CHALMERS | GÖTEBORGS UNIVERSITET

DATX02 - Bachelor's thesis in Computer science and engineering

Project group 69

Bury the Needle

A racing game in open terrain

Matz Johansson Bergström	matzjb@yahoo.se
Tommi Kerola	kerola@student.chalmers.se
Andrej Lamov	andrej.lamov@gmail.com
Ola Palholmen	ola@palholmen.se
Hugo Pila	mr_sueko@hotmail.com
Bahareh Sadeghi	baharehs@student.chalmers.se

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2011

Abstract

This bachelor's thesis describes the implementation of a 3D racing game for the Windows platform using the XNA framework. Presented are results and discussions concerning fundamental aspects of implementing a modern computer game: 3D modeling, computer graphics, physics, particle systems, audio, networking and artificial intelligence.

This thesis shows that developing a computer game using an iterative, incremental approach is a feasible task. We demonstrate possible solutions for solving tasks common to modern game development.

Contents

List of Figures

1 Introduction

With the mass market for computer games growing even bigger than the movie industry [1], computer games constitute an interesting software development challenge. Games are made up of complex parts, which require a broad variety of different knowledge: programming, graphical design, sound, physics, story telling and game play. The creation of a game is a challenging task and has therefore become a field of its own.

The problem examined is the process of creating a 3D-rendered vehicle racing game within a specific time limit. This includes research, design and implementation of advanced technology, handling different aspects of game development, and finding out suitable solutions to game related problems.

The ability to create large and complex software, with a satisfying result, is always a very demanding task. This problem is interesting for developers, but also for people working with the economic aspects of promoting and selling game software, and for people that are just interested in computer games as well. However, this report is intended for bachelor's level students.

1.1 Purpose

The purpose of this project is the creation of a modern 3D-rendered vehicle racing game with good-looking graphics and realistic collision detection and response. The target platforms are Windows and, if time permits, Xbox360. Additionally, the practical programmatic use of the related aspects of modern computer games are explored. These aspects include developing and coding with C# using XNA, and using HLSL for the creation of graphical effects. Also, the use of particle systems, physics for simulating realistic vehicle collision behaviour, managing low latency network programming and creating a suitably intelligent artificial intelligence system for simulating a challenging game experience is presented. The process of basic 3D model and texture creation for inhabiting the rendered 3D world, along with sound effects, is also explored. These aspects will ultimately contribute to a total game immersion. Additionally, the project is subject to much needed optimization techniques for keeping game performance requirements at a low end.

Nonetheless, the most important goal of this project is that the developed game will be perceived as both challenging, intriguing and enjoyable.

The purpose of this report is to answer questions about the implementation and use of the above mentioned aspects. This report presents several solutions to such game-related problems, which may be of use for future students and other interested people who wishes to explore the process of game creation.

1.2 Limitations

The end result of this project, a racing game, is targeted towards running on Windows and, if time permits, Xbox360. Since Xbox360 is made from fixed hardware, as opposed to a Windows PC, where a mix of old and modern hardware could exist, the aim is to make the game for the fixed hardware platform of Xbox360.

The hardware of the different platforms creates a limit on how many effects can be added to the game. Also, the mix of hardware and the flexibility of the PC market makes it important to be able to regulate the level of quality in the game so it can run smoothly on an older PC as well as adhere to the demands of a state-of-the-art PC.

For these reasons, it was decided to use an incremental model of development; to build a game that was initially small and visually modest, which would then grow into a complex and graphically impressive game. Therefore, as the project progressed, additional effects and sophisticated functionality were added. During the project's growth, an important benchmark that had to be met was attaining a frame rate of 60 frames per second on a Xbox360.

Complicated artificial intelligence, sound and optimization was not the main focus of this project. The decision of adding artificial intelligence, sound and optimization into this game was based on the mere necessity of creating an enjoyable game experience. Microsoft charges a fee for using the XNA network API [2]. Therefore, this project did not use their API. Instead, the lidgren-network-gen3 library was used. Furthermore, in order to not spend time implementing complicated physics code, the physics used in the project were handled by using a physics engine. The only requirement was for the physics engine to be realistic enough to create a resemblance to a real-world driving and collision experience. Lastly, less focus was directed at creating a massive amount of game content, such as levels and models of cars and other environmental objects. Instead, a sufficient amount of game objects was included to merely be able to demonstrate the implemented features of the game.

1.3 Report outline

This report is structured into several main sections, each discussing a separate aspect of this project. The report starts with a description of the game itself and its internal structure. Thereafter, solutions and ideas concerning fundamental aspects in modern game development are presented in the following sections:

- Modeling
- Real-time graphics
- Physics

- Level editor
- Optimizations
- Audio
- Networking
- Artificial intelligence

Each section will introduce its topic, after which results and discussions follow. The report ends with a conclusive discussion of the project and game programming in general.

2 The Game

This section is intended to introduce our game idea and theme to the reader and also give an explanation of the class hierarchy of our code.

2.1 Background story

While our game does not have an explicit plot, the basic gameplay layout is similar to a classic car racing game, where you compete against either computer-controlled vehicles or actual players through a network connection. In addition to concepts familiar to regular players of computer games, the game should feature realistic collisions, deformation and destruction of other vehicles.

2.2 Theme

The game is set in the rural areas of a post-apocalyptic earth. The reason for this is that we early on in the discussions wanted the game to be set in an environment that felt as open as possible and was not constrained to a road. Additionally, a game in a city environment would require a lot of modeling, texturing (see Section 3.1.1, Texturing) and design. This would generate an infeasible workload (with respect to this thesis' deadline) and would also go against our idea of an "open environment".

Often, when playing games and racing games in particular, a feeling of "what is beyond those trees?", or "I wonder if I can see what is beyond that fence..." can arise. Racing games such as "Need For Speed 3" (1998) [3] have been limited in that way since they are constrained to racing on a road. This constraint is due to the memory limitation of the PCs in the late nineties. However, today, with games such as "Mafia 2" (2010)[4], we start to see a larger explorable and more detailed environment, which we found appealing to use in our project.

The world in the game is "post-apocalyptic", meaning that it is set after a catastrophe, such as a nuclear or viral attack. Because of this, the population is effectively zero (apart from people driving the vehicles) and as a result, the man-built constructions wither away, squeak and fall apart. Using this theme means that the creation of animations of people can be avoided, which is also a very time consuming and difficult task.

The inspiration for the theme comes from movies such as "Resident Evil:Extinction" [5], "Mad Max 2" [6] and "The Book of Eli" [7], and games such as "Fallout 3" [8] and "Motorstorm: Apocalypse" [9], to mention a few.

2.3 Class Hierarchy

The architectural pattern used is the Model-View-Controller concept. Model View Controller (MVC) is a software pattern used to divide the code into logical parts [10]. MVC describes the three logical parts: The

Model which handles logic, View which handles the graphics and Controller which handles input via game pad/keyboard. The main goal of the method is to separate the concerns of different tasks and thereby increase code reuse and maintainability [10].

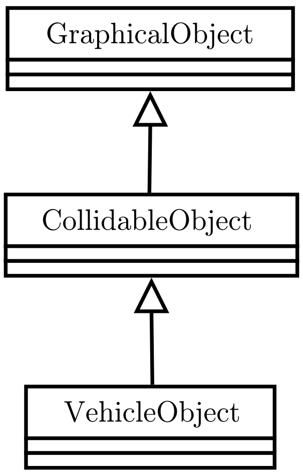


Figure 1: The fundamental line of inheritance for objects in this project.

The data models used are basically extensions of a graphical 3D model. The line of inheritance, as can be seen in Figure 1, is as follows: GraphicalObject contains information needed to render the graphical model on the screen such as position and model. CollidableObject is inherited from GraphicalObject, thus containing all properties of GraphicalObject. The main purpose of CollidableObject is to give an object that it is possible to perform collision detection on. This is done by integrating this object with our physics engine of choice (see Section 5.1) and performing updates on each of these object via a separate physics controller class.

Vehicle inherits from CollidableObject and its parents. This model is abstract, so it cannot be created without a using subclass because each vehicle has to define various parameters, which control the vehicle's physical steering behaviour (see Section 5.3, Car physics).

Simple data models that are self explanatory, e.g. a sound container and car type, are excluded from the diagram but are still part of the system.

As for controllers, we have one for each Vehicle object whose purpose is to keep track of a specific type of vehicles. In addition to this, there is a physics controller which keeps track of all objects that need physics calculations. Since there is a tremendous amount of work developing a physics engine, the physics controller uses, as previously stated, an external library called JigLibX to do most of the calculations involving collision detection and response. For sound and music in the game, we will have a sound controller whose purpose is to calculate where the sound is orienting from (for 3D sounds) and distance to the sound in order to regulate sound volumes levels (see Section 8, Audio). Networking will use the lidgren-network-gen3 library so there was no need to develop basic networking ourselves (see Section 9, Networking).

3 Modeling

While the aim of this project is to create a 3D game, we think the core part of our game is the graphical content. The process of creating 3D models, which are to be rendered to the screen, is called modeling, which is the topic of this section.

3.1 Model Creation

In this section, we will discuss modeling. Information about texturing, along with examples of the processes, will also be provided.

A 3D model consists of normals and triangles, and this piece of information defines the shape. The process of creating a 3D model requires the use of a 3D modeling software. There are several modeling editors available. There is free software, such as Blender [11], and proprietary software, such as Autodesk:Maya [12] and Autodesk:3D Studio Max [13] (3DS Max). The choice was 3DS Max due to prior knowledge.

3DS Max is a complex, yet easy-to-use software with an intuitive Graphical User Interface (GUI). With 3DS Max, you will have a virtual world in which you can create models that can be imported into the game environment in a simple way. The file format used for this is called FBX [14, 15].

3D models are made up from triangles and textures. A texture is basically a part of an image that is mapped onto the coordinates of a triangle face [16]. Creating an object usually require thousands of triangles to look convincing. Each texture element (texel) is mapped from 2D image onto a 3D object using different projections, so called UV maps. The UV is the coordinates (u,v) of the texture. The main task of the modeler is to create the triangles and to map the 2D image onto these triangles in a convincing way.

To create textures, photos and image-editing software is used. There are many different types of imaging software available. As with modeling software, the choice is between free and proprietary.

Free software sometime lack in features and intuitive, well-organized GUI design. An example of this can be seen in Blender. By the new official release of Blender 2.57, the GUI was fully reworked [17] and according to some the new version is even better than its professional counterparts.

Software like Gimp (GNU Image Manipulation Program) [18] and Paint.NET [19] has features such as layering images, a palette of different brushes and the basic set of tools needed to manipulate and combine images into textures in an easy fashion.

On the proprietary list of software there are, for instance, Corel Draw [20] and Photoshop [21]. As with the choice of modeling software, prior knowledge of Photoshop made this choice simple.

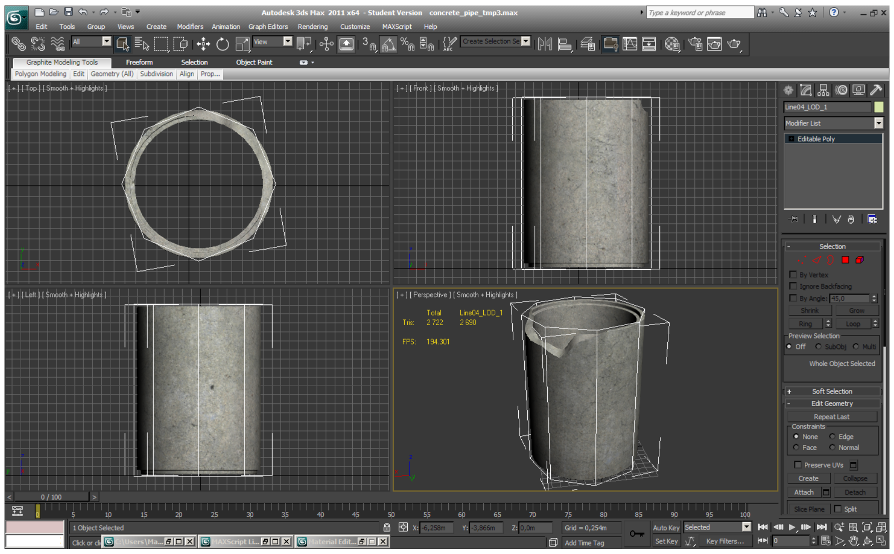


Figure 2: A screenshot of the Graphical User Interface of 3DS Max. Note the vast set of tools available for the modeler and the different views and toolbars.

3.1.1 Results

The process of learning 3DS Max is a complex and advanced task. The amount of tools available for the modeler makes 3DS Max complex and the advanced technical expertise needed to solve a modeling task demanding. Since one of the members of the group had several years of experience with both 3DS Max and Photoshop, video tutorials were created and used to communicate ideas and lay out a workflow of the modeling and texturing process.

Prior to describing conclusions, explanations of modeling and texturing workflow are presented with a few examples.

Modeling In 3DS Max, an interface allows the creation and modification of 3D objects in real time. There is also navigation in a total of four viewports: three of which are orthographical projections and one is a perspective view (i.e., the way the object would look in reality). Each time a 3D object is being modeled, we follow a set of internal rules. These rules were invented to make the modeling and texturing as efficient as possible. These are:

1. Keep things simple
2. For repetitive actions, create/use a script (MAXScript)
3. Keep everything in the right scale

4. Model as fast as possible
5. Never scale an object[1] directly.

The first rule is the most important, not only for modeling. Keeping things simple is not always a simple task, since there was a lot of 'trial and error' before we could find a simple modeling procedure. In modeling and texturing, it means to plan the modeling and to just use the right amount of detail to accomplish the task at hand. The texturing and modeling stages come hand in hand. When making a complex 3D object, the texturing stage will be cumbersome since projecting to a larger number of triangles has to be handled.

The second rule is about utilizing the 3DS Max's scripting language (see Section 3.2).

Third, always keep everything in the right scale, especially in a project with several members.

Fourth, the practice of modeling as fast as possible is to prevent the possibillity of modeling to perfection. Perfection is something to strive for, but that is never reached. Making a wooden box look perfect essentially means nobody will notice the difference, unless they have something to compare to. Focus should initially be put on just making the objects work in the environment and then on creating the right atmosphere.

As a fifth rule, never scale an object directly. This is because of the way 3DS Max is designed [22]. In 3DS Max, each model is represented in the world using a transformation matrix. A transformation matrix contains information of an object's rotation, scale and position [23]. If an object is rotated, the vertices will be displayed as rotated around a pivot point. It is not until the object is converted to a *mesh* that the matrices will be multiplied with the vertices and stored for each vertex. Now, if an object is scaled, the display will show the object as scaled, but in reality the scale is stored in the 'scale matrix' and not by the vertices themselves.

There are different (unofficial) disciplines in creating 3D objects. We will run through two ways of modeling and give the drawbacks of the techniques along with some examples.

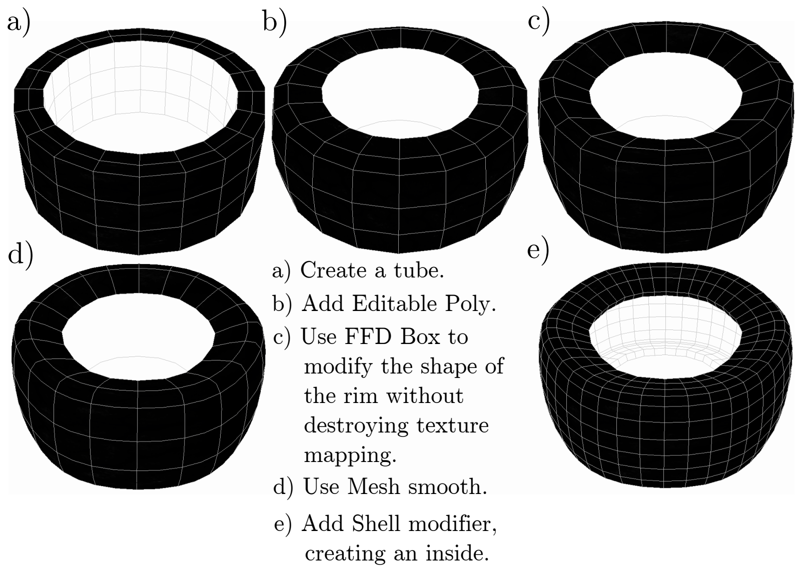
Box modeling: This is the more complicated of the two modeling techniques. It starts from a simple box and from this box more detail is added by using modifiers like extrusion and beveling. This technique is well suited for car modeling since it gives absolute control of the models and allows keeping the number of vertices down, which is good when unwrapping. A car is basically in the shape of a box.

Spline shape modeling: This method is mainly used for cylindrical objects like tires or tin cans. The main modifier used is called "shell". Shell makes a one-sided 3D object into a two-sided object and also provides the UV coordinates for the inside, making this modeling technique

[1]An object consists of several meshes.

very fast. Later, we will show how we modeled the tires in the game using spline shape modeling

Modeling, an example Modeling is best shown with an example. The modeling of a tire could be done using either of the techniques mentioned in Section 3. Here we use a lathed spline.



a) Create a tube.
b) Add Editable Poly.
c) Use FFD Box to modify the shape of the rim without destroying texture mapping.
d) Use Mesh smooth.
e) Add Shell modifier, creating an inside.

Figure 3: Tire modelling creation steps.

FFD Box modifies the shape of the object but keeps the UV-mapping intact.

All of the above steps are important for the appearance of the tire. We introduce UV mapping at step b), before we add shape and more vertices, since it is easier to modify the projection map with fewer vertices (see below).

Texturing The texture, or material, of each model is based on 2d images which are mapped onto the triangles that make up a model (see Figure 4). Since the texture is mapped from 2D image to a 3D model, the UV template is inevitably cut, resulting in seams. These seams are smoothed using different techniques to make the seams less visible. To enhance the visible appearance of a model, different types of textures are used such as: specular mapping, normal mapping and diffuse mapping. The diffuse map provides the basic color of the material, e.g., the blue color and brown dirt on a blue car hood. For more details, see Section 4.4.

After the modeling is complete, we need to define the UV projection technique (see Figure 5). More advanced texturing, involving specular and normal maps, is only used for the terrain. This is because the FBX format, used by the models, does not support multiple textures. Additionally, the heavy workload put on the modeling team has prevented adding these.

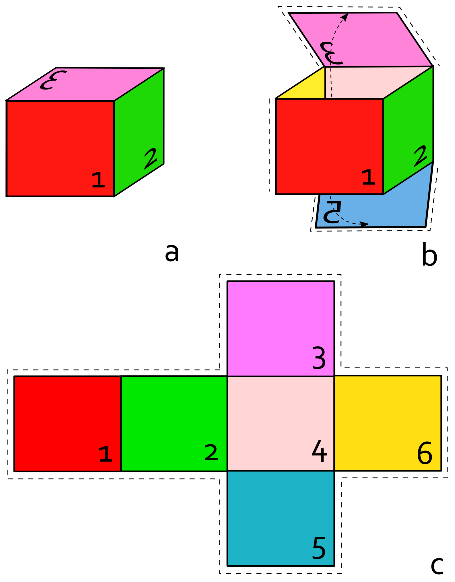


Figure 4: The UV mapping shown on a box, **a**. In **b**, the seams are marked with dashed lines as a result of flattening the box. **c** illustrates the box unfolded as a UV template.

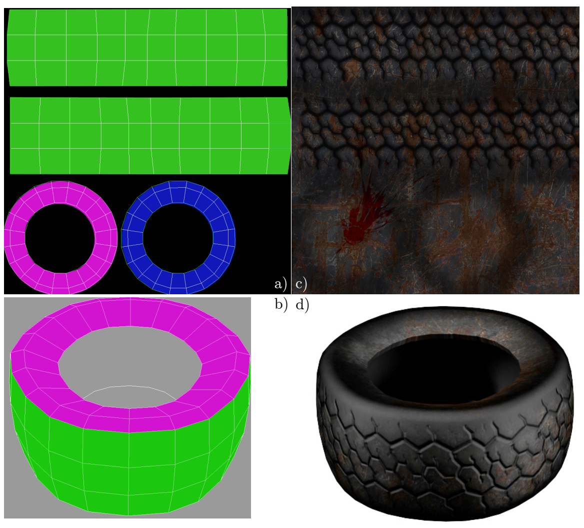


Figure 5: The texturing and mapping of a tire. **a** shows a colored UV template for the purpose of clarity. **b** shows how the projection is mapped onto the tire. **c** is the diffuse texture. Specular and normal is painted in the same manner. **d** shows the resulting model with normal, specular and diffuse materials applied. Please note how the indented rim catches the light (the specular and normal maps are only used by the terrain, see Section 4.4).

3.1.2 Discussion and conclusion

Modeling and texturing in 3DS Max and Photoshop is time consuming. Using the scripting language MAXScript saves time and has been useful for speeding up repetative modeling steps. Shading effects such as normal mapping and specular mapping adds a heightened sense of realism and would be interesting to implement in the future for all the objects in the world, not only the terrain.

