and the normal texture, the major bottleneck lies in the repeated position reads each occlusion sample contributes with. It was further revealed that removing one texture from the geometry buffer did not lower the frame time, contrary to what was previously thought. For the actual shadows, the dFdx/dFdy-method did not change the appearance or quality. However, on flat walls, there were some subtle disturbances visible only when the camera was moving. For future development, when the geometry buffer becomes full, these disturbances might be acceptable or even possible to remove. Until then, the dFdx/dFdy-method was removed.

The method of reconstructing position from depth and camera configuration was not chosen. Firstly, some tests revealed that the exit early-strategy did not work well: Ogre3D's normal-oriented hemisphere SSAO demo was modified to show areas where the shader would exit early. Some thin lines were observed, but considering how nearby pixels' fragment shader threads are scheduled together into a warp as described earlier, concern was raised whether there was any performance gain to be found at all. The SSAO solution was implemented based on these observations. Afterwards, the experiment was repeated, and it was revealed that the background resulted in early shader exit. As the scene had no skydome (a dome placed around the game world to represent the sky), the depth of the sky would correspond to the camera's far clipping plane, causing the early exit. However, our game has a skydome not placed far enough to trigger early exit. Instead, we tried masking away the skydome manually from the geometry buffer, but it ruined the bloom and motion blur effects. Considering the project time constraints, investing further research into a better masking solution could not be motivated.

Finishing the discussion on position reconstruction from a single depth value, the current layout of the geometry buffer need to be considered (Table 2.2). Shrinking position from three to one element, there are still five elements that need to be stored. Given hardware constraints, two textures with four elements each are still needed, leaving three slots empty instead of one. Trying to shrink the number of slots per texture to three is also without merit: such pixel formats (PF_FLOAT16_RGB) are unavailable on some of our target hardware. If they are available, it is possible that the hardware pads the data with an extra element anyway, in order to adapt the pixel-format to the internal memory width that is a power of two.

Table 2.2: The current layout of the geometry buffer.

	r	**g**	**b**	**a**
MRT 1	*position.x*	*position.y*	*position.z*	*emissive*
MRT 2	*normal.x*	*normal.y*	*normal.z*	(empty)

Given that the camera often moves forward in a rally game, essentially diving deeper into the scene, TSSAO seems like a good candidate for implementation. Reprojecting the SSAO texture from the last frame will often become an upscale in our case. A fast moving car imply that the upscaling happens quickly. The implementation by Mattausch et al. indicate that this might work badly for the visual fidelity we seek, as the rate of convergence need to be set high in this case. For a rally game, maybe the dynamic refinement pass method proposed by Bavoil and Sainz work better. Considering the project time constraints, we did not feel confident enough to try, a venture possibly ending in a badly looking SSAO with horrible performance.

2.11.3 Discussion

We investigated the possible options in the field of real-time ambient occlusion algorithms, and came to the conclusion that normal-oriented hemisphere SSAO with radial falloff weighting and Gaussian blur was the best solution for our project. The technique was implemented with good results. Implementation was not problem-free though. Most of the problems encountered were based on flawed assumptions about how random entities in the code behaved. As support for future work, these problems are discussed in detail below. Finally, we note that suggestions for future work have already been made, with the most notable one being adoption of HBAO.

In order to generate the high-frequency banding artefact-reducing noise, a simple function of the screen-space position was used as a pseudo-random generator. However, the probability distribution of this function was not uniform, leading to slightly more occlusion on the left side of every object. When generating the static random-vectors inside the hemisphere, this phenomenon was also observed. Sometimes, especially when using few samples (16 in our case), the random values were noticeably bad. We found that trial-and-error worked well when obtaining random-vectors: they should ideally represent a Monte-carlo integration of the hemisphere volume, meaning that there should

be a good balance between the vectors. Also, having too short vectors or having two vectors that are too close to each other is a waste of performance, as they will reference the texel to be occluded or the same texel repeatedly.

