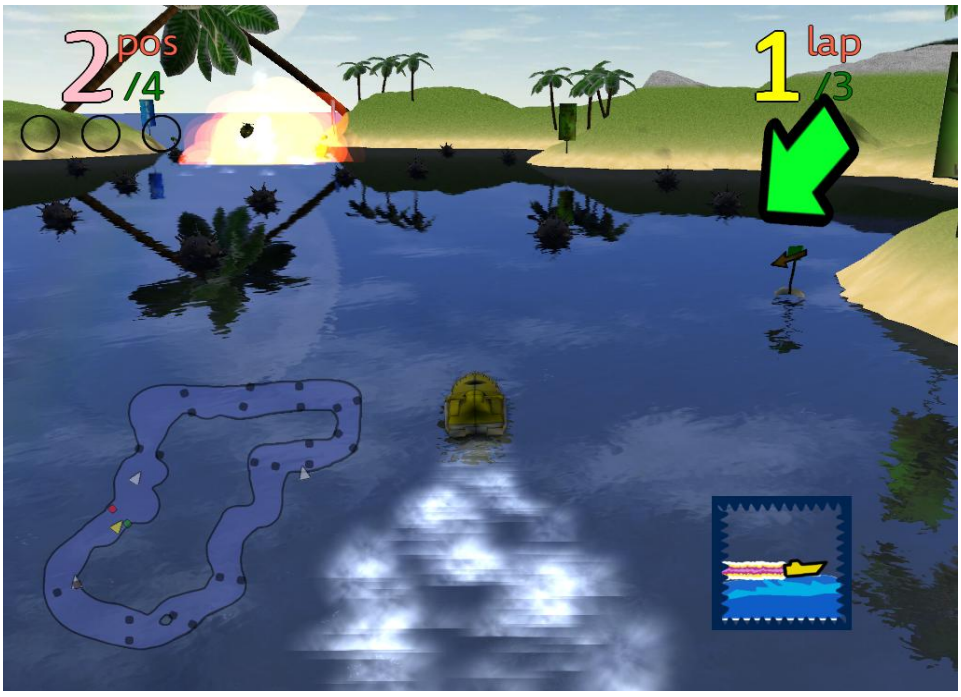
CHALMERS



Water Racing

A study and implementation of game development techniques for smaller projects

Bachelor´s Thesis

MATHIAS ANDERSSON
JAKOB BRATTÉN
TOBIAS FÄRDIG
SEBASTIAN GELOTTE
ROGER LJUNGBERG
DANIEL WOGELBERG

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2010

Abstract

This thesis examines which techniques are suitable for developing a game rapidly. The techniques involved come from the following areas concerning game development: the choice of development process, graphics, network, game sound, physics, artificial intelligence and user interface.

In order to examine which techniques are appropriate, a study was conducted where the six authors of this report formed a team. Using an Iterative process model and the framework XNA, they developed a game in four months. The three-dimensional boat racing game called Water Racing was the result.

It is shown that there are several tools and methods available, which assist in the task of developing games of modern standard at a rapid pace. Among these, the Iterative process model is shown to be suitable for smaller game projects. Also, it is shown that the genre of boat racing might have several advantages that reduce the amount of time and resources required for development, compared to other kinds of racing games.

Terminology

Texel - Abbreviation for "texture element". A point in a texture.

Billboard - A texture which always faces the camera.

CPU - Central Processing Unit.

GPU - Graphics Processing Unit.

Haptic Feedback – Information given to the user in the form of sense of touch.

Table of Contents

1 Introduction

Game development is a rapidly growing industry. In thirty years, the technology has developed considerably and raised the expectations of commercial computer games. The time and resources needed to make a commercially successful game have therefore increased over time. However, game development is a profitable business. The game industry made a profit of over twenty billion dollars in the United States in 2009 (Brightman 2010). It is therefore interesting to know what potential there is, not only for established companies, but also for minor projects that are led by programmers who aspire to advance from hobby level to professional level. If there is a sufficient amount of techniques available to assist game development in smaller projects, a talented programmer could have the possibility of making a living by forming a company on his own, that develops computer games.

