Night Racer
A case study in the small-scale development of a graphically refined game

Bachelor of Science Thesis in Computer Science and Engineering

MARCUS HULTMAN
JOHANNES KEINESTAM
ANDREAS LUNDQUIST
FREDRIK THANDER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2013

Night Racer
A case study in the small-scale development of a graphically refined game

Marcus Hultman,
Johannes Keinestam,
Andreas Lundquist,
Fredrik Thander.

Examiner: Sven Arne Andreasson

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

Cover:
Shows two players racing in the multiplayer mode of Night Racer.

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

This Bachelor's thesis details the development of a graphically refined action game by a small-scale development team under a six month time constraint. The goal is to create a visually pleasing game with basic playability and network game capabilities. The resulting game is also meant to be commercially viable, in the sense of having the potential of marketability and being proof of concept as part of a sales pitch.

The game in question is a racing game called Night Racer, which is created using Microsoft's XNA framework with the C# programming language. For making a visually pleasing game easier to develop, the target hardware is decided to be high-performing computers, which allows for large amounts of graphical effects to be implemented. Among the techniques implemented in the game are particle systems, post-processing effects such as bloom, shadows, Phong shading, normal maps, and environment maps. In the development of such a graphically refined game under time constraints, gameplay has to come second, but an extendable game engine is implemented to allow for further development of new, more interesting, game modes.

The result is a visually pleasing game, using graphical effects such as particle effects and pulsating light sources to provide a sense of constant screen movement. A multiplayer game mode allows up to four players to compete online, while providing a smooth gaming experience. The game is however very primitive gameplay-wise, and it is the opinion of the authors that developers working under a similar set of conditions that a more complete game engine solution than XNA would allow for further refinement of the graphics, while also allowing more time to be spent on gameplay.

Sammanfattning

Detta kandidatarbete beskriver utvecklingen av ett grafiskt snyggt datorspel, utfört av en liten projektgrupp under kort tid. Projektets mål är att skapa ett estetiskt tilltalande spel med grundläggande spelbarhet med möjlighet för nätverksspel. Det resulterande spelet är tänkt att vara kommersiellt gångbart, i bemärkelsen att det ska potentiellt kunna utgöra ett konceptbevis i syfte att rättfärdiga finansiering.

Spelet i fråga är ett racingspel med titeln Night Racer, och utvecklas med hjälp av Microsofts XNA-ramverk i C#. För att göra ett estetiskt tilltalande spel enklare att utveckla är projektet inriktat på hög-presterande datorer, vilket möjliggör implementation av en större mängd av grafiska effekter. Exempel på de grafiska tekniker vi har implementerat är partikelsystem, post-processing effekter som bloom, skuggor, Phong shading, normal maps och environment maps. I utvecklingen av ett så pass grafiskt snyggt spel under tidsbegränsning, så måste gameplay komma i andra hand. Dock har en spelmotor som gör vidare utveckling av nya, mer intressanta, spellägen enklare använts.

Resultatet av arbetet är ett visuellt tilltalande spel som använder partikeleffekter och pulserande ljuskällor för att skapa konstant rörelse på skärmen. Ett flerspelarläge tillåter upp till fyra spelare att spela emot varandra över nätverk utan att uppfattas hackigt. Spelmässigt är spelet dock väldigt primitivt, och åsikten av denna rapports författare är att spelutvecklare under liknande arbetsförhållanden bör överväga färdigskrivna spelmotorer. En sådan lösning skulle låta mer tid läggas på att polera grafiken, men också möjliggöra ytterligare utvecklingstid på gameplay.

Acknowledgements

We wish to thank all the people that have helped in the writing and proof-reading of this thesis and in the development of Night Racer by providing feedback and technical help. The supervisor of this thesis project, Ulf Assarsson, has in particular provided invaluable help in both technical as well as lingual matters all throughout the project. We express our gratitude to the fellow students in group 33 with whom we have shared the project room, who have provided valuable comments and feedback on the look of the game.

Table of Contents

1 Introduction **1**

1.1 Goal and Purpose 1
1.2 Problem 2
1.3 Limitations 3
1.4 Method 3
1.4.1 Night Racer 3
1.4.2 Agile Development 4
1.4.3 Libraries 4
1.4.4 Development Environment 4

2 Game Engine **5**

2.1 Game Loop 5
2.2 States 6
2.2.1 Results 6
2.2.2 Discussion 7
2.3 User Interface 7
2.3.1 Results 8
2.3.2 Discussion 10
2.4 Triggers 10
2.4.1 Event Triggers 10
2.4.2 Results 11
2.4.3 Discussion 11
2.5 Game Modes 11
2.5.1 Results 12
2.5.2 Discussion 13

3 Level Design **14**

3.1 Terrain 14
3.1.1 Results 14
3.1.2 Discussion 16
3.2 Track 17
3.2.1 Results 17
3.2.2 Discussion 17

3.3 Navigation Mesh . . . 18
3.3.1 Generating an Approximation . . . 18
3.3.2 Results . . . 19
3.4 Seed . . . 19
3.4.1 Results . . . 20
3.4.2 Discussion . . . 20

4 Real-time Graphics 21
4.1 Models . . . 21
4.1.1 Vertex Data . . . 21
4.1.2 3D Modeling Packages . . . 22
4.2 Graphics Pipeline . . . 22
4.2.1 Stages of the Pipeline . . . 22
4.2.2 Shaders . . . 23
4.3 Lighting . . . 24
4.3.1 The Rendering Equation . . . 24
4.3.2 Light Sources . . . 24
4.3.3 Reflected Light . . . 24
4.3.4 Shading Models . . . 26
4.3.5 Deferred Lighting . . . 27
4.3.6 Results . . . 27
4.3.7 Discussion . . . 28
4.4 Shadows . . . 29
4.4.1 Shadow Simulation . . . 30
4.4.2 Results . . . 30
4.4.3 Discussion . . . 31
4.5 Reflections . . . 31
4.5.1 Environment Maps . . . 31
4.5.2 Dynamic Environment Mapping . . . 32
4.5.3 Results . . . 33
4.5.4 Discussion . . . 33
4.6 Skybox . . . 34
4.6.1 Sky Representation . . . 35
4.6.2 Results . . . 35
4.6.3 Discussion . . . 36
4.7 Texturing . . . 36
4.7.1 Texture Mapping . . . 36
4.7.2 Diffuse Texture . . . 36
4.7.3 Normal Map . . . 36
4.7.4 Alpha Map . . . 37
4.7.5 Texture Filtering . . . 38
4.7.6 Depth Buffer . . . 39
4.7.7 Multitexturing . . . 39
4.7.8 Results . . . 40
4.7.9 Discussion . . . 40
4.8 Particles . . . 41
4.8.1 Results . . . 41

4.8.2 Discussion . . . 42
4.9 Post-processing Effects . . . 43
4.9.1 Render to Texture . . . 44
4.9.2 Blur . . . 44
4.9.3 Bloom . . . 45
4.9.4 Motion Blur . . . 45
4.9.5 Results . . . 46
4.9.6 Discussion . . . 48

5 Optimization **49**
5.1 Rendering Optimization . . . 49
5.1.1 Level of Detail . . . 49
5.1.2 Backface Culling . . . 50
5.1.3 Occlusion Culling . . . 50
5.1.4 Viewing Frustum Culling . . . 51
5.2 Results . . . 52
5.3 Discussion . . . 52

6 Audio **53**
6.1 3D Positional Effects . . . 53
6.2 Sound Libraries . . . 54
6.3 Results . . . 54
6.4 Discussion . . . 56

7 Multiplayer and Network **57**
7.1 Offline Multiplayer . . . 57
7.2 Online Multiplayer . . . 58
7.3 Network Architecture . . . 58
7.3.1 Client-server . . . 58
7.3.2 Peer-to-Peer . . . 59
7.4 Frameworks . . . 59
7.4.1 XNA Networking . . . 60
7.4.2 Lidgren . . . 60
7.5 Combating Latency . . . 60
7.5.1 Identifying Player Data . . . 60
7.5.2 Simulating Smooth Gameplay . . . 60
7.6 Results . . . 61
7.7 Discussion . . . 64

8 Results **66**

9 Discussion **67**

10 Conclusions **69**

11 Future Work **71**

12 References **73**

1 Introduction

Computer graphics is a fast-growing area of research, especially in the video game industry. The importance of producing visually pleasing games has increased, and the bar of what is considered visually pleasing is constantly raised. In recent years, computer processing power has increased tremendously, leading to the possibility of higher graphical fidelity, in turn leading to increased consumer demand for graphically competitive games. Companies, such as Microsoft, have made huge investments to meet this demand, and as a consequence, the area of research has developed very fast (Bracken and Skalski, 2009).

In recent years, many graphical techniques have been developed by research teams and game engine specialists to improve the graphical appearance of games. These techniques are meant to portray reality as close as possible. In real-time rendering, the visual phenomena found in reality, such as shadows, cannot be exactly emulated because of the time it would take to calculate each frame, and first at about six frames per second, the viewer starts to feel that the images are indeed moving (Akenine-Möller et al., 2008). The techniques used in games are thus merely visual approximations of the real world.

This project is a part of a Bachelor's thesis at Chalmers University of Technology and is carried out by four students in the Software Engineering department. The assignment received from the department was to develop a multiplayer racing video game with focus on visuals by programming graphical algorithms and effects, the specifics of which this chapter aims to expand upon.

1.1 Goal and Purpose

The goal of the project is to design a video game which fulfills a number of requirements.

- The game shall be visually pleasing.
- The game shall offer playability. In other words, the player shall be able to successfully compete in and finish a race.
- The game shall offer multiplayer capabilities over a local area network and the Internet.
- The game shall be commercially viable.

These requirements are largely self-explanatory, but the definition of what is visually pleasing in the context of a game is very subjective; accounting for this, the authors of this report offer the following definition. A visually pleasing game does not necessarily require high resolution textures and numerous polygons. However, such imperfections are easily noticeable in a scene with little movement, and therefore it is important to always provide a sense of movement even if the player is idle. Of course, there cannot be moving objects on the screen at every single moment, but a sense of movement can be achieved by using graphical effects. This can hide graphical blemishes and provide an interesting visual experience. Effects for accomplishing this can for example be pulsating or moving light sources and particles.

Another term that must be defined further is "commercially viable" which typically refers to if a product can be sold for profit. In the context of the product detailed in this report, commercial viability does not mean that the product should be ready for commercial sales by the end of the development cycle, nor that it is actually intended to be sold at some point. The term is only used to refer to a product that has the potential of achieving commercial marketability, and is in a state of development so that it viably could be used as a technological demonstration as a part of a sales pitch.

The project is developed under a certain set of conditions, in that the project team is composed of developers with no previous game development experience as well as a time frame of 16 weeks. With such conditions, the project scope must be well defined and development methods must be well chosen. It should be noted, however, that the goal is not to have produced a finished game by the end of the development cycle.

The purpose of this thesis will be to evaluate what methods, techniques and graphical effects can be used in conjunction with each other to create the previously described video game.

1.2 Problem

A number of interesting problems arise in the development of a game with the previously stated goals. The game must be visually pleasing, but furthermore also commercially viable, which means that it can not be merely a demonstration in graphics, but must also offer basic playability. These problems need to be researched, and the results as implemented in the game will be presented in this thesis. A number of problems have been identified, which will need to be researched in the report, and the resulting choices will be used in the game. The problems that will be researched have been identified, and are are presented below.

- **Game engine.** Implementation of a suitable game engine, supporting the needed game mechanics, logical structure, and graphical techniques chosen for the project. The engine must also provide an extendable software architecture for making future development possible.
- **Level design.** Crafting a diverse gaming experience for players by providing varying yet realistic levels, which presents the problem of finding an approach for making such levels efficiently.
- **Real-time graphics.** Identifying graphical algorithms for providing visually pleasing results, while also keeping acceptable frame rates on the target hardware. Since producing a visually pleasing game is one of the primary goals of the project, several types of effects need be studied and implemented; among them are effects for lighting, shadows, reflections, and particles. As such, the research of graphics will represent a large portion of this report.

- **Optimization.** Ensuring that the computer processing power is used efficiently while providing no degradation of visual quality, so that as many players as possible can viably experience the gameplay without major performance issues.
- **Audio.** Providing an immersive gaming experience by using sound in the game, and implementing sounds in the background as well as contextual and positioned sound effects in the game in an impactful way.
- **Multiplayer.** Allowing several players to compete against each other without issues with choppy or otherwise unnatural gameplay by simulating smoothness in cases where sufficient data is unavailable.

1.3 Limitations

In the development of games and software in general, some parts of the project have to be prioritized. Choosing a car racing game allows the game mechanics to be straightforward compared to those of other types of games, such as role-playing games, which allows for more time to be spent on making the game visually pleasing. As such, the gameplay will be quite limited and may not provide playability on the same level as commercial games.

The graphics of the game are restricted by the target hardware platform, which is relatively high performing consumer-grade computers. This is required since a visually pleasing game with graphical effects requires high-performance graphics processors, and the resulting game will be presented to fellow students on a set of computers using the *NVIDIA GeForce GT 640* graphics card.

1.4 Method

The thesis is structured as a case study, where each chapter encompasses a field of study relevant to what needs to be implemented in the game. Each chapter presents the techniques considered and concludes by presenting the result as implemented in the game. To present the theoretical parts of the report, applicable literature will be researched. Game development is a largely practical field, and as such, scientific as well as more informal literature (blog posts for example) from reputable sources will be part of the research presented in this thesis.

Chosen technologies will be presented by the state of the game after their implementation, typically with accompanying screenshots visualizing the results of the technology in applicable cases. Discussions of the efficiency of the technique and its results will then be presented. These sections are largely practical, and will thus present the findings as original research.

1.4.1 Night Racer

The working title of the game developed in the case study is *Night Racer*, and it is a car racing game set in a dark but colorful fantasy world. The choice of setting is made because it was recognized as rather original for a racing game, but also because it allows for numerous graphical effects to be integrated more naturally. Night Racer will allow players to drive with real-world cars on dynamically generated levels, and will also provide a simple multiplayer competitive racing mode.

1.4.2 Agile Development

To make a product as polished as possible within a short time frame where progress needs to be made constantly, it is important to work in an agile fashion. This means working iteratively, where in each iteration a set of features (*milestones*) shall be implemented; before and after each iteration, the product should be functioning and ready for deployment. Agile development allows for a very dynamic development process, where features can easily be added and removed, and tested continuously (Fowler, 2005).

1.4.3 Libraries

Existing libraries for car and racing games will be used in order to create a functioning game as quickly as possible. The purpose of the project is not to build a game from scratch, but rather to use applicable libraries for physics, collision detection, and network to lessen the workload and make a better quality game as well. These libraries will be presented in their respective chapters.

1.4.4 Development Environment

The C# programming language and the Visual Studio development environment will be used for development. XNA 4.0, developed by Microsoft, has been chosen as the development framework for Night Racer. XNA provides low-level code implementations for rendering graphics and deploying games on multiple platforms (Sanders, 2006) but does not provide an actual game engine. This, however, allows for greater control of the implementation of graphical effects and other game code at the cost of higher complexity (compared to using a finished game engine). However, it is the opinion of the authors that increased control allows a more thorough examination of the problem statement (see Section 1.2).

2

Game Engine

A major part of the game development process involves programming. As mentioned in Section 1.4.4, the XNA Framework hides most low-level graphics details, however it does not provide a full game engine. As a result, a game engine must be programmed from scratch for Night Racer. Thorn (2011) describes a game engine as a non-precise concept but typically involving code for managing rendering on the screen, loading resources, handling input from the user, applying physical laws on game entities, and scripting gameplay elements. Management of rendering, resources, and input are largely provided by XNA and are thus not examined further.

This chapter will examine the problems present in programming the base parts of the game engine that accommodates implementation of game mechanics by an extendable software architecture. These parts of the game engine are not directly part of the gameplay of Night Racer but a necessary code base which supports the development of the game. Among these core parts of the game engine are the overall structure of the game code, the management of different game states, implementation of game mechanics, game menus, and events. These will all be described in detail, and in the process, the game logic as implemented in the game will be presented and evaluated regarding extensibility.

2.1 Game Loop

The very base of a game is the game loop, which is how it updates its state. The frequency of this update is determined by the *game tick*, which may occur at a fixed interval or manually invoked by the game logic. Night Racer uses the default XNA game tick frequency, which is set to 60 Hz. XNA provides the *IUpdateable* interface for game objects that should be updated at each game tick and the *IDrawable* interface, if the object should be rendered as well. XNA allows for a highly modular and object-oriented approach through the *GameComponent* and *DrawableGameComponent* base classes, themselves implementing *IUpdateable* and *IDrawable* respectively, since objects implementing these interfaces are easily added to the game loop by registering them as components.

2.2 States

Night Racer does not consist of only one gameplay screen, but also additional screens showing menus, game over statistics, and a lobby for multiplayer games. A player must be able to navigate between these states to start a game, for example, as visualized in Figure 2.1. As such, there needs to be a way to represent these states in the game engine and make it possible to manage the transition from one state to the next.

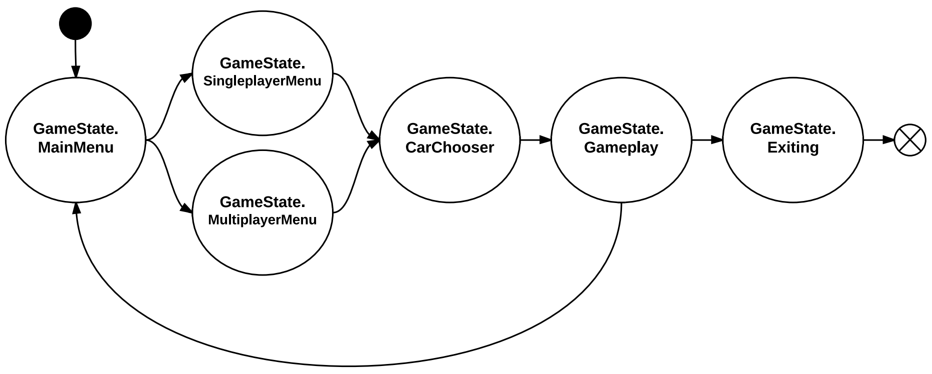


Figure 2.1: A state machine representing a possible state flow for a player of Night Racer, from start to finish.

2.2.1 Results

An extendable system for creating manageable states has been implemented in the game engine. To use the implemented state management system, each screen that needs to be shown must extend the implemented abstract class *GameStateView*, itself a subclass of *DrawableGameComponent*. Each *GameStateView* shall be associated with a constant from the globally accessible *GameState* enumerable, so that there is one instance of *GameStateView* corresponding to each state. The states are managed by each *GameStateView* through the *UpdateState* method, the code of which is visible in Listing 2.1.

```
// class MainMenuView extends GameStateView
public GameState UpdateState(GameTime time)
{
   if (input.IsPressed(Keys.Enter) && selectedItem is singleplayerButton)
      return GameState.SingleplayerMenu;
   // ...
   return GameState.MainMenu;
}
```

Listing 2.1: An abridged code snippet from the *UpdateState* method of the *MainMenuView* class, a subclass of *GameStateView* responsible for showing the menu when the game is started.

A *GameStateView* may handle state changes internally by not returning it to the state management system, such as when the menu system transitions a menu with another. However, some state transitions need to be handled by the state management system, implemented in the *GameManager* class whose game update loop is shown in Listing 2.2, so that new state views can be initialized.

```
// class GameManager
public override void Update(GameTime time)
{
   var nextState = currentView.UpdateState();
   if (currentState != nextState)
   {
      switch (nextState)
      {
         case GameState.Gameplay:
            currentView = new GameplayView();
            break;
         // ...
      }
   }
   currentState = nextState;
   base.Update();
}
```

Listing 2.2: An abridged code snippet from the *Update* method of the *GameManager* class, responsible for managing the game states.

2.2.2 Discussion

The implemented system can manage game states successfully, and it is also very extendable which is very important for further development. The concept of having a state return the next state when updated (as seen in Listing 2.1) is quite intuitive, but did present problems since these components can thus not be updated as typical XNA components by the *Update* method. This meant that the *GameStateViews* could not be added as components in the XNA game loop and had to be updated explicitly by the *GameManager* class (see Listing 2.2). However, not using the XNA component system lead to difficulties in disposing finished *GameStateViews* from the memory, and the resulting implementation uses the component system and both the *Update* and *UpdateState* methods.

The problems of this implementation occurred largely due to a lack of knowledge of details in the XNA framework structure, and in hindsight would have been better implemented after further research. An alternative implementation that would be more fitting with the component structure could for example be using a flag in each *GameStateView* to indicate change in state, which would mean not having to work around the existing structure of XNA.

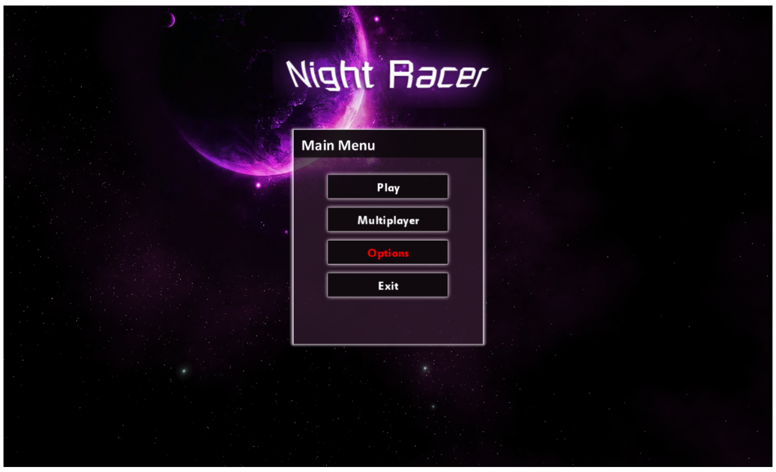
2.3 User Interface

Providing a user interface of some sort is essential in all but the most basic video games, so that players may set up the game and receive information that are not representable directly in the game world. A game will typically need two types of interfaces to accomplish these goals: a menu system (in form of front-end as well as in-game menus) and the heads-up display (”HUD”) respectively.

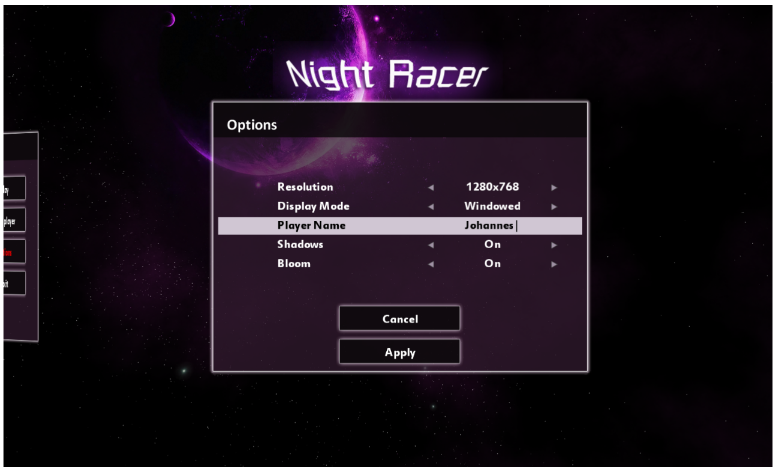
Fox (2004) presents goals of usability and design in the interface of a game, where simplicity and depth are paramount but at odds with one another. As for usability, it is important to present an easy to use menu that allows the player to access the game quickly and without unnecessary information and options, but at the same time allowing power users to change options in a more fine-grained fashion when needed. Therefore, it is important to keep advanced options away from the menu flow that will take the player to the game and provide *good defaults* for a player wanting to start a game immediately. Design-wise, the menu should be consistent with the theme and color scheme of the game and should not have large, distracting elements. This helps in making the interface feel like a seamless part of the game and not like a burden.