3.2 MAXScript

MAXScript is the scripting language of 3DS Max. The language is similar to the programming language BASIC, which also has a very simple syntax. MAXScript was developed by John Wainwright as an addition to 3D Studio Max R2 (1997) [24]. The main purpose of MaxScript is to ease the modeler's workload and make repetative tasks easy to program and run as a script.

The work flow of the modelers was carefully analyzed, and as a result, a script that makes the objects simpler and faster to export to the game was developed.

The modeler's workflow consists of the following:

1. Collecting reference data, as an aid in modeling
2. Collecting textures (in our texture library)
3. Planning the modeling, draw a rough version on paper
4. Create the model
5. Unwrap the model
6. Texture using the unwrapped template
7. Create LoDs
8. Create Collision Skins
9. Translate to the origin
10. Rescale
11. Export to FBX
12. Make sure textures are seen by the game

3.2.1 Results

In the project, a script coded in MAXScript was created to make the modeling more efficient and consistent as the models are imported into the game. One important result of the script is that the possibility of naming a bounding volume object (see Section 5.2.1) in the wrong way is eliminated.

Example code using MAXScript:

```
select shapes
for s in selection where
    s.modifiers.Count==0 do
    if s!=undefined do
                delete s
```

The code above selects all the shapes (splines and lines) and removes the shapes whose modifier stack is empty. Note that we cannot simply remove all shapes, since there might be models using splines (for which they always have at least one modifier).

The first points (in the modeling workflow list) were created continuously during the project and were accessible for all the members of the project via Dropbox [25]. The last part of the modeling workflow was repeated for each object and could therefore be scripted using MAXScript.

The script that was developed for this project makes the last steps of the modeling more efficient, since this part of the workflow is repetative and usually needs little new input. To keep the script simple and consistent, when using the script, a series of steps has to be performed in a specific order. For instance, when we name the bounding volumes, we know that a group named *_LOD_1 is placed on the origin, which is very important (this will be further discussed later). The script was written and given a GUI to enable an easy interaction with the 3d objects (see Figure 6). An order is enforced by the script by deactivating buttons and spinners that are not to be used yet.

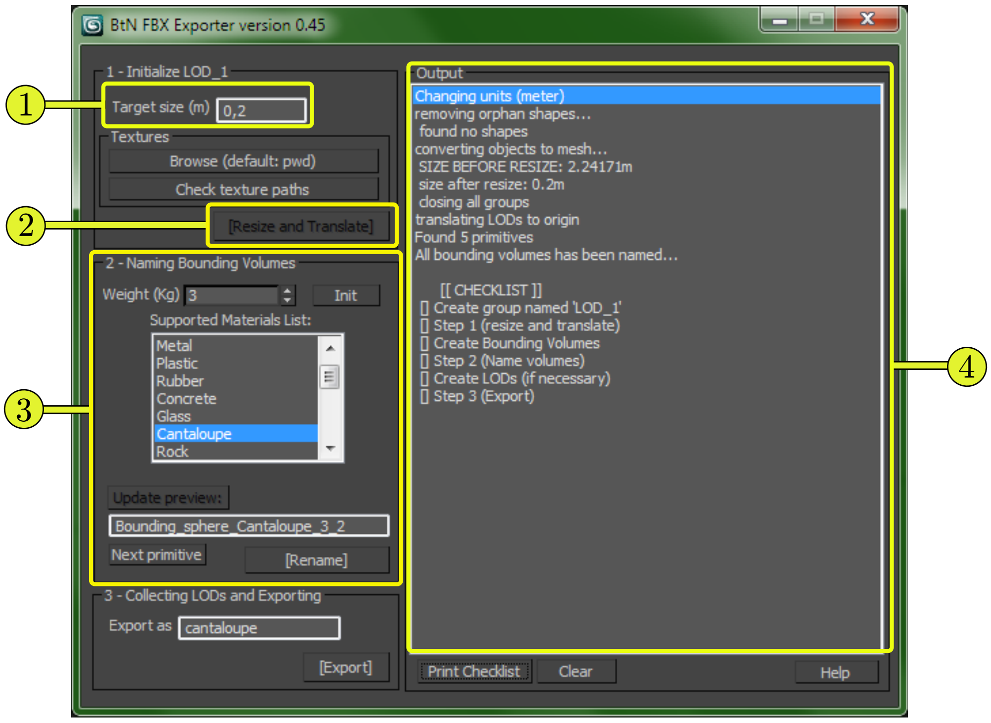
The grouping of the objects is used to automatically rename each object in the group with "_LOD", which is used by the game (see Section 7.3 for more information).

Below is a short list of the main features of the script. Some are used as a button and some are used internally.

Relocation of texture paths: The script tries to find textures given a texture path. This is saving a lot of time when using textures from different places and later relocate them. The alternative would be to use 3DS Max "Asset Tracker", which was too slow when changing many texture paths at once. This script is useful if an object uses many textures (such as specular, normal and diffuse).

Resizing an object: This is a simple, yet very useful button, which lets you use 3DS Max's "ReScale World units". The ReScale World units function rescales objects to a given the new scale. Instead of explicitly specifying the amount of scaling, the script computes the rescaling necessary to transform the object into a target size.

Translating to the origin: While this task may sound simple, translating to the origin is a very important task when using LoD groups. By using this code for origin translation, we can manage the LoD groups in a controlled way that is easy for the modeler to handle.



```
Changing units (meter)
removing orphan shapes...
 found no shapes
converting objects to mesh...
 SIZE BEFORE RESIZE: 2.24171m
 size after resize: 0.2m
 closing all groups
translating LODs to origin
Found 5 primitives
All bounding volumes has been named...

     [[ CHECKLIST ]]
 [] Create group named 'LOD_1'
 [] Step 1 (resize and translate)
 [] Create Bounding Volumes
 [] Step 2 (Name volumes)
 [] Create LODs (if necessary)
 [] Step 3 (Export)
```

Figure 6: The resulting GUI using MAXScript and a Visual MAXScript editor to create buttons and spinners. The target size (1) is specified, then the model is resized and translated (2). Next, the naming of the bounding volumes occurs (3). An output window (4) keeps track of the changes made to the scene.

Naming bounding objects: This is one of the most time consuming part of the preparation for export. In the project, we provide several materials for the bounding volumes, since they produce sound and also have a material property used in collision response. The materials accessed by the script are printed in a list, along with a spinner for weight and a way to select through the available primitives. This makes the bounding volume naming routine simple and efficient.

3.2.2 Discussion and conclusion

Although much time is spent modeling and texturing, scripting is generally difficult at the modeling and texturing stages, since it involves the artist in a non-predictive and non-repetitive way. However, the script was very helpful in minimizing errors and enforcing the internal standard.

Additional features such as stepping through the files to be exported in a conveyor belt fashion would be nice to have. However, there is a limit on how much automatic work one script can do. The script would have become too large and complex if each case was to be implemented as a button. Care has to be taken when developing an 'ultimate script' since it will easily grow into something large and unmaintainable.

4 Real-Time Graphics

This section describes the real-time graphics - an immensely important aspect of any 3D game. We will start this section with an introduction to HLSL, since this is central to all sections on graphical effects. The rest of this section assumes a basic understanding of HLSL. After the HLSL introduction, the graphical effects and features implemented in this project will be presented.

4.1 HLSL

HLSL stands for High Level Shading Language, meaning it provides functions at a high level from the hardware. This means that we do not have to produce low-level code to communicate with the hardware, but instead use HLSL as the communication bridge. The language HLSL was created by Microsoft and works on Windows and Xbox360.

HLSL is divided into three shaders. These are: vertex, geometry and pixel shaders. The layout of the graphics card can be seen in Figure 7. The vertex shader is responsible for moving individual vertices, while the geometry shader is responsible for creating new vertices. This is not supported by older graphics hardware (prior to Shadermodel 4) [26]. The last and most important part of the shading is the pixel shader. This shader manipulates the colors of the pixels on the screen. These three shaders are used in conjunction, similar to a pipeline, in order to achieve special effects.

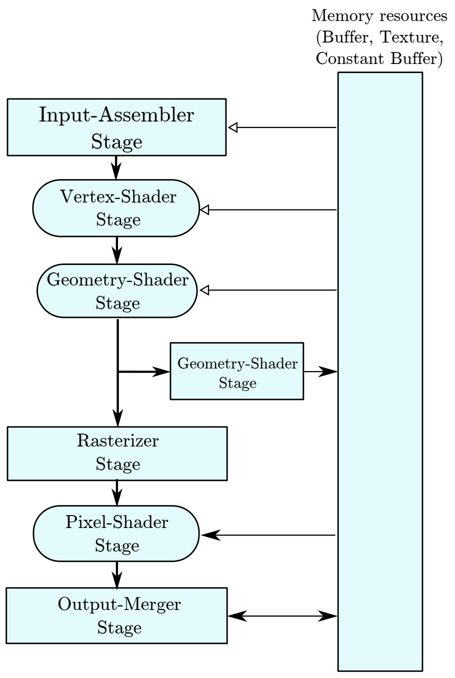


Figure 7: The pipeline of DX10 graphics card. Note the pipeline from vertex to geometry and pixel shader [27].

4.2 Terrain

One goal of our game was to let the player drive around in an open landscape. Rather than having explicit roads to travel on, the player should be able to drive around, without restrictions, in a 3D landscape of an arbitrary style.

A way of generating an arbitrary 3D terrain from a heightmap is proposed in [28]. The proposed technique in this sample utilizes the XNA Content Pipeline [29] to build the whole terrain as a single mesh by looking at a grayscale heightmap in order to see at which height each vertex of the terrain should be located. By using this heightmap-terrain solution, the map designer can easily modify the terrain through the heightmap texture.

4.2.1 Results

Although we made use of the heightmap-terrain technique, we soon discovered that creating a large terrain directly was not an efficient solution. Building large terrains required sending hundreds of thousands of vertices to the GPU. Also, since we used the XNA Reach Profile settings, we were limited to sending 65535 vertices per mesh [30].

This was solved by modifying the terrain generation code to divide the generated terrain mesh into a grid of several smaller meshes. By doing this, each smaller terrain mesh now only consists of blocks of 100x100 pixels on the heightmap.

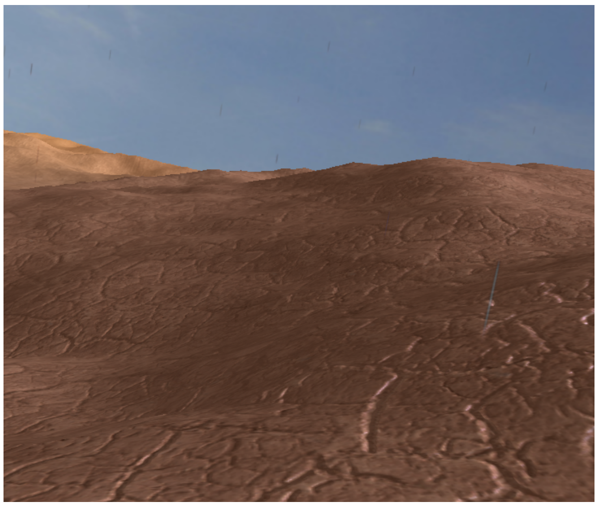


Figure 8: The resulting rendered terrain by using the heightmap-terrain technique.

4.2.2 Discussion and conclusion

The choice of utilizing the heightmap-terrain technique gave us a way of creating a result according to our goals. Although we had to modify the generated terrain in order to cope with larger terrains, this modification turned out to be useful later when performing frustum culling on the terrain (See Section 7.1, Frustum culling).

4.3 Splat Maps

Since our game contains a vast landscape, an easy way of texturing the terrain was needed. Rather than letting our modelers create specific 3D models of road parts to place on the terrain, we felt a need to be able to draw roads and vegetation directly onto the 3D landscape.

As described in [31], a technique called multiple texture mapping can be used to place arbitrary textures on a 3D landscape. This multitexturing technique is referred to by some as 'splat mapping' [32, 33]. Splat maps are used for specifying what kind of textures will be present in certain parts of the terrain ground. The world position of a pixel is mapped onto the splat map to determine how much of each available ground texture to use at a specific point. Each color channel (red, green, blue, alpha) represents one type of texture to place on the terrain. When specifying colors that are a blend of several of the channels, the textures are blended accordingly by using multitexturing in the pixel shader.

4.3.1 Results

We found splat mapping to be an efficient way of achieving our goal. With splat mapping, we could easily look at our height map and thereafter draw an accompanying splat map that defined where to plot certain textures onto the ground. The resulting color of each pixel is calculated through the following formula in the pixel shader.

```
color = (map.r * redTex) + (map.g * greenTex) +
        (map.b * blueTex) + ((1-map.a) * alphaTex)
```

As seen in the above formula, the red, green, blue and alpha components of the (splat) map determine how much of the red, green, blue and alpha textures should be used when rendering a pixel. When creating a splat map for a terrain, four different textures can be specified to correspond to each of the four RGBA channels of the splat map texture.

As an example, the splat map seen in Figure 9, applied to the height map in Figure 9, will result in the terrain seen in Figure 10.

To illustrate the underlying structure, Figure 11 shows the same scene as in Figure 10, but with splat map colors shown directly on the ground.

Splat maps turned out to be a very efficient solution to achieve our goal of easily specifying where on the terrain a certain texture should be used. Blending the red, green, blue and alpha colors on the splat map also allowed us to create a smooth transition between textures on the terrain.

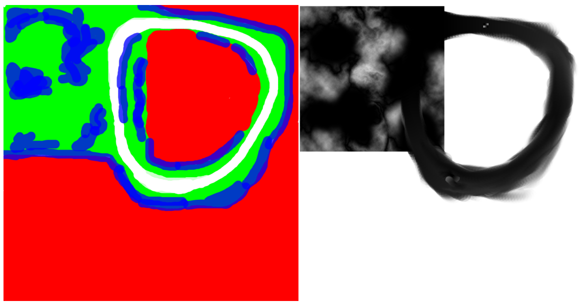
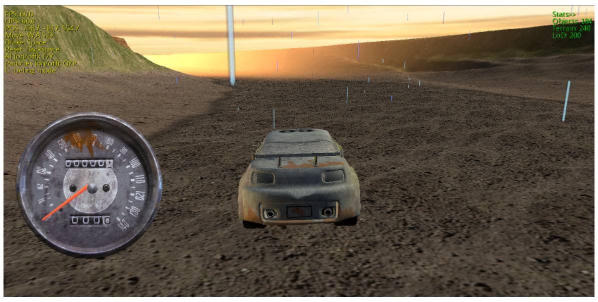


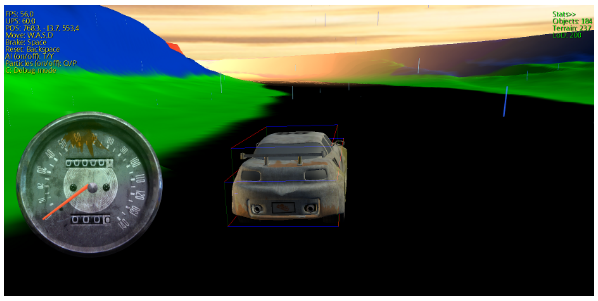
Figure 9: A splat map (left) and height map (right) representing the game terrain.



```
FPS: 60,0
UPS: 60,0
POS: 768,7, -13,7, 554,7
Move: W,A,S,D
Brake: Space
Reset: Backspace
AI (on/off): T/Y
Particles (on/off): O/P
C: Debug mode
```

```
Stats>>
Objects: 184
Terrain: 240
LoD: 200
```

Figure 10: The resulting terrain from applying the maps in Figure 9



```
FPS: 56,0
UPS: 60,0
POS: 768,3, -13,7, 553,4
Move: W,A,S,D
Brake: Space
Reset: Backspace
AI (on/off): T/Y
Particles (on/off): O/P
C: Debug mode
```

```
Stats>>
Objects: 184
Terrain: 237
LoD: 200
```

Figure 11: Game terrain with splat map colors shown on the ground.

4.3.2 Discussion and conclusion

While splat mapping is a technique which is easy to use and implement, it still suffers from only supporting four different terrain textures due to each pixels only having four color channels (red, green, blue, alpha). Although four ground textures were enough for our terrain to create a convincing 3D landscape, splat maps can perhaps be altered to support more textures. This is especially important in our game, where we map repeated textures on large areas.

One issue is that it is easy to see the repetitive patterns in a seamless texture if used over a larger area (see Figure 12). One solution to break the pattern of, say, sand ripples would be to use two versions of the sand ripples and map a large noise splat map to change between them.

Here is a possible idea of how the mapping would work to extend the

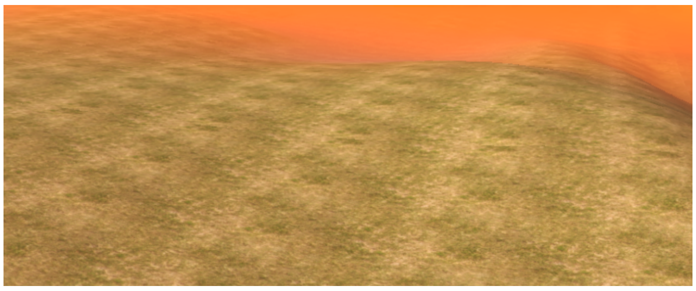


Figure 12: Seamless texture causing a visibly repeated pattern.

splat map to use eight textures, using the same map m:

```
f(m, r1, r2, g1, g2, b1, b2, a1, a2) -> Color

//contributions of the channels from map:
redc   = map.r<0.5;
greenc  = map.g<0.5;
bluec   = map.b<0.5;
alphac  = map.a<0.5;

color = (redc*r1 + (1-redc)*r2) + (greenc*g1 + (1-greenc)*g2) +
        (bluec*b1 + (1-bluec)*b2) + (alphac*a1 + (1-alphac)*a2);
//add contributions for green, blue and alpha
```

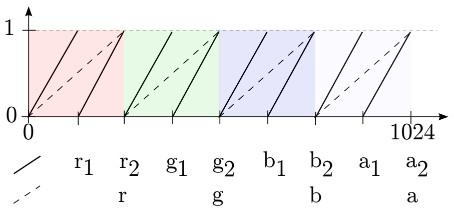


Figure 13: Mapping of (R, G, B, a) to eight textures.

Each zone in Figure 13 shows the mapping of the color for the current version (dashed line) and the proposed method (line). The drawback is that we are mapping the interpolation to half the precision. In HLSL, float precision is used, but ultimately, the "color" type is used which is 4 bytes, meaning 1024 shades, i.e. 256 shades for each channel. With the proposed method 128 shades will be used to interpolate between the textures.

Another idea is to let not each color channel on the splat map represent a ground texture, but rather a whole new splat map. This second splat map can then refer to ground textures or yet another splat map. This technique can be used to specify, in theory, an infinite number of textures on a terrain. However, the downside may be that an increasing number of splat maps have to be sent to the GPU.

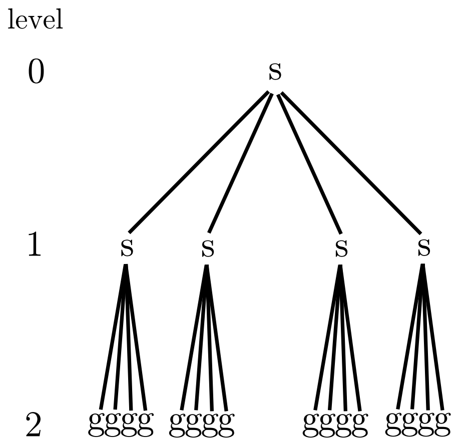


Figure 14: A figure showing the proposed technique for increasing the number of available ground textures (g) by using a tree structure relationship between the splat maps (s).

As can be seen in Figure 14, the number of available ground textures increases quickly for each level of splat-map-referencing splat maps. In fact, the number of available textures can be expressed through the expression 4^n, where n is the depth of the splat map tree. The number of splat maps sent to the GPU can be expressed as $\sum_{k=0}^{n-1} 4^k$. As this sum is a geometric sum, this can be stated explicitly as $\frac{4^n-1}{3}$. We can see that this proposed technique of increasing the number of available ground textures also requires an approximately extra 33% increase in the number of textures that are sent to the GPU. This increase is neccessary, since the pixel shader needs to have access to each used splat map. Another downside to this technique is the increased complexity of drawing the splat maps by hand. If a splat map references another four splat maps, the map designer has to keep track of the relationship between each of them. Perhaps this technique could be made easier to use, if a drawing tool, which is keeping track of the splat map relationships, was developed.

4.4 Lighting

Light itself is an immensely important phenomenon in our world. In order to create a racing game with convincing graphics, we consider it important to implement realistic lighting in our generated 3D landscape.

In the beginning of rendering, scenes were rendered using ray tracing. In ray tracing, a ray is traced through a screen and hit objects in the virtual world. Each ray hitting a surface is split and retraced through the scene until a ray hits a light or some set threshold of bouncing is reached. This method is simple, and hardware to make this method feasible has been proposed by [34]. Ray tracing is computationally expensive in real time rendering, and for games, approximations are still used. In real time rendering, lighting is defined as a sum of ambient (base lighting), diffuse (matte color) and specular (shininess) components [16].

$$i_{tot} = i_{amb} + i_{diff} + i_{spec} \tag{1}$$

In Equation 1, i stands for intensity, which is the light emitted from each pixel. In this section, it is assumed that the ambient component of the light source is constant (for more information about how the ambient component is calculated, see Section 4.5, Sunlight Model). The diffuse and specular components of the light interacts with the material's diffuse and specular properties. These properties are usually[2] defined by textures. Computing lighting and subsequently calculating the color of each pixel is called shading. There are several shading models available, which use normals to calculate the lighting for each triangle.