2.12 Motion Blur

In order to enhance the tunnel vision that humans experience when moving fast, such as when driving a rally car, motion blur was added to the game. The implementation focused on conveying the tunnel vision effect in real-time, rather than being accurate. Perfect motion blur would account for scene appearance between frames, but given that keeping the frame rate high enough even for the display frequency of around 60 Hz is troublesome, such intermediate computations cannot be justified for real-time applications. The solution available is interpolation between current and previous frame. This section explores some of the most popular interpolation techniques. Finally, our choice of technique, and the results we obtained in our game is presented.

2.12.1 Previous Work

Potmesil and Chakravarty noted that motion blur was caused by object movement during the exposure time in a camera, as well as the camera shutter's movement during start and stop of an exposure [50]. They chose the first cause of motion blur, object movement, and created a model for it. We note that the human eye, which we want to model to some degree, behaves similar to a camera but does not have any shutter that can cause motion blur. Potmesil and Chakravarty used ray tracing software for rendering scenes with different parameters. While not a real-time approach, there are still some interesting points in their research that can be applied to real-time applications. Firstly, they link motion blur to the velocity, not only of objects, but also the camera. They also show that moving objects with motion blur look like a blurred extrusion of an object. Lastly, they show that if the rendering is split into multiple exposures, the blur is still there, but without extrusion. Instead of looking extruded, the object looks duplicated.

Rosado noted that motion blur not only increases realism in games, it also help in hiding lag when the frame rate is low [51]. However, adding motion blur might in itself slow down rendering. Some research suggest that motion blur does not enhance the player experience in a high speed racing game [52]. Some users might also find the effect distracting. While our game is a rally game, with lower speeds on squiggling roads, the research might still apply. Other prior research in the ergonomics of scene blurring include the depth of field effect added to a virtual environment by Hillaire et al [53]. The depth of field effect is similar to motion blur but does vary with depth instead of velocity, rendering objects out of focus as blurred. It was found that some users described the blur as more fun, realistic and/or even addicting, while others found it annoying, discomforting or even a cause of headache.

Instead of blending the last frame with the current one, essentially applying a time variant low-pass filter of each pixel's individual color, other properties of what object is drawn at the pixel may be used to indirectly interpolate between frames. Using the per-pixel velocity, calculated as a difference between current and last frame world-space position, blurring of the current frame can be performed with individual blur strength on each pixel [51]. The important detail with this technique is that the last frame only indirectly participates in the final result, through the calculation of pixel velocity. The blur is calculated only in the current frame, by averaging in the pixel velocity direction. Instead of storing positions, the pixel depth of the current and previous scene can be used in conjunction with an inverse of its corresponding camera projection matrix, in order to reconstruct position and finally velocity. This is the same technique already investigated in the SSAO chapter, but this time applied in another context.

Not every object is in favor of motion blur. For example, the player's car might be left sharp by masking away motion blur for it, while the rest of the scene is blurred [54]. This technique is used in the car racing game Split/Second, published by Disney Interactive Studios. Our observations of motion blur in the racing game Burnout Paradise, published by Electronic Arts, suggest that the motion blur is also adapted to where on the screen it appears. As the camera always follows the car, this resembles a velocity weighted depth of field effect.

2.12.2 Result

Motion blur was deemed the least important among the graphical effects we support in our game. As a consequence, it also received the least development time. The first iteration used what was already available: the current and the last scene. A linear blend was performed between them based on the effect factor. Realizing that objects in the scene that were approaching the viewer looked duplicated rather than extruded, due to the lack of color integration between frames, we tried to keep the inner parts of surfaces smooth. Intuitively, instead of extruding objects and blurring them (the result of motion blur), we let the human eye perform the extrusion and blurring but enhanced it with some extra retained motion around object edges. The edge detection is already performed implicitly in the SSAO-shader, when the occlusion is calculated. Instead of throwing away the result, it is combined with the current effect factor and passed together with the amount of occlusion to the Gaussian blur pass. This required almost no development effort, reusing code we already had.

The edge-detection solution mentioned is presented in Figure 2.18. Note how the large wall to the right is clear while its borders are moving, but also how the thin trees look duplicated due to heavy edge detection in that area. No velocity calculations are performed for producing these scenes, just simple alpha blending weighted by how near an edge the painted object part is. Note that the skid marks in front of the car are not related to the current drive, the camera is traveling into the scene contrary to what they suggest.