The problem today, however, is that many of the tools and methods made to assist the development of games are too complex to be used in a project of a smaller size. The learning period to master these tools or techniques might be too long, or specialized skills may be required. To finish the project in time, a set of techniques would be required that allowed a rapid approach. Techniques being defined in this study as suitable for a smaller game development project have one or more of the following properties:

1) The time required for learning is short.

2) The technique uses a high degree of automation of tasks.

3) The framework shifts the focus to tasks of higher level.

In order to find out which tools and methods are appropriate for rapid game development, a study was conducted. This study was made by a team of six students at Chalmers University of Technology who were given the task of developing a three-dimensional boat racing game in four months. In this report, we choose to evaluate a number of techniques that could be suitable for this task. The goal was to make a game that held the highest possible quality that the team possibly could develop under their conditions. The team consisted of the authors of this report, who were Bachelor of Science students studying Informations Technology and Computer Science. They had previous experience of game development from several projects during college.

The purpose of this report is to, with results of the study as a basis, answer the question: ***Which techniques are appropriate to use when developing a game in a short time frame, and at the same time aspiring for a high quality result?*** This concerns all the important areas of game development:

The development process

The planning for the game project and the choice of process model.

Graphics

The visual techniques of the game.

Network

A game mode where several persons play simultaneously in the same game.

Sound

Playback of music and sound and the techniques involved.

Physics

The implementation of game physics.

Artificial Intelligence

The simulation of humanoid behaviour in the game.

User Interface

Parts of the program designed to be user-friendly.

1.2 Method

A number of methods were used which are not explained in detail later in this report, and some of these will be mentioned here. These were first and foremost the frameworks used when programming, as well as the synchronization tools.

The game in this study was built using the C# framework XNA. The reason for this choice was based on the fact that it is a framework specifically created as a platform for which to develop modern games (XNA 2007). Therefore, the programmer will not have to write as much code for the game-specific details as when only using an ordinary programming language, such as C++ or C#. Furthermore, the team had previous knowledge of XNA and would not have to spend much time learning a programming language prior to development. Visual Studio was the development environment used in order to get full support for the XNA framework.

In order to synchronize the code base, Tortoise SVN was used, which enabled several programmers to work simultaneously on the same code. This program enabled the updating of the project to be done in a flexible way. With just one click, the latest changes made to the project by another team member could be acquired. The word processor Google Docs was used to synchronize the writing of this report. In Google Docs, several persons may edit the same document at once from different systems. By using both Tortoise SVN and Google Docs, it was possible for the team members to work concurrently at the different project parts, and to work at different locations.

When developing the game, open source code was used to a considerable extent. It was used to implement the water (Hayward 2008) and particle effects (XNA Creators Club 2010), as well as for generating terrain based on a heightmap (Grootjans 2008).

1.3 Restrictions

The limited amount of time being available for the project led to restrictions of certain areas. Therefore, in this section these areas are listed along with motivations of why the team did not spend as much effort on these, compared to the rest of the project.

The most obvious restriction made was associated with game content. High quality models, interface designs, music and sound are desirable elements when striving for a game of high standard. However, the team consisted of mainly programmers and while some had specialized knowledge of some of the content areas, the game-play mechanics in the game was considered to be of higher priority. Because of this, only a limited amount of time was spent at content creation.

While the purpose of the report is to evaluate development tools, the XNA framework will not be extensively evaluated. XNA was the

only framework the development group had familiarity with, and was thus chosen without consideration of other tools.