To calculate the diffuse lighting, Lambert's law is used[16]. The law states that for a surface with a matte material, the reflected light is determined by the cosine between n and l, the dot product. By adding the material's properties, we get:

$$i_{diff} = (n \cdot l) * m_{diff} * s_{diff} \tag{2}$$

where s_{diff} stands for the diffuse color of the light source. The calculation of the diffuse component, i_{diff}, can be seen in Figure 15.

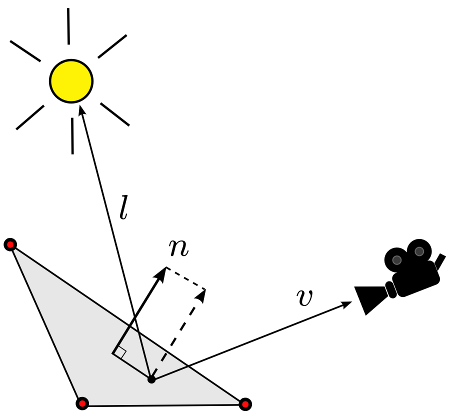


Figure 15: Diffuse lighting of a triangle with flat shading using only one normal per triangle. The normal used at each point of the surface is the thicker dashed line regardless of position.

A selection of these models include flat shading [16], which calculates the lighting per triangle using one normal per triangle. Gouraud shading [36] calculates the shading at each vertex and interpolates the coloring over the triangles (Figure 15). More advanced techniques, which calculate lighting at each pixel, include Blinn shading [37] and Phong shading [38].

[2]For some real time render applications, a BRDF (Bi-Direction Reflection Distribution Function) is used to define a multi dimensional "material function" in which high dimensional textures could be used [35].

A comparison between flat, Gouraud and Phong shading can be seen in Figure 16.

Bishop states that Phong shading is relatively slow [39]. XNA uses an optimized version of Phong shading called Blinn-Phong [40]. Blinn-Phong is faster and research comparing the method to physical experiments using isotropic BRDF measurements claim to be more accurate than Phong lighting [41].

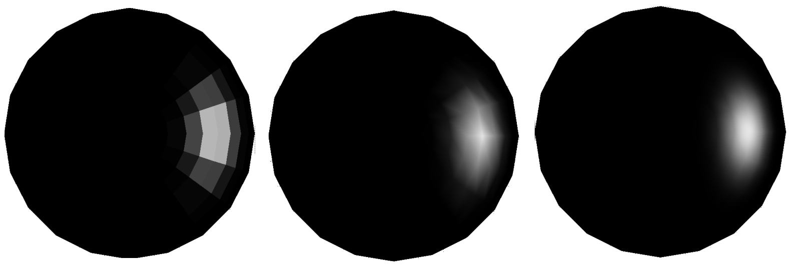


Figure 16: The three different shading techniques: flat, Gouraud and Phong seen from left to right. Note the highlight of the rightmost sphere compared to the Gouraud (middle).

4.4.1 Normal mapping

For the terrain in this project, normal maps were used to add extra detail such as small bumps and sand ripples. By using normal maps, the normals are changed across the surface to give an impression of greater geometric detail.

The existing normals, per triangle vertex, are changed according to the normal map, which is a texture containing the directions of the normals per pixel. These normals are mapped to the color channels (R,G,B). By sampling normals per pixel, additional details could be added without modeling them.

4.4.2 Specular mapping

Specular mapping is a mapping technique used to control the strength of the light reflection of a surface per pixel. The specular component is an approximation of a light reflection (see Figure 16). The following equation calculates the specular component of a lighting equation[3] [16]:

$$i_{spec} = (n \cdot h)^{m_{shi}} * m_{spec} * s_{spec} \tag{3}$$

where m_{shi} is the constant determining the strength of the shininiess. h is the half vector (see Figure 17). m_{spec} is given from the specular map. s_{spec} is the specular color of the light source, which is omitted in this project.

[3]Also called Phong lighting model. Not to be confused with Phong shading, which is about interpolating normals [16].

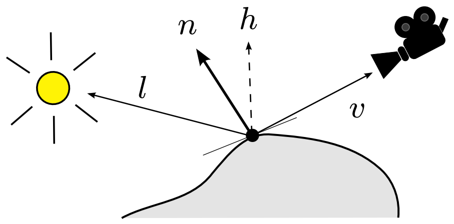


Figure 17: Specular lighting is using the half vector, which is the normalised vector between the normal and view vector v. Specular lighting is thus view dependent.

The slight coloration of the left render comes from the sunlight model (see Section 4.5).

4.4.3 Results

For each object in the game, the default XNA shader is used, but since the ground is larger and less complex, a custom shader was implemented using the above techniques.

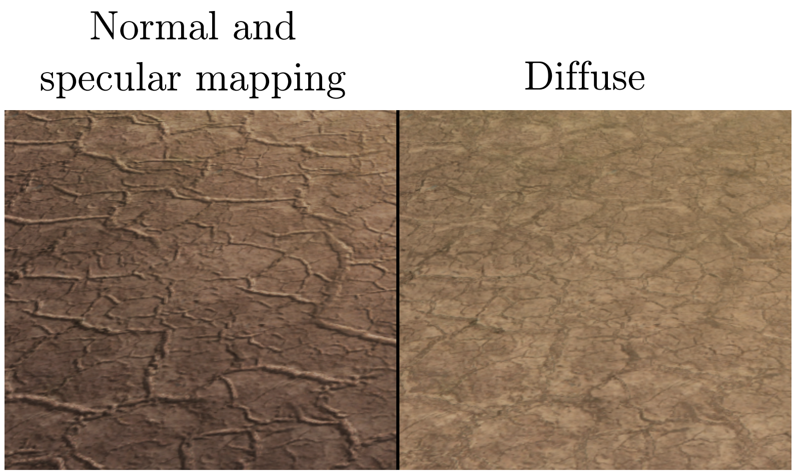


Figure 18: Normal and specular mapping on the terrain compared to only diffuse. The slight coloration of the left render comes from the sunlight model (see Section 4.5).

The specular color i_{spec} was created using $(n \cdot h)^{m_{shi}}$, where $h = \frac{l+v}{||l+v||}$, m_{shi} is the shininess of the surface, i.e. how strong and focused we want the specular reflection. In this case, $m_{shi} = 25$ which gave a good result.

Normal mapping depends on the specular material. If there is no specularity (light), then there are no effects from the normal map. Using the *n*Vidia Normal map filter [42], a normal map is created. The normal map is in so called tangent space. This is simply converted to a vector using $n = 2\, map - 1$ where map is the color. This is done for each channel (R,G,B).

One problem with normal maps is aliasing. Aliasing is the artifact due to the lack of sampling a signal [43]. To resolve this, a filter is used to add more samples. This is done for both diffuse and specular maps using a linear filter to access the samples from the mipmaps.

However, sampling with a linear sampler is not good enough for normal mapping. Since a normal map consists of vectors stored as color data, a linear sampling technique will smooth out the colors in the normal map, resulting in moving the vectors, giving the impression that the terrain is moving when the player is moving. By using a anisotropic filtering, this artifact can be reduced considerably.

4.4.4 Discussion and conclusion

Specular, normal and diffuse materials add finer detail and were relatively easy to implement. One issue with normal mapping was the filtering; using anisotropic filtering is more expensive but the improved visual quality, as a result of the extra samples, might justify the extra cost.

4.5 Sunlight Model

One of the project's aims was to create a dynamic living environment. This was considered important, since a living and breathing environment is not only more realistic but also more interesting to watch. One of the effects where a changing environment is used is the daylight cycle.

The lighting from the sun changes throughout the day. This is due to so called *R*ayleigh scattering [44]. As the photons enter the earth's atmosphere, they are scattered, and when they finally arrive at ground-level they will be colored in a slight blue tint. At sunset (and sunrise), the photons are scattered even more, since the light has to travel a longer distance through the atmosphere, hence giving the light a red/orange color.

There has been papers written on realistic sky models. Nishita et al [45] propose a sky light model based on physical measurements of the sky. Although this method is computationally expensive, a new and faster method of calculating a realistic sky model has lately been proposed in [46].

4.5.1 Results

Due to time constraints, it was chosen not to implement a fully lit sky model, but instead only focus on the ambient lighting contributed by the same scattering process as above. Additionally, we have not found any existing realistic model involving rendering of both sky and cloud coloring; fully developing our method would have required more time to test and implement. As seen in Figure 19, the sky light depends on the sun's angle.

To achieve the smooth color transition, the normal distribution func-

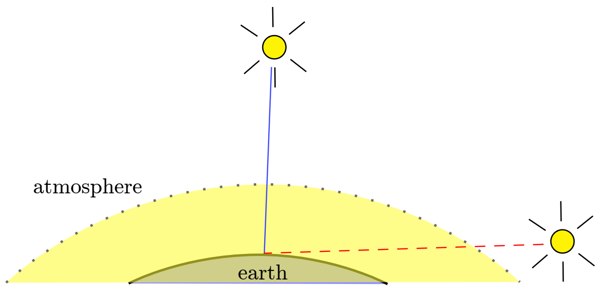


Figure 19: Lighting on earth depends on the sun's angle. At grazing angles, the sunlight is colored red, and at noon the light is tinted bright white.

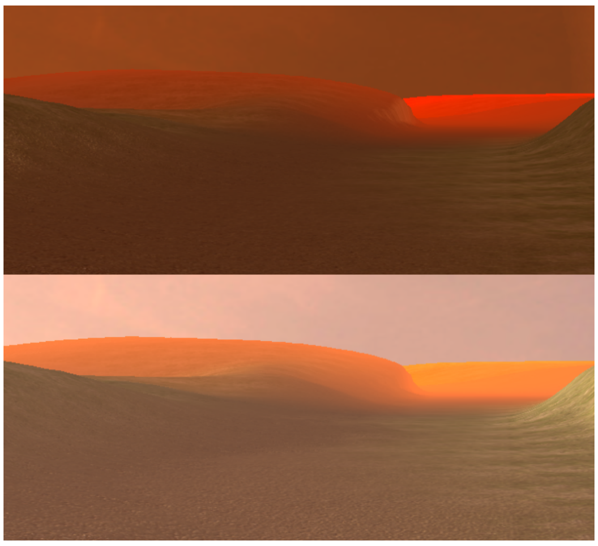


Figure 20: Sunrise and daylight using the sunlight model.

tion $f(x)$ is used

$$f(x) = \underbrace{\frac{1}{\sqrt{2\pi\sigma^2}}}_{a} e^{\frac{-(x-b)^2}{2c^2}} \tag{4}$$

where a is height, b translation and c is the width of the normal curve.

The resulting functions are describing the intensity of the colors red, green and blue, $r(\theta), g(\theta)$ and $b(\theta)$ where $\theta \in [0, 1]$, the sun is at angle $\theta \in [0, 360]$ as seen from earth.

To make the functions periodic and smooth and achieve a sunrise when the sun is on the horizon, an offset was introduced, effectively wrapping the function at $[-10, 350]$, since the function curvature is smooth at these places (see Figure 21).

4.5.2 Discussion and conclusion

The sunlight model is used to affect the ambient lighting on the sky and fog color. The effect created by the sun and sky is an important part of

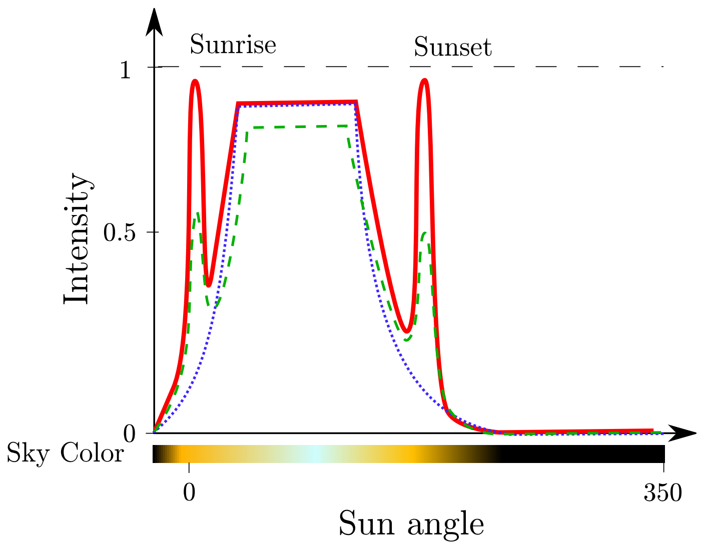


Figure 21: Sunlight model plot.

the game since it covers a large part of the screen. It is also seen as a reflection in metal and water (see Section 4.7, Water).

Since the normal distribution function is calculated at each frame, optimizations could perhaps include rendering the colors to a texture as a preprocessing step. This texture could then be used as a look-up table to increase performance.

Another detail worth mentioning is the lighting from the moon and the stars. While our model ignores the moon as a reflector of light, it is in reality a strong reflector and accounts for a lot of the illumination at night. Jensen et al [47] proposes a night rendering model, in which the moon and the night sky is providing the ambient lighting.

A simplified model of the night time renderer, along with a day time model of the whole sky, would be interesting to continue to work on, since the moon is actually lighting the sky itself and creates shadows like the sun does.

4.6 Fog

A good-looking fog effect can create a heightened sense of realism in the 3D landscape. Apart from creating a soft curtain, which shrouds objects which are far away, it also gives the game landscape an increased sense of depth.

As seen in [31], a basic good-looking fog effect can be created by measuring the depth of a pixel relative to the screen and then applying a linear fog effect to the pixel by linearly interpolating the color of the fog on this pixel, depending on the pixel's z-coordinate.

4.6.1 Results

The fog is calculated using a pixel shader which modifies the color of each pixel in the following way, thus creating the desired fog effect:

```
float depth = input.Depth;
float fog = (depth - FogStart) / (FogEnd - FogStart);
fog *= FogDensity;
fog = clamp(fog, MinFog, MaxFog);
final = lerp(final, float4(FogColor, 1), fog);
```

As can be seen, the fog effect is created by linear interpolation (the function 'lerp') of the fog color depending on the pixel's distance to the camera.

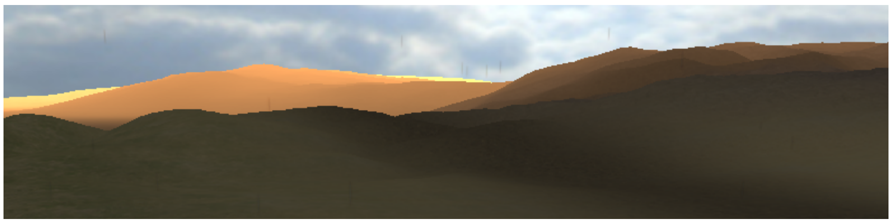


Figure 22: The resulting fog effect, as can be seen in the landscape in the distance.

4.6.2 Discussion and conclusion

The fog produced by this effect looks good and is also fast to render. Several improvements upon this simple technique can, however, be thought of. One way of making the fog seem even more realistic would have been to let the fog grow thicker when closer to the ground and then gradually dissolve as it appears higher above ground.

4.7 Water

Water has always been an interesting challenge in game development, most due to the performance required to render it. We will here present an implementation for water rendering used in this project.

As proposed by Grootjans [48], a good-looking water effect can be created using a combination of several render targets and bump mapping. The creation of the water relies on performing three separate steps of rendering.

4.7.1 Refraction Map

First, everything in the scene that is below the plane that lies on the water surface is rendered to a separate render target. This is called the refraction map. This rendering is done by specifying a clip plane, which lies on the water surface, and then clipping all pixels in the pixel shader which lie on the wrong side of the plane.

4.7.2 Reflection Map

The next step is to create the reflection map. While the refraction map may be thought of as a picture containing everything in the world below the water surface, the reflection map contains everything that is above the water surface. But, since the reflection map has to function as an image of what is reflected in the water surface, it cannot be rendered using the same camera as when rendering the refraction map.

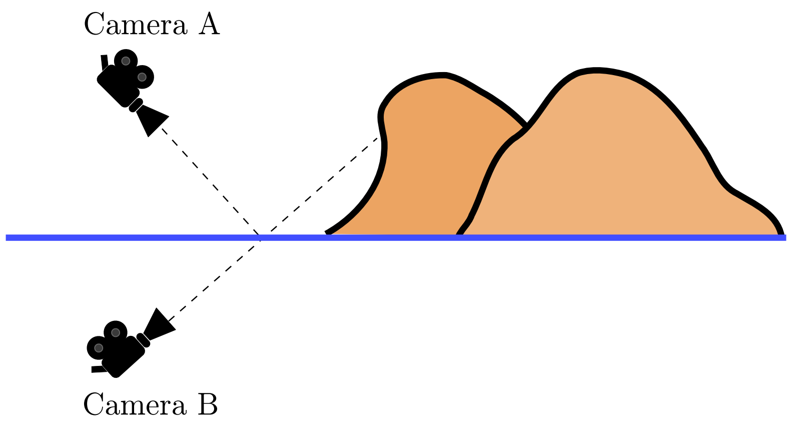


Figure 23: Cameras used during reflection map rendering. To achieve the mirrored image seen from camera A, the scene is rendered from camera B.

As seen in Figure 23, the scene reflected in the water as seen by camera A can be easily generated by rendering the scene from camera B instead, which is positioned right below camera A and looking toward the same point of interest.

4.7.3 Bump Mapping

When the refraction and reflection maps have been rendered, it is time to combine them together with a bump map in order to create the final water effect. A bump map is a texture which contains irregularities which look like ”bumps”. The bump map is read and the normals generated from the colors on the texture are then applied to the flat water surface in order to create a visual wave effect.

4.7.4 The Fresnel Term

When looking at a water surface, the angle between the eye and normal of the water surface determines how much will be reflected in the water. As proposed by Grootjans [48], the ratio of refraction and reflection can be found using the dot product of the inversed eye vector and the normal vector.

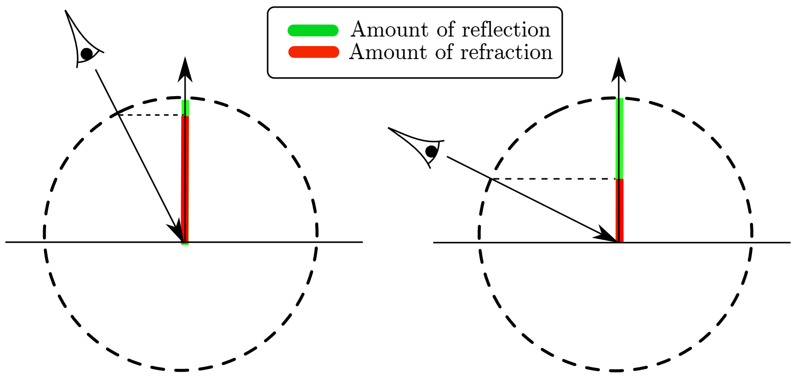


Figure 24: The Fresnel term describes the refraction and reflection relationship by using a dot product.

$$fresnelTerm = -eyeVector \cdot normalVector \tag{5}$$

As seen in Figure 24, the dot product of the inversed eye vector and normal vector results in the length of the red bar. This dot product is called the Fresnel term. By linearly interpolating the refractive and reflective color using the Fresnel term, we get the final color of the pixel.

```
float4 combinedColor = lerp(reflectiveColor,
                            refractiveColor,
                            fresnelTerm);
```

4.7.5 Specularity

As in specular maps (see Section 4.4.2) water also reflect the sunlight. The specularity effect is created by measuring the angle at which the sun rays are hitting the water surface and the angle at which the camera is looking at the water. If the angles are almost the same, then the sunlight will contribute to the shininess of the water.

4.7.6 Results

We decided to implement water rendering as described by the above technique. While this approach to resulted in good-looking rendered water (see Figure 25), the effect itself was expensive to compute. Basically, all objects in the 3D landscape were required to be rendered three times in order to gather enough data for the water effect. First, all objects were rendered to generate the refraction map. Secondly, a complete re-rendering of the objects was required since the creation of the reflection map requires the camera to be positioned below the water surface, thus creating an image showing objects from a different angle. Lastly, all objects were rendered normally along with the water.

Luckily, optimizations such as frustum culling (see Section 7.1, Frustum Culling) made sure that the number of objects required to be rendered were kept down.