Figure 2.18: Edge detection enhancement applied to simple motion blur.

There are some cases in which the car or objects in front of it becomes blurred too, where Figure 2.19 stands as an example. In order to remove the blur from these areas, a radial falloff factor was introduced, as shown in Figure 2.20. In our game, the car is always centered on screen (the screenshots presented have been cropped to the area of interest). Exploiting this fact, the radial falloff is computed as the square of the distance to the middle of the screen. The falloff factor was trivial to implement, compared to a masking solution that could exclude exactly the screen area occupied by the player's car. The asphalt under the car will still look blurry when driving fast in the game, but that aspect cannot be captured in print.

Figure 2.19: Edge detection-based motion blur sometimes become too aggressive.

Figure 2.20: Radial falloff applied to the motion blur.

Respecting users who do not want to have motion blur enabled for one or another reason, the effect was made possible opt in and out from. Considering that turning on or off enhanced tunnel vision could affect gameplay, the effect was made easily available for toggle by a button on the keyboard. If disabling the effect would have been hard for the average user, technically skilled users could have used this to their advantage, possibly creating a scenario similar to cheating.

2.12.3 Discussion

It is already mentioned that the motion blur technique we chose was a result of limited development time. While the result look good, it does not have the visual fidelity of a comparable real-time solution that use world-space velocity and blurring. This motivates implementation of such a solution as a suggestion for future work. Maybe the method of edge detection can be combined with world-space velocity blurring. What would result from such solution is outside what we can imagine, maybe the techniques are too different to work well together. If the world-space velocity solution is built on top of the existing one, it becomes trivial to test what the results would be.

Chapter 3

The Physics and Networking

In order to support the gameplay, a solution of car physics was needed. Physics simulations are driven by time, which will become an important topic of the discussion to follow. The kind of simulation used for real-time car physics is split into discrete steps, which all record a snapshot of the simulated world's state. The type of simulation suitable for our project is built on rigid body dynamics, corresponding to simple non-deformable shapes that can receive forces and collide with each other. Keeping the simulation entities simple not only lays the foundation for a fast simulation, supporting our real-time goal, it also simplifies the implementation, keeping our development efforts within the project time frame.

To support multi-player physics in our game, players are allowed to interact with each other in a networked client-server model. (Using a peer-to-peer model was discarded as too time consuming to implement.) Interaction is not limited to seeing each other with the correct car model and color, spinning wheels and skid marks, it also includes proper handling of collisions between players' cars. For correct (non-relativistic) physics simulation results, every part of the simulation must use the same clock as a reference when transitioning from one world snapshot to another. The reason for this is that the different parts of the simulation cannot run independently from each other, as all parts of the simulation could possibly collide, in extreme cases even at the same time and place. Since correct time management is essential to a physics simulation, multi-client solutions present challenges as every client has its own clock that cannot be synchronized to the other clients' clocks instantaneously, due to network latency. In order to deliver a smooth real-time experience to end users, this tight coupling between physics and networking requires a unitized solution.

The project goals dictates that while gameplay should work well, our attention should be focused on the computer graphics area. Therefore, an existing physics library was chosen and integrated into the project. What follows is a discussion on such libraries, with emphasis on real-time alternatives that are open-source, taking our other project goals into consideration. After a presentation of the available physics libraries, the

different alternatives for network management is presented. Finally, our choice is presented together with the results obtained.

3.1 Existing Open-Source Real-Time Dynamic Body Physics Libraries

IBM lists the most commonly used open-source 3D physics engines as: Bullet, Chrono::Engine, Dynamo, Moby, Newton Game Dynamics, Open Dynamics Engine, OpenTissue and Tokamak [55]. Being able to simulate a car is within the basic set of features offered by all of these engines. Some of these libraries have demos with car simulation, while other even include a car component in the core API [56], providing a customized interface for steering, throttle, breaking, speedometer reading etc. One of the libraries that has such a component is Bullet [57].