2.3.1 Results

When starting the game, the player is taken immediately to the front-end menu, displaying the main menu (shown in Figure 2.2a) which provides options for starting a game in singleplayer mode, starting a game in multiplayer mode, changing game options, and exiting the game. A player wishing to start a singleplayer game will be taken to the car chooser menu, where the game can be started after choosing a car; a player starting a multiplayer game will be asked to connect to a server and shown the lobby screen (see Section 7.6) before continuing to the car chooser menu. The options menu provides a few simple options, as seen in Figure 2.2b.



(a) Main menu.



Resolution 1280x768
Display Mode Windowed
Player Name Johannes
Shadows On
Bloom On

(b) Options menu.

Figure 2.2: The front-end menu system of Night Racer.

The menus are built around the *OverlayView* class, which is a drawable component representing the actual menu. This reusable class helps make the implementation of specific menus simpler, which allows for further and more complex menus as a future development. *OverlayView* objects may be used in more contexts than the front-end menu; one such context is in-game menus that are rendered on top of the gameplay when Night Racer is paused, which is shown in Figure 2.3.

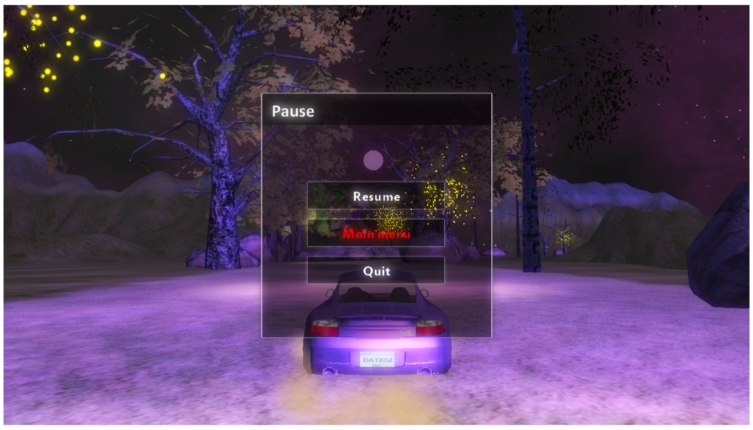


Figure 2.3: The pause menu shown in Night Racer, which allows the player to exit during gameplay.

A simple HUD has been implemented, which is used to show persistent displays as well as pop-up notifications, and is shown in Figure 2.4. Persistent HUD displays in the game include a component showing the player's place in the race, a speedometer, and a timer; pop-up notifications are shown contextually, for example when the player passes a checkpoint or the goal line, and when the player overtakes another player or is overtaken. Pop-up notifications are animated when appearing and disappearing to make the interface seem more kinetic, but care has been taken to not distract the player by making the notifications take up a minor part of the screen and showing them for no longer than three seconds. Pop-up notifications can easily be shown by a simple function call by the programmer.

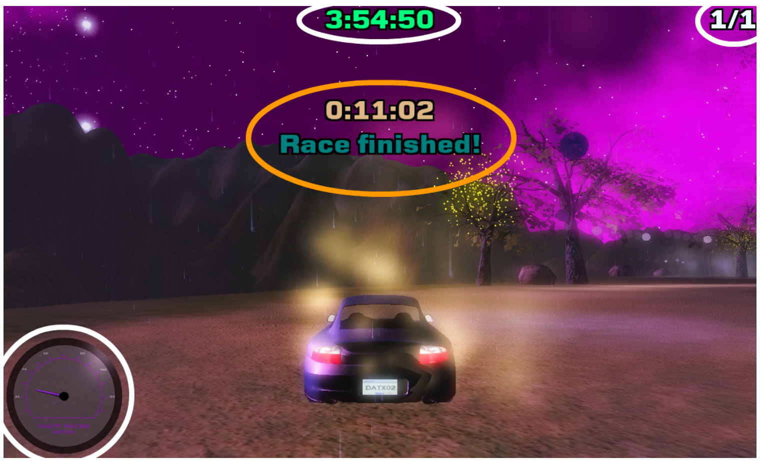


Figure 2.4: The HUD in Night Racer, showing persistent HUD displays marked white and pop-up notifications orange.

2.3.2 Discussion

The purpose of the menu system is to provide a simple interface for the user to start the game, while also providing a reusable interface for further development. In the first respect, the menus implemented in the game satisfy the simplicity test suggested by Fox (2004), since only two presses on the Enter button is enough to start the game with a default car chosen. However, the usability of the multiplayer connect menu is somewhat lacking since the user needs to input an IP address explicitly. Instead, a server browser of sorts could be implemented, but this would require major time and resource investments (such as procuring and running a centralized server). Overall, however, the implemented menus are deemed satisfactory.

Regarding code reusability, the extendable *OverlayView* class is mostly successful since simple menus can be implemented in a very simple manner by adding to a list of menu items. However, a typical placement of menu items may require some additional effort by developers, since the placement of menu items is standardized as being vertical and straight.

The choices and methods used in the development of the menu system were largely satisfactory, and provides a more professional quality to the game which assists in satisfying the goals of commercial viability and playability. The results are especially satisfactory because the initial vision of the menu system was much more primitive than the resulting implementation. The increased ambition in designing the menus arose essentially because of momentary inspiration, but if done again, however, this inspiration would have been better suppressed since it leads to more time being allocated to something that would have been better spent elsewhere.

2.4 Triggers

A fast-paced game executes many lines of code in every game tick. In such games, a system for supporting the triggering of multiple contextual events must be provided. Examples of such contextual events can be thunderstrikes occurring at a fixed time interval and falling trees when the player passes by. Dictating when events should occur can become very complex since a game could contain a very large number of events waiting to be executed in some given context. An approach for implementing such events effectively is using a trigger system (Orkin, 2002). This section describes the issues related to the implementation of the triggers and presents the implementation in Night Racer.

2.4.1 Event Triggers

The processor time consumed by a trigger system depends of the number of events due to be updated at once, and a trigger system with many events would thus need much processing power. However, in a typical game there are many events that need not be processed at all times. These events could occur at a time interval, or only in some context depending on the position of the player, for example.

Physical objects can trigger events, and the more of these objects' processes actively checking for the events to be triggered (*polling*), the slower the game. A cost efficient system for event handling that prevents duplicate computations is therefore needed, and thus the goal of the trigger system is to separate the event code of a game object, so that in can be invoked by the trigger system only when the event is triggered (Orkin, 2002).

2.4.2 Results

Using a variation of the observer design pattern as described by Gamma et al. (1994), trigger events declared as actions to be executed are then registered as events which will get notified whenever the event is triggered. The XNA component architecture allows a centralized trigger manager to be updated each game tick, which is used to update each trigger and, if triggered, notify the subscribing events.

In a racing game such as Night Racer, the progress the player can make in the gameplay is quite simple. This progress can be determined strictly geometrically by the player's position on the race track, and thus, tying the trigger system to the race track allows for events to be handled by the trigger manager based on the position of an object in the game. Such a solution is used in Night Racer, which allows for implementation of any behavior when the player passes a checkpoint. Night Racer uses triggers in the game mode system (described in Section 2.5), as well as for playing sounds contextually.

2.4.3 Discussion

Triggers are an essential part of Night Racer, since they are needed for reacting in contexts such as passing a checkpoint, which is a large part of the playability offered by the game. The method of using XNA as a framework has helped in the implementation of the centralized trigger system, reducing the processor time otherwise used in a polling-based implementation. As such, the trigger system in the game is quite efficient, and has been found to support very large amounts of triggers without a noticeable performance impact.

However, the trigger system is rather inflexible in its current iteration. This is mainly because the triggers can only be triggered based on an objects position on the race track. A more generalized solution could be triggers that evaluate any boolean expression in each game tick, and instead provide the race track-based trigger as an extension of this base class. Such a solution would allow more complex game modes, for example where the objective is destroying the opponents car, to be developed in the future.

2.5 Game Modes

The term game mode is used in this report to refer to a set of goals that must be achieved to make progress in the game. Some goals may need to be completed in a specific sequence, while others may be achievable throughout the gameplay or only contextually. In other words, a game mode is what the player is encouraged to play through to complete the game. Depending on the player's performance in completing the game mode, the outcome of the game may differ. Importantly, a game mode needs to provide a challenge and preferably a sense of enjoyment.

Some examples of game modes in general are *capture the flag* and *deathmatch*, which provides the same basic gameplay and in some games even uses the same levels, but with two different goals. In the context of a racing game, game modes may include racing against time, opponents, or avoiding obstacles.

To make development of game modes viable, both during the development process detailed in the report and as future work, it is important to provide an extendable software architecture that allows programmers to define and deploy game modes simply instead of a hard coded system.

This section will present the considerations that went into implementing game modes in Night Racer, and conclude by showing a high-level description of the implemented code.

2.5.1 Results

As the first step of implementing an extendable code base for game modes, what makes up a game mode and how these artifacts can be represented in code must be identified. In the Night Racer code, a game mode is considered to be split into a number of sequential *GameModeState*s, which in turn consists of a list of sequential triggers (see Section 2.4). When all triggers have been triggered in the *GameModeState*, the game mode advances to the next state, and when all states are completed, the game is over. This logic is represented in the *GameplayMode*, shown in Listing 2.3, which is the foundation of the game mode system.

```
// abstract class GameplayMode
public void Update()
{
    if (!allStatesFinished)
    {
        GameModeState current = states[currentStateIndex];
        current.Update();

        if (current.IsStateFinished())
            currentStateIndex++;
        if (currentStateIndex > states.Count - 1)
        {
            allStatesFinished = true;
            GameOverProcedure();
        }
    }
}
```

Listing 2.3: An abridged code snippet from the *Update* method of the abstract *GameplayMode* class, which represents a general game mode upon which others are derived.

A developer adding a game mode to the game needs to extend the *GameplayMode* class, and implement the abstract methods *Initialize* (which prepares the triggers and the list of *GameModeState*s) and *PrepareStatistics* (which calculates end-game information when the game is finished). However, this system merely keeps the game progressing and encodes the victory conditions. Some game modes may require further changes to the gameplay, such as adding obstacles to the race track. Such initializations could, however, easily be made in the *Initialize* method.

More tangibly, Night Racer uses this system to implement a simple race mode by encoding each lap around the track as a *GameModeState* containing a list of triggers where a trigger is positioned at each checkpoint, which needs to be triggered in a specific sequence by the player. At each trigger, the race time is recorded and then presented as end-game statistics. The multiplayer race mode implemented in the game extends this functionality by also keeping track of the lap times of network players, so that aplayer placements list may be shown as end-game statistics.

2.5.2 Discussion

The primary concern in developing an extendable code base for game modes was identifying and distilling what a game mode actually consists of. The concept of providing a set of trigger collections, where each trigger needs to be activated before the game can progress to the next set of triggers in the collection, is found to be quite universal of the game modes that have been considered. In this respect, the implemented solution is very much a success. However, there are some game modes that cannot be expressed with the current game mode system, but these are largely due to the rather limited trigger system, which can only express triggers based on an object's position on the race track. Using a more generalized concept of a trigger would allow virtually any game mode to be implemented using the *GameplayMode* class (for discussions regarding the trigger system, see Section 2.4.3).

3

Level Design

Video games need to take place in some sort of environment. In a car game, the environment consists of several levels with different terrain and track layouts. Designing a level is a time consuming process in game development and the importance thus needs to be evaluated. This chapter will cover the problem of providing a diverse gameplay experience, and in the process detail some alternatives for implementation of level design. The level design components detailed are the terrain and the track. Furthermore, an introduction will be provided to navigation meshes, which are used to keep the cars on the track, and details about the seeds, which is used for random generation.

3.1 Terrain

The creation of terrain can be taken in two directions. The first alternative is to design it by placing every object of the scenery manually, and the second alternative is to leave the creation up to randomized generation.

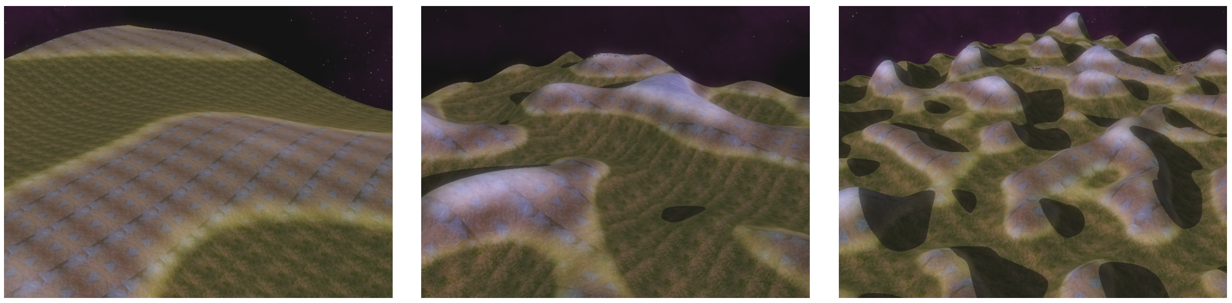
The design alternative can be further divided into two rough categories, the first being designing a complete level and the second is to create terrain blocks. A terrain block is a modeled, often square, area containing both a piece of the race track and all of the scenery; these blocks are then fused together to create a complete level. Both of these categories are quite time consuming and not entirely feasible for this project. Designing terrain blocks would be the preferable choice of the two, since a finished block could be used in several different tracks.

The final alternative is random generation, which has the potential to look good, if done well. What generation offers is acceptable quality of content for a relatively minor time investment. Because of how the generation works, it effectively offers infinite amount of levels, and a high level of replayability.

3.1.1 Results

The terrain of the game is randomly generated. If the levels were to be manually designed, a subset of the team would have to be assigned as level designers which would take valuable time away from other tasks. Dynamically generated levels, while being an interesting challenge (for both programmers and players), also increase the replayability of the game as the levels will never look the same after restarting unless a previously saved level was loaded.

To generate the terrain, a process called procedural generation is applied (Frade et al., 2012) using random values provided by a random number generator (see Section 3.4). In particular, a Perlin generator, which is a process that utilizes Perlin noise (Perlin, 1985, 2005), is used to calculate a height map which is then applied to the terrain of Night Racer, seen in Figure 3.1. A height map is a two dimensional grid array of float values describing the height of the different parts of the terrain. The Perlin noise follows a sine-like curve and is thus not realistic as an environment. In order to create a terrain that is not as continuous, the heights generated by the Perlin noise are perturbed (offset in a random direction) creating additional hills and small mountain ranges as illustrated by Figure 3.2.



(a) The result of a low level of Perlin noise, producing a terrain with few low height differences.

(b) The result of a medium level of Perlin noise, producing a terrain with a larger amount of height differences.

(c) The result of a high level of Perlin noise, producing a terrain with very frequent height differences.

Figure 3.1: A sequence of terrains with an increasing amount of Perlin noise.

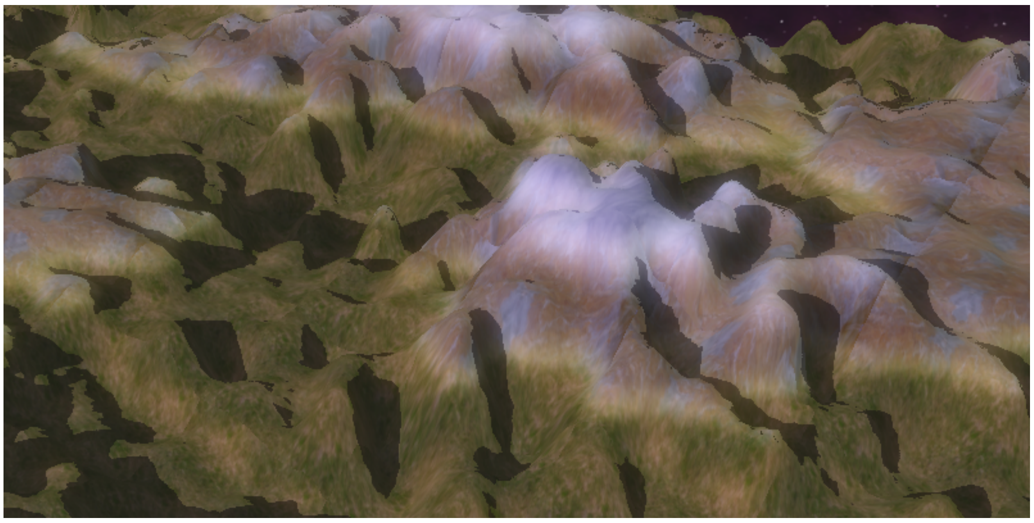


Figure 3.2: A terrain with a high level of perturbation. As can be seen, there is a higher number of hills, but there is also a larger difference in heights than the result of a high level of Perlin noise.

The resulting terrain contains a large amount of small hills and some larger mountains; a terrain that looks highly unnatural, even though it contains many interesting shapes. By using a function called erode the terrain near a point is flattened, and as a result smooth edges are eliminated. The result is that a more interesting terrain is created, with valleys, steep cliffs, and plateaus (see Figure 3.3). As a side effect, the terrain becomes spiked in places and looks very uninviting. To remedy this, the terrain is smoothened, reducing the maximum height distance between points, resulting in the more natural and softer looking terrain visualized in Figure 3.4.

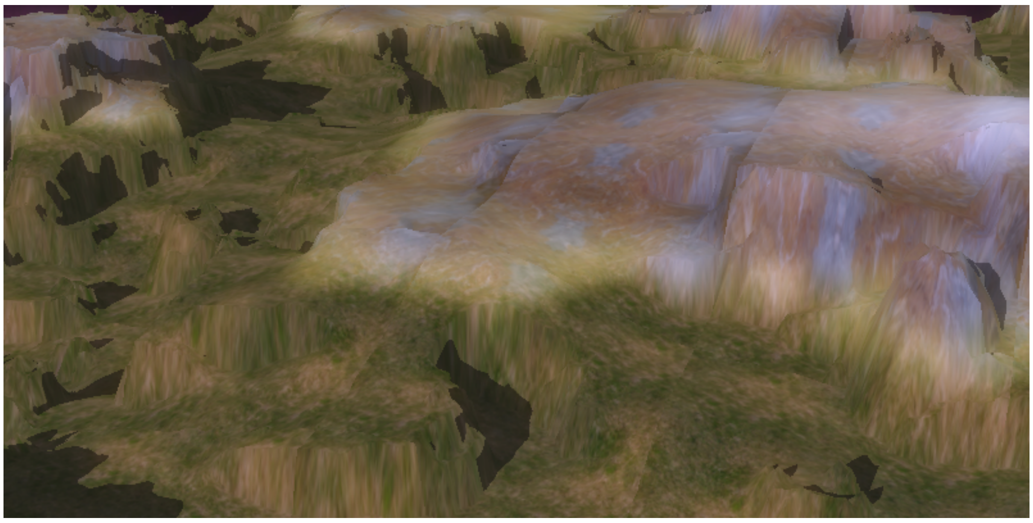


Figure 3.3: The result of an eroded terrain. The effect has created steep cliffs and a more jagged landscape, and the hills created from perturbation have been molded into a more interesting terrain.

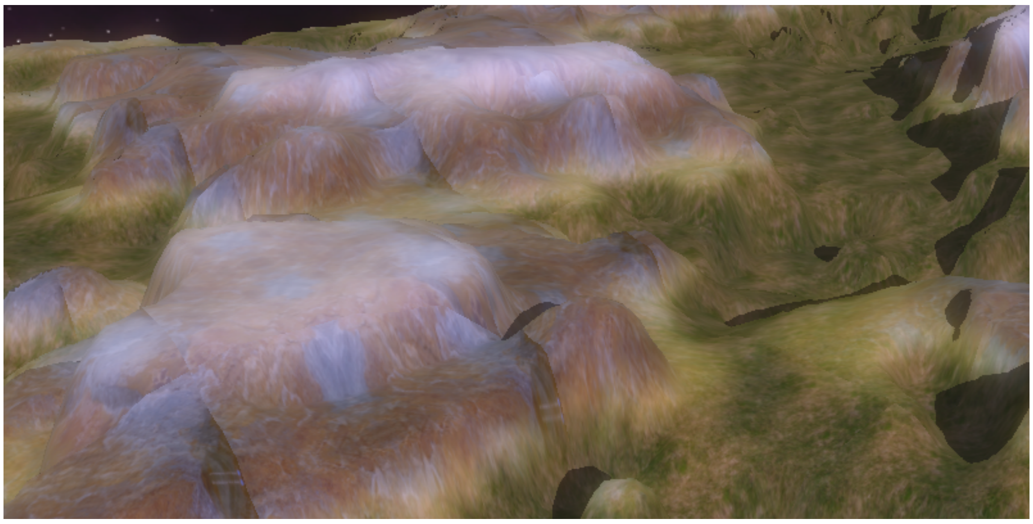


Figure 3.4: A smoothened terrain. The result from the erode function has been smoothened out, creating a more natural looking terrain.

3.1.2 Discussion

The final terrains generated are of an acceptable quality and offer interesting scenery for the player to drive in. The general shape of the terrain contains some oddities, but due to the lighting conditions of the game they are not as noticeable as they would be in a more well lit environment. The largest increase in visual quality would most likely be gained from textures (see Section 4.7) of a higher quality.

The largest problem with the current terrain is that it is loaded in its entirety into the memory, and as a consequence consumes a large amount of the available memory resources. If the terrain was instead split into smaller chunks, and only the necessary parts were loaded into the memory as needed, the memory consumption would be lowered. As a result, options such as a much larger terrain or a terrain with a higher level of detail (due to the increase in height point density) would be possible.

3.2 Track

Racing cars usually drive on roads, as such the created terrain is not an appropriate level to compete on. Thus there is a need to create a track for the cars to compete on. An easy to use representation of a track is a curve spline.

Splines are the vectorized implementation of lines or curves built by nodes and their directions. Compared to a polygon, which is a set of nodes with straight lines between them, splines with weighted normals affect the lines between the nodes making them more soft and curvy.

3.2.1 Results

Because the terrain was created with procedural generation the choice of creating the track naturally fell on automatic generation with curves, as the alternatives available are unfeasible in relation to the previous process used. The track is based on a curve spline that passes through a pre-determined number of nodes. These nodes can be located anywhere on the terrain and the curve passes through them in the order they were created. However, caution needs to be taken while placing the nodes, as the resulting curve may contain crossing roads or strange loops due to the spline's characteristics.

The resulting curve connects each node with straight lines. In order to get a more interesting track layout, the direction of the curve in each node is offset by a random value. Each node is also given a height value which affects the height of the curve. The curve is positioned on the generated terrain, and the curve receives a width appropriate to the game's race track. The height map's values that intersect with the track are then replaced with the heights associated with the curve, and the edges between the terrain and the track are interpolated, as illustrated in Figure 3.5.

The latest iteration of the track consists of five nodes which are positioned in a pentagon shape. The direction of each node is set to point towards the center of the shape in order to minimize the risk of the curve being located outside of the terrain perimeter, which would result in an exception. The direction the race takes place in is also randomized each time a level is created.

3.2.2 Discussion

The current version of the track fulfills the goal of offering the player basic playability, the ability to finish a race. The track's layout also varies a bit each time it is created and contributes in creating a more diverse gaming experience. The method, due to its efficiency and relative ease of use, is also well suited for the scope of the project.