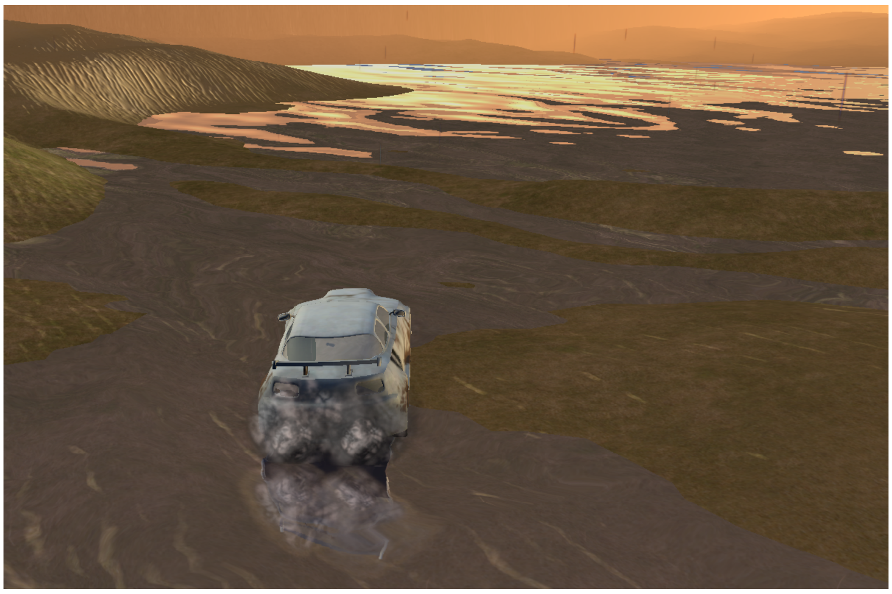


Figure 25: The resulting water effect with reflections of the car, exhaust smoke and the distant landscape shown in the water.

4.7.7 Discussion and conclusion

Although the use of a refraction, reflection and bump map results in a convincing water effect, we found it requires the hardware to be fairly up to date in order to keep a high framerate. A possible optimization to this water technique might be to create the reflection and refraction map at a lower resolution than the rest of the game. This approach might be justified due to the water reflections and refractions being distorted by the application of bump mapping, which gives them a less accurate appearance.

Future work could include implementing actual wave height, instead of faking it through a bump map. A possible approach might be to let the vertex shader dislocate the water surface's vertices, in order to create visual waves. Another interesting effect would be to let the water be affected by physical objects. For example, a desirable behaviour could be to let the water splash when driving a car through it. An interesting approach concerning this is discussed in [49].

4.8 Fire

Carter [50] shows a technique for creating live fire by using a combination of multiple render targets and the pixel shader. The approach renders the resulting texture onto a flat surface.

The effect is created by first rendering the ”warm” points from which the fire originates, called hotspots, to the first render target. Then, we render the hotspots with an offset to a second render target, which intensifies the hotspots and creates an intensifies color in the pixels surrounding each hot spot. Next, we use the pixel shader to blend each color together by averaging each pixel by setting its value to the average of it's four direct neighbours on the left, top, right and bottom sides. This creates the effect that pixels closer to the hotspot will have a more intensive color, while pixels far away will get a more subtle color, since these pixels have gotten their current values by following the chain of ”averaging” from each neighbour to the next. In order to kill off the intensifying effect and thus make the fire ”disappear” at the top of the texture, a constant value is subtracted from each pixel after the intensification.

4.8.1 Results

As a good exercise in learning HLSL, it was decided that this fire effect was to be implemented. Apart from creating the effect as described above, the effect was modified to better fit into a 3D landscape.

This need for modification arose as the technique proposed by [50] merely rendered the flames onto a 2D surface with a black background. Since we wanted to use this fire effect in the 3D world in conjunction with other 3D objects, the black background had to be made transparent in order for objects to be able to be seen through the fire. This was solved by applying another pixel shader when rendering the texture to a *billboard*. This shader decreases the alpha value depending on the amount of black in the texture. By doing this, a transparency is created which allows the player to actually see through the fire. If this had not been done, each fire billboard would have had a visible black background, which would made it look as if the fire was merely engulfing a black flat surface, which is not the desired effect.

Lastly, a smoke particle system was placed over the flames to improve the final look of the fire.

4.8.2 Discussion and conclusion

While the chosen approach did result in an effect which resembles live fire, the result was not as a convincing effect as we had hoped for. Particularly, the solution to decrease the alpha in dark areas of the generated fire texture caused some of the finer details of the fire to disappear. While it might be possible that this fire effect could be improved by creating the transparency of the fire billboard in a different way, it was decided that improving the fire effect beyond the current result was out of scope of this project.

Although this effect was implemented as a means of learning HLSL, future work could include improving the fire effect by making the fire itself a particle system (see Section 4.10), instead of a plain texture. Also, the effect could perhaps be modified to function as a source of light. A physically based implementation is discussed in [51], which also simulates the scattering of light inside the fire medium.

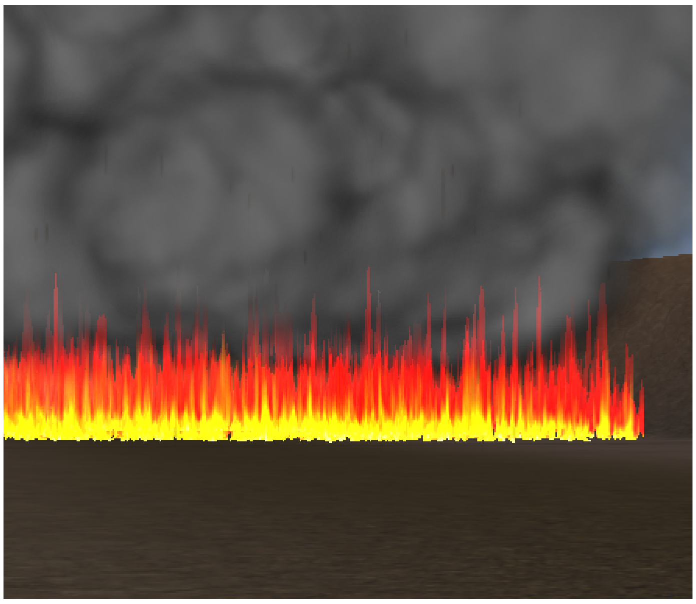


Figure 26: The resulting fire effect with a smoke particle system applied at the top.

4.9 Vegetation

Vegetation can help create a dynamic environment, which resembles the real world. This section discusses vegetation rendering in our game by presenting a solution to the problem of rendering grass in a 3D world.

Fernando [52] proposes a realistic way of rendering vegetation. Ideally, each grass blade should be processed and rendered with its own lighting, shadow casting and movement with the wind. This is acceptable if there are just a few plants to render, but modeling each grass blade in a whole wide meadow is not feasible, as the number of polygons required would be huge. Displaying a scene with thousands of blades of polygonal grass, is a task that will put a strain on today's graphics hardware. That leads to the first problem to solve:

- Many blades of grass must be represented by few polygons. This can by solved by grouping several grass blades in one quad (two triangles), thus faking each grass blade by applying a texture with several grass blades rendered in one single quad.

This technique will represent grass as a plane, giving it a weird appearance when you look at it from the side (being infinitely small). Therefore:

- Grass must appear dense from different lines of sight. To deal with this issue, there are two approaches. First, the simplest way would be billboarding, which means, rotating the grass to always face the camera. That would work for grass positioned far away, but would look weird on closer distances. The second approach would be to cross a set of polygons so that the bush never shows an edge. This technique does, however, require three or four times more polygons and needs to make the triangles visible from both sides (which requires disabling of back face culling).

Also, the grass could be rendered as a static polygon without animation, but to make it look more realistic, it could move slightly with the breeze.

4.9.1 Results

As areas with greater amounts of vegetation were needed in this project's open terrain, it was decided not to represent each plant as a single entity, but rather to group them to increase the frame rate of the game. In order to keep the amount of vertices down, it was decided to render quads with a grass texture applied.

To achieve a good visual result, independent of the current view, we cross the grass quads according to [52] (see Figure 27). Crossing the quads to form star shapes gave the best visual result. We also need to orient the normal vectors of all vertices parallel to the quads' edges. This is done in order to achieve correctly illuminated vegetation.

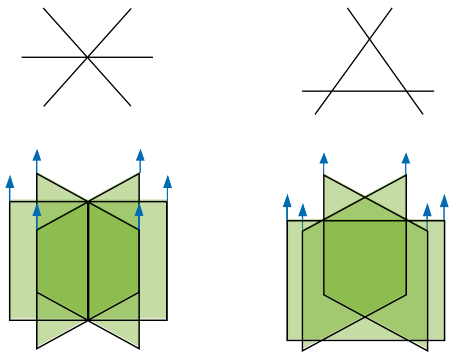


Figure 27: The blue lines represent normals, which are used to calculate lighting.

After forming star shapes, we place the grass objects together in a specific area, as shown in Figure 28. Thereafter, the objects are sorted back-to-front, which allows correct use of alpha blending. By doing this, an impression of a naturally and thickly grown meadow is attained.

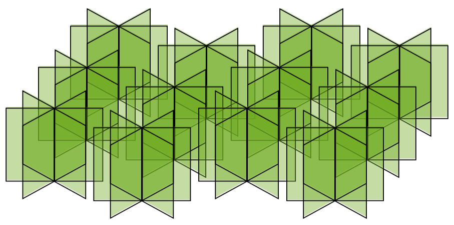


Figure 28: A collection of grass objects.

In order to enhance the performance of the game, vegetation far away from the camera is not rendered as a star of polygons (see Figure 29). Instead, it is rendered as a billboard, which is a lightweight graphical object. Since the grass is seen at a distance, the player will not be able to see the difference between the star-shaped grass object and the billboard version. Because of this, the faster billboard version may be used, and consequently, the performance will increase greatly as we render four times less polygons. Also, when vegetation is rendered as a billboard, it is not animated. This is because its movement will be insignificant and not noticeable from a long distance. Many types of vegetation can be created using this technique. In this project, however, only support for rendering grass was implemented.

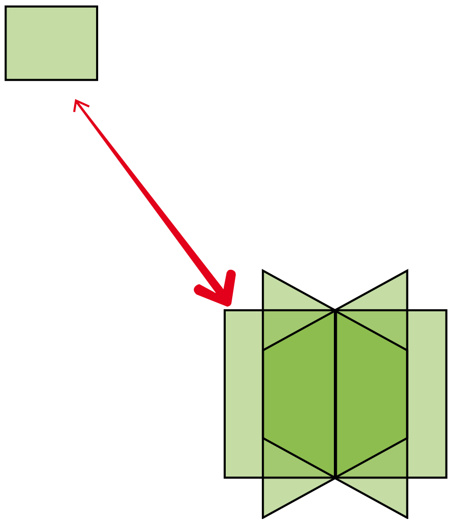


Figure 29: Distant grass is rendered as a billboard while adjacent grass objects use a more complex model.

Vegetation is animated by stretching the upper vertices of each quad. This will deform the vegetation image on the texture while still looking good. Since the movement of the waving grass is very small, the image will not look odd. To achieve a different animation for each grass bush, some randomness is applied to the wind direction and strength. These calculations are made on the vertex shader, relieving the CPU from work, thus enhancing performance.

Finally, vegetation is applied to the scene using splat mapping (see Section 4.3, Splat Maps), where each type of vegetation has its own color. Similar to a splat map, we create a ”vegetation map”, as explained previously. This map will be read by the game, after which (for example) flowers will be rendered on the positions where red is painted on our vegetation map. Vegetation density is also applied from the vegetation map, where darker colors represent denser vegetation zones. For example, a light red color represents zones with just a few flowers, and a dark red color represents zones thick with flowers). The end result can be seen in Figure 31.

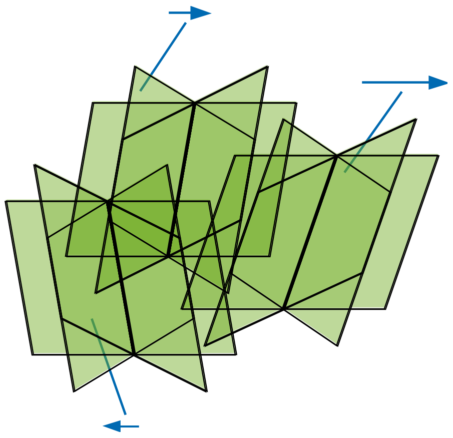


Figure 30: Grass animation, where the blue lines represent wind force.

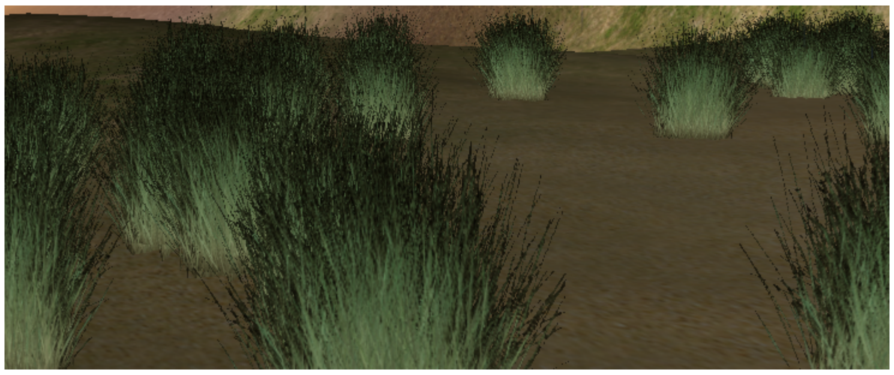


Figure 31: The resulting vegetation.

Combining billboarding with crossing polygons showed an increase in performance. For some types of vegetation (e.g. single flowers) that are not symmetric, crossing polygons is not an option, as it will look weird due to asymmetry. For other objects, such as leaves, neither animation, billboarding nor crossing polygons was implemented.

4.9.2 Discussion and conclusion

By using this technique, we achieved a nice looking vegetation in the form of grass bushes. More realistic vegetation can always be implemented, but due to time constraints, a trade-off between performance and good visual appearance has been done.

In addition to our work, a larger variety of vegetation could be implemented. As our program is built to support easy addition of new types of vegetation, multiple types of vegetation could be easily displayed on the terrain by just assigning it a color on the splat map. An interesting addition would be automatic generation of vegeation on other objects. [53] discusses a possible approach to this.

4.10 Particle Systems

A particle system is a way of handling the creation and execution of certain effects which involve particles. A particle can be a single billboard which has velocity, acceleration, age, color and texture. Given certain starting conditions, set for all particles in the particle system, these particles then behave independently of each other according to the set parameters until the particle's age becomes larger than its set lifetime. When that happens, the particle is marked as dead, and can thus be reused in the particle system.

This section shows an implementation of a particle system and discusses its limits and shortcomings.

As proposed by Carter [50], a particle system can be created by distinguishing between particles and the general rules for all particles in that system. The idea is structured into the following classes:

4.10.1 Particle

A particle is a textured square with color, velocity, acceleration, age and a certain lifetime. Each time a particle is updated, its age is increased and its attributes are modified.

4.10.2 ParticleSystem

The ParticleSystem class describes an actual particle system which is an abstract class, used as a building block for creating a certain desired effect. The particle system class handles the actual rendering calls of the particles. It also handles the particles in an optimized way. It is of utmost importance that a particle system acts with focus on maximizing performance. This is due the fact that most effects that use a particle system consist of thousands of particles. All of these particles have to be updated 60 times per second, which requires each update and draw call of the particles to be very efficient. To create this efficiency, the particle system is created with a maximum capacity of particles. The exact capacity depends on the desired particle effect. All particles are then created simultaneously, by allocating memory for each of them and putting them in an array of fixed size. The particle system is then set up to detect dead particles and then reset those particles' states so that they become alive and can yet again be used in the particle system effect. By allocating memory for all particles once and then reusing them when they become marked as dead, we save a lot of processing time which would have otherwise gone to allocating new memory and freeing memory each frame.

4.10.3 ParticleSystemSettings

This class is used to setup the initial values for the desired particle effect. Table 1 shows the available settings, of which all can be used to control how the particles in the system should behave.

Table 1: Particle system settings

Setting	**Explanation**
Texture	Which texture the particle should use.
RotateAmount	How much to rotate the particle each frame.
RunOnce	Should this particle system run just once or should dead particles be resurrected?
Capacity	The maximum amount of active particles in this system.
EmitPerSecond	How many particles to emit per second.
ExternalForce	An external force applied to each particle each frame. E.g. wind.
EmitPosition	The starting position each particle of the particle system in the system's local coordinates.
EmitRadius	The radius in which the particle will be created from the emit position.
EmitRange	A range in each axis in which the particle's starting position may be set, relative to the emit position.
MinimumVelocity / MaximumVelocity	Each particle will have its starting velocity set to a random value between these.
MinimumAcceleration / MaximumAcceleration	Each particle will have its starting acceleration set to a random value between these.
MinimumLifetime / MaximumLifetime	Each particle will have its lifetime set to a random value between these.
MinimumSize / MaximumSize	Each particle will have its size set to a random value between these.
Colors	An array of possible colors the particle can be displayed in.
DisplayColorsInOrder	If true, the particle will change color during its lifetime according to the Colors array. Otherwise, the particle's color is set to a random one of the available colors in the Colors array and will keep that color for its whole lifetime.

By setting these values differently, the overall behavior of each single particle is controlled effortlessly.

4.10.4 Results

We used the particle system structure described above to create a way of handling particle systems in an effortless way. Although the implementation of idea by [50] worked well to quickly achieve a working particle system, the idea was expanded on to make the particles themselves a bit more intelligent. For example, a modification was made so that each rain particle will keep track of its current height in the world. When the rain drop reaches the ground, it will automatically be marked as dead and then be recreated at the top of the 'rain cloud'. This behaviour could not have been achieved using the original particle system idea, since that idea merely specified rules for the particles in the system as a whole. Our modification gave each particle the freedom to modify the rules to some extent depending on the particle's own private variables. Figure 32 and Figure 33 give depictions of a selection of our implemented particle systems.

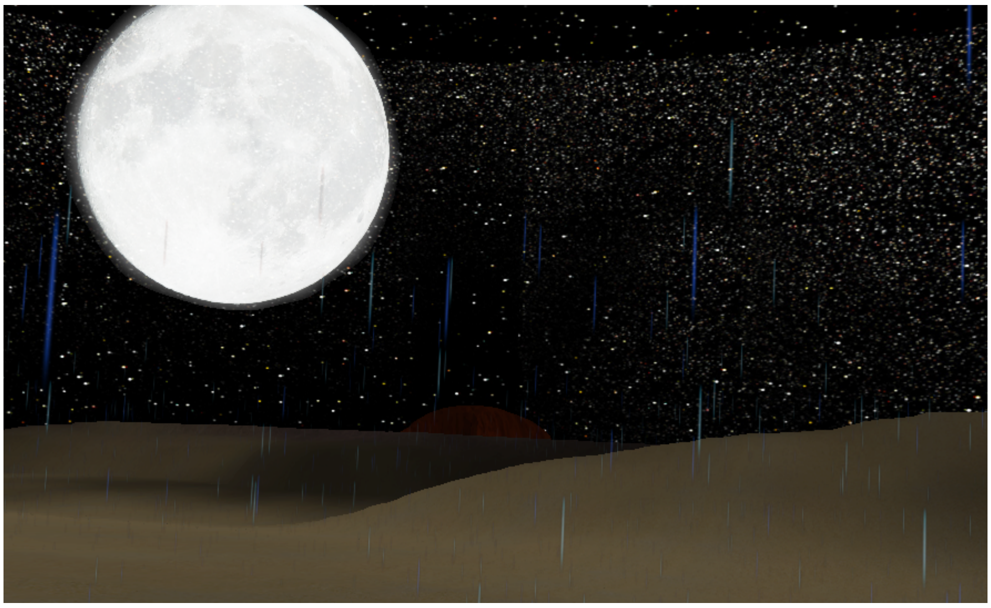


Figure 32: A particle system depicting rain during the night.

4.10.5 Discussion and conclusion

While the implemented particle system fulfilled its purpose in creating effects such as smoke, it could be further developed to improve the visual looks and behaviour of the particles. One idea is creating snow particles, which will collide with the terrain. A possible approach to adding collision detection to particles is discussed in [54]. Another future improvement of the particle system implementation could be a technique for generating realistic-looking dust behind a travelling vehicle, as discussed in [55].