3.2 Networking

Targeting consumer hardware, the only networking solution of interest is the de-facto communication system users' computers are connected to: the Internet. Assuming that our application needs to communicate over the Internet, the API used in modern operating systems to achieve that is BSD-sockets [58]. As modelled by BSD-sockets, data is transferred between computers in indivisible chunks known as packets. When packets are delivered over the Internet, they can arrive out-of-order, or even not arrive at all. There are two major transfer protocols that might aid developers in this regard: TCP and UDP.

TCP keeps a session open to the remote computer, reordering packets so that they arrive in-order as seen by the application. TCP also issues a re-transmit request to the remote computer if any packet becomes lost, still maintaining the in-order packet delivery guarantee. Lastly, TCP provides congestion control, trying not to flood the network with more packets than it can handle, as that only leads to many lost packets and delays. UDP, on the other hand, provides none of these features. What UDP provides is low overhead and real-time performance, especially since there is no stall in incoming data to the application when a packet becomes lost. However, the developer needs to ensure graceful application behavior for lost or out-of-order packets, as well as provide congestion control.

3.3 Combining Networking With Physics

Combining networking with physics introduces the problem of unsynchronized clocks and latency to the simulation [59]. Due to the non-deterministic nature of a physics simulation, the only option for reliable results in real-time is either running it on only one client, or on a server. One aspect of this choice is trust: the machine running the

simulation could potentially run modified code for cheating or spread malicious code to the other clients. These problems would still be present if the simulation would be put on a trusted server, but the server could at least try to provide an extra layer of security. However, running the simulation on a server not only adds latency between different clients and cars, it also adds latency between the player's client and the simulation of the player's own car. Furthermore, it requires that the server is fast enough to handle not only all clients' connections, but also the physics simulation itself. Finally, if the game should support offline single-player mode, the physics simulation needs to be adapted so that it is able to run on the client as well.

The simulation also needs to run on all clients, at least partially, if any kind of predictive reasoning about the future should be performed. Prediction can be used to momentarily fill in simulation results in the graphical representation of the game, before the results from the server actually arrive [59]. Prediction can greatly enhance the player experience, as it conceals the low update rate from the server caused by congestion. For non-deterministic simulations, such as the one we are faced with, the simulation needs to be rewound and corrected when the true result arrives, should it differ from the predicted one. Another potential problem with prediction is keeping the local prediction model synchronized with the server at all times. The floating-point numbers used in the real-time physics libraries of today has to be carefully accounted for in this regard, as their behavior may differ when deployed to different hardware platforms. Prediction can also be used on the server to reason about what the player is doing, reducing the effects of input latency.

Finally, if players should be able to race against each other, the time of when a race starts needs to be synchronized. By keeping the simulation on the server, all the cars can start at the same time and there is no ambiguity in which car won a race, due to unsynchronized clocks between clients and clock drift [59]. The car racing game Burnout Paradise, published by Electronic Arts, instead uses a clock-synchronization scheme in its peer-to-peer physics model, in order to synchronize clocks as accurately as possible. It should be noted that true clock synchronization is not possible to achieve over the Internet, due to its non-deterministic latency. Thus, there can be multiple clients claiming they won a race in Burnout Paradise. The game tries to solve this conflict by local timing of races, which is much more accurate but still not entirely fair, considering clock inaccuracies (read jitter, and to some degree, drift). Cheating is always possible, but this scheme makes it very easy by simple data tampering, compared to a cheating robot that has to honor game dynamics when a validating server is used.

3.3.1 Results

We chose Bullet as physics library, due to its active development, large user base, and the car simulation component and demo it had. Many of the other libraries have plenty, or all, of these qualities, and might have performed as good as (or even better than) Bullet did. In retrospect, we would have made a different choice. While Bullet had

good documentation explaining the general concepts, the API documentation was non-existent. Also, the car simulation component was littered with memory corruption bugs. Both of these shortcomings led to many development hours spent deep within Bullet's source code.

While the car simulation component provided by Bullet was accurate, it was not user-friendly: when the car tires lost traction of the road due to drifting, the car went into an uncontrolled spin. We solved this by implementation of an automatic drift control system, which corrects the steering so that the front wheels always are referenced to the velocity direction. At first, the system worked too well, at the expense of gameplay. Some low-pass filtering of the steering direction referencing were added to the system, in order to allow users to drift, but still not lose control of the car completely.