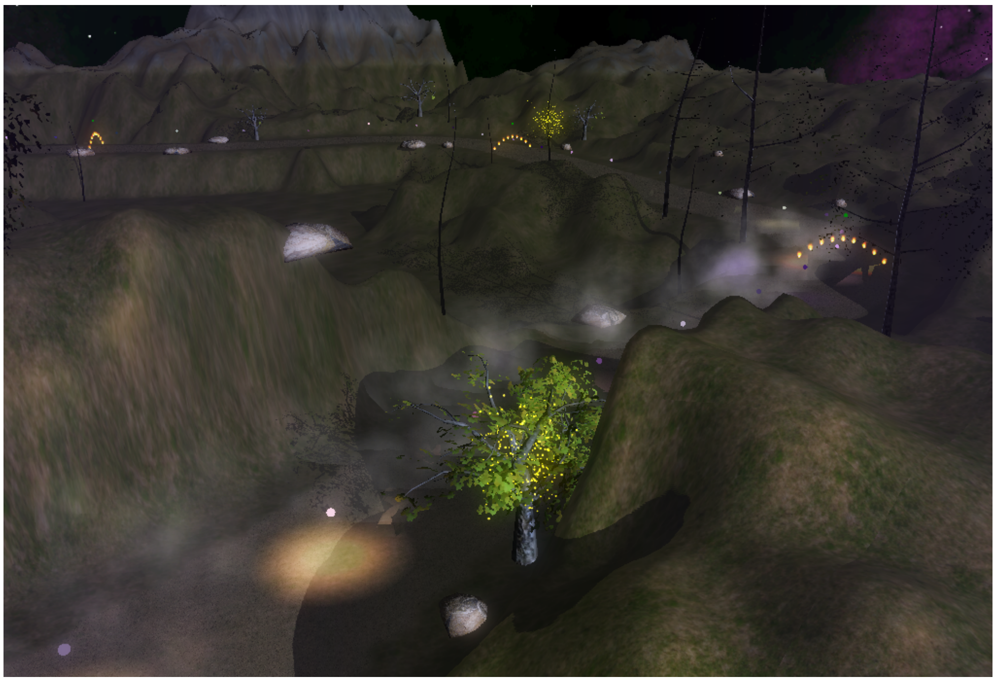


Figure 3.5: A piece of the track with an example the interpolation between the edges of the track and the terrain in the foreground.

Due to the low number of nodes, the height difference is applied over a large distance, and it is thus hard to notice at times. An additional drawback due to the low amount of nodes is the sparse occurrence of sharp turns. An increase in the number of nodes could result in a more action packed gameplay with more tight corners to pass, as well as small hills and inclines that result in the cars jumping into the air.

3.3 Navigation Mesh

The ability to walk around an environment seems quite trivial, but in graphics simulations, objects have to be defined as solid so other objects can collide with them. Collision detection consists of programming simulations, and with simpler computations the game can update faster. However, simplifying the terrain too much can in many cases compromise the visual quality that comes with a detailed model. Replacing rough areas of the terrain with an approximated surface and proceed with calculations on this plane will drastically reduce the number of operations required to determine a collision with an object. This approximation for collision detection is provided by a navigation mesh (Hale and Youngblood, 2011), the usage of which will be detailed in this section.

3.3.1 Generating an Approximation

To completely remove the rendered terrain from the equation and replace it with a virtual model on which collision detection is performed is a common way to minimize costly collision detection. The collision model is constructed using the track curve, either pre-calculated or generated dynamically during execution (Hale and Youngblood, 2011). The resulting model is not rendered, and only handled logically in the game code.

The term navigation mesh suggests that the approximated terrain is a mesh (see Section 4.1), but the representation can also exist in other shapes. Another case is where a navigation mesh is made up from a graph with nodes and edges, allowing widely known graph algorithms such as shortest path to be used.

3.3.2 Results

The navigation mesh used in the game is modeled as any other three-dimensional object which the car can perform collision detection against, and is used for keeping the player on the surface of the terrain and within the borders of the track. Thus, the navigation mesh is used as a tool for aiding user controlled input as well by keeping the player from leaving the level.

In Night Racer, the navigation mesh is represented as a mesh, and it is pre-calculated from the same race track data used to generate the terrain, so the car appears to follow the terrain when it is in fact following the navigation mesh, even though it is not rendered (a visualization can however be seen in Figure 3.6). The quality of the navigation mesh can be adjusted, where a higher quality is required for racetracks with sharp turns but a lower quality mesh is sufficient for the race track used in Night Racer.

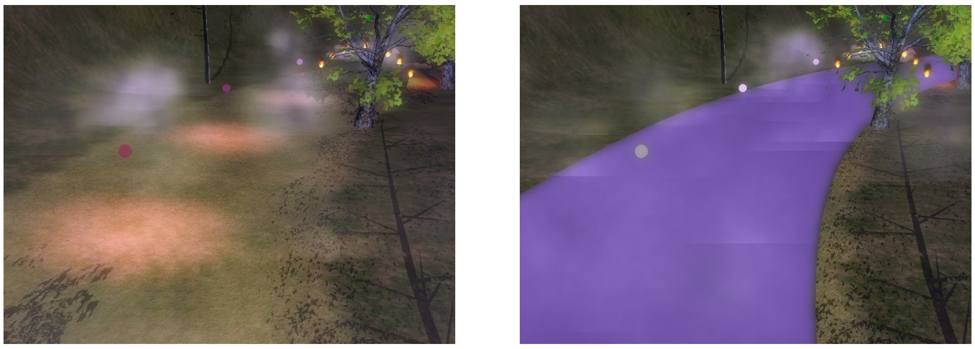

(a) The race track in the game, as normally seen by a player.
(b) The navigation mesh seen on top of the track, not visible by the player.

Figure 3.6: The navigation mesh used in Night Racer.

Night Racer contains no real physics engine, and as such, the navigation mesh is used in the game to allow for the simple collision detection needed for a car game. The behavior of the car is to merely follow the surface of the navigation mesh and no collision detection with other objects is provided. This does adversely affect the playability, especially in multiplayer game modes, where tactical collision with opponents may provide gameplay advantages. In future development, however, a physics engine could be integrated reusing the current mesh and this explicitly programmed collision detection could be replaced.

3.4 Seed

Randomly generated numbers are often the basis of any event based on chance in programming. These numbers are provided by a pseudo-random number generator. Generators follow a pattern and will output a sequence of numbers depending on a number (*seed*) that is provided to the generator. If it receives the same seed, it will output the same sequence of numbers (Knuth, 1997). This seed is conventionally the system clock and the sequence of numbers is thus perceived as practically unique each time.

3.4.1 Results

Night Racer uses a single universal number generator, whose seed is set at launch. The seed in the universal generator can be changed after startup, and since the levels of the game are calculated using this generator, it can be used as an effective way to distribute a level in the multiplayer mode. Instead of distributing the entire height map, the curve of the track, and the position of every piece of the scenery, a single number can be distributed. This is an incredibly efficient way of handling the data, the downside being that every player has to generate the level.

3.4.2 Discussion

The seed mechanic has several potential uses. A seed can be stored in order to have permanent access to a level and in effect creating a static set of levels to use in a future campaign mode, or as a way to challenge your friends. In the latter case, a high score list could be tied to the seed and thus offer a base for a competitive scene.

4

Real-time Graphics

Render engines used in games are different from those used in 3D modeling packages as they need to render an image every game tick. Such applications need efficient real-time graphics techniques to achieve smooth results. Therefore, these techniques need to find approaches to simulate the real world in a realistic way while keeping the frame rate high.

This chapter will start by giving an introduction to the composition of models and the graphics pipeline used in real-time graphics to render the frames. The remaining sections detail components used in Night Racer to achieve visually pleasing graphics, as it is one of the goals of this project. These components are lighting, shadows, reflections, sky visualization, texturing, particles, and post-processing effects which will be examined using a similar structure; this includes a short background, the results in the game, motivations of the techniques chosen, and a discussion of what could have been improved.

4.1 Models

Three-dimensional objects, known as models, are basic components creating the foundation of computer graphics by simulating the physical entities of the real world. Models are built in a 3D modeling application and then exported to be used in the game. The models consist of triangles with mutual corners forming a mesh, which are stored in a sparse data structure. The models are not volumetric, and instead they only represent what can be seen, which is the outer shell of the object.

This section will present some of the theoretical concepts that are necessary in the development of a 3D game and will be an aid in presenting the subsequent parts of this chapter.

4.1.1 Vertex Data

The corner of each triangle is called a vertex. In the simplest form, the vertex describes a point in three-dimensional space, stored as a vector, but as represented in a programming language, the data structure is very expandable, thus enabling any data to be tied to it. Among these data types is the vertex normal, defining where the vertex is facing.

4.1.2 3D Modeling Packages

The creation of models used in real-time applications are usually not performed in code, as writing code is not a very intuitive way to create something aesthetic and are thus made in a whole other process. Autodesk 3ds Max, Autodesk Maya, and Blender are examples 3D modeling packages which are used when modeling assets to games and film.

4.2 Graphics Pipeline

The graphics pipeline describes the process of producing a two-dimensional image that can be displayed on a screen from a three-dimensional scene, viewed from a virtual camera. When the pipeline was first introduced nearly 40 years ago the functions were fixed and an integrated part of the graphical processing unit (*GPU*) (Houston and Lefohn, 2011). Nearly a decade ago, the programmable pipeline was introduced which allowed for much more control over the pipeline. The programmable parts, known as shaders, are small programs with instructions to the GPU. It is important to know how the graphics pipeline is constructed in order to understand how shaders can be used. This section will describe what stages are included in the pipeline and the potential of programmable shaders.

4.2.1 Stages of the Pipeline

In a pipeline, the process needs to be completed sequentially. The three stages of the graphics pipeline are the application, geometry, and the rasterizer stages (Akenine-Möller et al., 2008), see Figure 4.1.

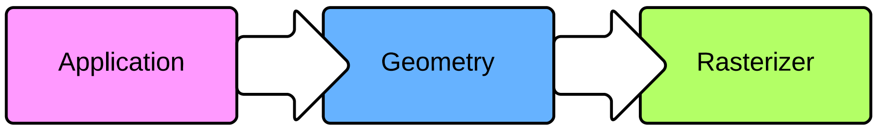


Figure 4.1: The three stages of the graphics pipeline.

Application The application stage of the pipeline is executed on the CPU and is fully programmable. This part of the process is able to handle operations such as animation, collision detection, and physics, but it is the developer's responsibility to implement. The main objective of the application stage is to determine what triangles, lines, or points (see Section 4.1) should proceed to the next stage, the geometry stage. As the implementation is left to the developer, the performance is dependent on the chosen algorithms.

Geometry The geometry stage is divided into sub-stages which handle different data operations from the application stage. This stage operates on the GPU, but can be controlled with programmable shaders. The main operations in the geometry stage relevant to this thesis include *transforms*, *vertex shading*, *clipping*, and *screen mapping*.

Initially, the incoming data is transformed from model space to view space via world space. Model space is the coordinate system in which the object was modeled. World space is the coordinate system in which all objects are when they are transformed in the scene, and view space is the coordinate system used when viewing the scene from the virtual camera.

For example, when a car model is positioned, scaled, and rotated in the world, the model's coordinates need to change from the local coordinates (model space) to the global coordinates (world space). When the car is seen from the camera, the coordinates change to view space. These transforms are calculated with matrix multiplications and are illustrated in Figure 4.2. The objects built with the data may then be shaded by applying a shading model, covered in Section 4.3.4.

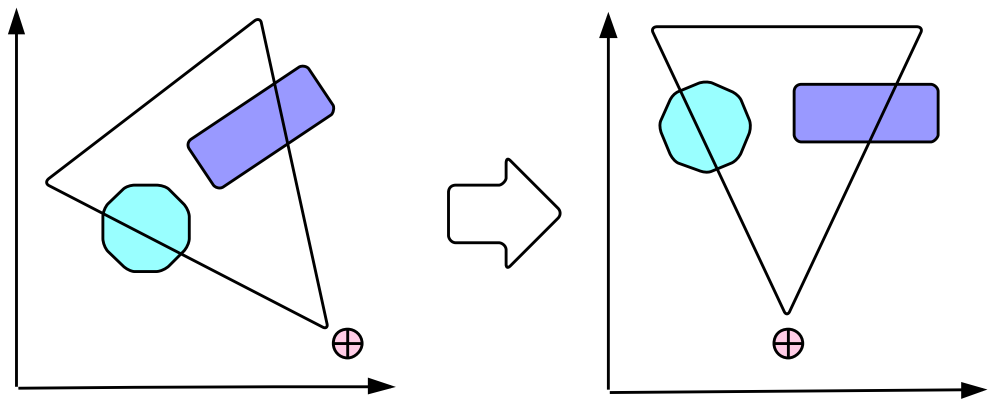


Figure 4.2: The scene as seen with world coordinates to the left and view coordinates to the right. The camera is represented by the circle and the triangle is the area visible to the camera, called the viewing frustum.

The next operation is clipping, which clips the scene so that only the parts visible to the camera are sent to the next stage. Finally, the coordinates are transformed to be adjusted to the screen.

Rasterizer When all geometry operations have been performed, the rasterizer converts the scene to colors for each pixel of the screen. The rasterizer traverses all triangles in the scene to find which pixels overlap the current triangle. After the traversal, shading models which operates on a per-pixel basis, can be applied to each pixel to compute the shaded color. If several triangles overlap the same pixel on the screen, all pixels are stored and merged to a color buffer with a technique known as the depth buffer algorithm to determine which pixels are in front and should thus be visible. The content of the color buffer is finally displayed on the screen.

4.2.2 Shaders

The two programmable shaders, vertex shader and pixel shader, make it possible to control operations in the graphics pipeline. The vertex shader is called once for each vertex in the scene, allowing the developer to use per-vertex shading. The most important calculations in the vertex shader are the transforms described in the geometry stage in Section 4.2.1. The output from this shader is then interpolated to get input to the pixel shader which is called once for every pixel. The pixel shader has the ability to decide which color each pixel should have based on the input information such as texture coordinates and normals.

4.3 Lighting

Lighting is an important part of making a three-dimensional scene look realistic, as light is essential for the ability to perceive shapes of objects. As such, lighting is one of the most researched areas of computer graphics, but it is computationally expensive (Sanchez and Canton, 2003); thus, simpler reflection models (or lighting models) are used to simulate real lighting. This section will examine how lighting works and how that is applied to real-time graphics. Furthermore, the lighting results of Night Racer will be evaluated in regards to aesthetics and performance.

4.3.1 The Rendering Equation

The light distribution in a scene can be described mathematically with the *rendering equation* (Immel et al., 1986; Kajiya, 1986). In a point x, the reflected light $L_r(x, \omega_o)$ in the outgoing direction ω_o is given by the following formula.

$$L_r(x, \omega_o) = \int_{\Omega_c} f_x(x, \omega_i, \omega_o) L(\omega_i) cos(N_x, \omega_i) \mathrm{d}\omega_i$$

The reflected light depends on three different variables; the material's bidirectional reflectance distribution function (BRDF), first defined by Nicodemus (1965) describing the material's reflective behavior, the amount of incoming light $L(\omega_i)$ to a point x in the scene from a direction ω_i, and the cosine of the normal in the point x and the incoming light direction. These attributes are integrated over all incoming directions ω_i from the hemisphere around the point x.

The goal is to fulfill the rendering equation and to calculate the reflected light in every point, if trying to achieve photo-realistic lighting. In real-time applications, the goal of photorealism is hard to achieve due to the large amount of calculations involved, but simpler lighting models are often used to approximate reality.

4.3.2 Light Sources

Light sources are entities which emit light. The light is typically in the visible range, meaning a wavelength of about 380 nm to 780 nm (Bao and Hua, 2011); outside this range, light is not visible to the human eye. In computer graphics, the simulated light is visible and is represented by a color and a light value called irradiance.

The *directional light* is the easiest light source to simulate (Akenine-Möller et al., 2008). This type of light source is best described as distant lights, like the sun, where the light beams are almost parallel to one another. These light sources are defined by a direction and an irradiance value.

Two other types of light sources are *point lights* and *spotlights*. These also have an irradiance attribute, but they differ from the directional light as they have a falloff (where the light intensity decreases). A point light can be defined by a position and radius, similar to a sphere, and the spotlight is similar to the directional light but the light is only radiated within a cone with a position and range.

4.3.3 Reflected Light

As stated in Section 4.3.1, there are several lighting models to approximate real lighting and one of the simpler models is the *Phong reflection model*, introduced by Phong (1975). The following paragraphs describe this model with small variations.

The light in a scene is either emitted or reflected. If an object does not emit light like a light source, it is reflective and can be described with three different components: ambient, diffuse, and specular lights. These components contribute to the total light in different ways depending on the material properties of the reflected object.

Ambient light, the first component, is the background light in the scene. When the light is reflected in the scene, a small amount of light is contributed to every surface even if it is not exposed directly to a light source, so called indirect light. This component is typically simulated by a constant color value, see the leftmost circle in Figure 4.3.

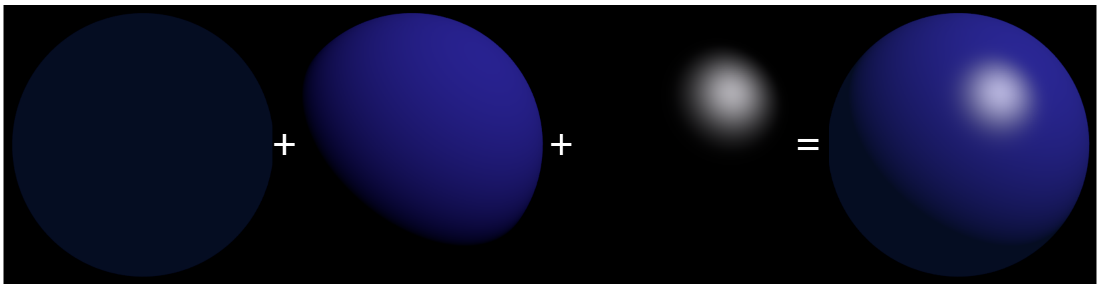


Figure 4.3: The ambient, diffuse, and specular component, in order left to right, of the light separated and then summed. The components are part of the Phong reflection model.

The second component, diffuse light, is the light which reflects equally in all directions from a surface. It arises from the fact that the light is reflected on the surface but continues iteratively beneath it and is reflected once again but with another angle, causing scattered light. The angle of incidence is important when calculating the intensity of the diffuse light (Sanchez and Canton, 2003). The intensity follows Lambert's cosine law, stating that it is proportional to the cosine of the angle of incidence and the normal of the surface (Smith, 2007), as seen in Figure 4.4. That means full intensity when the light is parallel with the normal and almost no intensity when the light is perpendicular. A surface which follows this law is called Lambertian surface. The diffuse light is one of the greatest contributor to the realism of the scene, and without diffuse light it is hard to sense form of three-dimensional objects.

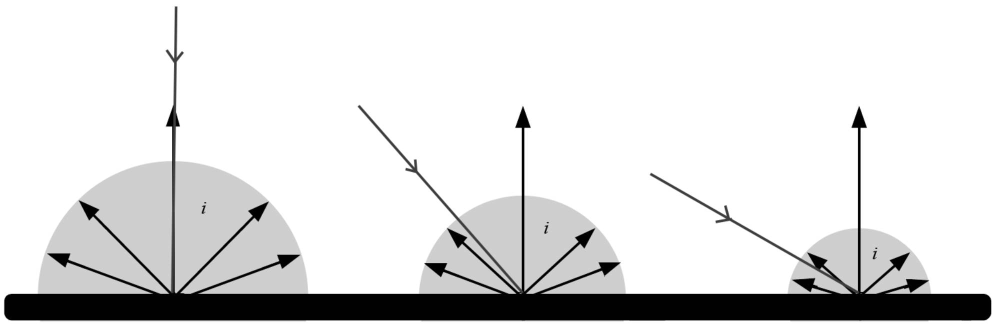


Figure 4.4: With diffuse light, the reflected light is scattered in all outgoing directions. The intensity i, illustrated in gray, of the diffuse light is calculated with the cosine of the surface's normal and the angle of the reflected light ray. The closer to the normal the ray is, the higher the intensity. The images are ordered by intensity, starting with highest intensity to the left.

The third and final component is specular light; the light which is reflected on a surface without scattering. Scattering means that the light comes from a single direction and is reflected in another, in contrast to the ambient and diffuse light which are reflected in every direction. Specular light causes the highlights in a scene, as illustrated in Figure 4.3.

4.3.4 Shading Models

Another type of model, shading models (not to be confused with shaders), describes how to determine the shading of a scene. Shading is used to give depth to three-dimensional objects to make it possible to perceive forms. Two categories of shading exists: flat and smooth shading. The difference between the shading models is what data is used for the lighting model.

Flat shading is the easiest technique to implement, but it does not give realistic results on smooth objects, as illustrated by Figure 4.5. Flat shading uses each surface's normal and the light direction to calculate how to shade the surface. Because the technique uses the same data for all points on a surface it will be shaded in the same color.

To give better results to smooth objects, smooth shading can be used. Smooth shading interpolates data to have more than one shade on each surface, thus creating an illusion of smoothness (Sanchez and Canton, 2003). However, this is more computationally expensive than flat shading due to the need for interpolation of data.

Gouraud (1971) first introduced the Gouraud shading. This shading calculates the lighting for each vertex in a triangle by using the same technique described by flat shading, but needs to interpolate the normals of the surrounding surfaces' normals to calculate the vertex normal. The results are then interpolated to calculate the shades of the surface's points. As an effect of the interpolation of the normals, Gouraud shading does not handle local specular highlights well.

Soon after the Gouraud shading was introduced, the Phong shading was described in the same paper as the Phong reflection model (Phong, 1975). Phong shading is the most popular shading technique and takes care of specular highlights better than Gouraud shading (Sanchez and Canton, 2003). This technique interpolates each surface vertices' normals to calculate a normal to each point on the surface. The lighting of each point is then calculated by using a lighting model, and in contrast to Gouraud shading, the results are not interpolated. This technique gives the best result to all kind of objects, including smooth surfaces (see Figure 4.5), but numerous calculations are involved.

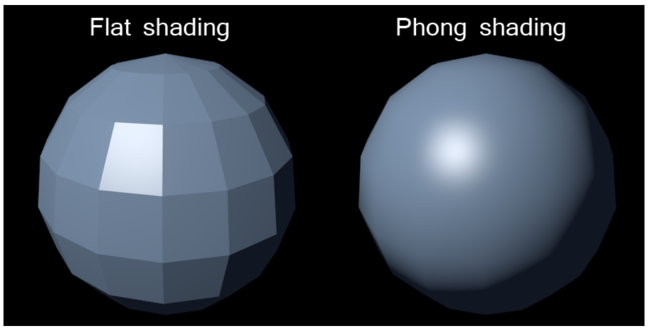


Figure 4.5: A comparison of shading models. Flat shading is shown on the left and Phong shading on the right.

4.3.5 Deferred Lighting

Deferred lighting, or pre-pass lighting, is a technique for handling multiple light sources. The technique is used in many commercial games, for example in the Ubisoft game engine called AnvilNext (Bertz, 2012). The naive technique to handle multiple light sources is to calculate the lighting on each object by adding the effect of all light sources. This has the disadvantage that numerous lighting calculations are performed even when the triangles are not visible (see the depth buffer technique in Section 4.7.6). Deferred lighting eliminates these problems by deferring the calculation of the light contribution until after the visibility check (Liang et al., 2000).

Quandt (2009) describes the technique in the following steps. First, geometry data from the scene is rendered to a depth texture and a normal texture. The light sources are then calculated as simple geometry objects (using a sphere for point lights) with the data from the textures, and thus light only needs to be calculated for the visible pixels. The light contribution is rendered to a light texture which is applied to the objects in the scene instead of calculating the light when rendering.

4.3.6 Results

As one of the requirements for this project is that the game shall be visually pleasing, Phong shading together with the Phong reflection model was implemented. This combination is almost always used in a project of this kind because the results of the other combinations are not satisfying enough.