5 Physics

As the computing power of consoles and computers evolve, so has the physics in the games, from the simple physics in the game “Pong” to

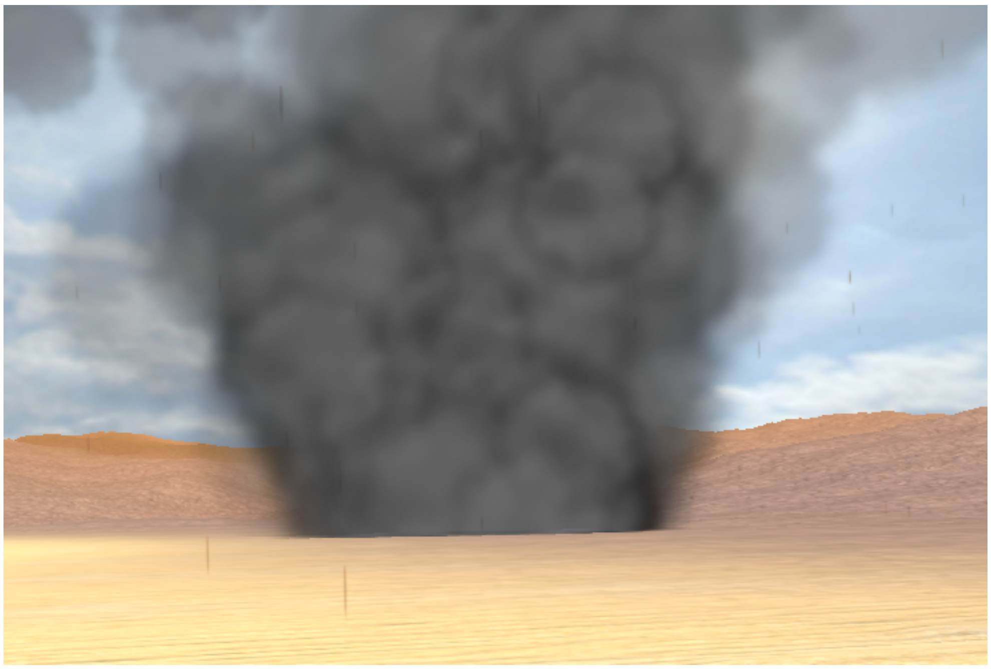


Figure 33: A particle system depicting rising smoke.

the impressive physics of “Star Wars: Force Unleashed” [56]. As a consequence of the increased processing power, gamers such as ourselves expect more realistic physics in today’s games. Since this project covers the creation of a car racing game, naturally, realistic handling of physics is something which will enhance the game experience.

5.1 Physics Engines

A physics engine is basically built on collision detection and collision response, where focus is put on making these calculations realistic and fast.

Physics in games today are realistically simulated using highly optimized professional physics engines. Examples of these are proprietary physics engines such as Havok [57] which is used in Half Life 2 [58], and PhysX [59] (*n*Vidia), used in Mafia 2 [4]. Due high licensing fees, we chose to use an open source physics engine with a free license. For the choice of a physics engine, the only really good alternative for C# was JigLibX, since it was compatible with XNA.

Other engines worth mentioning are Bullet [60][4] (used in Grand Theft Auto IV [61]), Open Dynamics Engine [62] (used in S.T.A.L.K.E.R [63]).

5.1.1 Results

JigLibX was used for detecting collisions and calculating basic collision responses. JigLibX is integrated with our game engine to achieve seam-

[4]written in C++

less physics response without having to deal with complicated physics coding, which would had required a vast amount of extra work.

5.1.2 Discussion and conclusion

Although using JigLibX worked very well, it placed certain constraints on the project code, which might have been avoided if a more extensive documentation on JigLibX was available. For example, when using the car physics of JigLibX (see Section 5.3) the actual forward acceleration vector of the car was mapped to a vector that XNA perceived as the vector pointing to the right. However, despite the lack of a thorough documentation, the fact that JigLibX is open source made modifications to the physics engine possible, should this had been desired.

5.2 Collision Detection

Collision detection is one of the most important parts of a physics engine. It provides a means of interacting with the environment. Some games use collision detection to immerse a player into a "as close to reality world as possible" frame of mind, while other games utilize collision detection as a way of simply guiding the player to the right path (for example, this is done in World Of Warcraft [64]). One instance of the latter would be a racing game, where, if driving in the wrong direction, the player would be gently pushed back towards the road.

As the games becomes more interactive, involving more objects and larger worlds, the need of a fast collision detection system is essential. Even though computing hardware is constantly getting better, the gamers' expectations are always pushing the limits.

In our game, there are three types of objects: moving collidable objects, static collidable objects, and collidable objects. Moving collidable objects are, for instance, vehicles, and they can interact with the terrain, which is a static collidable object (it will not move). We also have the collidable object. These are objects that are resting on the terrain and exist for the player to collide with. To appreciate the work being done by the collision detection system, we digress for a moment and present a simple, naïve method of a collision detection system.

Calculations involved Assume we have n moving cars and m static objects (environment); the number of operations to test each object against every other object is $\mathcal{O}(\binom{n}{2} + mn) \subseteq \mathcal{O}(n!)$. What we can see from this expression is that when the number of moving objects become large, the number of operations increases very quickly. Additionally, the system would then test each of these objects' triangles against every other triangle in the scene. This is essentially a dot product. [5] Assume that each vehicle contain 3000 triangles and there might be 10 cars and 1000

[5]The distance operation is a dot product and some squared distance calculations.

stationary objects in the environment. To perform collision detection for each frame, for each triangle, requires 450 million dot products per frame. According to [65], the number of dot products that can be calculated is 16 million per frame. This performance is insufficient for collision detection.

5.2.1 Results

In this project, we use JigLibX for physics calculation and collision detection. The designers has optimized the collision operation by using "collision skins" instead of detecting collision between each triangle as we did in our example.

A Collision Skin (or "Bounding Volume" as we will refer to it later) is an approximation of an object's shape. Instead of using the triangles, we use a simplified version of each object. The collision tests are reduced to testing collisions between boxes, cylinders and spheres. For example, a rock has a sphere as a collision skin. To test if a collision between two spheres has occurred, JigLibX calculates the distance, d_{old}, between the centers of the spheres for the current frame. For the next frame, if either d_{old} or the new distance d_{new} is shorter than the radii of the spheres, then a collision has occurred and a collision response will follow.

In 2D, for the purposes of this example, the collision detection of above example would be as in Figure 34.

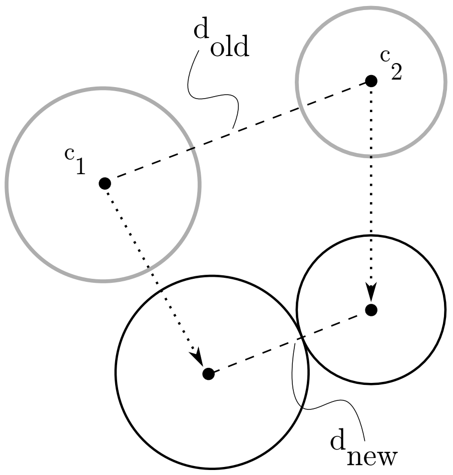


Figure 34: To test collision between two circles, JigLibX compares distances between points between frames (the radii are fixed).

Bounding Volumes In order to get realistic collision detection and response, we use JigLibX's CollisionSkin. In order to create the CollisionSkin for each model, a system was developed. This system allows the modelers to create the bounding volumes directly in the 3D editing software and export to the FBX format. After all, the modelers know which bounding primitives to use for collision detection, instead of creating an automated bounding volume code to determine this for us.

To create a bounding volume in 3DS Max, we name the bounding primitive according to the following scheme:

```
bounding_<type>_<material>_<mass>_<ID>

<type>  ::= box | sphere | cylinder | capsule
<material>::= Wood | Metal| Plastic | Rubber | Concrete |...

<mass>::= <int> (hecto grams)

<ID>    ::= <int> (unique number)
```

Our parser matches each mesh named "bounding" and parses the type, material and mass for each primitive.

An example of an ASCII FBX file (abbreviated):

```
...
Objects:  {
Model: "Model::bounding_box_metal_50", "Mesh" {
...
     Properties60:  {
...
Property: "Lcl Translation", "Lcl Translation", "A+",0,1.41348111629486,-0
         Property: "Lcl Rotation", "Lcl Rotation", "A+",0,0,0
}

  Vertices: -0.319378823041916,-0.0769001841545105,0,...
...
  NodeAttributeName: "Geometry::bounding_box_metal_50_ncl1_1"
}//Model

}//Objects
```

For each mesh, the FBX file only contains information about the position of the vertices in model space and how the model is rotated and translated in the 3D Modeling software. To extract the information, we need to create the bounding volumes from the FBX file. First, the vertices are read, and the dimensions are calculated by taking the bounding volume of the vertices. This creates the CollisionSkin, used by JigLibX. We then transform the bounding volume by using the Lcl (local) properties and applying them accordingly.

The bounding volumes are used by the collision detection system, but they also contain information, such as material and mass. These variables are used by JigLibX to simulate material properties. A material contains properties for its static and dynamic friction, as well as elasticity. The

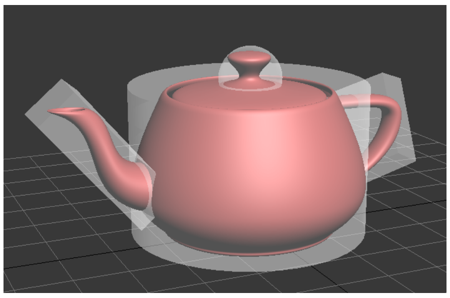


Figure 35: A teapot with bounding volumes in 3DS Max. Pay attention to the translation and rotation of the bounding volumes. These bounding volumes are written in the FBX file as “Lcl Translation” and “Lcl Rotation”.

material properties are also used to get the appropriate sound effect when collisions occur, see Section 8.

One of the most difficult parts of the project proved to be the collision skin parser implementation. One of the reasons parsing the FBX file is difficult is the lack of information from how XNA interpretates the objects. When an object is exported from 3DS Max in FBX format, the scaling of the object is written in both a transformation (Lcl Property) but also as a scaling property at another place in the FBX file. When an FBX file is imported into XNA, it assumes the units in the vertices are in centimeters, not in inches as is default in 3DS Max. Another minor difference is the way XNA’s coordinate system works. XNA uses a left handed cartesian system, while in 3DS Max, it is oriented in a right handed system. Another minor annoyance is that XNA uses y as the up vector and 3DS Max uses z, see Figure 36.

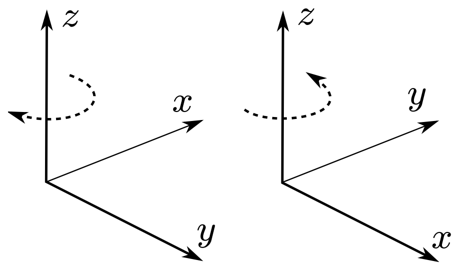


Figure 36: Left handed and right handed coordinate systems.

5.2.2 Discussion and conclusion

The collision system used in the project is based on integrating modeling with the physics of JigLibX. Connecting the two has proven more difficult than we first thought. However, apart from the difficulties, the big advantage is that we can easily add new objects and apply the material and physical properties to the objects in a simple way.

An interesting part of the collision detection that has not been implemented is the ability to destroy the bounding volume or making parts of it fall off, as on a real vehicle. These effects are present in games such as Grand Theft Auto IV [61] and Mafia II [4]. Unfortunately, not enough time was available to investigate the possibility of implementing features of this kind.

5.3 Car Physics

Naturally, an important part of any car racing game is car physics. The car needs to be able to behave in a convincing way and be responsive to the the player's steering commands.

Our chosen physics engine, JigLibX, included a class with support for simple car physics such as forward/backwards driving and handbraking. This included class had, however, no support for more realistic car driving behaviour such as drifting [66]. We found a way of extending the JigLibX car class to improve the car's driving behaviour by using the ideas presented by [67].

5.3.1 Results

By taking advantage of the built-in car physics class in JigLibX and applying the modifications proposed in [67], we were able to easily combine the car physics with a car model and thereby create a responsive vehicle with a natural-feeling steering.

Although later, an inherit error in the car physics was discovered. While driving at high speed, the car would sooner or later start to wobble. This bug is previously known and a solution has been proposed in [67]. The proposed solution did, however, not eliminate the problem from our game. Since the aim of this project was not to dig into advanced car physics, we decided, due to time constraints, to not spend time trying to correct the car's behaviour. Rather, we circumvented the bug by lowering the maximum speed of the vehicle.

5.3.2 Discussion and conclusion

Overall, using the built-in car physics class in JigLibX to achieve a convincing car driving behaviour worked very well. Although modifying the car class itself requires a deeper knowledge of physics than any of the group members possessed, the basic structure of the class was understandable. While this rendered us able to modify various parameters of the car such as roll resistance, maximum speed and friction, some parameters were very hard to understand and seemed to have complex relationships. Basically, altering one parameter affected another parameter, thus making it a complicated task to improve a specific aspect of the car physics. Perhaps, with a deeper understanding of the underlying physics, this would have been an easier task to assess.

5.4 Deformation of vehicles

A point of interest in this project was to investigate the possibility of implementing realistic deformation of vehicles during collisions. We will here present our corresponding findings.

Several methods of possible vehicle deformation exist. Lee B. et al [68] discusses using a space-partitioning data structure called a KD-tree [69], while Yinghui C. et al [70] proposes a method using linear modal analysis for performing the deformation. We decided to investigate an approach using a KD-tree, since implementing a space-partitioning data structure can be useful in other aspects of a game, such as optimizations [16].

JigLibX (our chosen physics engine), allows the game engine to sample the processed physics data, which means the penetration points in a collision can be found.

A KD-tree is a variant of a binary search tree that divides each level of nodes into a separate spatial dimension [69] (a 2D representation can be seen in Figure 37). In our three dimensional case, $k = 3$, the root node is sorted in accordance with its X-axis coordinate. The children of the root are sorted according to their Y-axis coordinates. The next level of children are sorted by their Z-axis coordinate. Their children are then again sorted by their X-axis coordinate.

By using a KD tree instead of a brute force search for the nearest neighbour to a point, we reduce the search time complexity from $\mathcal{O}(n^2)$ down to $\mathcal{O}(\log N)$ [69].

This is a great improvement and a necessary one, since we think performance is one of a game's main focus points.

5.4.1 Results

We explored an idea of how a KD–Tree could be used in order to achieve the goal of vehicle deformation. The idea entails using the found penetration points from a collision. These points can be used to find the closest vertices in each 3D model. In order to find the closest vertices, all vertices in the model's meshes have to be put into a 3D KD–Tree.

Since code correctness was of great importance when implementing a performance-critical algorithm such as a KD-tree, it was decided to translate the C++ implementation by Milev [71] into C# code. Extensive testing showed that the resulting algorithm generated the same search results as a brute force search.

Our implementation searches for an arbitrary number of closest vertices and subsequently applies a physical force to them, which causes them to dislocate themselves by an amount relative to the collision force. Since the search in the KD-tree returns a selected number of vertices in the order "closest first", it was decided to apply a greater force to the closest vertices, creating a result which resembles a dent.

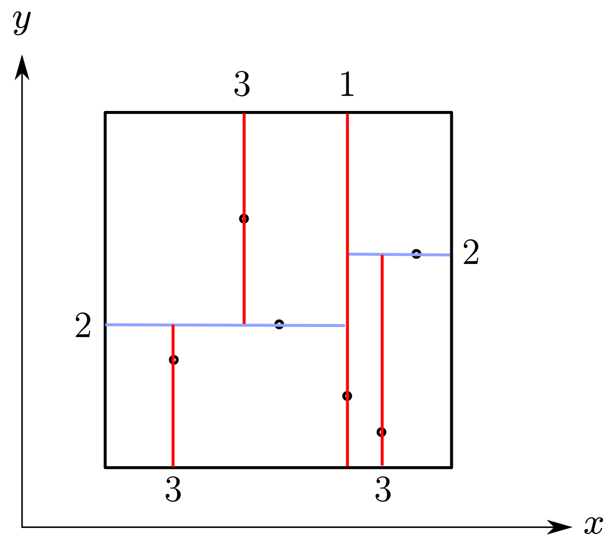


Figure 37: A 2-dimensional KD–Tree showing three sets of dividing planes. The first y-plane (red) divides the data set (the points) into two smallers sets. Thereafter, the second x-planes (blue) are dividing each of the two smaller planes into two additional planes, resulting in four planes. Lastly, the third y-planes (red) divide the remaining three points, resulting in six new, smaller planes. Note how each plane is dividing a different axis than the previous.

This dislocation of vertices creates an effect of model deformation, which allows vehicles to be bent and dented (see Figure 38).

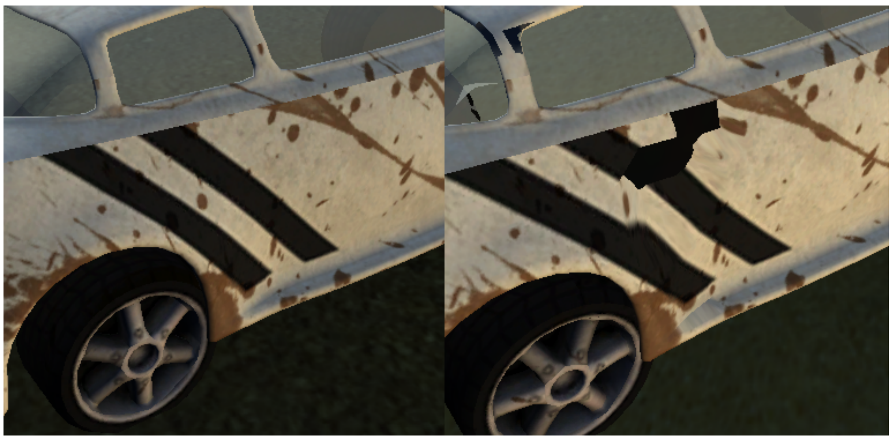


Figure 38: After a collision, vertex dislocation is applied to the unharmed vehicle (left), causing a dent, as seen in the striped area on the collided vehicle (right).

5.4.2 Discussion and conclusion

While the chosen method of vehicle deformation dislocates vertices in a desired way, in our opinion, the created dents are sometimes hard to see, since the normals of the dislocated vertices are unchanged by the deformation. In order to create deformations that still are applicable to effects such as shadowing and lighting, this method should perhaps have been expanded to recalculate the normals of the model after a collision. However, due to time constraints, this issue was decided not to be expanded upon.

Another possible approach might have been to perform the deformation calculations entirely on the GPU side. While our KD-tree approach works entirely on the CPU side, the linear modal analysis method by Yinghui C. et al in [70] utilizes the GPU to gain improved performance.

5.5 Camera System

The world in a game is built on points in 3D space, and to be able to see the world on a 2D screen, we need to project these points from the virtual world onto the monitor's flat surface. The projection of points in 3D to 2D is done by a projection matrix.

The camera is defined using 3D vectors describing its position, target, and up vector. The camera also has a FoV (Field of View). We can get interesting effects if we change this value. Changing FoV to 180° will allow seeing from the sides of the camera, the distorsion will be distracting because we are not used to this. Changing this value down toward 0 will eliminate the perspective distorsion and introduce a schematic look of the world. This can be compared to using a telephoto lens, where the angles and distances are preserved (basically, the sense of depth is ignored). The latter is used in the game to achieve a telephoto lens effect (see Section 5.5.3).

To move the camera smoothly, a spring system is used. For all cameras in the game, different approaches to positioning and handling are used, but they all involve damped springs. In the following text, the game's spring system is presented.

5.5.1 Camera spring system

Hooke's law states that $F = -ky$, where k is the spring coefficient, telling how stiff the spring is. The equation means "force equals the negative displacement of a spring" [72]. This means that the spring force is always pointed toward its rest position. Since we do not want to know just the displacement in one dimension, we use 3D vectors. The law can then be combined with Newton's second law $F = ma$. We set $m = 1$ and effectively eliminate it from subsequent equations. We get

$$F = \underbrace{\frac{\mathrm{d}^2 y}{\mathrm{d}t^2}}_{a} \tag{6}$$

and with Hooke's law

$$F = -ky \tag{7}$$

we can now express the equation in terms of the function $y(t)$

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -ky \tag{8}$$

For the camera movements, oscillating springs are unwanted, and therefore, a dampening force $F_b \propto F$ is introduced. We get

$$F_b = -\mathrm{d}v$$

where d is the dampening coefficient and v is the velocity of the displacement. We add this to Equation 8 and get

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -ky - d\frac{\mathrm{d}y}{\mathrm{d}t}. \tag{9}$$

By carefully choosing coefficients d and k, we can get a "critically damped spring", which is the type of spring used for the cameras. What makes a critically damped spring work well for camera movements is that the spring does not oscillate and it reaches its rest position in the shortest amount of time possible, which is what we want. Springs such as these are used when smooth but fast recovery of sudden movements are necessary, for instance, in a vehicle [73]. For example, a shock absorber of a car has a strong spring with a damper made from a system of pressurized gas or oil, which is forced through a small hole. This spring system suspends the wheels from the chassi for each wheel, making the trip less bumpy.

We have in equation x transformed our spring system to an ODE (Ordinary Differential Equation) which we want to solve for each time step (60 frames per second).

5.5.2 Euler's method of integration

The most simple, and fastest, method of numerically solving ODEs is called Euler's method. The method is commonly used because of its speed and simplicity in games [74]. Therefore, we chose to implement Euler's method.