For networking, considering the project time constraints and that physics and networking was prioritized lower than the already big area of computer graphics, we chose a hybrid solution. The player's car is simulated locally on the client, but its state is also announced via a server to other connected clients. Apart from managing user sessions and hiding the other player's identity, the server broadcasts incoming player state to all other connected players. Filtering of old packets that have arrived out-of-order is not only done on clients, but also on the server, possibly reducing bandwidth usage. Both the client and server use raw BSD-sockets for communication with the network, sending and receiving UDP packets asynchronously. The data synchronized over the network is car position, orientation, velocity, model/color and wheel traction. The position and velocity is used for client-side prediction of position, and calculation of wheel rotation.

The main reason for choosing the hybrid solution was that network latency was assumed to become much more apparent for the player's own car, rather than for other players' cars. By keeping the player's car model local, we could focus on one simulation, omitting much of the state synchronization between server and client that would otherwise be needed to provide a smooth and stable real-time experience for the user. Single-player offline mode was also made available by this method, without any extra development effort. One problem with client-local physics is that fair multi-player races cannot be hosted, and therefore we decided to only support local racing, with simple timekeeping triggered by driving through start and goal checkpoints.

It was our original intention to disable collision between cars, but, apart from some issues where one car can push another through walls, it worked well enough that we kept it enabled. Figure 3.1 shows such player interaction.

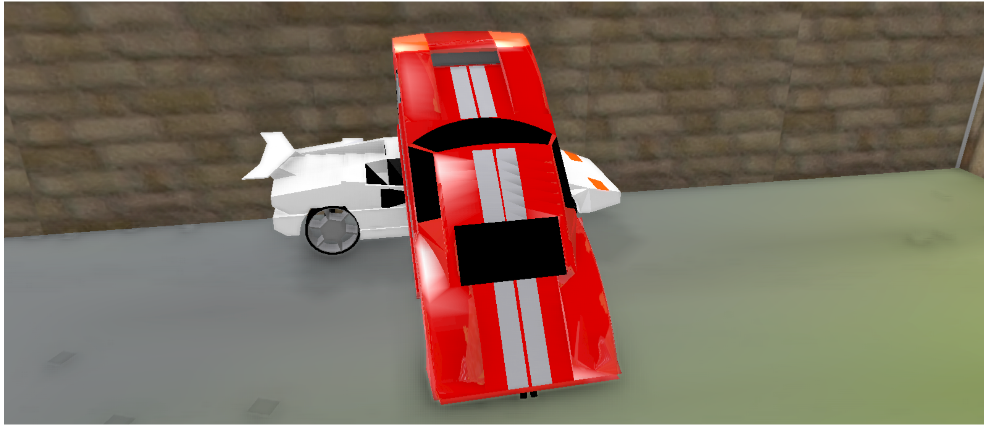

Figure 3.1: Multi-player interaction in our game.

3.3.2 Future Work

Even though the simulation works relatively well, we suggest that the physics simulation should be moved to the server in the future. The main advantages with a server-side simulation is proper handling of car collisions, and the possibility of hosting fair multi-player races.

Chapter 4

General Results

A racing track, which sets the theme of the game, was modelled using the open-source 3D modelling tool Blender. The racing track, a closed circuit within the Johanneberg Campus, allows players to race around the track and record their time. A multi-player solution was also developed, enabling players to interact with others.

Having a car that handles well and is affected by gravity and other forces in a realistic manner improved gameplay. Using the open-source physics library Bullet allowed for the car to behave more realistically and physically accurate. However, an even more user-friendly solution was developed, as the accuracy from Bullet was proven to harm gameplay. Adding a physics library to the game also allowed for easy integration of collision detection and response, enabling players to crash into each other's cars, improving the quality of the multi-player support.

Using the open-source Ogre3D as a graphics framework did provide a range of features that aided the development process, including mesh loading support, particle systems, shadows and custom materials. A number of visual quality-enhancing post-processing effects were developed, improving the overall gameplay experience.