Night Racer uses one directional light to simulate the moonlight, which changes color and intensity to match the pulsating sky. The rest of the light sources in the game are point lights positioned above the track. The point lights change colors to match a pattern where the lights appear to move forward in the intended race direction as illustrated by Figure 4.6. At the start of each race, all point lights start to visualize the count down by changing from red to green in four steps, see Figure 4.7.

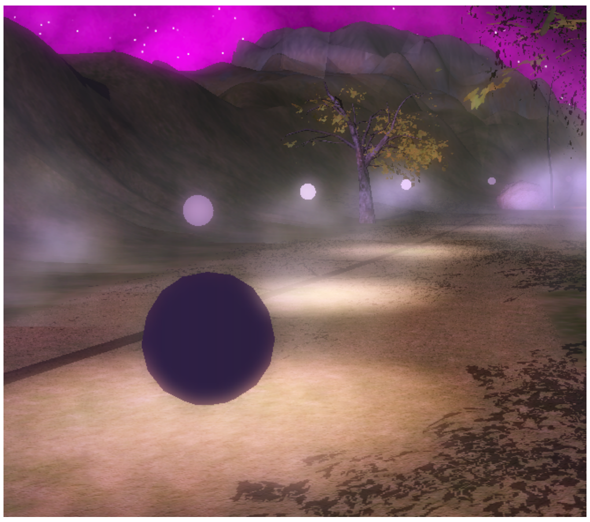
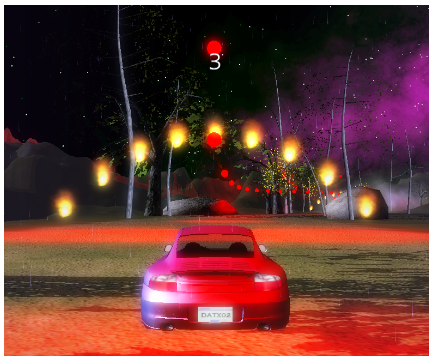
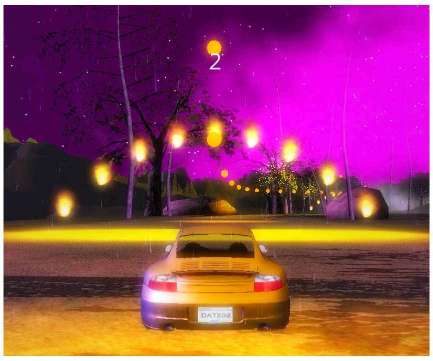


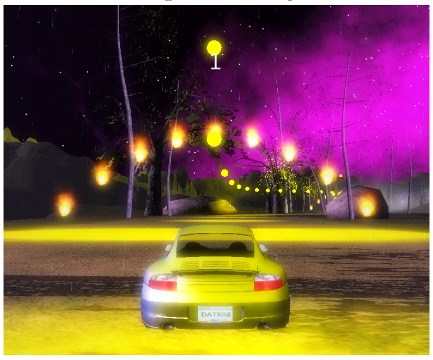
Figure 4.6: The point lights above the track indicating the intended direction of the race with the change of color.



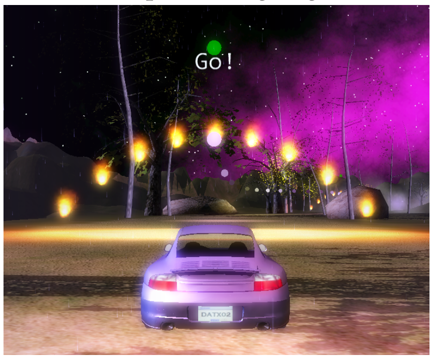
(a) Step 1: Red lights.



(b) Step 2: Orange lights.



(c) Step 3: Yellow lights.



(d) Step 4: A green light, with the rest of the lights in default mode.

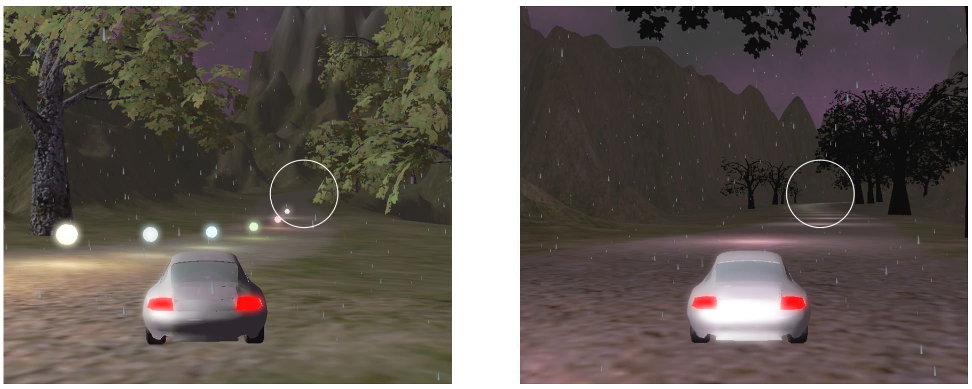
Figure 4.7: The results of the countdown lights.

Early in the project, lighting on each object was calculated by adding the effects of all light sources in the scene. This proved to be too complex and allowed up to approximately ten light sources affecting the objects at the same time. The light sources were then sorted by distance to the car to find the ten closest light sources to use. This worked when observing the car, but with the effect that distant light sources were not contributing to the lighting of the scene, as illustrated by Figure 4.8a.

The solution to the problem with multiple lights was to implement deferred lighting. This technique, in combination with the high-end target hardware, allowed all objects in the scene to be affected by up to about 200 light sources. Using this technique, all light sources are contributing to the lighting in the scene and the environment looks more realistic than before (see Figure 4.8b).

4.3.7 Discussion

The lighting used in Night Racer resulted in a satisfactory graphical effect after the switch to deferred lighting as mentioned in the problem statement of this thesis. Many iterations of lighting implementation were needed before the decision to use deferred lighting was made. However, this approach is not without problems as the scene was needed to be rendered again with geometry data, and thus performance was decreased. Deferred lighting is still performance-wise the best solution of the techniques tested.



(a) Due to the limit of ten light sources, the lights closest to the car were used when calculating the lighting. The circle shows the intended position of the distant lights which are not contributing.

(b) Deferred lighting allowed more light sources to affect objects in the scene and the distant light sources are now contributing.

Figure 4.8: The results of the deferred lighting.

Another solution to get even better performance can be to use a static approach where lighting is calculated and saved to a texture once in the first frame, and on subsequent frames the information in this texture can be used to light the objects. This works only if the light sources and the objects to be lit are static and thus not move. However, even though all the lights in this game are static they change their color every frame, thus static light textures would not work in this context.

Aesthetically, the point lights could have been a more integrated part of the environment rather than only be positioned above the track to guide the player in the right direction, but the focus has been more on the actual light effect rather than the light models themselves. An improvement to the lighting of the scene would be to implement spotlights. These could be used to light up the track when driving with the car's headlights, but the lights already cover large parts of the terrain and the track. Thus, the spotlights would not give the same effect as if the world was unlit.

4.4 Shadows

Imagine a world where objects do not interact with one another: mirrors would be empty, film projectors would leave movie screens blank, and shadows would never be seen on the wall. This is the world if only the most basic functionality of the graphics pipeline was used. To make realistic looking graphics, programmers try to replicate these physical phenomena rather than using actual physical formulas to represent the world, as that would require more computing resources than is viable in today's systems. There are many phenomena that have to be considered, commonly gathered under the category Global Illumination or *GI* (Akenine-Möller et al., 2008).

The manner in which physical objects cast shadows on other objects is an example of a GI phenomenon commonly implemented in real time graphics to provide a sense of depth to a scene by indicated how objects are positioned in relation to each other (Akenine-Möller et al., 2008). This section will detail the shadow techniques considered, and present the way shadows have been implemented in the game.

4.4.1 Shadow Simulation

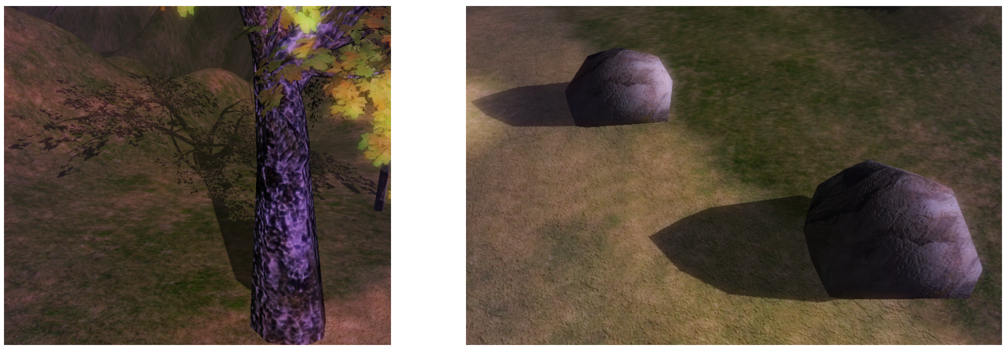
Lighting plays a key part in the shadow phenomena, where it is the lack of light that fills in the contour of the objects that block the light sources. However, calculating light as individual rays as in the study of optics is unsuitable for most real-time graphics applications due to high computational costs, which in turn forces programmers to use different algorithms for providing shadows.

Shadow mapping One of these algorithms is shadow mapping. Shadows in a three-dimensional scene calculated by the shadow mapping algorithm are evaluated in a set of steps, starting with information gathering. The first step of the process consists of projecting the shadow caster's distance from the light source onto a texture called the shadow map, which is then provided to the rendering stage. In the pixel shader of the rendering stage, the distance to the light from the current pixel and the distance of the corresponding pixel on the shadow map is evaluated. If the distance of the shadow map is lesser, the rendered pixel thus lies behind the pixel in the shadow map, appearing in shadow.

Shadow volumes Another shadow technique is the shadow volume. Much like the shadow mapping algorithm storing shadow data in a texture, the shadow volume constructs shadow geometry by projecting the shadow caster's vertices to create a volume behind the object. This shadow volume is then used in the rendering stage by evaluating whether the point is inside it, and if so, is covered in shadows.

4.4.2 Results

Shadows have been implemented in Night Racer (see Figure 4.9) and are calculated using the shadow mapping algorithm. Due to the scenery and the numerous visual effects implemented, a static approach is used instead of a dynamic one to conserve computing power. This means that the shadow map is only calculated for the first frame, and then reused for all subsequent frames. The light source used for the shadow map is the directional light implemented to simulate the moon, and the shadow maps are calculated separately for each terrain segment (see Section 5.2) so that the resolution of each shadow map segment can be of a higher than if a single map was calculated for the entire terrain.



(a) A tree casting a shadow on the ground. **(b)** A simpler object casting a shadow.

Figure 4.9: The shadows in Night Racer.

4.4.3 Discussion

The shadow algorithm was chosen because of the relatively easy implementation, but still achieving the goal of visually pleasing results. However, the algorithm was first implemented with dynamic lighting in mind before the decision to use the static approach was made. The method of working in iterations was important as the final approach proved to be a good optimization, allowing a higher frame rate. On the other hand, the static approach is the reason why the car does not cast shadows on the ground as the car is dynamic, but the decision was made that the shadows of the car could be separated from the other shadows in the future.

The choice of a high shadow map resolution by calculating one shadow map for each terrain segment allows for smoother shadows, compared to the jagged results of low resolution maps. However, the splitting of shadow maps proved to be problematic when applying the calculated shadows onto the car because of the movement between several terrain segments, making it hard to find which segment's shadow map should be used; for example, when the car transitions from one segment to another, the shadows on the car need to be calculated from both the corresponding shadow maps.

4.5 Reflections

Reflections help in making a scene look more photo-realistic, and are thus a natural part of a visually pleasing game. This section examines environment maps used for reflections and describe their two different formats (cube maps and sphere maps). Furthermore, a technique to generate maps in real-time known as dynamic environment maps will be studied in order to evaluate which techniques are most suitable for the purpose of this game.

Regarding the implementation of reflection, the implementation of planar reflections and reflections in curved surfaces differ. Reflections in curved surfaces can be implemented by generating an environment map, and using the viewer's angle to the object, texture lookups on the environment map are performed to see what should be reflected in the object. Planar reflections are often implemented by rendering a copy of the object mirrored in the plane.

4.5.1 Environment Maps

Blinn and Newell (1976) first introduced the technique called environment mapping. An environment map contains information about the environment around the objects, and it is used when calculating what a curved surface reflects. Without too much effort, an environment map can make a scene more visually interesting (Llopis, 2009). It is important to note that the goal is not to have physically accurate reflections, but rather to trick the eye to believe the reflections are real (Andaur et al., 2002). Reflections are view-dependent, meaning a surface reflects differently when looking at it in different directions. Each format of environment maps have specific mathematical methods to calculate what is being reflected.

Cube maps is one of the formats of environment maps. A cube map is a texture where six faces of a cube are represented to get a full environment representation, and can be seen in Figure 4.10. The cube map generation is view-independent and can be generated once regardless of the view direction. The first step to get the reflection from a cube map is to trace a ray from the camera to the surface. Then the reflected ray is calculated based on the incoming ray angle and the surface's normal according to the formula:

$$r = d - 2(d \bullet n)n$$

where r is the reflection vector, d is the view direction, and n is the normal. Next, the largest absolute component of the reflection vector determines which face the reflected ray hits. When the target face is determined, the UV coordinates (see Section 4.7.1) are calculated by dividing the other two components of r with the largest absolute component.

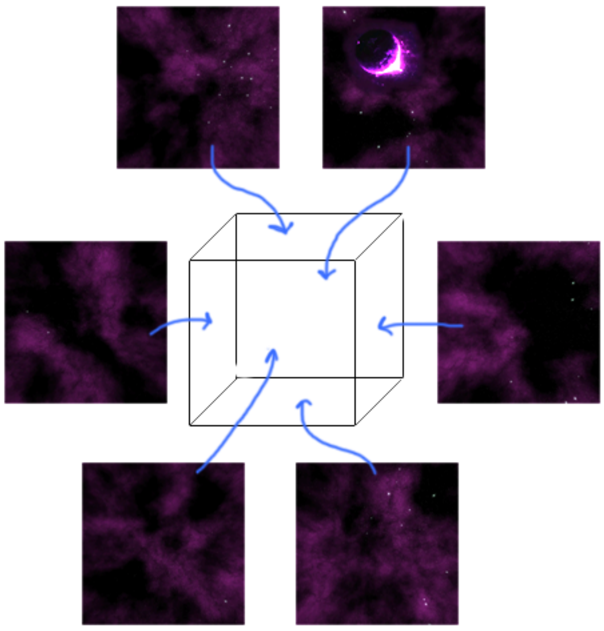


Figure 4.10: The six textures of a cube map used to calculate reflections.

Sphere maps is another format of environment maps. Sphere maps are textures representing the environment by a circle where the center is representing the environment in the forward direction and the edges in the backward direction (see Figure 4.11). Sphere mapping has one major drawback that may explain why the format is, in general, not used (NVIDIA Corporation, 1999); the drawback is that, in contrast to cube maps, sphere maps are view-dependent and need to to be regenerated every time the camera moves. Another drawback with sphere maps is that it is harder to generate the textures compared to cube maps.

Despite all the disadvantages of sphere mapping, they could still prove to be useful as a fall back in situations where cube mapping cannot be used. Cube mapping requires better hardware and it may not always be available (NVIDIA Corporation, 1999).

4.5.2 Dynamic Environment Mapping

Standard environment maps are static, meaning they are generated once before the application runs. In the case of cube maps, the result can be improved if the maps are generated in real-time.

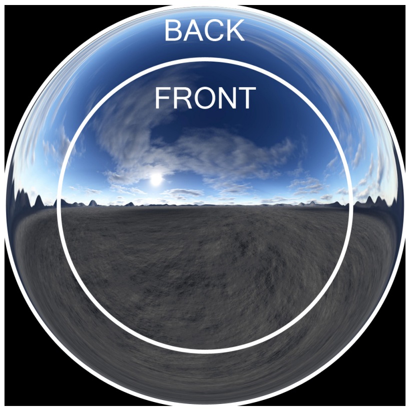


Figure 4.11: A sphere map where the center represents the front and the edges represent the back of the sphere.

Every generation will need six renders, but allows objects to actually reflect the real environment and not just an approximation generated outside the application. However, this approach is computationally expensive for the graphics card, as each time the generation is performed, six renders in addition to the default one are needed.

One way to improve the performance is to reduce the number of objects rendered in each of the six renders, by performing stricter frustum culling (described in Section 5.1.4) and choosing to exclude certain objects in the renders.

4.5.3 Results

In order to achieve visually pleasing graphics, reflections in the car were essential to get a look of a bright and smooth surface. None of the other objects needed these reflections because they are mostly organic non-reflective objects. Cube mapping was the implementation used in this project because the targeting hardware supports it, thus no alternative was for consideration.

One goal with the reflections on the car was to implement dynamic environment map to always reflect the actual surrounding environment on the surface. In the early development stage of the game, the ability to generate environment maps dynamically was implemented, but the technique requires powerful hardware to run smoothly. One solution to this problem, used in the final version of the game, was to use the the cube map representing the sky as an environment map. As seen in Figure 4.12, this proved to be sufficient enough, as there are no, or a small amount, of objects blocking the sky for reflections.

4.5.4 Discussion

Reflections are a good way to achieve a shiny look. The results of the dynamic environment mapping were visually pleasing, but without advanced hardware, the performance was lacking. By taking the cube map generated for the sky cube instead the performance was greatly improved. The static approach was satisfying enough for the goal of this project and the dynamic environment maps were not needed in the game because of what the environment looks like. The method of using XNA helped with cube maps as the framework contained support for saving and using them efficiently.

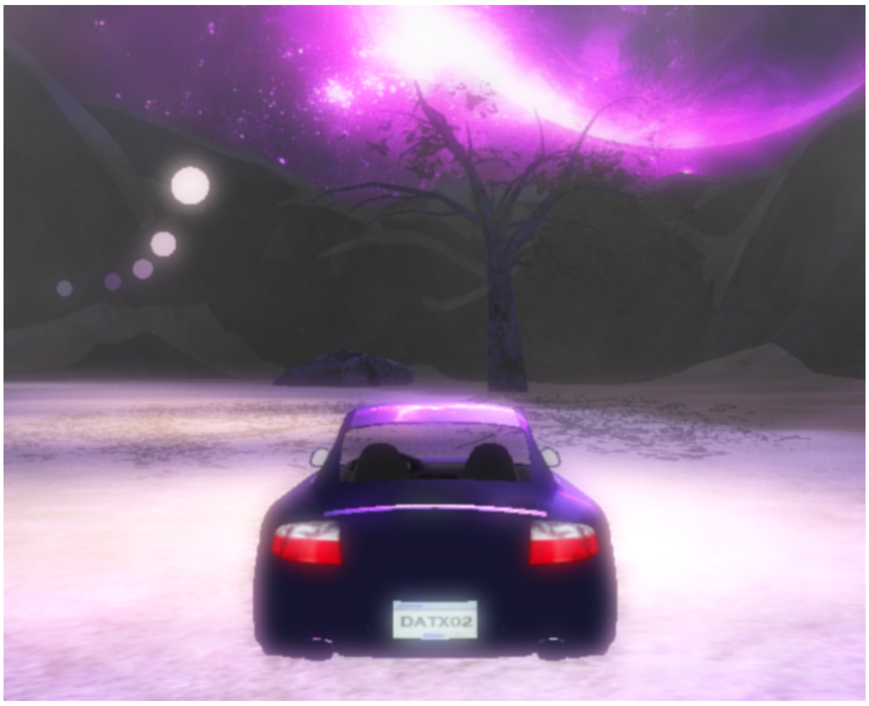


Figure 4.12: The sky is used as an environment map, and as a result, the moon and the stars are reflected on the car.

If more real looking reflections were prioritized, the performance when generating the dynamic environment map could have been improved. As described in Section 4.5.2, the six renders of the dynamic environment maps can be simplified to take less rendering time. The environment map could also have been generated less frequently and probably still get satisfying results.

Another improvement considered for implementation was to use several pre-generated cube maps for different sections of the world, and thus having better representations of the actual environment when driving. While improving the results per section, this technique would most likely lead to transition problems when traveling to another section.

4.6 Skybox

In a scene which simulates the real world, there is often some kind of terrain model to represent the ground, in addition to the other objects in the scene. This is one of the fundamental model representations to form an earth-like appearance. If the scene is not enclosed by objects (such as in a typical outdoor scene), all parts on the screen are not guaranteed to be covered and may show up as empty, which is the color used for clearing the display between frames.

To ensure that no part of the screen remains empty in the outdoor scenes of the game, some graphical entity can be used to envelop the world and then be used to cover up areas in which no other object exists. This entity could be referred to the sky of the game world, since it will be mostly visible on the top of the screen if the terrain and objects constitute the rest of the scene. The sky, however, is not a single physical object and the player can not move behind it or see it from the side, which makes the sky quite a unique part of the game.

This section will describe the implementation of the sky in the game, and evaluate its efficiency and visual appeal.

4.6.1 Sky Representation

Considering games often lack the presence of actual models in the sky, a simple simulated representation is typically sufficient. This can be accomplished using an enclosing object centered around the camera of the player, and does not change in appearance through rotation in relation to the world space.

This representation is called the skybox, or skysphere since the object may be a box with a cubical texture applied which could look similar to a sphere. Additionally, the skybox is rendered without being affected by light sources, so that it is not perceived as just another physical object in the scene. Since the sky needs to be in the far background of the scene, the rendering of the skybox is performed as the first object in the scene with no depth write value, so that all the other objects will be drawn on top of the sky.

4.6.2 Results

A skybox has been implemented in Night Racer, representing the purple sky envisioned for the game. The model was exported from a 3D modeling package, but could very well be constructed in code as it consists of eight vertices. The XNA standard class for texture cubes is used for constructing the skybox from the six textures making up the sky. The skybox used in the game is shown in Figure 4.13. However, what cannot be seen in the figure is that the sky is made to constantly pulsate, providing a more dynamic visual appearance. The pulsation follows a sine curve and adjusts the color intensity accordingly.

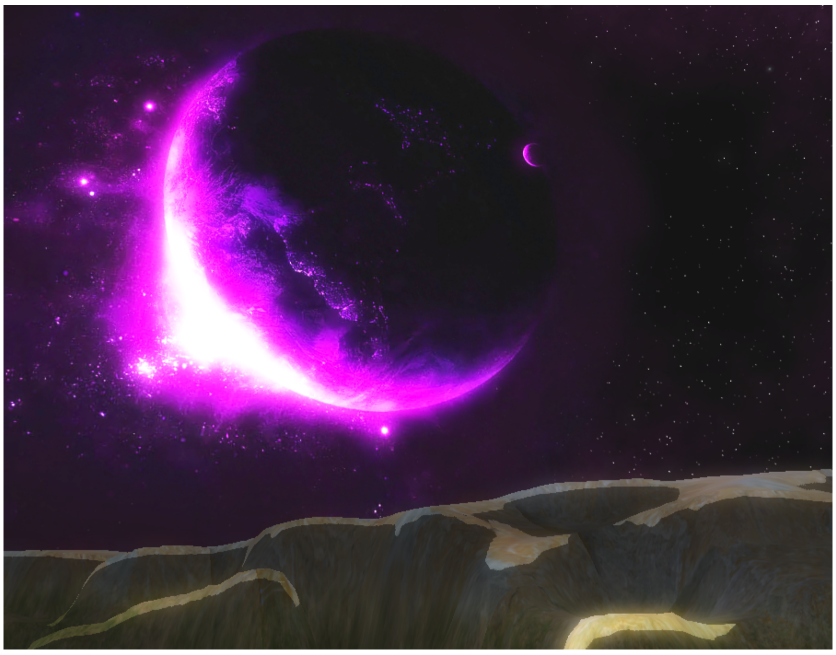


Figure 4.13: The skybox used in Night Racer.