The method interpolates the solution of the ODE by calculating the solution in small time increments and linearly interpolates the new value using the previous value. Using Euler's method on the above equation, we get with $t_{n+1} = t + h$

$$\underbrace{y_{n+1}}_{velocity} = \underbrace{y_n}_{old\ velocity} + h\, \underbrace{f(t_n, y_n)}_{acceleration} \tag{10}$$

5.5.3 Results

Since the method is integrating for each frame a fixed time step h, a sudden drop in frame rate would offset this time step and introduce errors in the solution (see Figure 40). The inaccuracies are visible as the

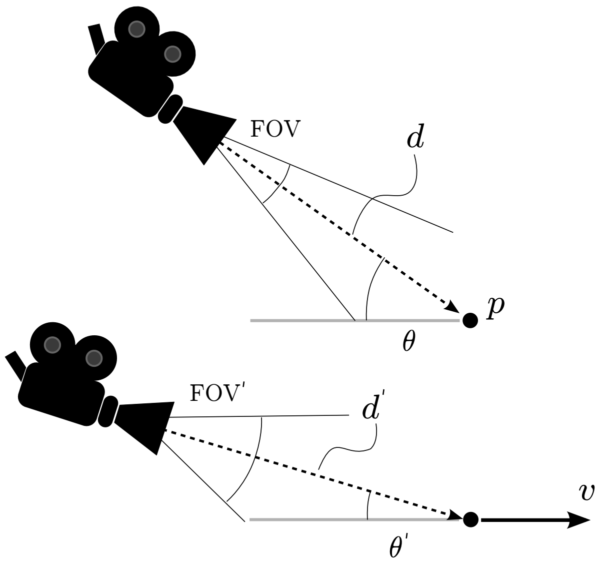


Figure 39: The camera spring system with a moving point P. Note that the angle $\theta' < \theta$ and $d' > d$ when $v \neq 0$. This works when the point is travelling backward compressing the spring increasing the angle θ.

camera starts to drift away from its intended position. This is clear when the camera is used on a moving object (see *Tail camera* below).

One way to reduce the inaccuracies caused by low frame rate (large time steps) is to calculate the elapsed time since last frame update in the game and perform the time steps between the frames. In this way, the integration calculates at small predetermined fixed time steps, which makes the method very accurate.

Tail camera One of the cameras that use the spring system (see Figure 39) is the tail camera. This camera is the default camera used in the game, placed behind the car, looking forward. This position, called the "ideal position", is positioned so that the camera will never tip over the vehicle, and always maintain an upright position.

The spring system used by the tail camera interpolates the current camera position to the ideal position; the spring physics makes the interpolation smooth. One issue with the tail camera is that when the car is spinning or tipping over, the ideal position will also spin. However, the camera will not rotate, since its up-vector is the world's up-vector, but rather move in a circle. To combat this behavior, the tail camera uses the velocity direction for high speeds, instead of the car's forward vector to position the "ideal position".

Movie camera Another camera, which uses a slightly different spring system, is the movie camera. The movie camera is used as a replay view, resembling the way in which a televised car race would have telescope cameras positioned throughout the track.

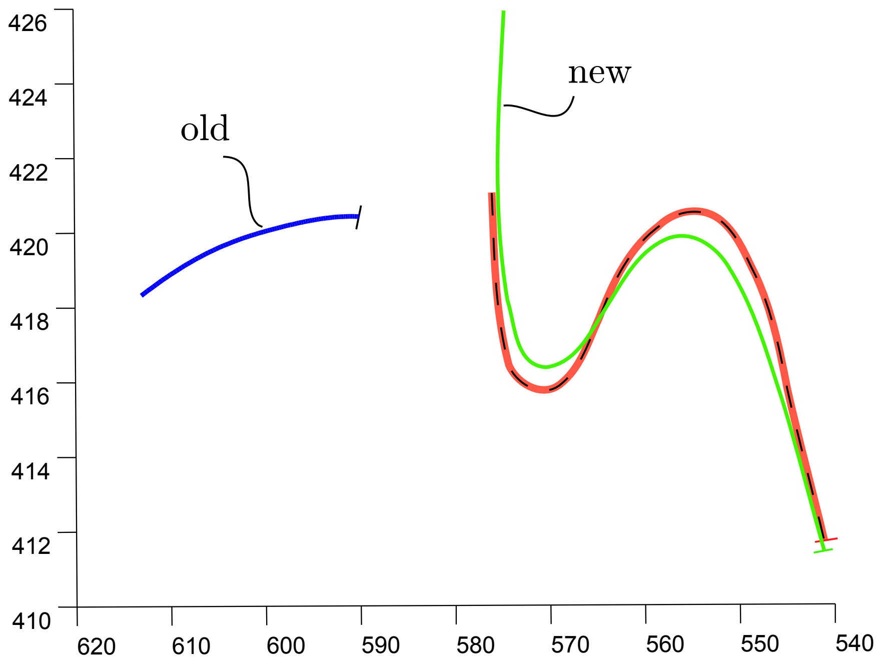


Figure 40: Using the old solution, with a large time step of $dt = 0.001$ (blue curve), with Euler's method, results in a poor approximation of the ODE, while our new method (green curve) renders a better result. This can be compared with the exact result (thick red curve).

First, the camera has predefined positions on the track. When the car is close to one of those positions, the camera will move to the new position, simulating a camera switch.

The camera is static and its "look at" vector (the direction the camera is looking at) is pointing at the car. However, it still needs to provide smooth movements. Therefore, the car's position is used as an ideal position. To emulate the way a movie camera zooms in on the car, a one dimensional spring is used to achieve this effect. Instead of a 3D vector, a scalar is used. This scalar value is a position on the "lookat" vector, which gives the desired zooming action.

To increase the feeling of a movie camera, the FoV is changed depending on the zooming. This value starts at 42 and goes down to 20 when at maximum zoom.

5.5.4 Discussion and conclusion

As mentioned, the Euler method of integration is inaccurate and should be avoided at all cost [75]. Our method of fixing the time step problem effectively eliminates the error introduced by too large time steps, Tests of the game using the spring system shows that the method is not as bad as we first thought, if using our modified method (see Figure 41).

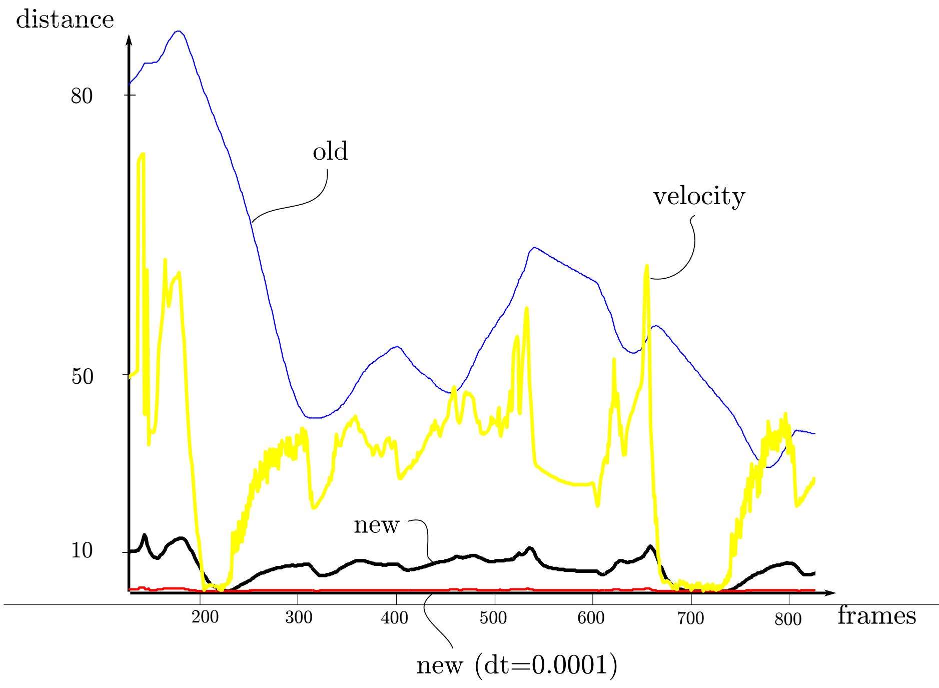


Figure 41: Comparison of the different camera solutions. The y axis is the distance from the ideal position. As can be seen, the plot for the old solution, where $dt = 0.001$, differs a lot from the ideal position, whereas the new technique stays very close. If $dt = 0.0001$, the new solution will tangent the ideal position almost perfectly. However, this requires more processing power. Note that as the velocity (yellow) of the ideal position increases, the precision of the old camera solution decreases.

The conclusion is that we found no reason to implement any higher order approximation, for instance RK4 [76] or any higher order numerical integration method. For our purposes, the Euler method of integration works well.

One feature that was tested, but ultimately not added to the game, was collision detection for the camera. This is an issue when the car is driving up a steep hill, since the tail camera will penetrate the terrain. One way of solving this was researched, but further analysis is required. The idea was to use JigLibX's collision system and wrap the ideal position with a collision volume.

One issue with this method is the time step compensation system. The system will linearly interpolate between ideal positions, which will result in a faulty penetration of the terrain.

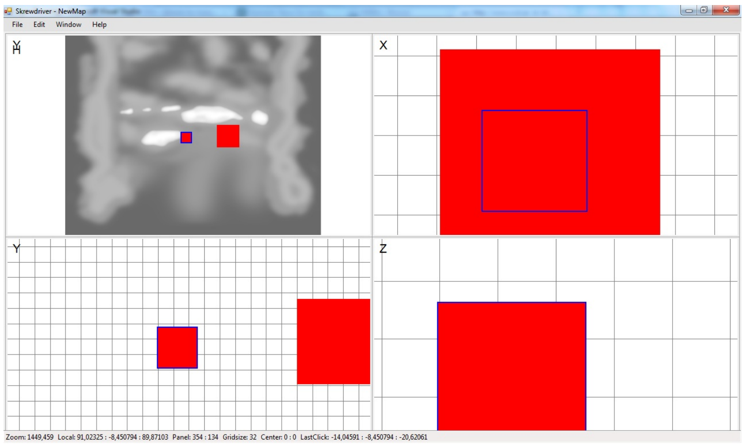
6 Level Editor

Since the game takes place in an open world environment, where objects such as rocks, traffic signs, and houses are placed on a predefined map, we discovered a need for a way to quickly and easily place these objects into the world.

6.1 Results

To fulfill this need, a level editor was developed (see Figure 42), which main purpose was to give the operator an easy way to click and drag objects onto the map.

The level editor saves the data in an easy-to-read XML file which basically informs the game what object should go where; this enables the possibility of creating completely different maps without the need of recompiling the game. Having no hard coded map data within the game is considered a feat because we can then expand the game with user generated data.



```
Zoom: 1449,459  Local: 91,02325 : -8,450794 : 89,87103  Panel: 354 : 134  Gridsize: 32  Center: 0 : 0  LastClick: -14,04591 : -8,450794 : -20,62061
```

Figure 42: The four viewports of the level editor.

The level editor has four screens viewing the map from the axes x, y, z and $y + heightmap$. When we move an object on one screen, its position is dynamically updated on the others at the same time. When the operator drags an object on the screen, its position in height (y) is automatically calculated from the heightmap. In the editor, we can also adjust scale and rotation of the objects.

At this point, we also discovered another purpose of the level editor: events. Events can be described as objects, but without any graphical output. An example of events can be starting points and checkpoints.

Thinking further, we came to realize that we could use events to describe objects that should be dynamic on the map. With this, the game can now support objects such as birds or trains that should move on a predefined path. Events in the level editor can also be used to describe paths for the AI so the developer does not need to make them static in the game.

6.2 Discussion and conclusion

Further development of the level editor should include a world camera where the operator can move around in the world and see the creation without the need of starting the game. This is needed mostly, because as it is now, it may be hard to discover cracks between the terrain and objects placed in a slope. These objects need to be rotated so that they touch the ground's tangent perfectly.

7 Optimizations

While games are striving towards looking as beautiful as possible, they still have to be enjoyable to play. A large part of this enjoyable experience comes from the game feeling smooth and responsive to the player's actions. Therefore, in order to cope with utilizing beautiful but demanding graphical effects while still making the game feel responsive, several optimization techniques have to be implemented. This section presents a selection of possible optimization techniques for this project, of which a few have been implemented.

7.1 Frustum Culling

Frustum culling is a relatively simple but efficient technique for speeding up game engine performance. This is done by checking every renderable object for intersection with the active camera's view frustum. By doing this, we gain information on which objects can actually be seen by the camera. With this information, we can simply refuse to send the unseen objects to the graphics card at all, thus creating a major performance boost, reducing the amount of data sent to the GPU.

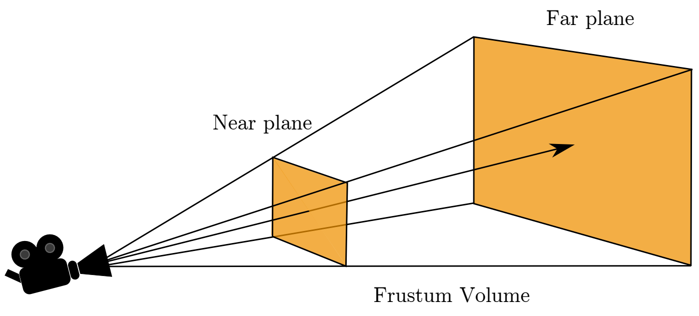


Figure 43: A view plane frustum.

As seen in Figure 43, the camera's field of view can be though of as a frustum emerging from the camera. With frustum culling, all objects are first made sure to be inside this view frustum otherwise they are filtered and not sent to the GPU.

Frustum culling was implemented in this project by using the aforementioned technique.

7.2 Occlusion Culling

Basically, this technique filters objects from being sent to the GPU by checking if they are occluded by another object in the 3D landscape, whose Z-coordinate is closer to the camera in the view matrix. I.e. with the camera looking straight at a mountain and a car being positioned right behind the mountain, the car is occluded by the mountain and should therefore not be sent to the GPU for rendering. While this technique may have had increased game performance, it was decided, due to time constraints, that occlusion culling was not to be implemented as part of this project.

7.3 DLoD

DLoD or Discrete Level of Detail is a method used to decrease the complexity of a model as it moves away from the camera, since it covers less pixels on the screen.

In [77], Clark concludes that varying the geometric detail based on screen area is an effective optimization.

7.3.1 Results

We decided to create each model in three versions (see Figure 44). For each version of a model, the number of triangles is reduced by 60-70% depending on the overall shape. In a way, LoD is to models as mipmapping is to textures.

To accomplish this, we use 3DS Max "ProOptimize", a modifier which reduces the number of triangles of a model automatically, without destroying the texture mapping.

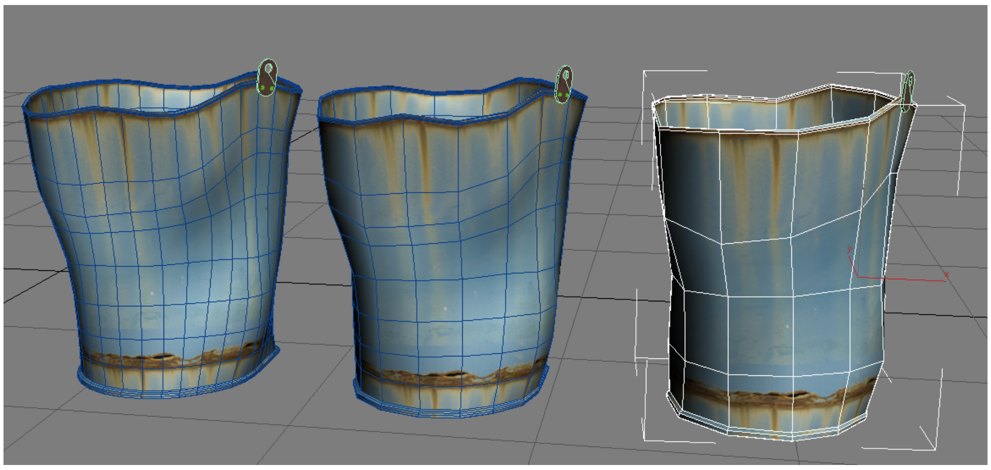


Figure 44: A screenshot of 3DS Max with a model of a bucket showing three LoDs. Pay close attention to the preservation of texture mapping while using the ProOptimize modifier.

The models are named automatically in 3DS Max using the exporter script (see Section 3.2) by appending "_LOD_n" where n is 1,2 or 3 to each mesh. For each frame in the game, the mesh is chosen depending on the distance to the camera according to a simple mapping scheme, see Figure 45.

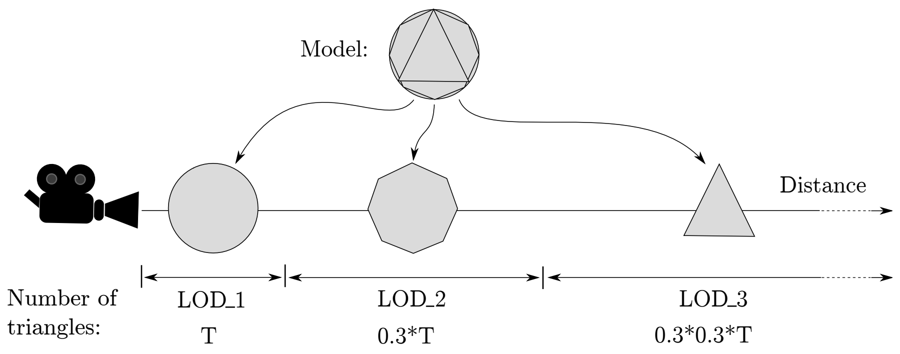


Figure 45: The LoD scheme used in the project. Observe the increasing distance between LoD switching.

The original model contains not more than $0.3 + 0.3 \cdot 0.3 = 0.39$ times more triangles than the original model. In Figure 45, the distance between the LoDs is increasing.

The way the mesh is selected is by calculating the distance between the camera and each model that is within the view of the camera (see frustum culling at Section 7.1), this is done using squared distance, which is a well known optimization.

7.3.2 Discussion and conclusion

One of the problems with DLoD is that each model has to be created in three different versions. There are, however, other techniques where LoDs are created by collapsing the triangles of a model in realtime [78].

To further optimize DLoD and reduce the number of distance calculations a data structure could be used. The method is called HLoD (Hiearchical Level of Detail). Instead of calculating the distance from the camera to each mesh, the distance to a group would be performed and several models would become part of the same LoD [16].

Another issue with DLoD is the switch between LoD levels. This visual artifact could be reduced by fading between the objects to make the transition as smooth as possible [79]. Using this method means, however, that at one point there will exist two versions of an object at the same time in the scene. Thus, requiring the same performance as rendering two separate objects, which is getting us back to the original reason for using LoD.

8 Audio

Usage of audio can help creating a heightened sense of realism. This section discusses audio in game programming, and also presents the audio implementation used in this project.

There are two approaches toward implementing audio handling in XNA[80]. The first requires manually importing and creating sound objects through the Content Pipeline. This implies that the programmer builds a sound bank manually, in the code, by adding files and folders. The *Content Pipeline* supports .wav, .wma and .mp3.[81]

The difference between these three audio formats is that .wav (wave) usually contains uncompressed audio, while .mp3 and .wma use compression. This means that .wav files require more memory than .mp3 and .wma, but result in better sound quality.

Since XNA is wav-based [80], only imported sounds of .wav format can be modified through the following parameters:

- Volume
- Pitch - This means changing the frequency of the tone to become higher or lower[82]
- Panning - This is the 2D position (on the screen), from where the sound is emitted, e.g. the upper-right corner, which affects the volume the stereo speakers are operating on. For example, if the sound is emitted from the right, the right speaker will get a higher volume than the left one. [83]

These parameters can be changed during run-time.[80] Mp3 and Wma files can be used for playing music. However, during run-time, only the volume can be modifed. [84]

The second way of implementing sound is to use XACT (Cross-platform Audio Creation Tool), which is a graphical tool for authoring audio content. Audio imported into the XACT-tool is expected to be fully developed and processed [85]. XACT is described as a sound studio that has functionality for creating and combining complex sound structures [86].

The sound bank created in XACT is loaded into the game, after which the sounds can be accessed through cues. A cue is composed of one or more sounds. The cue can be played, paused, resumed or stopped. While volume and pitch can be changed during run-time, they can also be randomly set to a value within an interval, specified in XACT, each time the cue is executed. These properties can be adjusted in XACT and different sounds can be bound to cues without the programmer needing to change code or to rename sound files. [87]

The drawback with XACT is that it only supports .wav files. Wave files are more suitable for short sounds because of the file format being uncompressed and occupying more memory than compressed formats.

Mp3 files are more suitable for music that will not be modified during run-time, but they cannot be managed by XACT. [88]

We chose to manage the entire sound system with XACT to easily test different sounds without changing or renaming sound files. An advantage of using XACT is that handling sounds in the project code is easy. The sounds and their names are only modified in XACT.

The program used for editing sounds before adding them to XACT was WavePad (see Figure 46). In WavePad, wav-files are processed by the following effects [89]:

- Amplification - This is changing the volume
- Normalization - This means making sound files use the same amplification
- Pitch
- Speed
- Fade in/fade out - Meaning to change the volume gradually only at the beginning and at the end of a file
- Echo
- Reverberation - The persistence of a sound in particular space.
- Noise Reduction - The process of eliminating distracting background noises.