Testing was another are being limited. Structuring the program around methods such as unit testing, would take a considerable amount of effort. In the study, no systematic method for testing was used. Instead, the program was tested at the end of each iteration. Utilizing their previous experience of games, the team members test-played the game until they had identified a number of defects, if they found any. Therefore, the application was considered to be highly dependable while the fact remained that it could not be guaranteed of containing no errors. This approach is partly based on the well-known fact that testing can only show the presence of errors and not the absence of them (Jeffries 2009). So even if a systematic approach for testing would be used, it would still not be possible to guarantee a game with no errors in it.

Artificial intelligence is another area where much time can be spent on the implementation of advanced and robust methods. In order to avoid time consuming and complex coding, the method selected for AI behavior is limited in its access to complex maneuvers that the player characters are able to perform.

Regarding graphics, the reason for the lack of sections concerning lighting or shadowing is that XNA's default settings were used for this. While being important for the realism, lightning and shadowing were not prioritized for the game. The reason for this is further explained in section 3.6.

Furthermore, the application was developed for personal computers with modern hardware. While certain Xbox support is inherently given due to our development tools (XNA Gamer Services 2009), no testing has been done to ensure that the application can be executed on any other hardware than the one used in the development. The hardware consisted of an AMD Athlon 64 X2 Dual Core Processor 5000+ (2.61 GHz), an nVidia GeForce 8800 GS graphics card and 2.0 GB of RAM.

2 The Development Process

This chapter will examine a fundamental area of any major project in information technology - the development process. This is basically a template for the project plan.

To describe how to proceed making a game, one cannot tell exactly how to carry out each and every step in advance. Therefore, an abstract representation of the development process is needed, telling what phases and results will occur at certain given times. This is known as a *software process model* (Sommerville 2010).

Some of the most common and widely used process models will be covered in this report. Section 2.1 will discuss the Waterfall Model - one of the most universally recognized process models. In 2.2, the Spiral Model is discussed. This is most widely used in the area of game development (Schell 2008). Agile development is covered in 2.3 and Iterative development in 2.4.

2.1 Waterfall Model

One of the most well-known software processes is the Waterfall Model. In the Waterfall Model, the different stages of development occur one after another,

resembling a finite state machine (Scacchi 2001). In the *requirements* phase, it it specified what kind of services the system should provide. This might include a feasibility study with potential customers. How the system should work is then described in the *design* phase. This can be described as the architectural phase of the development process. When having this skeleton for the system, the *implementation* phase begins. Here, the system is being constructed. *Testing* is then carried out to detect errors being present in the system. When the system is complete, *maintenance* is the final step. This might include the delivery of the system to a customer, and updating it when necessary.

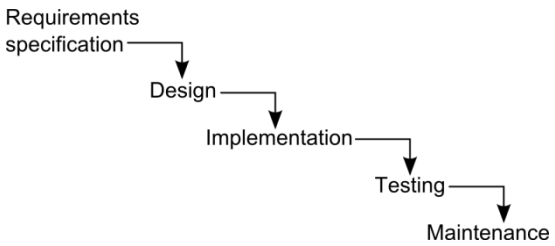


Figure 1: *The Waterfall Model*

Klein mentions both several advantages as well disadvantages with the Waterfall Model. There are two particular advantages. With the large amount of planning, it is a low-risk approach. Also, with the all the documentation being produced, it is easy to continue the project even if team members are replaced. However, there are several disadvantages as well. The most obvious is the inflexibility - when one phase is completed, it is impossible to go back (Klein 2008).

2.3 Spiral Model

The Spiral Model of software development is interesting in the area of game development, since this is the area where it is most popular (Schell 2008). Therefore, this was initially a model being considered for the project.