4.6.3 Discussion

The modeling and implementation of the skybox used in Night Racer was a relatively easy task. The method of using XNA was a benefit in the implementation, since it provided a standard class which made the creation of the texture cube representing the sky of the game world much easier as it is equivalent to loading six 2D textures.

The sky used in the game greatly affects the visual impact of the game, and having it pulsate provides the sense of movement which was central to the definition of "visually pleasing" described as a goal (see Section 1.1). The sky further assists in portraying the fantasy universe which is the setting of the game.

4.7 Texturing

Texturing is a method to save data as a picture, every pixel corresponding to a data index. Data is stored as a color, which means it can have up to 255 units of precision (8-bit) in three variables, or four in the case of a picture with transparency. How the data stored in a texture is interpreted is defined in the code, and can thus be used to describe any data collection, but is most often associated with describing color.

Textures are applied to the surface of a model to describe it. A surface is not limited to a single texture as they describe different characteristics, thus several textures can work in conjunction to create the final look.

This section will focus on describing the texturing methods used to describe the surface of a model and the terrain. Considering a texture can be interpreted in any given way in the code this section will describe how the different types of textures are coded in this project.

4.7.1 Texture Mapping

As first pioneered by Catmull (1974), texture mapping is the act of applying the texture to the model's surface. A texture is stored in a two-dimensional space, and thus it needs to be recalculated into the three-dimensional space of the game. This is accomplished by projecting each pixel of the texture, or texel, onto the model according to predefined texture coordinates generated by the 3D modeling application. The texture coordinates are defined in UV space, representing the X and Y axes of the texture.

4.7.2 Diffuse Texture

A diffuse texture is one of the most basic types of texturing, containing the color description of a surface without any additional effects such as shadows or light reflections, see Figure 4.14. The texture contains the pure, unaffected, color of the object as three components, red, green, and blue (RGB).

4.7.3 Normal Map

Models in real-time graphics are often of a lower resolution than in other types of applications in order to save computational power. As a consequence, there exists a loss of structural precision, and in order to regain some of the loss a normal map can be used. A normal map describes

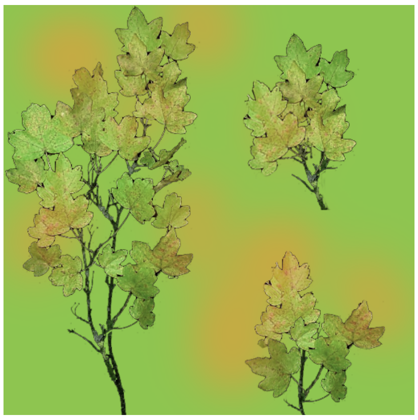


Figure 4.14: A diffuse texture used in the game which shows the colors of the leaves for one of the trees.

how a surface actually looks, by describing the normal of every point of the area, see Figure 4.15. This allows the model to look more complex than it actually is, by using the new normals when calculating how light affects the object, while still being a cheap method to compute.

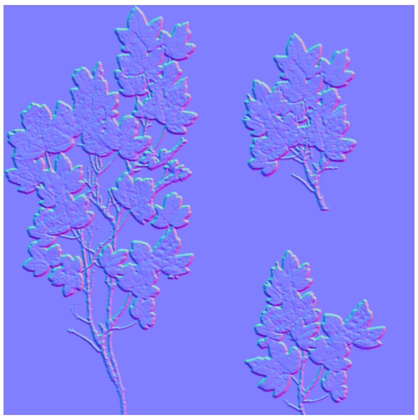


Figure 4.15: A normal map used in the game. The difference in color determines the direction of the normals, and thus the structure of the normally flat surface of the leaves.

A subgroup of the normal map is the bump map, which is a technique used for achieving a more realistic surface on objects, such as a wrinkled cloth or a dirt road. The effect is achieved by distorting the normals of the surface in pattern-like shapes, and thus creating a more lifelike appearance.

4.7.4 Alpha Map

The alpha map is a texture that represents the transparency of an object. Common implementations include saving the transparency as grayscale images or in the assigned alpha layer of an image with transparency. A black area is used to describe complete opaqueness, while a white area is used to describe complete transparency, see Figure 4.16. Values in between are increasingly more transparent going from black to white, and are blended together with the diffuse texture for the area.

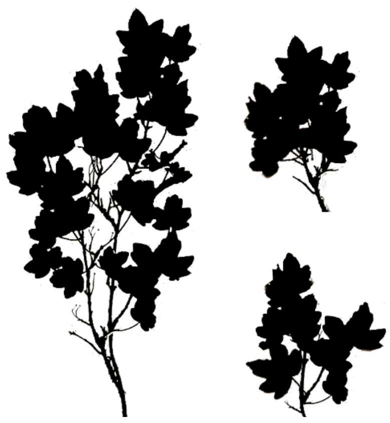


Figure 4.16: An alpha map used in the game. The black area outlines what parts of the diffuse texture should be visible in the game.

4.7.5 Texture Filtering

A texture's resolution is often equivalent to the largest possible version of an object that is going to be shown in a game. A common occurrence, however, is that the object is further away, which leads to the problem of the object the texture is mapped to occupies a smaller screen space than the texture was created for. The consequence of this is that several texels occupy the same pixel space, and the program needs to determine which texel to be used; filtering the available texels or blending them together. The result is often subpar and graphical artifacts may become apparent.

One of the solutions to the texture filtering problem is mipmaps, a commonly used technique to optimize rendering. Mipmapping is based on the concept of storing several versions of the same texture in the memory, each smaller than the previous (see Figure 4.17), and using the one best suited for the current situation. Despite storing additional versions of every texture the memory consumption is quite low, the increase is only about one third of the original size due to how the sum of the division scales.

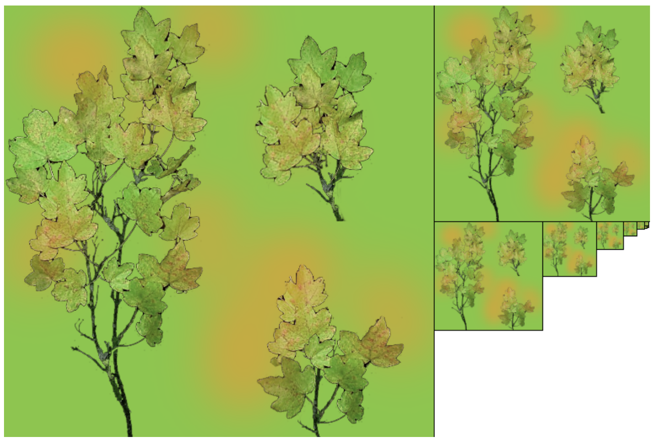


Figure 4.17: An example of a mipmapped diffuse texture. The smaller versions are tiled next to each other, and because of how they are tiled it is easy to see how the algorithm scales.

4.7.6 Depth Buffer

A depth value is generated for each pixel whenever an object is rendered. If another object were to occupy the same space, the value of their depths are compared, if the new object is closer to the camera it is chosen instead, while if it is further away it is discarded. The result is stored in a two dimensional array, and can thus be considered a texture. This is called depth buffer, as referred to in Sections 4.2.1 and 4.9.1.

4.7.7 Multitexturing

Multitexturing is not a texturing method, but a technique that can be utilized to create smoother transitions between textures. Most models have a texture mapped to them, but terrain often has a flexible texture determined by variables such as height in the world. This makes it possible to have all terrain above a certain height be covered in snow. Another determiner could be the slope of the terrain; a steep incline might be uncovered bare rock or dirt instead of a smooth grass texture.

Each vertex thus has a specific texture calculated and assigned, and if two adjacent vertices have different textures assigned to them there will be a very sharp edge between them (see Figure 4.18). To remedy this multitexturing can be used. For every vertex a weighted value is assigned to each texture, and on vertices where more than one texture carries a value the textures are blended together, see Figure 4.19.

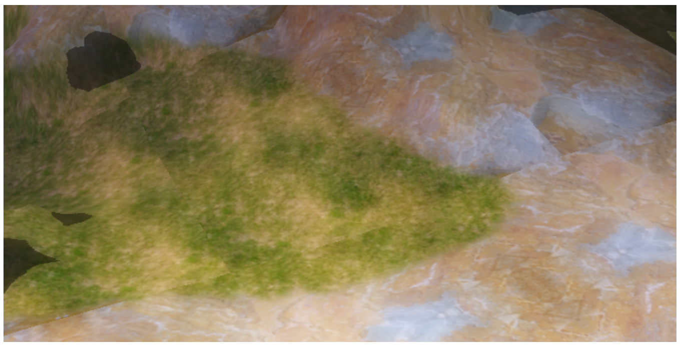


Figure 4.18: The terrain without the use of multitexturing. A sharp edge can be seen between the mountain texture and the grass texture.

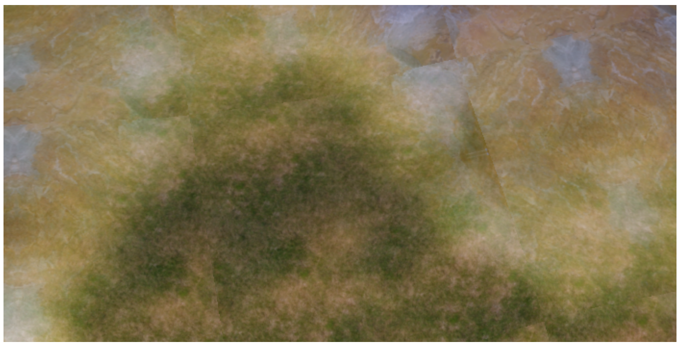
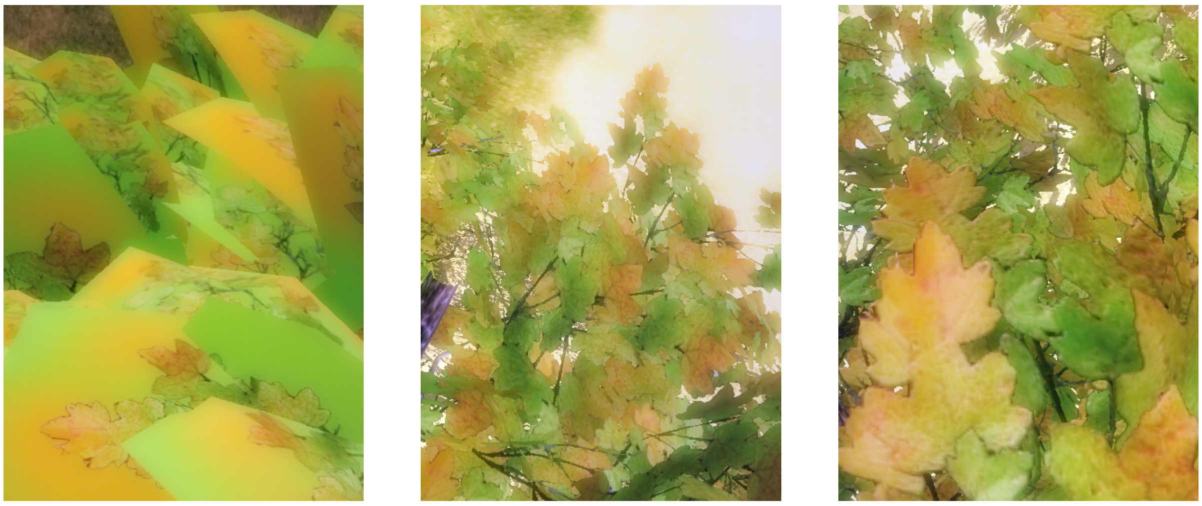


Figure 4.19: The terrain with the use of multitexturing. The edge between the two textures has been smoothened, creating a better transition between materials.

4.7.8 Results

Textures are an essential part of creating additional detail and depth to the game. Textures are at the same time an efficient format, and basic textures are thus of a relatively high priority to implement. That said, high-resolution textures are time consuming to create and offer little additional content in comparison to a lower resolution alternative. Since the game takes place in a low light environment, the quality of texture is not as noticeable as in a more well lit environment. Thus the quality of the diffuse textures has not been a high priority.

All objects in the game have at least a diffuse texture and a normal texture or bump map in order to achieve additional depth. One-dimensional objects, such as the leaves on the trees also have an alpha map to gain transparency where the object does not actually occupy any space, see Figure 4.20 for an example of these texture types. Mipmapping is automatically computed by the XNA framework when a texture is added to the Content Pipeline, and thus does not need to be created manually.



(a) Leaves with only a diffuse texture.

(b) Adding an alpha map to the leaves make the areas that are not leaves see-through.

(c) A normal map is added, resulting in the leaves gaining some additional depth.

Figure 4.20: Different textures applied to the leaves of a tree.

4.7.9 Discussion

The implementation of texturing in Night Racer is satisfactory, and all of the textures needed to provide a good game experience are included. A texturing type called specular map has not been discussed due to the low light environment of the game. This was an intentional choice due to the limited amount of additional detail it would provide for the game, but as a future implementation it could be a worthwhile pursuit.

The most definitive improvement on the textures in Night Racer would be to create textures of a higher resolution. An effort that would allow for not only additional details in the current game, but also a potential for an alternative setting, such as a more well lit environment or a different type of terrain without making the game look worse.

4.8 Particles

Some objects are more difficult to visualize with the use of models, because their shapes are irregular and their appearance is often perceived as random (although there is a physical explanation). This includes effects such as fire, smoke, water, and dust to name a few. Instead, particle systems are used in these cases. The first use of a particle system was described by Reeves (1983), and it is a collection of particles represented by a texture which are generated continuously. Each particle has its own set of attributes and lifetime and is typically independent from the other particles in the system. The particles can also be animated to get an effect such as pulsations. This section will describe the effects fire, smoke clouds, rain, and fireflies implemented with particle systems in this game and what context they were used.

4.8.1 Results

One of the goals with this project is to make the game visually pleasing, and one approach to that is to implement a set of particle systems that make the game look more interesting. The particle systems implemented in this game is presented below.

Fire The fire effect uses two particle systems, one for the fire and another for the smoke. The fire particle system was made up of red colored textures spawned with a random velocity upwards to simulate the rising flames. The additive blend state was used to give the effect of that it is hotter in the middle of the fire, because all particles are adding up their colors to get a brighter result. The smoke system is very similar but spawns gray particles with a bit higher velocity to get the smoke above the fire. The fire system was used in this game to indicate checkpoints along the race track, positioned in an arch over the track as illustrated by Figure 4.21. A detailed view of the fire effect can be seen in Figure 4.23a.

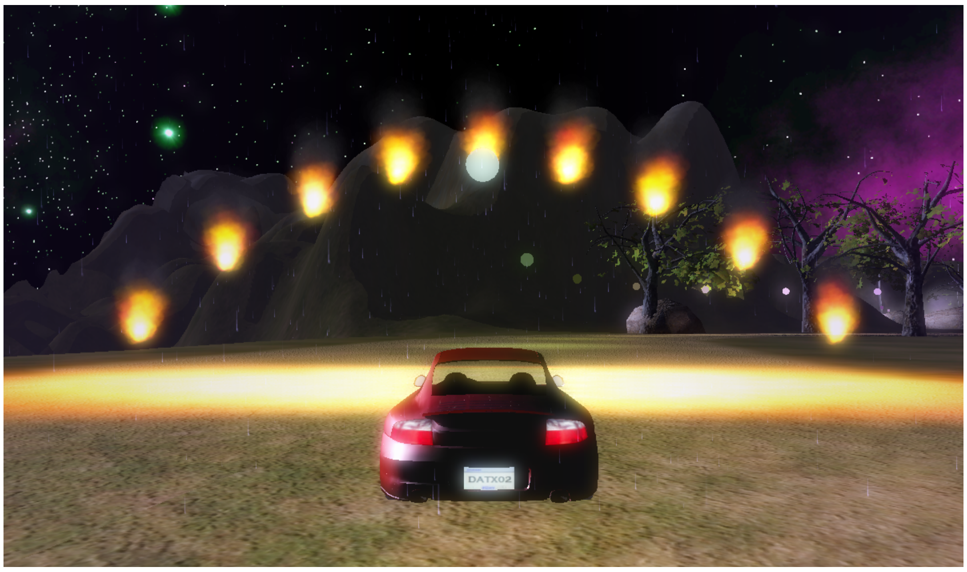


Figure 4.21: Checkpoints indicated by an arch of fire.

Smoke clouds An effect of large smoke clouds was implemented by a particle system with large particles instead of the more common small particles. Gray smoke particles were spawned in an area on the track to simulate fog with a velocity sideways to simulate wind. Figure 4.22 and Figure 4.23c illustrate the effect.

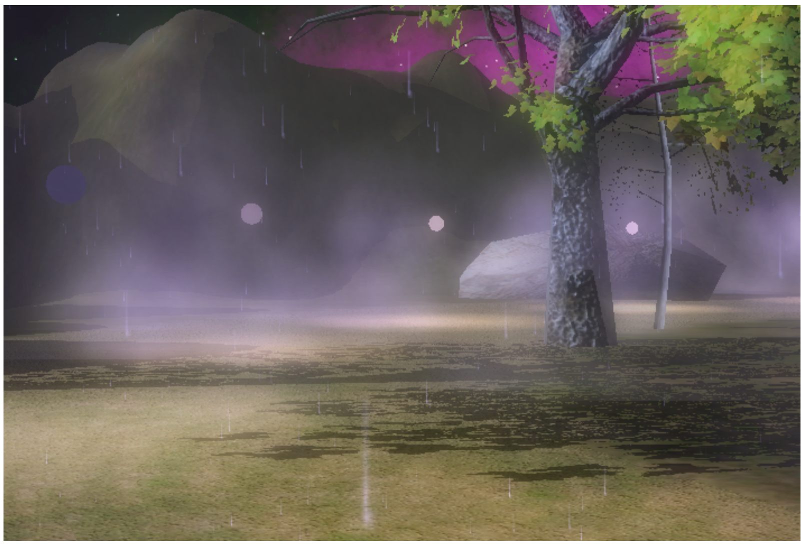


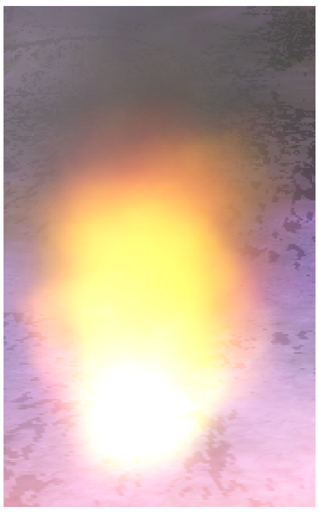
Figure 4.22: Smoke clouds along the track.

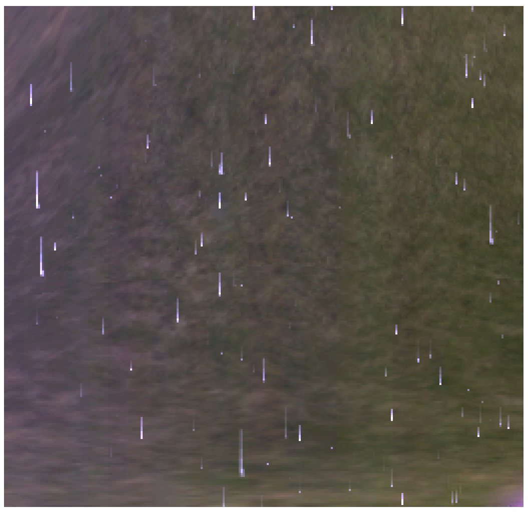
Rain Rain was implemented by spawning raindrops in the air at random heights. To limit the amount of rain particles to use in the scene, particles are only spawned in an area around the car, providing the appearance of rain everywhere. The particle texture is made up of a blue straight line to simulate the motion blur when the rain falls down from the sky. See the final result in Figure 4.23b.

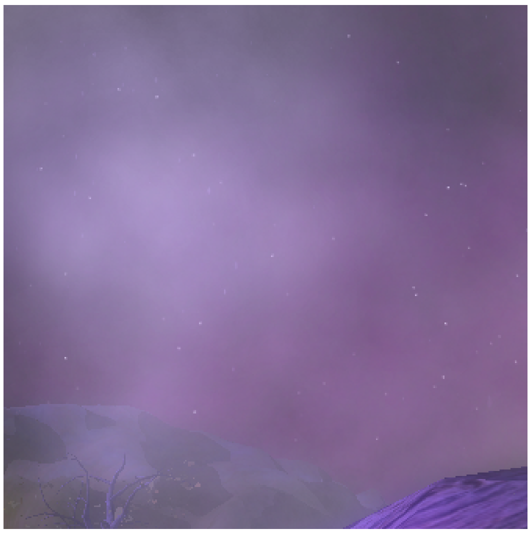
Fireflies Fireflies were created by a particle system with small round yellow particles. These are spawned around objects on the side of the road, such as the trees. New particles are given random positions within a bounding cube and random velocity in any direction. The particle system uses the bloom effect to simulate the bright lights (see Section 4.9.3). Figure 4.23d show the result of the implementation.

4.8.2 Discussion

Overall, the particle systems created was looking good and the method used was satisfactory, but some improvement could have been made to get closer to the goal of visually pleasing graphics. For example, the firefly particles could be improved to look more like real fireflies rather than glowing spheres. To improve even further, an animation of flapping wings could be added. Furthermore, the rain particles could be more prominent in their appearance to get more intense rainy weather. On the same track, the rain texture is drawn in a stripe form trying to fake motion blur when the rain is falling. However, the results were inadequate when the particles fell towards the player (such as when driving forward), due to no possibility of rotation in the particle system


(a) The result of the fire effect.


(b) The result of the rain effect.


(c) The result of the smoke cloud effect.

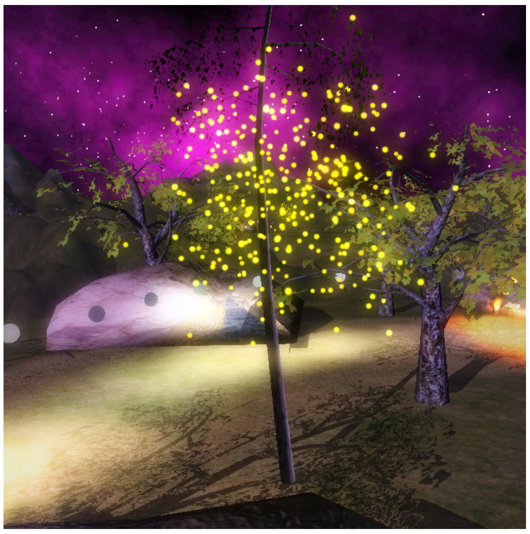

(d) The result of the firefly effect.

Figure 4.23: The results of the particle system effects.

engine. With real motion blur instead, the particles could look more like stripes using a texture of a round raindrop.

To support the improved particle systems, additional development of the particle system engine would be needed; features such as different colors depending on the lifetime, non-uniform scaling, and animations would augment the engine.

4.9 Post-processing Effects

Post-processing is a term to describe a collection of techniques used to improve the visual appearance of images after the rendering is complete. The techniques operate on a two-dimensional image of the three-dimensional scene, but some of them require additional textures. This section starts with introducing the basic method to enable post-processing, and then continues with three different effects implemented in Night Racer.

4.9.1 Render to Texture

Render to texture is a method for providing the means for post-processing. Instead of rendering the scene to the screen directly, this method renders it to a texture and thus enable additions of graphical effects to improve the visual appearance. After adding the effects to the texture, this texture is rendered to the screen. The implementation details, however, are specific to each graphics library.