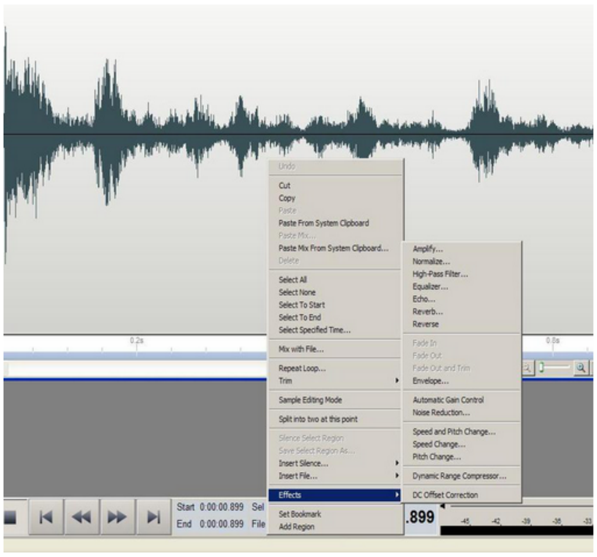
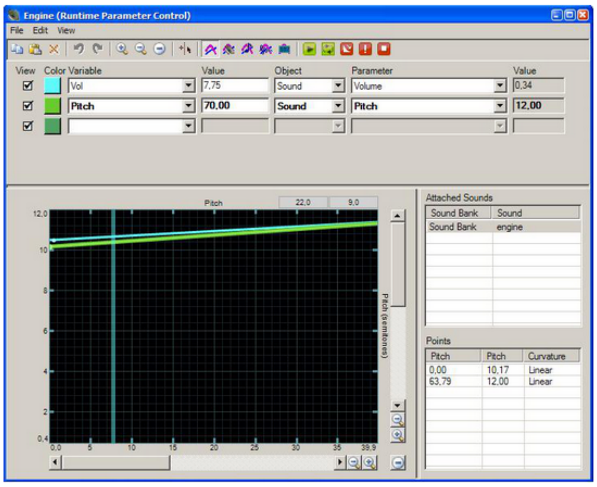


Figure 46: WavePad showing a part of a wav-file that will be processed by an effect.

The processed file is then imported into XACT, where the run-time variables that should be visible for the programmer are specified. The variables hidden from the programmer could be either set to a fixed value, or be set to an interval. [80]



View	Color	Variable	Value	Object	Parameter	Value
✓	🟦	Vol	7,75	Sound	Volume	0,34
✓	🟩	Pitch	70,00	Sound	Pitch	12,00
✓	🟩					

Attached Sounds

Sound Bank	Sound
Sound Bank	engine

Points

Pitch	Pitch	Curvature
0,00	10,17	Linear
63,79	12,00	Linear

Figure 47: The XACT-tool, showing variables, their interval and how they change into different values.

Our current raw sounds are from the [90] sound bank that goes under the Creative Commons License, which says that sound under this license can be shared, remixed and reused legally.

8.1 Results

Every sound in the game features effects applied through WavePad. Whether the player should hear a sound or not, depends on the emitter's distance to the player. The volume of each sound is calculated by using the distance between the emitter and the player. The minimum distance (the "field of sound") between the player and an object, in order to hear the sound, is set individually, for each sound. The volume and pitch of every sound have default values, and also an interval within which they can change.

There are three types of sounds in the game, that differ in design. These will be presented in the following three sections.

8.1.1 Sounds bound to movable objects

A sound bound to a movable object is, for example, the engine sound. An engine sound is repetitive. Therefore it is necessary to find a part in the sound file that will sound good while being looped. The issue with looping parts is of the same kind as with repetitive patterns in seamless textures (see Section 4.3.2), where the modeler needs to make the textures' parts repeat, and fade into each other smoothly. The volume and pitch variables are set to change during run-time and will use the speed of the car while being modified. When the speed goes up, the pitch and volume will also rise, while remaining between the minimum and maximum range. The panning will also be set by the 3D position of the car. In every iteration of the game loop, the engine sound will be updated with new values of the speed, distance to the player car and 3D position of the car. In order to determine the correct sound to play when the car is driving on different types of ground (grass, sand and gravel), the car's position is compared to the color on the splatmap (see Section 4.3).

8.1.2 Sounds bound to static objects

A static object could be fire, a cave or some emitter that is not moving. Default volume, pitch and range of sound is set by the programmer and should depend on the size of the static object. A big fire will have higher values for volume, and the characteristics of the fire is set by the pitch. The only thing that needs to be calculated during run-time is the volume and checking if the player is within the object's field of sound.

8.1.3 Collision sounds

To generate the correct sound during a collision or an event in the game, the bounding volumes (see Section, 5.2.1) will have tags that can indicate which sound (and how the volume and pitch parameters) should be changed. Every point involved in a collision will carry the following information:

```
bounding_<type>_<material>_<mass>

<type>::= box | sphere | cylinder | capsule
<material>::= wood | metal | plastic | rubber | woodhollow |...
metalhollow | plastichollow | rubberhollow...
<mass>::= <int> (hecto grams)
```

By taking the material, mass and velocity of a colliding point, and the distance between the point and the player car, the appropriate volume, pitch and 3D position can be calculated. For example, if a car hits a tree, two sounds will be played: one with a crashing/bumping metal sound and one depicting crashing/bumping towards wood. The mass and velocity will set the volume and pitch.

8.2 Discussion and conclusion

Several improvements to the audio implementation could be made. The volume and hearing range of a sound could be mapped to a texture on the terrain. The texture can dynamically change during the game. XACT could be combined with manual insertion of sound to gain benefits from the mp3 format.

Possible future work could include the addition of dynamic music, which is not bound to an area on the map, but rather to events. A possible improvement could be intensifying the music when the car is approaching a situation that could involve an event of action and suspense. Another highly interesting, but perhaps complicated addition, is creating sound dynamically during run-time, based on the emitter object's motions. [91] discusses an interesting approach to this, where a swinged sword will automatically create the sound of a sword slash, taking in account the motion and shape of the sword.

9 Networking

Playing a game by oneself can be a rewarding experience. That is, however, nothing compared to experiencing a game together with some friends. Therefore, to further enhance the dynamic experience of the game, networking support was added. This section presents our chosen implementation for adding networking capabilities to this project.

9.1 Results

The first problem encountered was poor library support in the XNA framework. This was due to the XNA network library only working on PCs where user had the XNA software development kit installed. Since installing the XNA software development kit on the client side was not an option, we chose a 3rd-party software library called Lidgren-Network [92] instead.

The networking support was built upon UDP with the external library as an interface to the raw data. There exists no message reliance or sequencing in UDP [93]. However, these were included by default in the external library, making it easy to force the client and server to validate the data sent.

As seen in Figure 48, the game sends updates of coordinates each cycle. This means that each client will send its new position 60 times per second and the server will send back all coordinates of the opponents at the same rate. When sending the car position, we also send the rotation in the x, y and z axes of the vehicle.

9.2 Discussion and conclusion

Some collision testing between cars has been performed. The test results indicate that while the system in its current state works in practice, it is not perfect. The collision that occurs between a local player and a remote player is calculated on both computers. However, only the location of the local player is sent back to the server. If the network is unstable, this could trigger an effect where only one of the players gets the affected by a collision. The solution to this problem could be making the server verify all collisions. However, that was not within the scope of this project.

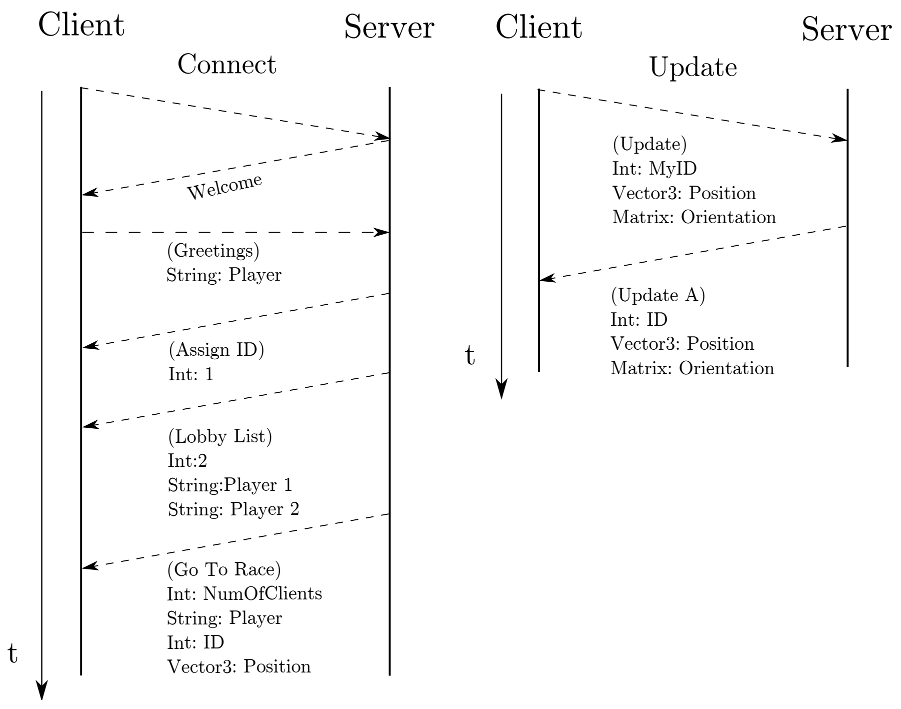


Figure 48: Diagram of network communication, showing what is being sent each cycle

Collision between an object and a player suffers from the same defect. This, however, is more noticeable. All object collisions are calculated on all the client computers, thus making it possible and probable that an object ends up in two different locations on two different clients.

Further work in the area of networking could be adding support for events. As of now, the networking is limited to only showing where the different vehicles are located in the 3D world. An event could be, for example, a player triggering a traffic light and thereby making some animations appear.

10 Artificial Intelligence

Game Artificial Intelligence (from now on AI) refers to all techniques used in computer and video games to produce the illusion of intelligence in the behavior of non-player characters (NPCs). These NPCs will be making different decisions based upon the situation, trying to take the most accurate decision each time. In this section, the AI implemented in this project will be presented.

AI can be pretty simple for standard, automated NPCs. For example, a woodcutter in a game can have a behavior such as: "go and cut where there is plenty of trees". But, it gets really complicated when the NPC's task is to simulate the behaviour of a real player. It is in this situation where AI gets an important role in the development of a game. The role of AI is important in this situation because NPC behavior affects the playability and interactiveness of the game, which will help bringing a joyful experience to the player. This section discusses and presents our chosen implementation of AI into this project.

AI can be as complicated and close to reality as a programmer wishes it to be. AI can be simple and easy to implement or very complicated. Fuzzy Logic or Neural Networks [94] are some examples of advanced AI structures. AI is a very broad field since there are tons of different AI types; each one adapted to each game. These can, however, be categorized by the type of game (Shooter, Strategy, Sports, Racing, etc). Each of those AI have different algorithms and are implemented in their own way.

While focusing on AI algorithms for racing games, there are some basic rules. In every racing game there is some vehicle that has to follow a path. Path Following is the main algorithm a racing game AI has to handle. Reynolds [95] enumerates different steering behaviors for autonomous characters. Even if those algorithms are useful for very different games, we think path following is especially interesting for racing games.

10.1 Path following

Instead of making NPCs detect every slope or turn on the track, having to calculate every deformation on the terrain to deal with it, or having to calculate directions and shortest paths between points, this algorithm just makes the NPC chase a target. Then the target is moved along some predefined path (See Figure 49 for a better understanding of the algorithm).

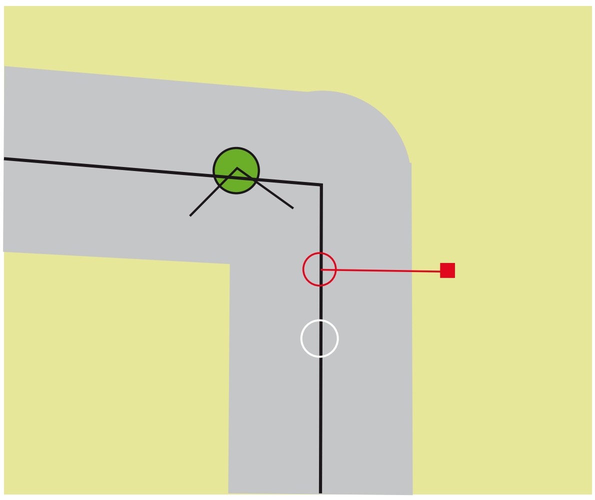


Figure 49: Algorithm with the NPC (green) chasing the target (white).

Path following is implemented here by performing corrective steering only when the vehicle (green circle) begins to head off the path radius (gray region). A prediction of the vehicle's future position (red dot) is made based on its current velocity. This predicted future position is mapped onto the nearest point (red circle) on the path (black line). When the distance between these points exceeds the path radius, corrective steering is required. In the diagram presented here, an equivalent condition is that the red dot moves outside the grey region. Corrective steering is obtained using a "seek" behavior on a target point (white circle) further down the path. For a very smooth path, or a straight line, the red circle could always be used as the seek target, but as Figure 50 shows, we need to make some correction for sharp bends on the path.

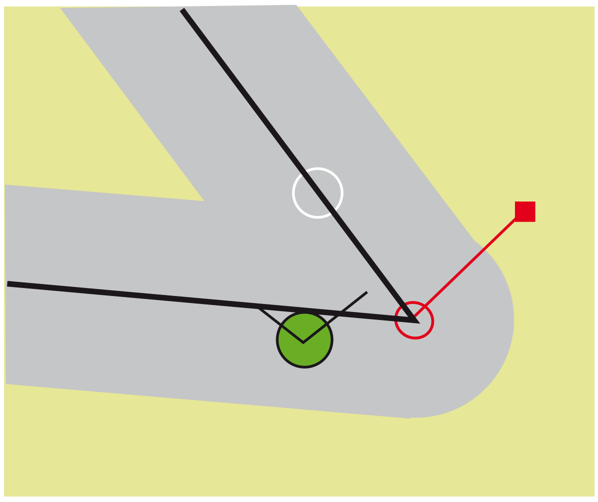


Figure 50: A scenario where the NPC is projected to end up outside the path, and a correction is applied.

As can be seen in Figure 50, the NPC can head off the path if chasing the red circle. Should the NPC end up outside the path, a correction is applied as necessary. The "seek" behavior implementation will correct the car's steering until its direction matches the desired direction. As can be seen in Figure 51, this is when dot product of A and B is zero.

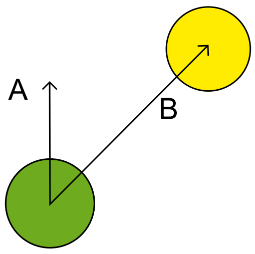


Figure 51: Representation of AI direction vectors A and B.

The dot product is used for mapping the future position of our car onto the path, also called the normal point (red circle). This algorithm ensures robustness against sudden situations where the car drives outside of the track. In Figure 52 is a presentation of how the car (green) is facing a completely wrong direction (violet line) while the target (white) is correctly calculated from the close normal (red circle). The blue line shows the car's desired direction, or the direction to its target.

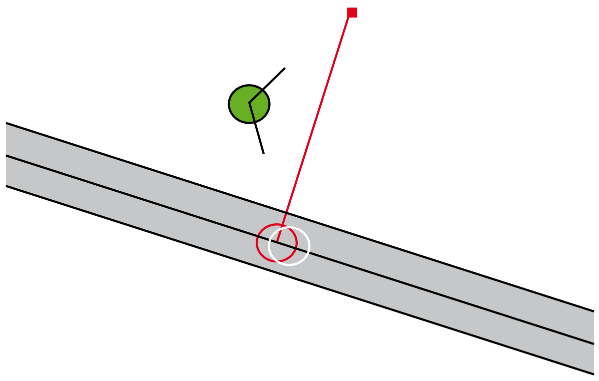


Figure 52: A completely wrong-facing situation.

Once a racing game AI takes care about following the track, it is time to expand it with handling of different scenarios needed to offer a good gameplay. Those situations are countless and depends mostly on the type of game implemented (e.g. Shooting algorithms have to be implemented if the game is intended to have shooting vehicles). Collision is a common situation in racing games, and to have vehicles act "intelligent" a collision avoidance algorithm is needed. Once again, collision avoidance can be implemented in lots of different ways. A possible approach is presented in the next section.

10.2 Collision avoidance

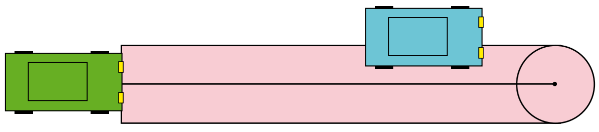


Figure 53: Red area shows collision avoidance area.

Each NPC detects and tries to avoid any object entering their collision area (light red, see Figure 53). This area is calculated from the future projection of the car (red dot as discussed in the previous section). Along the line from the car to this point, any object entering a certain radius will end up on a collision statement, which means a future collision is imminent, making the NPC react by correcting their steering. This correction is made by switching the NPC's path to an alternative path, calculating a new normal and new target on this new path. Adding an alternative path means that we will have two alternative paths present on our track. The track is created from a list of check points, or nodes. Each node is connected to the next node by a straight line, which is the path the AI follows. Linking the last node with the first completes our track, creating a loop. In order to avoid collisions, two of these looping tracks will be needed so that the AI can choose an alternative path around a collision.

10.3 Adaptive driving

In order to have a fun and interactive game for all kind of players, no matter their ability, racing games tend to regulate their speed according to players position [96]. This means that if the player is not very skilled, the NPC cars will drive slower, giving the player the opportunity to catch up. If the player is a good driver the NPCs will increase their speed to offer the experienced player a challenging situation. This technique is called Rubber Banding [96] as it is used to keep all cars in the game together, enhance interactivity, create more collisions and in general make the game more interesting to play.

10.4 Results

We designed our game to be played both in network mode and solo mode. Playing in solo mode means that the player competes with other cars driven by the computer (NPCs). Those NPCs are substituting real players. Therefore, the NPCs should be able to behave as close as possible to real players. This makes AI in our game as demanding as we want. But because of the time limit of this project, and the "simple-to-complicated" development pattern our group chose to work with, our game covers merely some basic behaviour which creates an interesting gameplay experience.

Situations where a NPC car gets stuck trying to drive through a wall is not an acceptable situation as it would make the game look very unrealistic. In order to avoid situations like this and to offer an acceptable gameplay, some of the basic AI behaviors implemented in this project are as follows:

- Path following - NPCs are clever enough to complete a route or follow a path.
- Stuck situation behavior - When a NPC gets stuck, it has to be able to detect this situation and correct it.
- Collision avoidance - When there is some object or another car on a NPC's path, it should be able to overtake another cars or avoid a collision.
- Collision seek - In order to have a more challenging game experience, some NPC's will hit each other or the player's car.
- Adaptive driving - In order to have an always thrilling game experience, NPCs adapt their speed around the track to match the player's speed.

Depending on the different situations that can happen during gameplay, NPCs will have to choose whether to perform one action or another. Every NPC is independent and therefore has to take care of its own actions. The project is structured around a central AIController class that is updated once every game loop and checks each NPC's situation and updates its behaviour if needed by passing them a new "brain", or a new behavior, to follow. Each NPC has a default "brain" or behavior. The default behaviour is to complete a lap by following the path. As seen in Figure 54, depending on the situation, AIController will pass a new brain to a certain NPC, putting this brain first in line in a brain-priority list. This causes the NPC to act according to this behavior until finished, after which it will return to its previous behavior. For example: an NPC is following the track, trying to complete laps, and another car is blocking its path. The AIController detects this situation and gives the NPC a new behavior: "collision avoidance". The NPC will then immediately respond, following this new behavior, and, when completed, return to its previous behavior; completing laps.

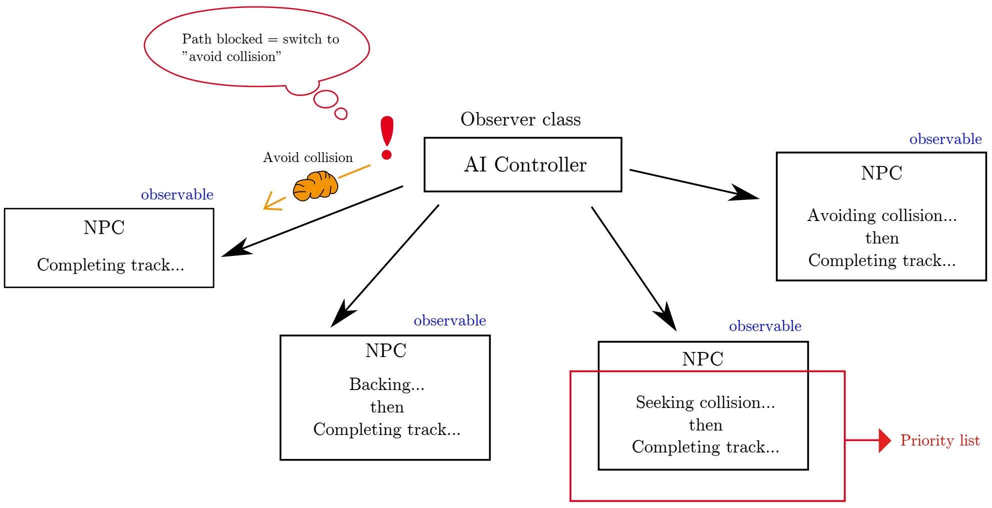


Figure 54: Graphical explanation of the AI structure.

The following five sections present an overview of the implemented AI behaviors.

10.4.1 Path following

As in every racing game, the main goal in our game is to reach the goal at the end of the track. Therefore, we want the NPC cars to drive along this track. To achieve this, we have implemented a path following algorithm based on Reynolds' path following algorithm [95].