Chapter 5

Conclusion

The purpose of this thesis was to investigate limitations and possibilities in creating a visually appealing multi-player racing game with focus on computer graphics, using open-source tools and libraries. A wide range of graphical effects were considered and examined further. This resulted in a combination of graphical effects that seemingly improve the appearance of the game. Modelling the world, cars, and textures, as well as using open-source tools and free software, enables the game to eventually become a commercially viable product.

Deciding to use Ogre3D as the graphics engine was proven to be a valuable decision. Using it allowed for increased freedom, compared to similar alternatives, contributing to the success. Considering Ogre3D is open-source also allowed for the final product to fit the original purpose.

Development of a more advanced gameplay model is one of the aspects we recommend for future work. Also, many of the graphical effects can be developed further. A proper implementation the geometry buffer unlocks development of many future improvements, such as deferred shading in HDR, allowing a tone mapping implementation. Other proposed improvements include support for HBAO, flare, cascaded shadow maps, world-space velocity blurred motion-blur, and a faster car enamel reflection solution.

Even though the game is not a finished product, we think that the result is a success in regards to creating a graphically pleasing multi-player racing game, using limited development resources and money. The possibility of using open-source tools and libraries for such a task was also confirmed.

Acknowledgements

We want to thank our supervisors Ulf Assarsson and Erik Sintorn for their invaluable help during the whole process. We also want to thank everyone in the open-source community for their contributions to Ogre3D, Bullet, Blender, and all the other projects we used. Finally, we want to thank our friend David Gardtman for his ideas about gameplay and graphics in the beginning of the project.

Bibliography

[1] Ogre3D, "About." http://www.ogre3d.org/about, 2015. [Accessed: 01- Jun- 2015].

[2] B. G. Baumgart, "A polyhedron representation for computer vision," in *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pp. 589–596, ACM, 1975.

[3] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering.* CRC Press, 2008.

[4] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, pp. 143–150, Aug. 1986.

[5] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, pp. 311–317, June 1975.

[6] R. L. Cook and K. E. Torrance, "A reflectance model for computer graphics," *SIGGRAPH Comput. Graph.*, vol. 15, pp. 307–316, Aug. 1981.

[7] J. F. Blinn, "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, pp. 192–198, July 1977.

[8] B.-D.-N. to Pro, "Blender 3d: Noob to pro/uv map basics." http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/UV_Map_Basics, 2015. [Accessed: 18- May- 2015].

[9] Ogre3D-Manual, "Pixel formats." http://www.ogre3d.org/docs/manual/manual_67.html#Pixel-Formats, 2012. [Accessed: 14- Apr- 2015].

[10] G. Junker, *Pro OGRE 3D Programming.* Apress, 2006.

[11] T. McReynolds and D. Blythe, *Advanced graphics programming using OpenGL.* Elsevier, 2005.

[12] Ogre3D-Wiki, "The use of sprites in a 3d environment." http://www.ogre3d.org/tikiwiki/tiki-index.php?page=-billboard, 2011. [Accessed: 27- May- 2015].

[13] W. T. Reeves, “Particle systems—a technique for modeling a class of fuzzy objects,” *ACM Trans. Graph.*, vol. 2, pp. 91–108, Apr. 1983.

[14] S. Le Grand, “Compendium of vertex shader tricks,” *Direct3d ShaderX: Vertex and Pixel Shader Tips and Tricks*, pp. 228–231, 2002.

[15] J. Blinn and M. Newell, “Texture and reflection in computer generated images,” *Communications of the ACM*, vol. 19, pp. 542–547, October 1976.

[16] N. Greene, “Environment mapping and other applications of world projections,” *EEE Computer Graphics and Applications*, vol. 6, pp. 21–29, November 1986.

[17] L. Wanger, “The effect of shadow quality on the perception of spatial relationships in computer generated imagery,” in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, I3D ’92, (New York, NY, USA), pp. 39–42, ACM, 1992.

[18] L. Williams, “Casting curved shadows on curved surfaces,” *SIGGRAPH Comput. Graph.*, vol. 12, pp. 270–274, Aug. 1978.