Spiral development is based upon an iteration of four steps (Gooma, Kerschberg 1995):

1. Determining object and constraints
2. Analyzing Risks
3. Developing Product
4. Spiral Planning

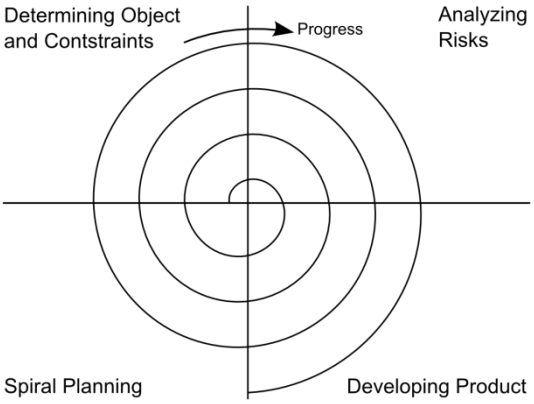


Figure 2: *The Spiral Process Model*

The characteristic properties about the Spiral Model are the first two steps. Step 1 is about finding the approach for the next iteration. This could be done by prototyping the product. In step 2, a risk assessment is done. By doing this, one can plan beforehand how to proceed the project if having issues with time management or resources. Next follows the implementation of the product. Finally, planning is done for the next iterations.

According to Boehm, there are two disadvantages of the Spiral Model. The first is the need for risk assessment expertise. The second is the need for further elaboration on the process steps when different levels of experience are present in the team (Boehm 1988). Another problem stems from the fact that the project may not afford to spend enough time on risk assessment and

prototyping to adhere to the spiral process model.

2.4 Agile Development

Agile development is one of the more modern process models. Four aspects make a software development model agile (VTT Electronics, 2002). The model has to fulfill four criteria. It has to be:

1. Incremental
2. Have customer involvement
3. Straightforward and easy to learn
4. Adaptive for sudden changes

Besides these criteria, most Agile development processes build upon several principles according to Craig Larman. Simplicity - not making more than necessary - is important. Also, teamwork is a cornerstone. The primary factor to consider when measuring the progress is how well the product itself works (Larman 2003).

There are several forms of agile development to choose from. A popular form today is *Extreme programming.* This method is largely based on teamwork (Extreme Programming 2009). Communication is an important aspect between the team members. The customer himself is considered a team member, and has an important role during the development when it comes to stating requirements and evaluating the product. As according to criteria 1, parts of a working system (increments) are delivered to the customer on a regular basis.

2.5 Iterative Development

In Iterative Development, the software development life cycle is divided into several iterations. The reason for this is that problems or faulty assumptions can be discovered early in each iteration, and thus be corrected at an early stage (Hung 2007). The different stages are basically the same as in the Waterfall Model: planning, implementation, testing and evaluation are usually carried out in each iteration. There are different types of Iterative Development depending on what factor is of concern. Craig Larman mentions these four types (Larman 2003):

Risk-driven Iterative development

This approach deals with the riskiest or most difficult task in the beginning of each iteration, and thus aims for the safest way of finishing the development.

Client-driven Iterative development

When the client has the possibility of choosing features in each iteration, this variant is used.

Evolutionary Iterative development

This is chosen when one wants to remain flexible in changing the different parts of the project - be it requirements, estimates or major parts of the implementation.

Adaptive Iterative development

Adaptive Iterative development is used when feedback from work already done is what requires an adaptation in response. This could be for instance when the programmers, after some programming sessions, discover that a particular section needs more attention than initially expected, and thus change the requirements.

Many similarities can be found between the agile and iterative development processes. Both methods emphasize the reworking of the system in cycles. However, Agile Development is largely based on team collaboration and

customer involvement, which is an important distinction (AgileCollab 2008).

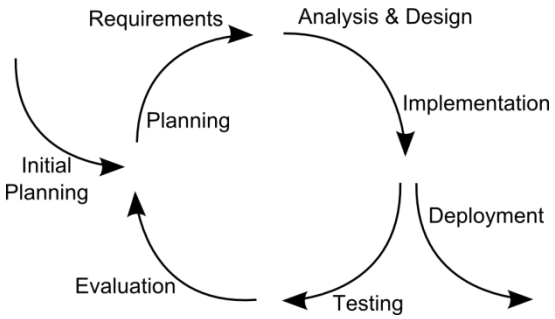


Figure 3: *An example of an Iterative development iteration cycle*

2.6 Results

The Iterative process model was the process model of choice for the study. The variant of choice was the *Adaptive Iterative Process Model*. This was because the team's work would lead to the change of requirements. The development team did not know beforehand exactly which features they would have time to implement, which ones that would be too hard or which areas the team would be able to lay more focus on than initially expected. Therefore, in each iteration they wanted to be able to change the requirements if necessary.