The most common texture to render is the color texture representing the colors of the scene. The color data is the base of every post-process technique, but additional data, such as depth and normals, can be rendered to textures to provide the techniques with more possibilities.

4.9.2 Blur

Blur is a technique for smoothening out the texture and thus making it less sharp. A simple approach to calculate blur is to take, for each pixel, the average of its neighboring pixels in the texture. Montabone and Wickes (2009) states that this technique works better with smaller textures because the number of neighboring pixels stays the same regardless of the size of the texture.

A variant of blur that gives better results than the simple approach is Gaussian blur (Montabone and Wickes, 2009). The main idea is the same, but the neighboring pixels are instead weighted by the Gaussian function (see Figure 4.24). Gaussian blur uses a radius which determines how many pixels should affect the calculation. In order to get a full screen effect, the Gaussian blur process is performed in both the horizontal and vertical direction, and the final result is the sum of the two processes.

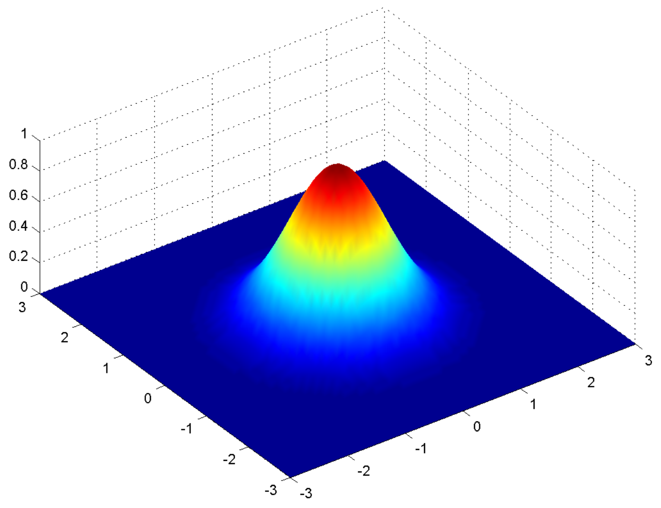


Figure 4.24: Example of a two-dimensional Gaussian function. This function determines how the neighboring pixel values should be weighted.

4.9.3 Bloom

Bloom is an effect for simulating how bright lights tend to spread their light on their surroundings to get more photo-realistic images. The effect simulates how cameras perceive lights rather than the human eye. The effect is prominent in Figure 4.25, where the characters look like they are luminous.

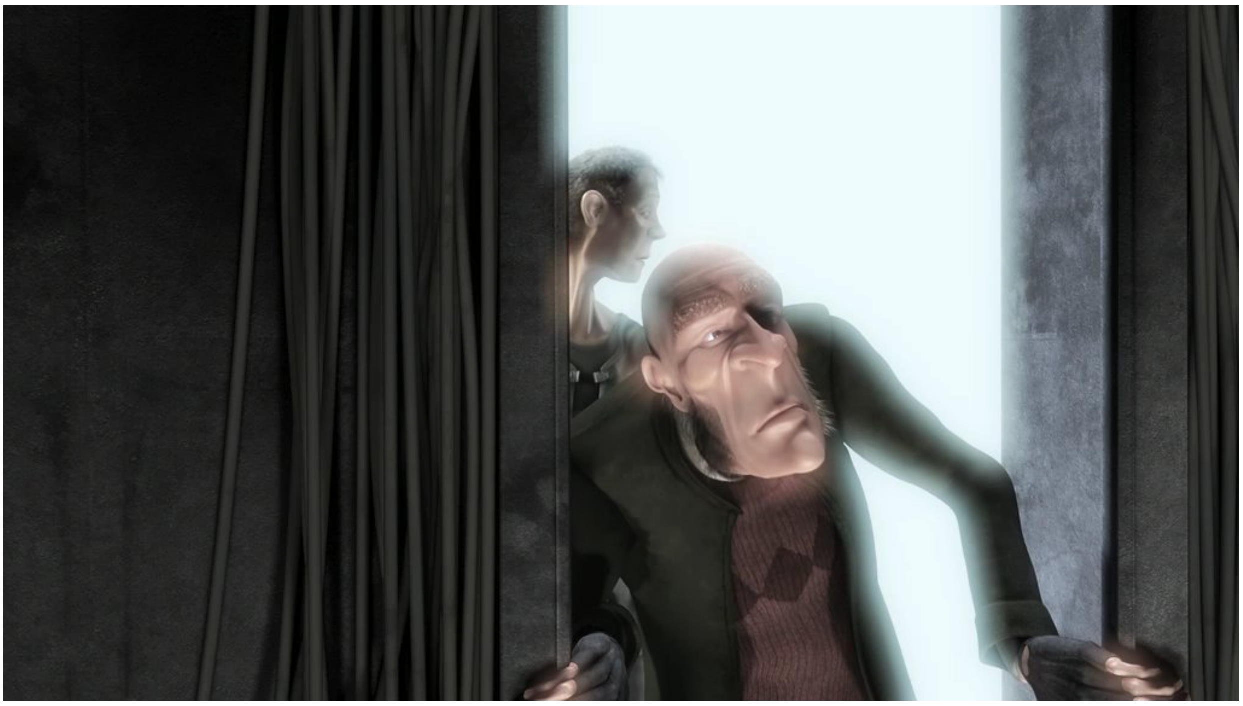


Figure 4.25: An example of the bloom effect. The bright light spreads light on the surrounding objects. *Image courtesy of Blender Foundation.*

Kalogirou (2006) describes the implementation of bloom in four steps:

1. Rescale the original texture down to a reasonable size.
2. Render the bright areas of the scene into a texture. This can be done by using a threshold value and only rendering the areas where the light is brighter than the threshold.
3. Blur the created texture to smooth out the bright areas.
4. Combine the original texture with the new texture by performing linear interpolation between them.

4.9.4 Motion Blur

Motion blur is an effect used for evoking a sense of speed by blurring parts of the screen. Motion blur is important in games where something moves really fast, and thus, especially in racing games. Though, if implemented poorly or overused, this effect can have negative consequences for players, such as nausea (Li, 2011).

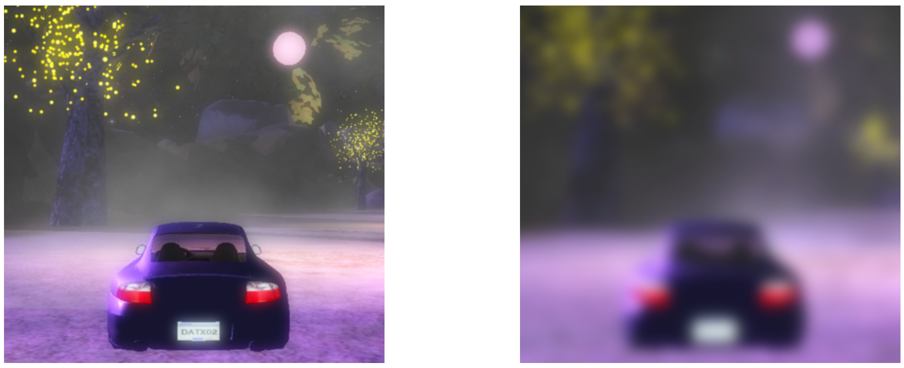
The effect tries to smooth out the rendered scene by taking the average value of several frames for each pixel. In order to calculate where the pixel was in the previous frames, the velocity of that pixel needs to be determined. Several techniques to find out the velocity exist, and they often include additional rendering that may cause some applications to run more slowly (Rosado, 2008).

Rosado (2008) describes another technique in the book *GPU Gems 3*, which uses the depth buffer to calculate the per-pixel velocity. This enables motion blur to be implemented as a post-process effect. The depth buffer is first used to find the current 3D position of each pixel, and then the previous frame's view and projection matrices (see Section 4.2.1) are used to calculate each pixel's 3D position in the previous frame. With both of these positions available, the velocity is the difference between them.

4.9.5 Results

XNA provides methods for implementing post-processing effects easily, and thus making it easy to adopt well-known techniques to improve the visual appearance of the game. The effects were implemented with instructions to GPU through pixel shaders (see Section 4.2.1), with additional XNA method calls to render the different textures used.

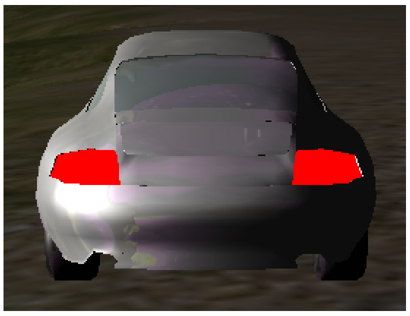
The effect blur was implemented in the game with the simple method that calculates the average of all pixels surrounding the current pixel. The result is shown in Figure 4.26a, and as the effect was not as prominent as thought, Gaussian blur was implemented instead. The radius and number of samples were tweaked to give an even better result. A comparison between the different approaches is visualized in Figure 4.26. Blur was only used in this game as a helper effect for the bloom effect.

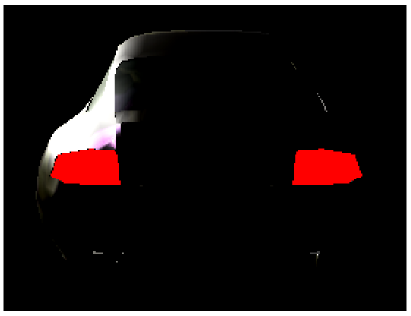


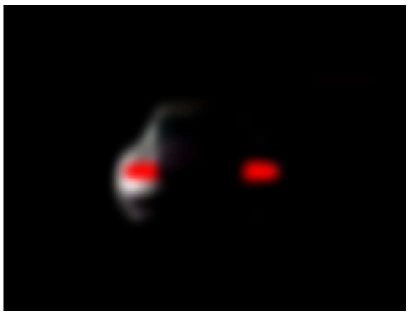
(a) The simple approach did not give sufficient result. **(b)** Gaussian blur gave much better result.

Figure 4.26: A comparison between the two blur methods (blur is not used in this context in the game).

When implementing the bloom effect, a variant of the implementation described in Section 4.9.3 was used. The bright areas of the scene are rendered to a texture which is then downscaled and blurred with Gaussian blur to get the bleeding effect. The texture combined with the original image makes up the final result, and the process can be seen in Figure 4.27. The result looks convincing enough to feel that the lights are bright instead of just flat and dull. The bloom effect in Night Racer can be found mainly on the point lights and the particle systems like the fireflies, but everything that is bright in the scene will be affected by bloom, as illustrated by Figure 4.28.


(a) The scene without bloom.


(b) Texture with the bright areas of the scene.


(c) The texture downscaled and blurred.

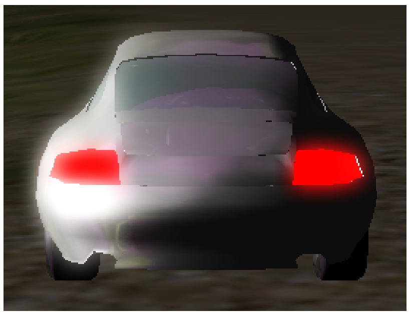

(d) The final result when combining the original image with the bloom texture.

Figure 4.27: The results of the bloom process.

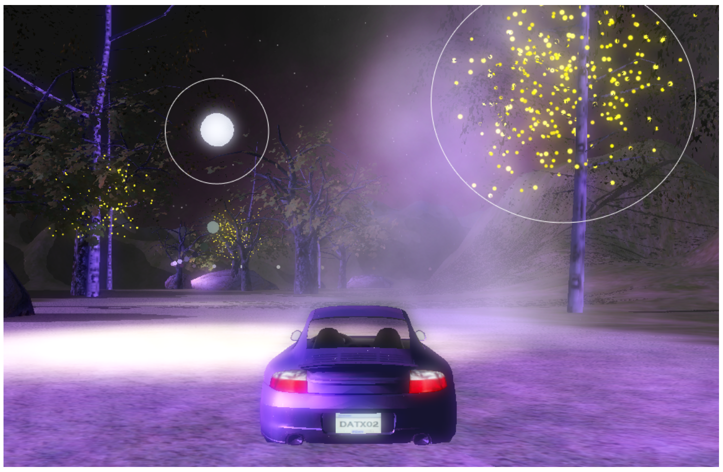


Figure 4.28: Bloom effect on the point lights and the fireflies in the tree.

Motion blur was implemented using the process described by Rosado (2008), summarized in Section 4.9.4. This technique proved to be efficient, as it is a post-process effect so no additional rendering other than a depth texture was required, but the result of the effect is not entirely satisfactory. The effect appeared as several distinct copies of the scene rendered in a direction, but when decreasing the distance between the copies, and instead increasing the number of copies, the results were sufficient. Motion blur was only used on the sides of the screen when driving to still get a clear appearance of the car. The final result can be seen in Figure 4.29.

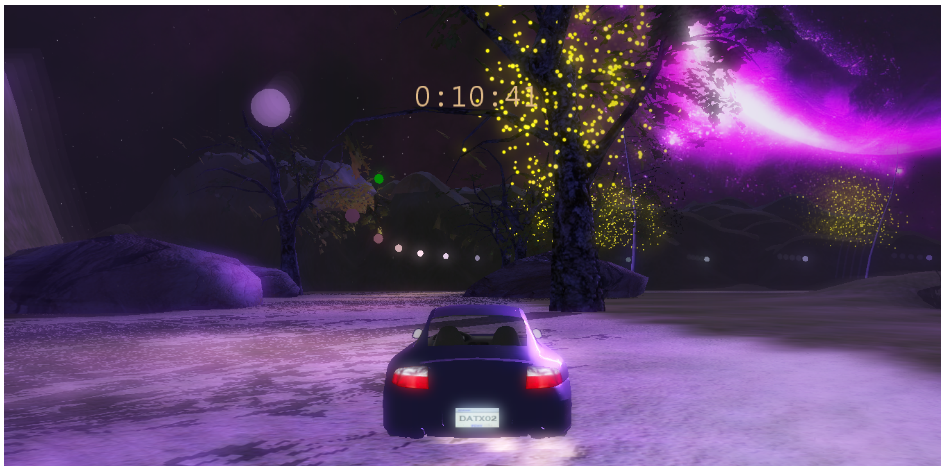


Figure 4.29: Motion blur seen on the sides of the screen. The effect appears as several copies of the scene mixed together to get the feeling of additional speed.

4.9.6 Discussion

The post-process effects implemented improved the graphics slightly, but they did not dramatically change the look. However, the goal of choosing visually pleasing graphical techniques was still met. If the effects were missing though, it would be noticeable as they can be found in the real life. If, for example, bloom was not implemented, the point lights would not look like real lights; they would instead appear as regular smooth-shaded spheres.

If the project was repeated, another technique for the motion blur effect would probably be used. The post-processing technique used in the game was easy to implement, but gave limiting results. A technique which uses multiple renders would most likely give better results and should thus be used if the hardware supports it. Another approach worth trying is to use Gaussian blur to mix the copies in the motion blur technique to get smoother and blurrier results.

5 Optimization

As the size of a game increases, it will also require additional time to compute all the necessary calculations in each game tick. In order to minimize this time, optimizations can be a valuable time investment. Optimization may thus help to solve the problem of using the computer's processing power effectively by limiting the number of objects that need to be considered each update, or by simply making the used algorithms more efficient.

A large part of this chapter will be dedicated to rendering optimization, as this project is focused on creating a visually pleasing game. The optimization of the terrain, which is closely related to how rendering optimization is utilized, will also be covered.

5.1 Rendering Optimization

One of the oldest problems encountered in computer graphics is the problem of visibility (Bittner and Wonka, 2003), and by extension, the optimization of rendering. A scene often contains far more objects than what is currently visible to the camera. The solution to this problem is to accurately and efficiently determine what objects are currently in view of the player, upon which the remaining information can be discarded from the render cycle. This process is referred to as culling.

This section will present an assortment of available methods which offer several different solutions to the problem (of which all can be used in conjunction with each other) and outline the details behind their idea and usage. The first method to be covered is used to determine on what scale the detail of an object needs to be to still read as the same object at various distances. The remaining methods discussed are different options for implementation of culling, starting with backface, occlusion, and finally viewing frustum culling.

5.1.1 Level of Detail

As first explored by Clark (1976), an object at a distance does not need to be as complex as an object that is closer to the viewpoint; both objects will appear to have the same shape even if the object further away has far fewer polygons. The level of detail (*LOD*) of the object is determined by the distance, and this is the origin of the name.

In its most basic form, several models representing the same object are created, and distance is used to determine which level of detail the model needs to be used in order to accomplish more optimized rendering (Danovaro et al., 2006), see Figure 5.1.

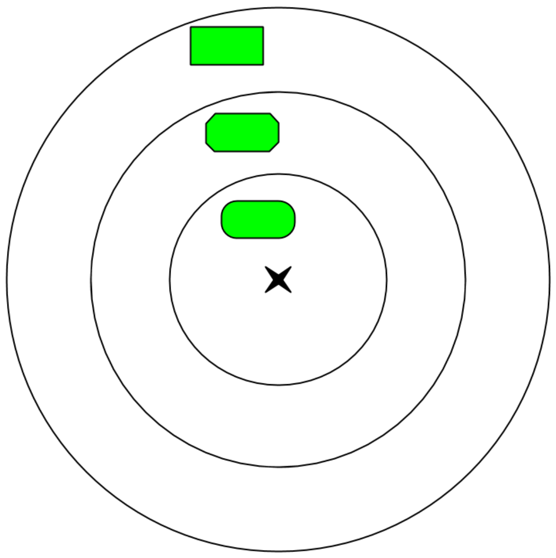


Figure 5.1: An example of level of detail optimization. Each circle represents a discrete distance from the viewer at the center, and the model is simplified as the distance increases. The green shapes represent a possible version of the same object at the different distances, each using a simpler shape than the previous.

5.1.2 Backface Culling

Backface culling is the concept of determining which polygons are currently visible to the player, which is easily accomplished by observing the normal of the polygon (Lamberta, 2011). The normal of a polygon is equivalent to the direction it is facing, and thus a polygon which is currently facing in the direction of the player will have a normal pointing towards the current viewpoint. All other polygons are then ignored during rendering, as they are obscured by the side of the object actually facing the player.

The algorithm, due to its nature has the potential to roughly halve the amount of polygons needed to be rendered, but is subject to several limitations (Blinn, 1993). First and foremost, the object needs to be completely opaque; a model with, at least, partial transparency are by their very nature see-through. The polygons facing away from the player are therefore still visible, even though they are not facing towards the player. The object's polygons also need to form a complete shell where there should be no holes or openings in the surface, as these perforations create the same problem as transparency, the inside of the object, and thus the backside of the polygons might still be visible to the player.

5.1.3 Occlusion Culling

The concept behind occlusion culling is that if an object is obscured by an object closer to the view point, it should not be rendered as it is not visible from that vantage point (Frahling and Krokowski, 2005). The process of occlusion culling is thus to determine which objects are visible to the player, and, in more advanced algorithms, what parts of said objects are visible, see Figure 5.2.

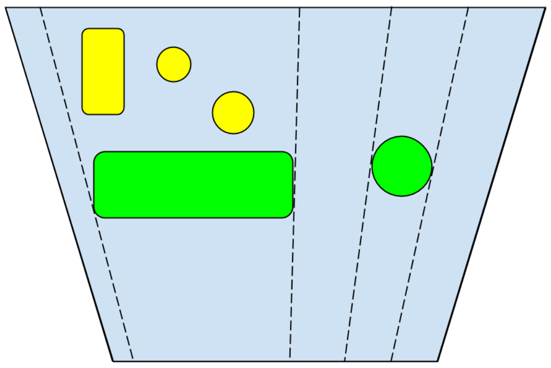


Figure 5.2: An example of occlusion culling. The blue area is the visible screenspace, with the viewer located at the bottom of the image. The green shapes are visible to the viewer and the yellow shapes are hidden (or *occluded*) by the objects in front of them.

5.1.4 Viewing Frustum Culling

A frustum is the geometrical shape received when a solid, such as a cone or pyramid, is cut by two parallel planes. The viewing frustum is the pyramid shaped frustum equivalent to the visible screen space. Viewing frustum culling is thus the process of eliminating polygons that are located outside of the projected screen space (Bittner and Wonka, 2003). This is accomplished by checking which objects and polygons intersect with the current viewing frustum, see Figure 5.3.

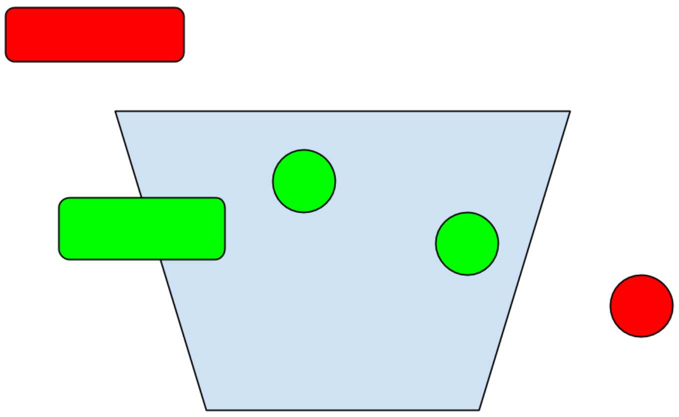


Figure 5.3: An example of frustum culling. The blue area is the viewing frustum. The green shapes are at least partially inside the frustum and are thus included in rendering. The red shapes are outside of the current viewing frustum and are ignored during rendering.

5.2 Results

The implemented rendering optimizations have led to better performance and an increase in the game's framerate, and have thus been a valuable pursuit. However, only two of the methods outlined above have been implemented, which is due to the complexity of the algorithms. The first of the two is backface culling, a relatively simple algorithm that is included in the XNA framework. The second optimization is viewing frustum culling, of which a simple version has been created. The bounding box of each object (which is a cube containing the whole object) is examined whether it intersects with the viewing frustum or not, and if it occupies a part of the screen it is considered during rendering.

Occlusion culling has not been implemented in the game due to the difficulty of the algorithms involved, the choice was instead made to focus on other aspects of the game. The inclusion of the level of detail algorithm was discussed and the models were tested if they could be simplified in an acceptable manner with the 3D modeling program. As the result of the simplification was not deemed to be of an acceptable level and all models were gained from outside sources, as well as the line of sight in the game being fairly limited by the small scale terrain, it was decided that level of detail would not be implemented.

Another optimization implemented in the project is used along with rendering of the terrain into the world. Originally, the terrain consisted of a single model, which created a problem when it was subjected to the simple, but quick, frustum culling algorithm used in this project. The problem was that since a part of the terrain was always inside of the viewing frustum the entire terrain was considered as a possible candidate for rendering. The solution was to divide the terrain into several subsections, where each subsection is its own model. This lead to a noticeable increase in performance as a large part of the terrain is no longer considered during rendering.

5.3 Discussion

As optimization is an important part of making the game playable for a majority of the potential userbase, it is a valuable part of any project. As is apparent from the results, the method of using the XNA framework has been very useful because of the built-in implementations of the culling algorithms. While the concrete optimizations applied in this project are quite few, numerous smaller optimizations have been applied continuously on individual algorithms used in the project.

Despite this, additional optimizations would be a very valuable area of development if further development of the game is considered. In such a case, the rendering optimizations discussed in this chapter may be of specific interest due to their large effect on performance. Furthermore, to develop a the game into a more visually pleasing one, the performance gains by the algorithms implemented allow for even more effects to be considered in the future.