10.4.2 Stuck situation behaviour

Should an NPC end up facing a wall or find itself in the middle of a car melee, it will change its "mind" and try to reverse. After spending some time reversing, the NPC will try once again to drive forward.

10.4.3 Collision avoidance

In order to enhance the intelligence of the NPCs and thereby achieve a richer playing experience, NPCs in our game are able to detect other cars and objects, after which they try to avoid them if they are blocking the NPC's path.

As our nodes are connected by straight lines, it might be possible to think that a NPC will turn very harshly at the corners where the nodes are. But that is not the case, as our algorithm projects its position in the future (red dot) and as soon as the target begins to follow the new path the car will begin to turn, causing NPCs to take smooth curves.

10.4.4 Collision seek

As stated in the beginning of this report, our game was meant to feature nice collisions, making collisions one of the main features. Because of that, it was decided that NPCs in this game will have to behave in this way, seeking collisions when an opportunity presents itself. NPCs in our game will try to crash into other cars when they discover an opportunity. An opportunity was defined to be a situation where other cars enter an NPC's 180° front field of sight. This field of sight detects targets in a limited range. For an NPC having another car in his collision seek area, the probability of taking a collision seek action is given by a random factor. This factor is then adjusted to increment or decrease the NPC's aggression towards other vehicles. When an NPC takes the decision of trying to hit another car, it will forget about anything it was doing and instead change its target (white circle in path following explanation) to this car and begin to chase it. The chasing car will experience a small speed boost and follow its target for a set amount of time. When both cars collide or when this time runs out, the NPC will continue to perform the action it was doing before targeting the opponent car. However, special care has to be taken at this point. After a chase, the NPC will not be at the same location as before the chase, and in order to behave in a human way the NPC will have to recalculate its position and determine the most intelligent way to return to following the track again. For example: before a chase, an NPC has a task to go from checkpoint 3 to 4. But after the chase, the NPC has already passed checkpoint 4. In this situation, a far more suited decision would be to aim for checkpoint 5. Therefore, the NPC recalculates its path and updates its checkpoint list.

10.4.5 Adaptive driving

Adaptive driving was implemented by checking where on the track the player's car is located. Thereafter, by comparing the NPC's own checkpoint with the player's closest checkpoint, the speed of the NPC is incremented or decreased depending on if the NPC's own checkpoint is behind or ahead of the player.

10.5 Discussion and conclusion

By implementing *Path Following*, *Collision Avoidance*, *Collision seek*, *Stuck situation behavior* and *Adaptive driving*, we have obtained an, in our opinion, enjoyable and competitive AI. NPC vehicles in our game behave in a sufficiently intelligent way, and are able to face most situations. Hours of testing has been done to ensure the implemented AI algorithms are solid and trustworthy. NPC vehicles have been forced into sensitive or complicated situations in order to study the stability of the algorithms. Although this game's artificial intelligence algorithms could have been developed further, there has been a trade-off between quality and time for this project. More complex approaches could have been chosen. Our opinion is that implementing artificial intelligence algorithms using Fuzzy Logic or Neural Networks [94] would have most likely improved gameplay, but more time would have been needed for that approach to be feasible.

While the implemented AI works, it renders the game in some situations a bit predictable. Future work can be performed on adding new behaviors to NPCs and adding complexity to decision-making algorithms. By adding probabilistic weighting to some actions will easily add an element of randomness to the game and decrease predictability of the AI. It is also possible to add a basic "learning pattern" to NPCs. While this might not be implemented in as a complicated manner as Neural Networks, it will still enhance the game play experience.

11 Conclusion

All good things come to an end, as does this report. This section presents overall results and discusses the project as a whole.

11.1 Results

Overall, the aims of this project have been satisfied. We have created a working 3D racing game with basic gameplay that also features several graphical effects, such as water, fog and a sunlight cycle (see Sections 4.7, 4.6, 4.5). In addition to this, collision detection and reponse has been implemented through JigLibX and bounding volumes (see Sections 5.1, 5.2.1). The game currently runs successfully on a Windows PC. However, we did not succeed in running it on a Xbox360, due to lack of time. Working with C# worked well most of the time. There were, however, some hindrances on the way, e.g. a mysterious System.OutOfMemoryException in the XNA Content Pipeline [29], which appeared at random throughout the development of the project. As this error did, strangely enough, not occur all the time, but due to unknown reasons, our faith in the stableness of XNA has been slightly lowered. Using HLSL in this project worked without complications once the required knowledge of the language was acquired. Also, networking, artificial intelligence and sound handling was implemented without major problems.

There were several benefits of using an iterative programming style during this project. Namely, since much of the programming concerned implementations of features new to all of the members of this project, creating an initially ugly, but working solution allowed the project to make quick progress. Once a solution was proven to work, it was modified to work well with the rest of the project. Although using the MVC pattern to design the code structure enabled us to quickly start working, some minor rewrites had to be done to cope with the increasing size of the project.

In the end, this project resulted in a good-looking 3D game. While presenting the game to test subjects, it was perceived as intriguing with several impressive graphical effects. Some people especially complimented the water effect and lighting.

11.2 Discussion

While the effectiveness of the team could be increased by dividing tasks, we also noticed the importance of all team members having a unified vision of the project. Using an incremental development model worked well in this case, since we had regular meetings several times per week, which served to keep the project on the right track.

The use of custom bounding volumes gave us an easy way of linking the bridge between the 3d model and the game engine. However, the lack of a specification of the FBX file structure caused complications when importing bounding volumes. Many hours were spent on trying to get the bounding volume parser working correctly. Nonetheless, some bounding volumes were still translated in a faulty way.

Choosing to use XNA caused some complications, since XNA went through a major upgrade from version 3 to 4 in 2010, with many breaking changes [97]. Since the vast majority of literature concerning XNA was about version 3, we experienced some complications when trying to implement certain effects, as the upgrade modified some of the internal functionality of XNA.

Future work might include continuing the research for achieving physically correct deformations, since the current solution only works moderately well. Shadows is another subject that might be interesting to implement. Interesting solutions for rendering shadows include soft shadows [98] and Screen Space Ambient Occlusion [99]. Another interesting aspect for future consideration might be to implement actual gameplay, since, as of now, the game merely functions as a visual demonstration of graphical and physical effects.

Glossary

3DS Max	3D editing software., 45, 57
AI	Artificial Intelligence., 55, 67
ASCII	American Standard Code for Information Interchange, a encoding format., 45
billboard	A simple rectangular image which is always rotated towards the camera., 34
FBX	Filmbox, a 3D object file format used by XNA., 45
FoV	Field of view, used in cameras., 50
GPU	Graphics Processing Unit. The piece of hardware responsible for quickly outputting images onto a computer screen., 56
HLSL	High Level Shading Language., 5, 71
JigLibX	A C# port of the physics engine JigLib., 42–44, 46, 47, 54, 71
KD–Tree	A tree data structure., 48
LoD	Level of detail, geometry based simplification., 57, 58
mesh	A collection of edges, triangles and vertices, which defines the shape of a 3D object., 12
MVC	The Model View Controller development pattern., 72
texel	The smallest element of a texture., 10
UDP	User Datagram Protocol. A fast, but unreliable network protocol., 63
XACT	Cross-platform Audio Creation Tool., 59, 60
XNA	A set of tools developed by Microsoft to accelerate game development., 5, 43, 45, 59, 71

References

[1] Lewis M. and Jacobson J. Game engines in scientific research. *Communications of the ACM*, Vol. 45 issue 1, 2002.

[2] Klucher M. XNA framework networking and live requirements. http://blogs.msdn.com/b/xna/archive/2007/11/16/xna-framework-networking-and-live-requirements.aspx, 2011. Online; accessed 15-May-2011.

[3] Need for Speed 3: Hot Pursuit. http://en.wikipedia.org/wiki/Need_for_Speed_III:_Hot_Pursuit, 2011. Online; accessed 15-May-2011.

[4] Mafia II. http://www.mafia2game.com/, 2011. Online; accessed 15-May-2011.

[5] Resident Evil: Extinction. http://www.imdb.com/title/tt0432021/, 2011. Online; accessed 15-May-2011.

[6] Mad Max 2. http://www.imdb.com/title/tt0082694/, 1981. Online; accessed 15-May-2011.

[7] The Book of Eli. http://www.imdb.com/title/tt1037705/, 2010. Online; accessed 15-May-2011.

[8] Fallout 3. http://fallout.bethsoft.com/, 2008. Online; accessed 15-May-2011.

[9] MotorStorm: Apocalypse. http://en.wikipedia.org/wiki/MotorStorm:_Apocalypse, 2011. Online; accessed 15-May-2011.

[10] Hansen S. and Fossum T.V. Refactoring model-view-controller. *Journal of Computing Sciences in Colleges*, Vol. 21 issue 1, 2005.

[11] Blender Foundation. Blender. http://www.blender.org/, 2011. Online; accessed 15-May-2011.

[12] Autodesk. Maya. http://usa.autodesk.com/maya/, 2011. Online; accessed 15-May-2011.

[13] Autodesk. 3D Studio Max. http://usa.autodesk.com/3ds-max/, 2011. Online; accessed 15-May-2011.

[14] Autodesk. FBX file format. http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7478532, 2011. Online; accessed 15-May-2011.

[15] Wikipedia. FBX. http://en.wikipedia.org/wiki/FBX, 2011. Online; accessed 15-May-2011.

[16] Moller T., Haines E., and Akenine-Moller T. *Real-Time Rendering.* AKPeters, 2002.

[17] Blender Foundation. Blender 2.57. http://www.blender.org/development/release-logs/blender-257/, 2011. Online; accessed 15-May-2011.

[18] GIMP. http://www.gimp.org/, 2011. Online; accessed 15-May-2011.

[19] Wikipedia. Paint.NET. `http://en.wikipedia.org/wiki/Paint.NET`, 2011. Online; accessed 15-May-2011.

[20] Corel. Corel Draw. `http://www.corel.com/servlet/Satellite/us/en/Product/1191272117978`, 2011. Online; accessed 15-May-2011.

[21] Wikipedia. Adobe Photoshop. `http://en.wikipedia.org/wiki/Adobe_Photoshop`, 2011. Online; accessed 15-May-2011.

[22] Boardman T. *3ds max 6, fundamentals.* New Riders, 2004.

[23] The matrix page. `http://classic-web.archive.org/web/20091027131421/http://geocities.com/evilsnack/matrix.htm`, 2009. Online; accessed 13-May-2011.

[24] Max script origin. `http://wiki.cgsociety.org/index.php/3ds_Max_History#3D_Studio_MAX_R2`, 2011. Online; accessed 15-May-2011.

[25] Dropbox. `https://www.dropbox.com`, 2011. Online; accessed 15-May-2011.

[26] Shader model 4. `http://msdn.microsoft.com/en-us/library/bb509657(v=vs.85).aspx`, 2011. Online; accessed 15-May-2011.

[27] The Direct3D pipeline. `http://www.viznet.ac.uk/reports/gpu/6`, 2011. Online; accessed 15-May-2011.

[28] Microsoft. XNA generated geometry. `http://create.msdn.com/en-US/education/catalog/sample/generated_geometry`, 2007. Online; accessed 15-May-2011.

[29] Microsoft. XNA content pipeline. `http://msdn.microsoft.com/en-us/library/bb447745.aspx`, 2011. Online; accessed 15-May-2011.

[30] Hargreaves S. XNA reach vs. hidef. `http://blogs.msdn.com/b/shawnhar/archive/2010/03/12/reach-vs-hidef.aspx`, 2010. Online; accessed 15-May-2011.

[31] James S. *3D Graphics with XNA Game Studio 4.0.* Packt Publishing, first edition, 2010.

[32] GameDev.net remigus. Splat map definition. `http://www.gamedev.net/topic/505887-xnamanipulating-uv-coordinates-on-a-height-map/page__view__findpost__p__4299450`, 2008. Online; accessed 15-May-2011.

[33] GameDev.net haegarr. Splat map definition. `http://www.gamedev.net/topic/382840-i-dont-understand-splatmaps/page__view__findpost__p__3530521`, 2006. Online; accessed 15-May-2011.

[34] Woop S. A Ray Tracing Hardware Architecture for Dynamic Scenes. Technical report, Saarland University, 2004.

[35] Ignatenko A. et al. A real-time 3d rendering system with brdf materials and natural lighting, 2004.

[36] Gouraud H. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, 20:623–629, 1971.

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

[38] Phong B.T. Illumination for computer generated pictures. *Commun. ACM*, 18:311–317, June 1975.

[39] Bishop G. and Weimer D.M. Fast phong shading. *SIGGRAPH Comput. Graph.*, 20:103–106, August 1986.

[40] Hargreaves S. XNA specularity. `http://blogs.msdn.com/b/shawnhar/archive/2007/04/12/specularity.aspx`, 2011. Online; accessed 15-May-2011.

[41] Ngan A. et al. Experimental validation of analytical brdf models. In *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, pages 90–, New York, NY, USA, 2004. ACM.

[42] Nvidia texture tools for adobe photoshop. `http://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop`, 2011. Online; accessed 15-May-2011.

[43] *Reconstruction filters in computer-graphics.* Mitchell, D.P. and Netravali, A.N., 1988.

[44] Sneep M. and Ubachs W. Direct measurement of the rayleigh scattering cross section in various gases. *Journal of Quantitative Spectroscopy and Radiative Transfer*, Vol. 92 issue 293, 2005.

[45] Nishita et al. Display of the earth taking into account atmospheric scattering. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 175–182, New York, NY, USA, 1993. ACM.

[46] Pharr M. and Fernando R. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional, first edition, 2005.

[47] Jensen H.W. et al. Night rendering. 2000.

[48] Grootjans R. *XNA 3.0 Game Programming Recipes: A Problem-Solution Approach.* Apress, Berkely, CA, USA, 2009.

[49] Enright D. et al. Animation and rendering of complex water surfaces. *ACM Trans. Graph.*, 21:736–744, July 2002.

[50] Carter C. *Microsoft XNA Unleashed: Graphics and Game Programming for Xbox360 and Windows.* Sams, Indianapolis, IN, USA, first edition, 2007.

[51] Nguyen D.Q., Fedkiw R., and Jensen H.W. Physically based modeling and animation of fire. *ACM Trans. Graph.*, 21:721–728, July 2002.

[52] Fernando R. *GPU gems: programming techniques, tips, and tricks for real-time graphics.* Addison-Wesley, pub-AW:adr, 2004.

[53] Knutzen J. Generating climbing plants using L-systems. Master's thesis, Chalmers University of Technology, 2003.

[54] Knott D. et al. Particle system collision detection using graphics hardware. In *ACM SIGGRAPH 2003 Sketches & Applications*, SIGGRAPH '03, pages 1–1, New York, NY, USA, 2003. ACM.

[55] J.X. Chen, Fu X., and Wegman J. Real-time simulation of dust behavior generated by a fast traveling vehicle. *ACM Trans. Model. Comput. Simul.*, 9:81–104, April 1999.

[56] Star Wars: Force Unleashed. `http://www.lucasarts.com/games/theforceunleashed/`, 2011. Online; accessed 15-May-2011.

[57] Havok Physics Engine. `http://www.havok.com/`, 2011. Online; accessed 15-May-2011.

[58] Half life 2. `http://orange.half-life2.com/`, 2011. Online; accessed 15-May-2011.

[59] PhysX physics engine. `http://www.nvidia.com/object/physx_new.html`, 2011. Online; accessed 15-May-2011.

[60] Bullet physics engine. `http://bulletphysics.org/wordpress/`, 2011. Online; accessed 15-May-2011.

[61] Grand theft auto IV. `http://www.rockstargames.com/grandtheftauto/`, 2011. Online; accessed 15-May-2011.

[62] Open Dynamics Engine. `http://www.ode.org/`, 2011. Online; accessed 15-May-2011.

[63] S.T.A.L.K.E.R. `http://www.stalker-game.com/`, 2011. Online; accessed 15-May-2011.

[64] World of Warcraft. `http://eu.battle.net/wow/en/`, 2011. Online; accessed 15-May-2011.

[65] `http://www.msxbox-world.com/xbox360-specification.php`.

[66] Drifting. `http://en.wikipedia.org/wiki/Drifting_%28motorsport%29`, 2011. Online; accessed 15-May-2011.

[67] Wobbly vehicle car physics. `http://forums.create.msdn.com/forums/p/29540/175820.aspx`. Online; accessed 15-May-2011.

[68] Siggraph. *Haptic Deformation using Graphics Hardware and KD-Trees*, 2006.

[69] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[70] Che Y., Wang J., and Liang X. Real-time deformation using modal analysis on graphics hardware. In Y. T. Lee, Siti Mariyam Hj. Shamsuddin, Diego Gutierrez, and Norhaida Mohd. Suaib, editors, *GRAPHITE, Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia 2006, Kuala Lumpur, Malaysia, November 29 - December 2, 2006*, pages 173–176. ACM, 2006.

[71] Milev A. `http://www.codeproject.com/KB/architecture/KDTree.aspx`, 2007. Online; accessed 15-May-2011.

[72] Jönsson G. and Nilsson E. *Våglära och Optik*. Teach Support, 2008.

[73] Longhurst C. The suspension bible. `http://www.carbibles.com/suspension_bible.html`, 2011. Online; accessed 2-May-2011.

[74] Kirmse A., editor. *Game Programming Gems 4*. Charles River Media, 2004.

[75] Kincaid D. and Cheney W. *Numerical analysis: mathematics of scientific computing (4th ed)*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1999.

[76] Millington I. *Game Physics Engine Development; electronic version*. Elsevier, San Diego, CA, 2007.

[77] Clark J.H. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, October 1976.

[78] Duchaineau M. et al. Roaming terrain: real-time optimally adapting meshes. In *Proceedings of the 8th conference on Visualization '97*, VIS '97, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[79] Giegl M. and Wimmer M. Unpopping: Solving the image-space blend problem for smooth discrete lod transitions, March 2007.

[80] Audio in XNA game studio. `http://msdn.microsoft.com/en-us/library/bb203895(v=XNAGameStudio.31).aspx`, 2011. Online; accessed 15-May-2011.

[81] `http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.content.pipeline.audio.audiofiletype.aspx`, 2011. Online; accessed 15-May-2011.

[82] Pitch. `http://en.wikipedia.org/wiki/Pitch_(music)`, 2011. Online; accessed 15-May-2011.

[83] Panning. `http://en.wikipedia.org/wiki/Panning_(audio)`, 2011. Online;accessed 15-May-2011.

[84] MediaPlayer Members. `http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.media.mediaplayer_members.aspx`, 2011. Online; accessed 15-May-2011.

[85] Audio overview of XNA. `http://msdn.microsoft.com/en-us/library/bb195055(v=XNAGameStudio.31).aspx`, 2011. Online; accessed 15-May-2011.

[86] Reed A. *Learning XNA 3.0 - Game Development for the PC, Xbox 360, and Zune*. O'Reilly, 2009.

[87] Audio cues. `http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.audio.cue(v=xnagamestudio.20).aspx`, 2011. Online; accessed 15-May-2011.

[88] XNA game studio: mp3. `http://msdn.microsoft.com/en-us/library/bb203895(v=xnagamestudio.20).aspx`, 2011. Online; accessed 15-May-2011.

[89] Wavepad. `http://www.nch.com.au/wavepad/index.html`, 2011. Online; accessed 15-May-2011.

[90] The freesound project. `http://www.freesound.org/`, 2011. Online; accessed 15-May-2011.

[91] Dobashi Y., Yamamoto T., and Nishita T. Real-time rendering of aerodynamic sound using sound textures based on computational fluid dynamics. *ACM Trans. Graph.*, 22:732–740, July 2003.

[92] Lidgren networking library generation 3. `http://code.google.com/p/lidgren-network-gen3/`, 2011. Online; accessed 15-May-2011.

[93] Postel J. User datagram protocol rfc 768. `http://tools.ietf.org/html/rfc768`, 1980. Online; accessed 13-May-2011.

[94] Zadeh L.A. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37:77–84, March 1994.

[95] Reynolds C.W. Steering behaviors for autonomous characters, May 01 1999.

[96] Missura O. and Gaertner T. Player modeling for intelligent difficulty adjustment, October 03 2009.

[97] Hargreaves S. Breaking changes in XNA Game Studio 4.0. `http://blogs.msdn.com/b/shawnhar/archive/2010/03/16/breaking-changes-in-xna-game-studio-4-0.aspx`, 2011. Online; accessed 15-May-2011.

[98] Hasenfratz J. et al. A survey of Real-Time Soft Shadows Algorithms. *Computer Graphics Forum*, 22:753–774, 12 2003. I.: Computing Methodologies/I.3: COMPUTER GRAPHICS/I.3.7: Three-Dimensional Graphics and Realism, I.: Computing Methodologies/I.3: COMPUTER GRAPHICS/I.3.1: Hardware Architecture, I.: Computing Methodologies/I.3: COMPUTER GRAPHICS/I.3.3: Picture/Image Generation.

[99] Bavoil L. Screen Space Ambient Occlusion. Whitepaper, `http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf`, August 2008.