[19] T. Heidmann, “Real shadows, real time,” *Iris Universe*, vol. 18, pp. 28–31, 1991.

[20] C. Schüler, “Eliminating surface acne with gradient shadow mapping,” *ShaderX4: Advanced Rendering Techniques (edited by W. Engel)*, pp. 289–297, 2005.

[21] R. Dimitrov, “Cascaded shadow maps.” `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`, 2007. [Accessed: 27- Apr- 2015].

[22] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, “The triangle processor and normal vector shader: A vlsi system for high performance graphics,” *SIGGRAPH Comput. Graph.*, vol. 22, pp. 21–30, June 1988.

[23] T. Saito and T. Takahashi, “Comprehensible rendering of 3-d shapes,” *SIGGRAPH Comput. Graph.*, vol. 24, pp. 197–206, Sept. 1990.

[24] S. Kircher and A. Lawrance, “Inferred lighting: Fast dynamic lighting and shadows for opaque and translucent objects,” in *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox ’09, (New York, NY, USA), pp. 39–45, ACM, 2009.

[25] M. van de Hoef and B. Zalmstra, “Comparison of multiple rendering techniques.” `http://www.marries.nl/wp-content/uploads/2011/02/Comparison-of-multiple-rendering-techniques-by-Marries-van-de-Hoef-and-Bas-Zalmstra.pdf`, 2010. [Accessed: 01- Jun- 2015].

[26] Ogre3D-Manual, “Ogre3d manual: Material techniques.” `http://www.ogre3d.org/docs/manual/manual_30.html#compositor_005ftexture`, 2012. [Accessed: 18- May- 2015].

[27] G. Spencer, P. Shirley, K. Zimmerman, and D. P. Greenberg, "Physically-based glare effects for digital images," in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, (New York, NY, USA), pp. 325–334, ACM, 1995.

[28] W.-m. W. Hwu, *GPU Computing Gems Emerald Edition*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2011.

[29] A. O. Akyüz, "High dynamic range imaging pipeline on the gpu," *Journal of Real-Time Image Processing*, pp. 1–15, 2012.

[30] P. Ledda, A. Chalmers, T. Troscianko, and H. Seetzen, "Evaluation of tone mapping operators using a high dynamic range display," *ACM Trans. Graph.*, vol. 24, pp. 640–648, July 2005.

[31] L. Bellia, A. Cesarano, G. F. Iuliano, and G. Spada, "Daylight glare: a review of discomfort indexes." `http://www.fedoa.unina.it/1312/1/Bellia_paper.pdf`, 2008. [Accessed: 2015-06-02].

[32] A. Majumder and S. Irani, "Contrast enhancement of images using human contrast sensitivity," in *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, APGV '06, (New York, NY, USA), pp. 69–76, ACM, 2006.

[33] P. Kovesi, "Fast almost-gaussian filtering," in *Proceedings of the 2010 International Conference on Digital Image Computing: Techniques and Applications*, DICTA '10, (Washington, DC, USA), pp. 121–125, IEEE Computer Society, 2010.

[34] S. Zhukov, A. Iones, and G. Kronin, "An ambient light illumination model," in *Rendering Techniques '98* (G. Drettakis and N. Max, eds.), Eurographics, pp. 45–55, Springer Vienna, 1998.

[35] M. Mittring, "Finding next gen: Cryengine 2," in *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, (New York, NY, USA), pp. 97–121, ACM, 2007.

[36] M. Fox and S. Compton, "Ambient occlusive crease shading." Game Developer Magazine, March 2008.

[37] *ACM SIGGRAPH 2008 Talks: Image-space horizon-based ambient occlusion*, (New York, NY, USA), ACM, 2008.

[38] J. Chapman, "Ssao tutorial." `http://john-chapman-graphics.blogspot.se/2013/01/ssao-tutorial.html`, 2013. [Accessed: 01- Jun- 2015].

[39] Ogre3D-Demos, "Normal-oriented hemisphere ssao shader implementation in ogre3d demos." `https://bitbucket.org/sinbad/ogre/src/default/Samples/Media/materials/scripts/SSAO/HemisphereMCFP.glsl`, 2012. [Accessed: 01-Jun- 2015].