The iterations were approximately two weeks long. Each iteration had a certain set of *milestones*, being certain features that would have to be implemented before the deadline of the iteration. *Planning* would be the first step of each iteration. Most planning was done after the initial planning phase. It could occur, however, that somebody finished a milestone earlier than expected and needed more work, or got stuck and needed support. It could also be about the addition of features, and the removal of features depending on time management and level of difficulty. An important planning tool that was used was the *Gantt chart*. By using a Gantt chart, it could be planned when to carry out the different tasks of the project. This is a tool that is not only useful in the initial planning, but also throughout the project as a way of monitoring whether the project is on schedule or not (MindTools 2010).

In each iteration, each person was given an individual milestone. Each person was responsible for one of totally six areas, the areas being the following:

- User interface
- Graphics
- Artificial intelligence
- Sound
- Network
- Input and controls

In the end of the iteration, *testing* was done on the results. After detecting possible errors, an *evaluation* was made where it was stated whether the results matched the requirements so far. The evaluation was mostly done at a meeting along with the supervisor for the project.

2.7 Discussion

Iterative development showed to fit very well to the project under its circumstances. The Waterfall Model would be inflexible, and the Spiral Model appeared to only suit larger projects. While Agile development was one of the candidate models, there was no customer to work against. Also, the focus of team collaboration was an aspect which did not match our conditions well. Often the team members worked individually, since the school schedules were different. Overtime was also present at different stages in the project, something that is not typical of Agile Development. Therefore, only Iterative Development remained, and there were a

number of aspects that made it optimal for rapid game development.

Due to the time constraint being present in the study, the team needed to work in a way that made the implementation phase active most of the project duration, without the need of making all the requirements and planning beforehand. An important aspect was that the team worked to produce the best game it possibly could in the given time frame, without having a detailed specification of a final product. Therefore, requirements specification and implementation had to be done concurrently.

Another important aspect of why iterative development was suitable for the study was that the focus could be mostly on programming rather than formalities. Also, adaptability was an important aspect, since the project showed to be very dynamic with many changes occurring during development. With agile development, it was easy to change the plan during development.

A particular advantage of the *Iterative development* model for this study was that parts of a working system could be delivered on a regular basis. This meant that there was a guarantee of having a program to show during the implementation, something that appeared to be a rather important aspect. There were frequently sessions where the team showed the game for the supervisor, and there was a half time review in the middle of the implementation phase where the project was to be presented for all the other candidate project groups at Chalmers University of Technology.

3. Graphics

The graphics of a game is sometimes considered an indicator of the overall game quality as well as the most important factor in order to give a good first impression of a game, according to lead programmer Jake Simpson of Raven Software (Simpson, 2010). Therefore, it was considered a high priority in the study. With good graphics, the game looks impressive, so that people remembers it and wants to play it again. The idea of a water racing game was largely based on the assumption that it is relatively easy to make a game of that kind look appealing. Therefore, the most important graphical techniques used in the study will be presented.

Particle systems will be examined in section 3.1. These were used to create most effects in the game, such as water splashes, fires and explosions. Section 3.2 will examine different techniques of rendering realistic water, and describes the technique that was chosen for the application. Different types of culling, meaning ways of saving system resources when drawing objects, are covered in section 3.3. Section 3.4 will describe the best ways of acquiring graphical content for the game; how to generate a landscape as well as game characters.