6 Audio

Sound surrounds us at all times; even when asleep there are several sounds that go unnoticed: the running of water in plumbing and radiators, the low buzz of the refrigerator or the occasional noise of nightly traffic. All these sounds are taken for granted, and the same is often true in video games. As such, the audio section in a small project can sometimes be seen as a burden to be fulfilled, when it instead should be seen as an asset.

Research has shown that sound can be used to influence visual perception; sound of good quality can be used to hide temporary drops in visual performance such as a decrease in framerate (Hulusic et al., 2011), or even improve the perceived visual quality (Simpson, 2000a). Audio is therefore an integral part in creating a believable game experience by making the game feel alive. A well made audial experience thus enhances the feeling of motion and makes the player feel immersed.

This chapter will cover how the three-dimensional nature of the game influences the audio of the game and what the alternatives are for implementing sound in an XNA project. The problem of providing an immersive gameplay experience by adding sounds appropriately will also be examined.

6.1 3D Positional Effects

Physics and the visual quality in video games are growing ever more realistic. Physics calculations can be done in real-time, but audio is still represented by pre-recorded samples. Sound as well is just a physical phenomenon, and the capability to accurately synthesize and calculate the sounds and audial abilities of an area has recently been explored (Raghuvanshi et al., 2007), but it requires immense amounts of computational power.

As the audio of a game is represented by samples, it can therefore become quite one-dimensional. However, the gains from accurately calculating the physical phenomena of 3D sound can be regained in a satisfactory fashion from far simpler methods. Most available sound libraries have the capabilities of adjusting both volume and pitch, which is two of the key variables to adjust in order to create a sound experience that is perceived as real.

The change in volume is used to make sounds diminish as they move further away from the player, which is easily applicable in a racing game. The volume can either be changed manually, or by using the methods available in the sound library that calculate the distance between the source and the listener and adjust the volume accordingly. These methods will also determine the position of the source of a sound in the game world, by changing the amount of sound emanating from each speaker (panning) in order to better simulate the direction in the game world the sound is coming from. These two techniques can thus be applied together to play sounds in a manner able to model their distance as well as direction, which is sufficient for simulating positioned sounds in a 3D world.

An additional effect that occurs when moving is the Doppler effect, which is the result when either the listener or the source of a sound is moving, which is an occurrence that is quite common in a racing game. An example of this is the sound of an ambulance's siren; when the ambulance is moving towards the listener the sound is different from when it is moving away. This is due to the frequency becoming offset from the velocity of the source; the frequency is increased when it moves towards the listeners, and decreased when it moves away.

6.2 Sound Libraries

There are several alternative sound libraries available for C#, some of which require a license to use if the goal of the project is a commercial game, such as FMOD. These licenses are an unnecessary expense, and since the goal of the game is merely to have basic functionality fit for a technical demonstration. The choice has thus been to evaluate the free alternatives available, and to narrow down the selection further it seemed prudent to use the alternatives that are either native or well integrated to XNA. The two remaining alternatives are then Microsoft's XACT, which is a high level sound library and authoring tool, and the XNA Content Pipeline (Microsoft, 2013b).

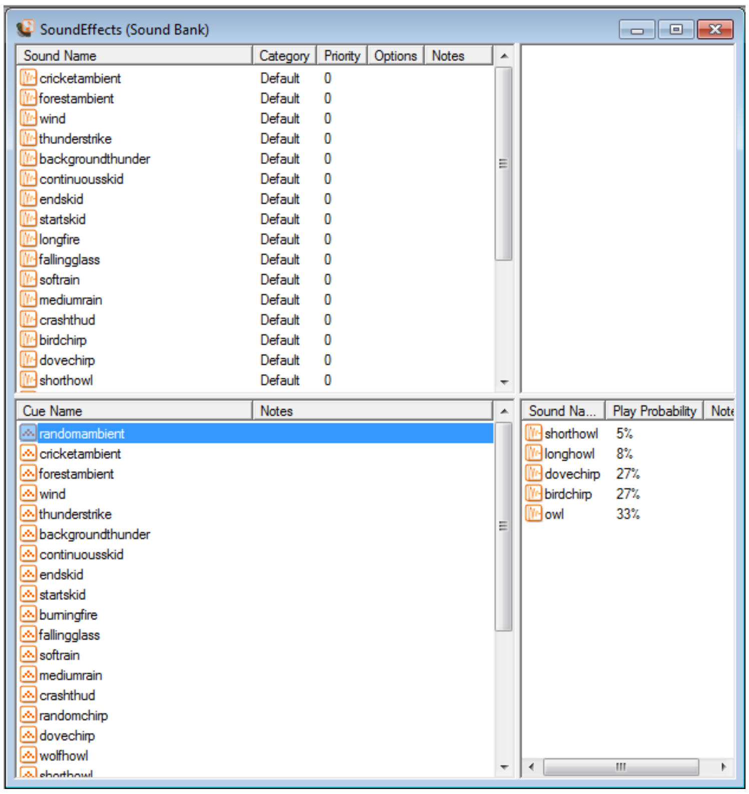
The XNA Content Pipeline has the advantage of being able to play MP3 files in addition to the formats supported by both alternatives, but carries with it the disadvantage of only being able to adjust volume during runtime. Each sound also needs to be managed and stored individually in the code, which becomes rather cumbersome if there are a large amount of sounds.

XACT only offers its full functionality when using WAVE files, but XACT has several advantages in its favor. It not only gives the ability to adjust volume during runtime, but also allows the user to change pitch and apply 3D positional effects. One of the key features of XACT is the ability to build wavebanks and soundbanks. The former is, as implied by the name, a collection of sound files of the WAVE format, making management easier. The soundbank contains instructions known as cues for how to access the wavebank, and how a single sound is changed when 3D effects are applied to it. These banks are presented in a separate management application, seen in Figure 6.1.

6.3 Results

Due to the similarities in the scope of the project and the capabilities of XACT, it was elected to be used for the sound implementation. It offers a high level of customization of sounds for a relatively small investment of time.

Since none of the team members had any actual experience working with sound creation or recording, as well as limited access to decent recording equipment, the choice was made to use


SoundEffects (Sound Bank)

Sound Name	Category	Priority	Options	Notes
cricketambient	Default	0		
forestambient	Default	0		
wind	Default	0		
thunderstrike	Default	0		
backgroundthunder	Default	0		
continuousskid	Default	0		
endskid	Default	0		
startskid	Default	0		
longfire	Default	0		
fallingglass	Default	0		
softrain	Default	0		
mediumrain	Default	0		
crashthud	Default	0		
birdchirp	Default	0		
dovechirp	Default	0		
shorthowl	Default	0		

Cue Name	Notes
randomambient	
cricketambient	
forestambient	
wind	
thunderstrike	
backgroundthunder	
continuousskid	
endskid	
startskid	
burningfire	
fallingglass	
softrain	
mediumrain	
crashthud	
randomchirp	
dovechirp	
wolfhowl	
shorthowl	

Sound Na...	Play Probability	Note
shorthowl	5%	
longhowl	8%	
dovechirp	27%	
birdchirp	27%	
owl	33%	

Figure 6.1: The soundbank view in XACT. The top half of the interface is the sound collection, each sound has a probability (bottom-right) to be played when a cue is played in the game. The cues are located at the bottom of the interface, each cue has its own set of probabilities.

crowd-sourcing websites dedicated to sharing free sound effects. Examples of such sites that were used are freesounds.org, findsounds.com and soundbible.com. Some of the sound assets have then been edited, where unwanted parts have been removed and seamless looping has been ensured.

A large wavebank has been created and is utilized in the game, however, not all of the available assets are used yet. The most developed area is the ambient sounds as they help the most with creating the atmosphere for Night Racer's fantasy theme. A general forest ambient was decided to be well associated with the fantasy setting, and thus sounds such as crickets and various bird calls have been woven together to create the basis for the audial scene. To create a more living world, night sounds such as a wolf's howl or an owl's hoot are played at random intervals. Additional ambient sounds include falling rain, which varies slightly in sound intensity as to not create a flat effect.

The fires in the checkpoint arches (see Section 4.8) are currently the only sound-emitting objects subject to the Doppler effect. Among the other sounds that would benefit from this effect are the engine sounds of the opponents.

6.4 Discussion

In an optimization scenario where the available memory is limited, XACT suffers from the file type limitation due to the uncompressed nature of the WAVE format. In particular, long sounds such as background music are an issue. As an alternative, the XNA Content Pipeline could have been utilized as it has full support for additional filetypes, such as MP3. Because of the need to create additional interfaces to easily interact with the systems of such a solution, the choice of only using XACT is however well justified. XACT does have an alternative to set a wavebank to streaming mode, which allows the files to be streamed from the harddrive into the smemory as more of it is needed, vastly reducing the memory footprint. However, this is not used in the game, but is a possible future optimization.

While the sound effects currently in use do provide a more immersive gaming experience, the game experience could be improved by acquiring better sound sources. The current sounds are far from optimal and do not fulfill the predetermined level of quality the project is aiming to achieve. This is because none of the effects used are customized for the project, instead available sounds have been edited with results of varying quality. In particular, good engine sounds were incredibly hard to find and are thus not part of the game, but is very central to the experience of a racing game. New sounds could be obtained by either utilizing services that allow access to their databases for a licensing fee, or by acquiring high quality sound recording equipment in order to record sounds.

7

Multiplayer and Network

All types of games need to provide some form of challenge for the player to overcome. When designing games, the designers need to decide what challenge to focus on. These challenges may take several different forms, among them are artificial intelligence opponents and non-physical opponents such as time or hurdles, but also other players through multiplayer game modes.

Competitiveness is part of human nature, and multiplayer gaming is a relatively simple way of providing a forum for competition. Furthermore, as network connectivity increases around the world, online gaming has become a major selling-point for games (Smed, 2008) and research has shown that the majority of young gamers engage in online gaming (Lenhart et al., 2008). Therefore, including an online multiplayer mode in a video game that is to be commercially viable is of obvious importance.

This chapter will examine techniques for implementing a multiplayer mode in Night Racer. However, multiplayer games can take on many different shapes, and as such, different types of multiplayer game modes that have been considered will be presented. The type of multiplayer gameplay, network architecture, and programming framework that was decided upon for the project are then examined in detail, followed by a discussion of the results. Furthermore, approaches for combating latency issues, so that the game can provide a smooth gameplay experience, will be presented and evaluated.

7.1 Offline Multiplayer

A basic multiplayer game mode is where two or more players play together on the same device. In this case, both players are shown on the same screen, sometimes splitting the screen into different areas if not all players can be shown in the same screenspace. Players use individual controls connected to the same device which control their respective characters.

Non-networked multiplayer games provide some advantages over networked games, such as being able to accommodate virtually latency-free communication between player. Since Night Racer cannot realistically show several players with the same camera while keeping the gameplay the same, splitscreen would need to be implemented to provide offline multiplayer gaming in the game. Splitscreen game modes require the game to be rendered one time for each camera, which makes splitscreen very taxing for a computer, and for a graphically intense game such as Night Racer, it makes it unrealistically so.

7.2 Online Multiplayer

Online (or "networked") multiplayer allows players to compete against each other from different computers connected through a network, either a local network (such as a home router) or a larger one (such as the Internet). Connecting players over a local area network or over a larger network is logically equivalent, since both examples work by connecting to a remote computer with an IP address, distance between the two notwithstanding. However, an implementation for local area network gaming might not work very well over the Internet, since such connections introduce data exchange issues, usually caused by congested connections between players (Coulouris et al., 2012).

7.3 Network Architecture

The architecture of a network refers to how the hosts in the network are connected to each other. In terms of networked games, there are two different network architectures to consider, as outlined below.

7.3.1 Client-server

A client-server network consists of a set of hosts (clients) and one centralized host (server). Clients communicate via the server and never directly with other clients. Clients run identical client applications which distribute and receive data from the separate server application that is running on the server (Kurose and Ross, 2010). The topology is visualized in Figure 7.1.

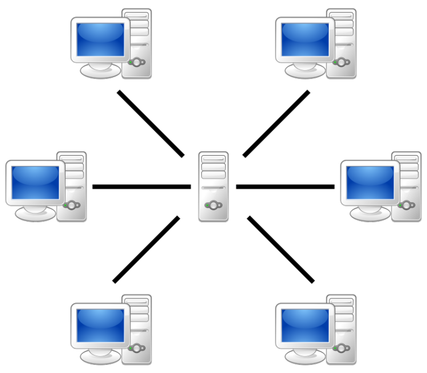


Figure 7.1: Visualization of an example client-server network.

Among the primary advantages of a client-server architecture is the possibility of having a single authoritative host, the *authoritative server* (Gambetta, 2010), since all data must pass through the server. In such implementations, the client-side player does not necessarily decide its own position, which may be decided by sending a request to move a number of units to the server, where upon the server responds by stating the absolute position in which the player can move to. In less naive implementations, the position of a client-side player may instead be corrected by the server. An authoritative server can thus allow for centralized anti-cheating measures, in a trade-off with increasing server complexity.

7.3.2 Peer-to-Peer

In a peer-to-peer network, all hosts (peers) communicate directly with each other (Kurose and Ross, 2010). A host wishing to connect to the peer-to-peer network connects to one of the peers already in the network, who then distributes a list of available peers to the newly connected host, as well as to the previously connected peers. At this point, all connected peers can exchange data directly with one another, without having to go through a central server. This topology is visualized in Figure 7.2.

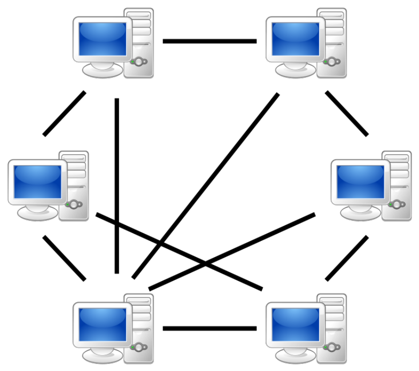


Figure 7.2: Visualization of an example peer-to-peer network.

A peer-to-peer network allows data to be passed directly to each player instead of requiring data to be sent through a central server, which can reduce the connection latency since $p * r$ packets (where p is the number of players and r is the rate of packets per second) being sent to and from the server every second may present a bottleneck. However, a peer-to-peer architecture provides additional complexity in implementation (Simpson, 2000b) and state synchronization (Fiedler, 2010).

7.4 Frameworks

There are numerous ways to implement network functionality in a game made with XNA. The most basic choice is socket programming using the *System.Net* library which is a part of the .NET Framework. This, however, involves low-level network programming which is not ideal for Night Racer since time could be spent on developing a more complete network game. Such low-level detail is already implemented in other libraries, some of which will be examined below.

7.4.1 XNA Networking

The *Microsoft.Xna.Framework.Net* library is part of the XNA framework and hides some of the low-level details previously discussed. It is specially tailored for network games, and thus has features such as lobbies, scoreboards and player management. In XNA games developed for Windows, it does not allow for players outside the same local network to connect to each other without Games for Windows Live. Developing for Games for Windows Live also requires a XNA Creators Club account (Microsoft, 2013a).

7.4.2 Lidgren

Lidgren networking library generation 3 is an open-source third-party library for providing network functionality in the .NET Framework. It is not specifically designed for networked gaming, but does provide a higher-level API for connecting to a server or peers and exchanging data messages, coupled with efficient use of resources (Lidgren, 2011). Furthermore, it has been used to implement multiplayer functionality in games, such as AI War (Park, 2010).

7.5 Combating Latency

Latency is defined as the time a message takes from the sender to the receiver over a network, and is a very important part of network games where the aim is that the state of a game should always be practically equal for all players connected. A congested network will present higher latencies (Coulouris et al., 2012), and it is therefore very important to find ways of alleviating congestion. This section aims to detail three different approaches to combat latency: sending smaller amounts of data, sending data less often, and simulating smooth gameplay when no data is available.

7.5.1 Identifying Player Data

When dealing with sending data over networks, every single bit counts. Networks can often be overwhelmed by data, and the less data that is sent, the less likely it is for such problems to arise. Online action games suffer from this problem as they typically need to deal with very rapid data exchanges, as they (in the case of Night Racer) update the screen 60 times a second, and therefore would preferably need new player data from all players at the same rate. As such, in designing network code for video games, it is very important to decide exactly what information the players need to properly display their co-players on their screens (Simpson, 2000a).

With an ideal connection, it would be possible to send the whole object representing the car on the player side, information of the surrounding terrain as well as various details from the physics engines. Such data would make it possible to react more consistently to complex situations such as collisions between players. However, that would entail several players sending roughly hundreds of bytes 60 times a second, which would put considerably more strain on the network, as well as making the server code vastly more complex (Fiedler, 2006).

7.5.2 Simulating Smooth Gameplay

However conservatively the exchanged player data is chosen, a latency-free connection can never be guaranteed since networks can become unexpectedly congested (Kurose and Ross, 2010). It is therefore difficult to provide consistently smooth gameplay in such situations. Furthermore, if algorithms can facilitate sending less than a full 60 packages per second in the game, congestion could be prevented in the first place. This following text will describe three considered techniques for simulating smoothness in gameplay when sufficient data is unavailable.

Interpolation Interpolation regards the latest messages received from the server and creates interpolated messages in between them, simulating a full 60 messages per second (Valve Developer Community, 2012). For example, if a client receives 30 messages per second with the position of a player, it will create one interpolated position in between positions P_n and P_{n-1}, where n is the number of the latest received message:

$$P_{n-0.5} = lerp(P_{n-1}, P_n)$$

where *lerp* is the linear interpolation function. The rendering of the moving player on the screen of the client is delayed twice the time between server messages, which makes the movement of the player smooth even if one message is lost; if more than one consecutive message is lost, the client will experience choppy gameplay (Valve Developer Community, 2012). This interpolation delay can make the game feel unresponsive to some degree since the player does not see the latest game state on the screen, and the interpolation of player positions can make rapid movements seem flattened (Bernier, 2001).

Extrapolation Extrapolation can be used when a new frame should be rendered, but no new player position has been received from the server. Instead of simply having the player keep their position (which would make the player's movements seem jerky), the client can extrapolate the player's new position based on the movement in the previous frames (Bernier, 2001). This uses the velocity (and acceleration, if possible) of the player, which is applied to the previous position P_n, producing an extrapolated position P_{n+1}. When an updated player position is received from the server, the client can correct the player accordingly.

Dead reckoning Using an agreed-upon extrapolation algorithm, dead reckoning as described by the IEEE Standards Association (2012) is used to simulate player positions when no new position has been received from the network (see extrapolation). However, a dead reckoning solution does not use a fixed number of position updates sent per second, but rather a distance threshold that each client keeps track of. This threshold is reached when the player has diverged a fixed distance from its position in the (by extrapolation) simulated car based on their previous player position message sent to the server. When reached, the client sends a new player position message to the server which is then distributed to the other clients. Thus, player positions are only distributed when they are actually needed, and if a player keeps well to the path of the extrapolation algorithm, that may rarely if ever happen, and the consequence is that network traffic can be kept to a minimum. This, however, makes player position updates less deterministic and implementation more complex.

7.6 Results

A fully functioning multiplayer mode has been implemented in Night Racer, seen in Figure 7.3. It is playable by a maximum of four players in the form of a simple racetrack mode, where the first player passing the goal line of the last lap wins. A multiplayer game is started by connecting to a server via the main menu, upon which the game lobby is presented (see Figure 7.4). Pressing the start button will take the player to the gameplay view, and when all players connected to the server have started the game, the race countdown commences.

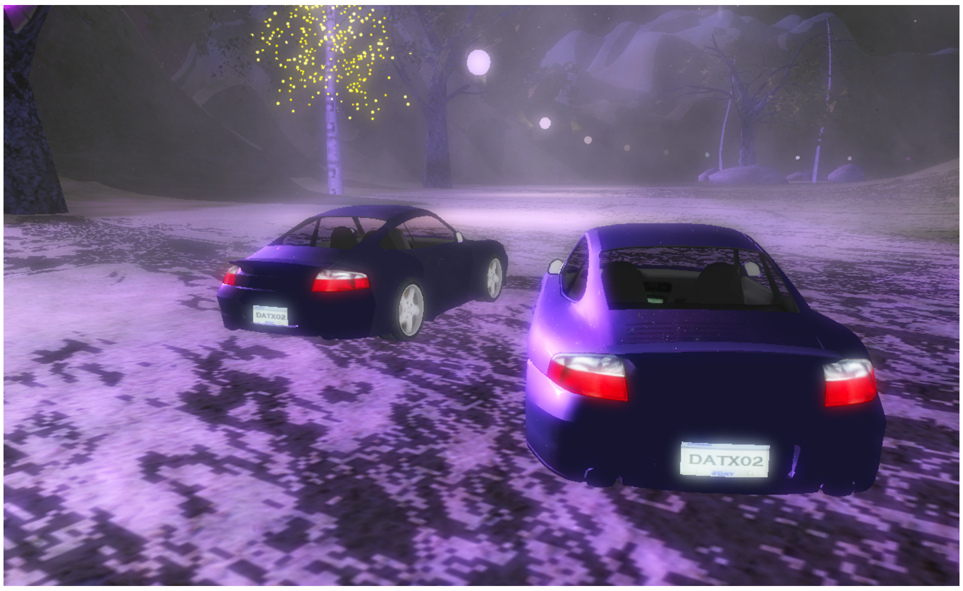
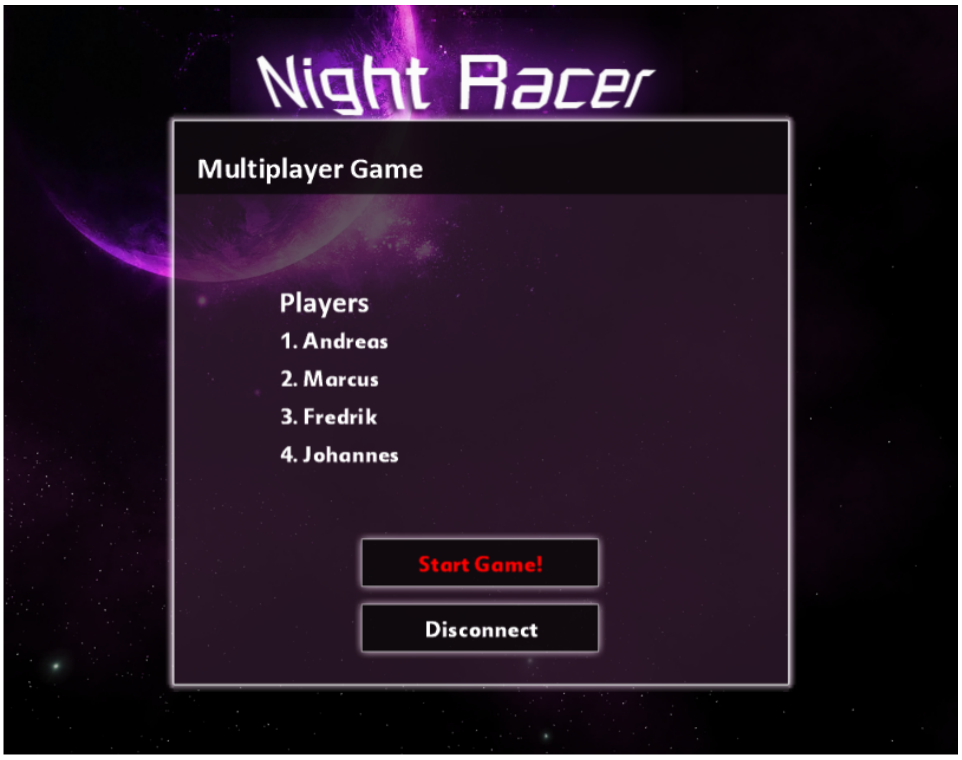


Figure 7.3: Two cars competing in the multiplayer mode.



Players
1. Andreas
2. Marcus
3. Fredrik
4. Johannes

Figure 7.4: The lobby screen, showing all players connected to the game server.