[40] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion," in *ACM SIGGRAPH 2008 Talks*, SIGGRAPH '08, (New York, NY, USA), pp. 22:1–22:1, ACM, 2008.

[41] L. Szirmay-Kalos, T. Umenhoffer, B. Toth, L. Szecsi, and M. Casasayas, "Volumetric ambient occlusion," *Computer Graphics and Applications, IEEE*, vol. PP, no. 99, pp. 1–1, 2009.

[42] G. Papaioannou, M. Menexi, and C. Papadopoulos, "Real-time volume-based ambient occlusion," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, pp. 752–762, Sept 2010.

[43] M. Slomp, T. Tamaki, and K. Kaneda, "Screen-space ambient occlusion through summed-area tables," in *Networking and Computing (ICNC), 2010 First International Conference on*, pp. 1–8, Nov 2010.

[44] Nvidia, "Cuda c programming guide." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#maximize-instruction-throughput`, 2015. [Accessed: 01- Jun- 2015].

[45] Ogre3D-Demos, "Crytek ssao shader implementation in ogre3d demos." `https://bitbucket.org/sinbad/ogre/src/default/Samples/Media/materials/scripts/SSAO/CrytekFP.glsl`, 2012. [Accessed: 18- May- 2015].

[46] OpenGl-Standard, "Opengl glsl standard reference manual: dfdx, dfdy." `https://www.opengl.org/sdk/docs/man/html/dFdx.xhtml`, 2015. [Accessed: 01- Jun- 2015].

[47] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk, and J. R. Isidoro, "Accelerating real-time shading with reverse reprojection caching," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, (Aire-la-Ville, Switzerland, Switzerland), pp. 25–35, Eurographics Association, 2007.

[48] O. Mattausch, D. Scherzer, and M. Wimmer, "High-quality screen-space ambient occlusion using temporal coherence," in *Computer Graphics Forum*, vol. 29, pp. 2492–2503, Wiley Online Library, 2010.

[49] L. Bavoil and M. Sainz, "Multi-layer dual-resolution screen-space ambient occlusion," in *SIGGRAPH 2009: Talks*, p. 45, ACM, 2009.

[50] M. Potmesil and I. Chakravarty, "Modeling motion blur in computer-generated images," *SIGGRAPH Comput. Graph.*, vol. 17, pp. 389–399, July 1983.

[51] G. Rosado, "Motion blur as a post-processing effect," *GPU gems*, vol. 3, pp. 575–581, 2007.

[52] L. Sharan, Z. H. Neo, K. Mitchell, and J. K. Hodgins, "Simulated motion blur does not improve player experience in racing game," in *Proceedings of Motion on Games*, MIG '13, (New York, NY, USA), pp. 149:149–149:154, ACM, 2013.

[53] S. Hillaire, A. Lécuyer, R. Cozot, and G. Casiez, "Depth-of-field blur effects for first-person navigation in virtual environments," in *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, VRST '07, (New York, NY, USA), pp. 203–206, ACM, 2007.

[54] M. Ritchie, G. Modern, and K. Mitchell, "Split second motion blur," in *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, (New York, NY, USA), pp. 17:1–17:1, ACM, 2010.

[55] T. M. Jones, "Open source physics engines." `http://www.ibm.com/developerworks/library/os-physicsengines/os-physicsengines-pdf.pdf`, 2015. [Accessed: 01- Jun- 2015].

[56] A. Boeing and T. Bräunl, "Evaluation of real-time physics simulation systems," in *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, GRAPHITE '07, (New York, NY, USA), pp. 281–288, ACM, 2007.

[57] E. Coumans, "Bullet 2.83 physics sdk manual." `https://github.com/bulletphysics/bullet3/raw/master/docs/Bullet_User_Manual.pdf`, 2015. [Accessed: 01- Jun- 2015].

[58] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 6th ed., 2012.

[59] A. Steed, "Introduction to networked graphics," in *SIGGRAPH Asia 2011 Courses*, SA '11, (New York, NY, USA), pp. 12:1–12:159, ACM, 2011.