3.1 Particle Systems

While most of the graphics in computer games consists of textured 3D models, there are some effects that must be achieved in different ways in order to ensure visual quality and performance. The purpose of a particle system is to render more complex effects such as explosions, smoke, water spray and fog.

A particle system is basically a set of points in a three-dimensional space. These points define the positions of particles being generated. The particles have life cycles; they are born, they live for a certain time and they die. Each particle has properties to describe the way it moves. The particles' movements are not always completely predetermined though, because in many cases they have a randomizing element which creates the feeling of a very chaotic and natural effect (Lander, 1998). What the particles form on the screen varies with the properties one desires. They can be everything from just painted pixels at the particles position in a given color, to fully rendered 3D-models at each position being specified. The different methods of particle rendering all have different problems. The next section focuses on problems with particles when they intersect with the terrain.

3.1.1 Soft Particles

When billboard particles intersect with geometry in the scene, they will partially disappear. This makes it very obvious to the observer that the particles are nothing but textured planes.

Figure 4: *The dashed circle marks an area where the non-soft particles intersect with the terrain mesh.*

Soft particles are particles where this artifact is not present. One of the simplest ways of implementing soft particles is by fading out the particles' corners when they intersect with geometry. In this way the particles will not appear to intersect with the geometry (Soft Particles, 2009). This, however, means that all the geometry in the scene being rendered to the screen also has to be rendered to a texture which holds depth information. Next, the particles depth per pixel is compared to the depth of the scene at this pixel. If the geometry and the particle overlap, the currently rendered particle is being faded.

3.1.2 Results

The particle system in the game is implemented as an extension of a 3D-particle system sample found on the XNA Creators Club (XNA Creators Club Online 2010). This particle system is designed to minimize the CPU overhead by calculating the particles' movement on the GPU. The CPU is only responsible for adding new particles to a vertex structure where start time, position and velocity is stored. The GPU can then, by reading this vertex buffer, calculate the age of each particle from the creation time and the current time, where the two times are set each frame as vertex shader parameters. From these values, the shader can calculate where the particle should be and draw the specified image at this position.

Billboards were the chosen method for the rendering of each particle used in the game. This means that the particles would look the same regardless of from which direction they are observed, but since the particles in the game were always used for very complex looking effects this did not cause any problems. The billboard methodology is

efficient since it only uses two triangles and a texture.

The soft particles-method was never used, since it was a graphics enhancement that was not in top priority. Using soft particles would also create a loss in performance, since all the geometry would have to be rendered twice. The simple pixel shader that was used in the game would also have to be more complex in order to handle the fading of the particles' corners.

3.2 Water Rendering

In computer games, one of the most important goals is to immerse the play into the virtual world. Because of this, for a game that is set on water, realistic behavior of the water is important (Stam, 2003). The following sections describe different techniques to achieve appealing, realistic water.

3.2.1 Water Geometry

The geometrical form of the water is usually represented by triangles, as most else in a game scene. Having a complex, animated mesh of water will improve realism and enable heavy waves. It may be preferable to have a complex mesh if water has a central role in the game such as in the submarine simulator game Silent Hunter 3 (Ubisoft Romania, 2005). Further, physical waves may affect the gameplay dynamics of a game, such as in the GameCube game Wave Race: Blue Storm (NST, 2001) where waves can be used to perform high jumps.

In certain situations, rendering a plane to represent water geometry is considered to be graphically convincing enough. When representing a small body of water, or a larger body unaffected by wind or seen from a great distance, no physical waves are necessary to give an impression of realism. The team observed that many games that are considered graphically impressive uses a mere plane as water geometry, and rely on realistic shaders and effects to improve the visual quality of the scene. Such games include Half-Life 2 (Valve Software, 2004) and Bioshock (Irrational Games, 2007).

3.2.2 The Fresnel Effect

In the real world, the degree of reflectance of smooth surfaces varies depending on the refractive index and the viewing angle (Westin, 2007). A high angle ensures high reflectance, and a low angle ensures low reflectance. This phenomenon is called the Fresnel effect. A method that is often used in order to render water, is drawing a plane and calculating the angle between the camera and the plane's normal (Toman, 2009). Since the refractive index of air and water remains constant enough to be approximated, this angle can be used to determine Fresnel reflectance.