Night Racer uses a client-server architecture with a non-authoritative server, which means that the server essentially receives messages from the players, and distributes them to all other players directly after appending an identifier of the sender. This keeps the complexity of the server software to a minimum since it contains no game logic, while allowing extension of the server with anti-cheating measures at a later date. At this point in time, no network security is provided, which makes it possible for resourceful players to cheat, but this is made less effective since the only data that is being exchanged is the absolute positions of the player, and the winner of a race is calculated client-side. Both the client-side network code and the server software uses the Lidgren library, which allowed for further focus on high-level architectural programming and provided a rather simple API.

The network game is divided into several states, where the first state (Lobby) is where information about players is exchanged, followed by the countdown state where players are notified when they may start moving, leading to the race state, which is concluded by the game over state, where all statistics are calculated on the client-side to be presented after the race. The race state is of particular interest, since it is only at that point player positions must be exchanged, which is the only type of message that needs to be delivered consistently and quickly and are sent in such frequency that they may cause latency issues.

For Night Racer, the data that needs to be sent for each player has been identified, and is presented in Table 7.1. The first byte indicates which type of message it is, and informs the receiving client how the rest of the message should be parsed. In the case of a player data message (or ”PlayerPos” as they are named in the code), it is encoded as 0x00. The solution used in Night Racer opts to distribute player positions from the server as individual messages, as opposed to sending positions of all the players in the same message. This choice was made because the gameplay will seem more natural if some of the players move, instead of none, in case of network congestion. The player positions in the 3D space as well as the rotation are enough to represent the player accurately on a game level, and are sent as floating point numbers with 32-bit precision, as they are represented in the Night Racer graphics engine.

Table 7.1: The bit fields of a PlayerPos message, as used in Night Racer.

Field name	**Data type**	**Size (bits)**
MessageType	Byte	8
PlayerID	Byte	8
Player position X	Float	32
Player position Y	Float	32
Player position Z	Float	32
Player rotation	Float	32
Velocity	Float	32

As the table indicates, the choice was made to not send any physics engine data (such as collision detection) in the data packets to conserve bandwidth. This means that all physics decisions are made on each client’s side rather than on the server, which could in rare edge cases produce rather amusing unexpected behaviors (such as when one car detects a collision, but the other car does not). However, the current iteration of the game does not feature collision detection, which makes these issues void. Further measures could be taken to conserve bandwidth, such as abbreviating the 32-bit values to a 16-bit data type (such as *short*), but it was determined that the loss of precision is unlikely to make up for the minor gain in bandwidth.

For simulating smooth gameplay, an adapted version of the dead reckoning technique described by Aronson (1997) is used with a linear extrapolation algorithm. The choice to use dead reckoning was made because it provides both enforceable player position accuracy and little network traffic. The downside is that dead reckoning is comparatively complex to implement, but using a simpler extrapolation algorithm makes implementation more time efficient. Changing the threshold (that is, the maximum allowable distance between the player car and the simulated car before a new player position message is sent) allowing for finding a balance between network congestion and smoothness of the gameplay. This technique has been found to produce satisfactory results, as it allows for smooth gameplay while keeping the network traffic quite low.

7.7 Discussion

The point of the techniques applied in Night Racer is to hide latency and packet loss, and as such, a good implementation should mean that the results are invisible to the players. However, there are some ways of simulating packet loss and latency built into the Lidgren library. Testing indicates that a packet loss of 5 to 10 percent is acceptable from a subjective impression, but higher loss probabilities make the movements of players seem too jerky. A non-simulated example of the impressive efficiency of the dead reckoning implementation was experienced when a client with a very low frame-rate was connected to a server. The movements of that client was then observed by a high frame-rate client, where the movements appeared almost entirely smooth even though network updates could only be sent very seldom by the low frame-rate client.

An evaluation of the usage of network resources by the algorithm has been done by measuring the number of PlayerPos packages sent per second. These measurements have been made by driving around the track as smoothly as possible to produce a consistent result, and using different arbitrarily chosen threshold levels. The graph in Figure 7.5 shows the results, along with a base line showing a simple extrapolation algorithm with a fixed packet send rate of 30 updates per second.

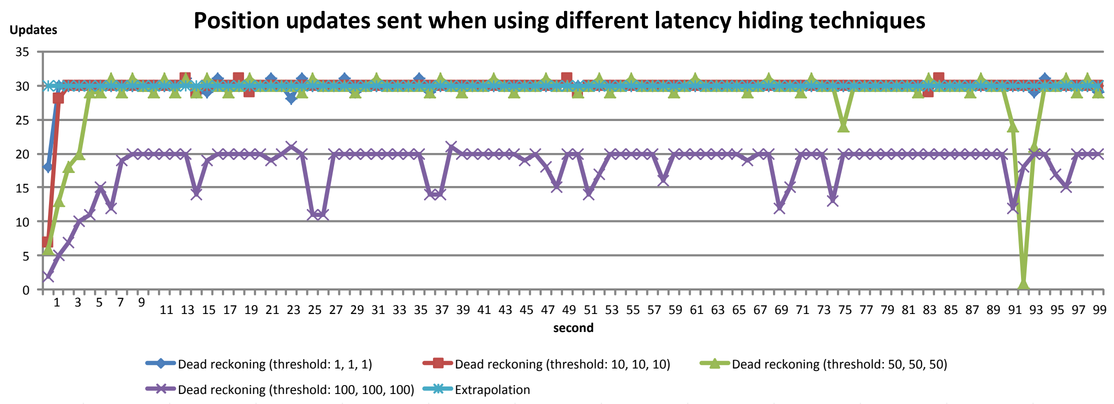


Figure 7.5: A comparison of the rate of PlayerPos updates, using a simple extrapolation technique and a dead reckoning technique with different threshold values. Threshold values are in an arbitrary (x, y, z) unit.

The measurements indicate that the implemented dead reckoning algorithm differs very little in network usage from the standard extrapolation algorithm in practice outside the occasional deviation (such as in the dead reckoning algorithm with a threshold of (50, 50, 50) units at around second 92) which may be explained by erratic driving during the test. Only when using very large threshold values, such as (100, 100, 100) units, does the send rate differ considerably. However, such high thresholds produces jagged player movements, and are thus not viable.

In conclusion, the network techniques implemented in the game have been found to be largely well chosen and the results are satisfactory, while allowing possible improvements in the future. However, the implementation of dead reckoning might not have been as time efficient as previously thought, as a simple extrapolation technique would have been much easier to implement and would produce very similar results. Relating to the goals stated for the project, the networking functionality has been successfully implemented, and is also found to add greatly to the playability of the game and thus the commercial appeal and viability as well.

8

Results

Night Racer set out with quite ambitious goals: most centrally creating a visually pleasing multi-player racing game, and has largely succeeded.

The game in action is graphically refined; even though the graphical fidelity is not up to par with commercial games, it fulfills the conditions stated by the authors for a visually pleasing game. Night Racer offers the player a basic gameplay experience, with a single type of game mode and simple controls. The car handling is not advanced but serves to give the player both a sense of speed and a satisfying sense of control.

The game mode, which is a simple time trial, is available for both singleplayer and multiplayer use. Multiplayer is available through network resources such as LAN or the Internet.

The game is able to run smoothly on the target hardware and can thus be said to successfully fulfill the requirements set for the project. The game does however suffer a framerate drop on weaker machines, such as laptops or older desktop PCs. This, and the fact that some assets (models and sounds) used in the game belong to a legal grayzone (obtained from the Internet) means that the game is not entirely commercially viable. However, Night Racer does mostly adhere to the definition of commercial viability presented in Section 1.1; the game could work as a technical demonstration that could be used to gather funds from a potential investor or a crowdsourcing project for its continued development.

The goals stated for the product can therefore be said to be fulfilled. The problems that were set to be examined in this thesis have been evaluated and answered in their respective chapters, and the reader would be advised to refer to each section in order to receive an accurate assessment to each problem. The authors' opinion on the finished product, not necessarily tied to the goals and problems, will be presented in the discussion section.

9
Discussion

Night Racer is by no means a finished game, nor was it intended to be. It was recognized very early on in the development phase that making a polished and complete game, and making a graphically refined game are goals that are at odds with one another. As such, making a finished game was never the purpose of this project. Nonetheless, it can be of interest to reflect on the state of the game today, and what could have been done differently.

The game is deemed by the authors as visually pleasing, with some very effective use of particle effects, reflections, colors, and extensive lighting. In particular, always trying to incorporate a sense of movement (whether it is by light sources changing intensity or color, or with particle effects such as drifting smoke) has made the game feel much more alive, and spending time on such effects were a largely productive use of time.

There are some lacking parts of the graphical impression, and the most prominent is a lack of diversity in the game world. Firstly, the generated levels are mostly flat since the lack of a physics engine would make racing on more hilly terrain ineffectual. Secondly, there is a lack of graphical objects placed around the map which makes the world seem very empty. Lastly, procedural generation is not used to its full extent since all generated race tracks and terrains differ very little from each run. While there never was a large focus on these areas, the project may have benefited by at least some time being allocated for reviewing this issue.

As for playability, the simple race mode provided with the game is not particularly interesting, especially since no collision detection between cars nor realistic car handling has been implemented. This, however, is where software extensibility is of importance, since more interesting game modes can be implemented with the provided interfaces for game modes and triggers. Software extension of Night Racer in general may present a few problems though, since the code is largely undocumented, and some parts of the software architecture is quite disorganized. This is largely due to a lack of previous experience in XNA, and that coupled with having a very early deadline of producing a first playable demo version was quite dissuading of proper planning of code structure. Doing the project over, some time would be spent in the software modeling stages.

Using XNA for the project has been a mostly positive experience, but for a game with the express goal of being visually pleasing, using a more complete engine would have been advisable. Game engines such as Unity or Unreal Engine would have most likely allowed more time to be focused on the graphical fidelity in a much more detailed way, and more time could be spent perfecting the graphics instead of constantly building in more techniques. This would of course not facilitate the in-depth research of graphical techniques conducted during this project, but it is the recommendation of the authors to game developers operating under similar conditions to extensively consider such an option, as the results will be more pleasing.

It is easy to get stuck discussing what could have been executed better, but the authors of this thesis consider the project and Night Racer itself a success, and while it will not attract many players, it may have the potential to do so with some further work.

10 Conclusions

This Bachelor's thesis has described the development of a racing game with focus on visually pleasing 3D graphics and plausibility of future commercial viability. Secondarily, providing basic playability as well as a multiplayer mode has been goals of the project. The game, called Night Racer, has been developed by a small-scale team under a 16 week time constraint, and programmed using Microsoft's XNA framework with the C# programming language.

The problem areas which have been researched and presented in their respective chapters in this thesis are the game engine implementation, level design methodology, identification of applicable graphical algorithms for providing visually pleasing results, optimization without sacrificing visual quality, game world immersion with audio, and techniques for providing smooth gameplay over a network.

Night Racer features a game engine consisting of a game state manager, a user interface with clear and non-distracting elements, a game mode with both singleplayer and multiplayer capabilities, and a trigger system which is used for checkpoints. While not featuring several game modes within a singleplayer campaign, the game engine is very extendable for such additions. The game also features procedural generation of terrain using a Perlin noise generator, creating greater replayability with diverse levels. As the terrain is randomly generated, the generator uses a unique seed value which allows for levels to be efficiently distributed to other players over the network.

The goal of a visually pleasing game has been reached by extensive use of graphical effects such as lighting, shadows, reflections with environment maps, and particle systems. To further improve the results, post-processing effects (namely blur, bloom, and motion blur) are used. The audio assets of the game are not complete, however, the sound engine used in this game enables the addition of new assets with ease. As mentioned earlier, network play is implemented providing smooth gameplay through use of the technique known as dead reckoning. Finally, rendering optimizations like viewing frustum culling and backface culling are implemented to make the game run smoother by identifying areas not visible to the camera.

The results are deemed by the authors as quite good considering the choice of a development environment that provided relatively few complete game functionalities. The authors would advice that a more complete development kit, such as Unity or UDK, would be considered in order to quickly create a working game, instead of building the game from scratch. While not ideal for developing a visually pleasing game under a time constraint, XNA does allow for interesting insight into the underlying techniques of game development.

11
Future Work

Although the results of this project were successful, the game still misses some of the features discussed at the start and during the project. Some of these features are quite easy to implement, such as polishing of the interface, and could make an apparent difference to the overall game experience. Other features are more substantial and would require numerous working hours to implement.

The main component missing in the game is content, as discussed before. Content can be added to several areas, such as adding simple assets throughout the game world using simple billboard models. Additionally, providing more cars in the car chooser is arbitrarily easy if using pre-made models, but has not been a focus during the development. Making the levels more interesting could be accomplished by using the pseudo-random generator for more environmental variables such as ground texture, height differences on the track, and spawning events such as thunder-storms. Related to the variety in content, the algorithm to position the assets in the world can be greatly improved. Convincing positioning of assets is not trivial, as each asset needs to be dependent on the already positioned assets.

Another feature lacking in order to attract the players to continue playing is a single player campaign featuring multiple game modes with varying difficulty levels. The goal of each level could be to finish varying objectives, such as destroying the opponent's car. The game could also feature some sort of collection-based awards to achieve further replayability. This is quite easy to implement using the game mode system implemented in the game engine.

When providing single player gaming, the ability to save levels is valuable. This could either be implemented by saving the generated components of the map (such as the height map of the terrain) or simply saving the random seed used to generate these, discussed in Section 3.4. The advantage of saving the seed is the portability; the seed is of very limited size, thus problems with distribution and memory do not arise. However, the disadvantage being increased load times as the components need to be regenerated every time the player starts a level to play.

The ability to save maps is also applicable to the multiplayer game modes as players probably would like to compete on the same map to improve their high scores. The results can be saved attached to the seed to get a scoreboard easily distributable to a central server.

As mentioned in the Discussion section, the game lacks real physics simulations. With a real physics engine implemented, the game would feel more realistic and fun, as it may enable the players to jump with their cars and provide more realistic car handling. The players in multiplayer game modes could also collide with one another to make the other player crash instead of competing against each other under the equal circumstances presented when not having a collision response between cars.

Finally, more optimizations could be implemented to support less powerful hardware. This could be done by choosing models with lower resolutions and optimize the prelighting process as they have been found to require the most computational power.

12

References

Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering*. Natick, MA: A K Peters, Ltd.

Andaur, C., et al. (2002). Blender Documentation Volume I - User Guide. *Stichting Blender Foundation*. `ftp://ftp.ucr.ac.cr/blender/documentation/htmlIold/book1.html`. (28 Apr. 2013).

Aronson, J. (1997). Dead Reckoning: Latency Hiding for Networked Games. *Gamasutra*. `http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php`. (27 Apr. 2013).

Bao, H. and Hua, W. (2011). Basics of Real-Time Rendering. In *Real-Time Graphics Rendering Engine*, pp. 7–20. Berlin: Springer-Verlag.

Bernier, Y. W. (2001). Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. *Valve Developer Community*. `https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization`. (26 Apr. 2013).

Bertz, M. (2012). Assassin's Creed III: The Redesigned Anvil Engine. *Game Informer*. `http://www.gameinformer.com/b/features/archive/2012/03/28/ac-iii-the-redesigned-anvil-engine.aspx`. (12 May 2013).

Bittner, J. and Wonka, P. (2003). Visibility in computer graphics. In *Environment and Planning B: Planning and Design*, vol. 30, no. 5, pp. 729–755.

Blinn, J. F. (1993). Backface culling snags (rendering algorithm). In *Computer Graphics and Applications, IEEE*, vol. 13, no. 6, pp. 94–97.

Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. In *Commun. ACM*, vol. 19, no. 10, pp. 542–547.

Bracken, C. C. and Skalski, P. (2009). Telepresence and Video Games: The Impact of Image Quality. In *PsychNology Journal*, vol. 7, no. 1, pp. 101–112.

Catmull, E. E. (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Salt Lake City: University of Utah.

Clark, J. H. (1976). Hierarchical geometric models for visible surface algorithms. In *Commun. ACM*, vol. 19, no. 10, pp. 547–554.

Coulouris, G., et al. (2012). *Distributed Systems: Concepts and Design*. Harlow, Essex: Pearson Education.

Danovaro, E., et al. (2006). Level-of-detail for data analysis and exploration: A historical overview and some new perspectives. In *Computers & Graphics*, vol. 30, no. 3, pp. 334–344.

Fiedler, G. (2006). Networked Physics. *Gaffer on Games*. `http://gafferongames.com/game-physics/networked-physics/`. (26 Apr. 2013).

Fiedler, G. (2010). What every programmer needs to know about game networking. *Gaffer on Games*. `http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/`. (24 Apr. 2013).

Fowler, M. (2005). The New Methodology. *Martin Fowler*. `http://martinfowler.com/articles/newMethodology.html`. (3 Jun. 2013).

Fox, B. (2004). *Game Interface Design*. Boston: Course Technology/Cengage Learning.

Frade, M., et al. (2012). Automatic evolution of programs for procedural generation of terrains for video games. In *Soft computing*, vol. 16, no. 11, pp. 1893–1914.

Frahling, G. and Krokowski, J. (2005). Online Occlusion Culling. In *Algorithms – ESA 2005*, pp. 758–769. Berlin: Springer-Verlag.

Gambetta, G. (2010). Fast-paced multiplayer (part I): Introduction. *Gabriel Gambetta*. `http://www.gabrielgambetta.com/?p=11`. (4 May 2013).

Gamma, E., et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Pearson Education.

Gouraud, H. (1971). Continuous Shading of Curved Surfaces. In *IEEE Transactions on Computers*, vol. C-20, no. 6, pp. 623–629.

Hale, D. H. and Youngblood, G. M. (2011). Using navigation meshes for collision detection. In *International Conference on Foundations of Digital Games; 18 Jun-1 July 2011, Bordeaux*, pp. 316–318.

Houston, M. and Lefohn, A. (2011). Beyond Programmable Shading Introduction. *Stanford Computer Graphics Laboratory*. `https://graphics.stanford.edu/wikis/cs448s-10/FrontPage?action=AttachFile&do=get&target=01-intro-BPS-2011-spring.pdf`. (18 May 2013).

Hulusic, V., et al. (2011). Maintaining frame rate perception in interactive environments by exploiting audio-visual cross-modal interaction. In *The Visual Computer*, vol. 27, no. 1, pp. 57–66.

IEEE Standards Association (2012). IEEE Standard for Distributed Interactive Simulation: Application Protocols. In *IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995)*.

Immel, D. S., Cohen, M. F., and Greenberg, D. P. (1986). A radiosity method for non-diffuse environments. In *Computer Graphics*, vol. 20, no. 4, pp. 133–142.

Kajiya, J. (1986). The rendering equation. In *Computer Graphics*, vol. 20, no. 4, pp. 143–150.

Kalogirou, C. (2006). How to do good bloom for HDR rendering. *Thoughts Serializer*. `http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/`. (1 May 2013).

Knuth, D. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Reading, MA: Addison-Wesley.

Kurose, J. and Ross, K. (2010). *Computer Networking: A Top-Down Approach*. Boston: Addison Wesley Higher Education.

Lamberta, B. (2011). Backface Culling and 3D Lighting. In *Foundation HTML5 Animation with JavaScript*, pp. 413–427. New York: Apress.

Lenhart, A., Jones, S., and Macgill, A. (2008). Adults and Video Games. *Pew Internet & American Life Project*. `http://www.pewinternet.org/Reports/2008/Adults-and-Video-Games.aspx`. (19 Apr. 2013).

Li, A. (2011). Pixel Smashers Gaming Sickness Guide. *Pixel Smashers*. `http://pixelsmashers.com/wordpress/?p=4391`. (5 May 2013).

Liang, B.-S., et al. (2000). Deferred lighting: a computation-efficient approach for real-time 3-D graphics. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, vol. 4, pp. 657–660.

Lidgren, M. (2011). Features. *Google Code Wiki: lidgren-network-gen3*. `https://code.google.com/p/lidgren-network-gen3/wiki/Features`. (5 May 2013).

Llopis, N. (2009). Shine On: Environment Mapping and Reflections with OpenGL ES. In D. Mark, ed., *iPhone Advanced Projects*, pp. 345–364. New York: Apress.

Microsoft (2013a). Prerequisites for Developing Xbox LIVE Multiplayer Games on Xbox 360. *Microsoft Developer Network*. `http://msdn.microsoft.com/en-us/library/bb975642(v=xnagamestudio.40).aspx`. (5 May 2013).

Microsoft (2013b). Sounds Overview. *Microsoft Developer Network*. `http://msdn.microsoft.com/en-us/library/bb195055(v=xnagamestudio.40).aspx`. (4 Jun. 2013).

Montabone, S. and Wickes, R. (2009). *Beginning Digital Image Processing: Using Free Tools for Photographers*. New York: Apress.

Nicodemus, F. E. (1965). Directional Reflectance and Emissivity of an Opaque Surface. In *Appl. Opt.*, vol. 4, no. 7, pp. 767–773.

NVIDIA Corporation (1999). OpenGL Cube Map Texturing. *NVIDIA*. `http://www.nvidia.com/object/cube_map_ogl_tutorial.html`. (29 Apr. 2013).

Orkin, J. (2002). A General-Purpose Trigger System. In *AI Game Programming Wisdom*, vol. 1, pp. 46–54.

Park, C. M. (2010). Choosing A Network Library in C#. *Games By Design*. `http://christophermpark.blogspot.se/2010/02/chosing-network-library-in-c.html`. (5 May 2013).

Perlin, K. (1985). An image synthesizer. In *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 287–296.

Perlin, K. (2005). *Standard for perlin noise*. US 6867776 B2.

Phong, B. T. (1975). Illumination for computer generated pictures. In *Commun. ACM*, vol. 18, no. 6, pp. 311–317.

Quandt, M. (2009). Light Pre-Pass in XNA [Part 1]. *MQuandt.Blog*. `http://mquandt.com/blog/2009/11/light-pre-pass-in-xna-part-1/`. (12 May 2013).

Raghuvanshi, N., et al. (2007). Real-time sound synthesis and propagation for games. In *Commun. ACM*, vol. 50, no. 7, pp. 66–73.

Reeves, W. T. (1983). Particle Systems - a Technique for Modeling a Class of Fuzzy Objects. In *ACM Transactions on Graphics*, vol. 2, no. 2, pp. 91–108.

Rosado, G. (2008). Motion Blur as a Post-processing effect. In *GPU Gems 3*. NVIDIA.

Sanchez, J. and Canton, M. P. (2003). Lighting and Shading. In *The PC Graphics Handbook*. CRC Press.

Sanders, K. (2006). What is the XNA Framework. *XNA Game Studio Team Blog*. `http://blogs.msdn.com/b/xna/archive/2006/08/25/what-is-the-xna-framework.aspx`. (3 Jun. 2013).

Simpson, J. (2000a). 3D Sound in Games. *gamedev.net*. `http://www.gamedev.net/page/resources/_/technical/game-programming/3d-sound-in-games-r1130`. (17 May 2013).

Simpson, J. (2000b). Networking for Games 101. *gamedev.net*. `http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/networking-for-games-101-r1138`. (26 Apr. 2013).

Smed, J. (2008). Networking for Computer Games. In *International Journal of Computer Games Technology*, vol. 2008, p. 8.

Smith, W. J. (2007). *Modern Optical Engineering: The Design of Optical Systems*. New York: McGraw Hill Professional.

Thorn, A. (2011). *Game Engine Design and Implementation*. Sudbury, MA: Jones & Bartlett Learning.

Valve Developer Community (2012). Source Multiplayer Networking. *Valve Developer Community Wiki*. `https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`. (26 Apr. 2013).