3.2.3 Reflection and Refraction of Light

Besides the shape of the water, the actual surface of the liquid also affects the way it is perceived. It is of concern that the pixels representing the surface of water are colored in a way that looks plausible. A naive approach is to simply apply a texture and animate it. In the real world however, water both reflect and refract incoming light. If realistic visuals are important, the surface must take the surrounding environment into consideration.

3.2.4 Reflection by Cube Map

A common approach when rendering reflections is to apply an environment map, such as a cube map (Lombard, 2004). A panorama of the environment is placed into six-square-two dimensional textures arranged like a cube (nVidia, 1999). A vector from the eye to a point in the reflective object's surface, together with this point's normal is used to create a vector reflecting the original vector. The texel of the point of interception of the cube map, and the reflection vector (whose location of origin is set in the center of the cube) is then used to color the given area of the object. The textures used in the cube map are preferably based on the scene in which it is used, in order to increase realism.

However, it is of note that reflections based on the environment map technique appear as if the objects they are based on are infinitely far away. While this work well for certain scenes, local objects of the scene are not present in the rendered reflections.

3.2.5 Planar Reflection

The technique known as "planar reflection", or "stencil reflection" is based on very simple concepts. The basic premise is to initially render a version of the scene flipped around the axis of reflection, using the stencil buffer to ensure reflections is only present in the pixels occupied by the reflective surface (OpenGL, 2001). A disadvantage of this technique is that it only works for planar surfaces, but can therefore be used for rendering flat mirror reflections.

3.2.6 Distorting Reflection and Refraction of Light

A variation of the planar reflection technique involves rendering a refraction texture, which is rendered from the same location as the camera is set but does not contain the water plane or any geometry above this plane.

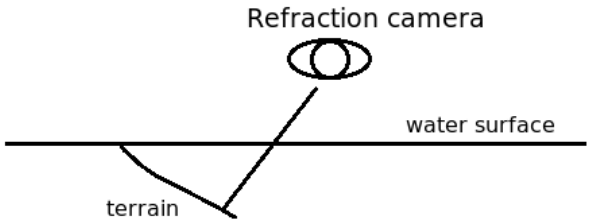


Figure 5: *The refraction camera's position and orientation are identical to the position and orientation of the player camera.*

A reflection texture is also rendered in order for the algorithm to later be able to render reflections. The location of the camera is set with the same world location and the view camera, but with mirrored Y-coordinate.

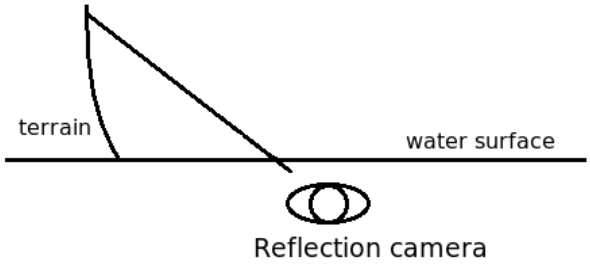


Figure 6: *The reflection camera's position and angle is mirrored in the Y-plane compared to the position and angle of the refraction camera (figure 5).*

Water refracts and reflects light depending on viewing angle, and the reflection and refraction textures are thus set according to the Fresnel term. By using this technique to render a mirror, it becomes simple to create an illusion of moving waves by adding a displacement term to the projection texture coordinates (Toman, 2009).

3.2.7 Results

The water graphics in the game was achieved by implementing a system distorting a planar reflection to achieve convincing wave effects on a plane. As the gameplay took place in a plane, only techniques based on plane water geometry was considered for the application.

Initially, an algorithm heavily based on the code presented in *XNA Tutorial using C# and HLSL Series* 4 was implemented. The result rendered convincing reflection and refraction of light and simulated waves by displacement of texture coordinates as described in previous section. While functional, it was later decided that the application would benefit from using the version of the algorithm presented by Kyle Hayward in 2008. This algorithm supported the same features, and was conveniently written as a game object. This suited well for the game, as it enabled quick implementation and tweaking for improved visual quality and robustness.

3.3 Culling

When drawing the objects in the scene, culling techniques are often implemented to disable rendering of triangles whose contribution to the resulting screen would be minimal or non-existent. There are several common culling techniques, with varying degrees of complexity and efficiency. Some are more viable than others depending on the application they are to be implemented in.

3.3.1 Culling Techniques

View frustum culling is a rather intuitive approach to reduce the number of rendered triangles in a scene. A viewing volume is generated depending on the camera position, and any object whose bounding volume is completely outside this volume is not rendered. A naive implementation of such a technique would require a collision check between all objects on the scene and the viewing volume. Notable performance gain could be achieved if the game scene would be divided into smaller volumes, so the system could check which volumes could be visible for the user (Pietari, 2000). If a given volume would not be visible, no further collision checks would be required for any object within it.

In order to hide triangles that are not visible to the user since they are facing away from the camera, a technique called back-face culling (Kumar et al, 1996) can be used. Back-face culling means, in mathematical terms, that the system does not render triangles with normals facing more than 90 degrees away from the camera. This means that if a person in the game is facing the camera his back is not drawn, as it cannot be seen. The rendering API often does this automatically. While back-face culling generally increases the rendering speed considerably by discarding roughly half of the triangles, many models are designed in a manner that makes back-face culling undesirable. (Pietari, 2000).

Occlusion culling enables the system to skip the rendering of an object's triangles that are completely occluded by another object (Zhang, 1998). For an occlusion culling technique to be considered viable for most implementations, it must also perform in a way that increases the frame rate of the application, and also be general enough to be used in any type of scene. A functional and fast implementation of this algorithm would free up GPU resources considerably.

Unfortunately, effective occlusion culling can be difficult to implement and has several

inherent problems, such as requiring a fairly fast CPU (Sekulic, 2004).

The process of not rendering objects whose contribution of the scene is minimum, such as when their location is far away from the camera, is aptly named contribution culling. When the clipping plane is located too close to the camera the user can recognize objects appearing from nowhere, which is undesirable. This can be hidden by drawing fog on the screen.

The technique known as "level of detail" can be applied to reduce the number of particles in a scene dynamically depending on an object's distance to the viewer. The game contains several versions of each model that has different complexities. Models far from the camera are drawn as the less complex approximations of the given model.

3.3.2 Results

For most of the development process, no culling techniques were planned to be implemented in the application. In the final weeks, the application experienced low frame rates due to the amount of objects that had been gradually added to each level. A view frustum culling algorithm was implemented, using bounding spheres to enclose objects. While both being simple and giving a notable performance boost, it was decided that no space partitioning was necessary since the resulting frame rate was considered acceptable (only below 60 fps at very rare circumstances). Back-face culling was used for obvious reasons. A minimum and maximum camera draw distance was set. Unfortunately, since the game is set in fairly open water, objects crossing the maximum draw distance as the player approaches are rarely covered by geometry. This makes the transition between not rendering and rendering an object to be very apparent to the player. To reduce the visibility of this effect, the maximum drawing distance is set very large, which could lead to performance issues for less powerful machines. Since further optimization was not prioritized, no other culling techniques were applied.

3.4 Graphical Content

Apart from applying render techniques to display the graphics, there is also a need for acquiring graphical content. In the following sections, it will be discussed what techniques can be used to fill the game with various graphical content.

3.4.1 Models

The 3D-models can be acquired in two ways for a project with no budget. Either they can be downloaded at a website providing them royalty free, as for instance The 3D Studio (The 3D Studio 2010), or they can be designed. When creating models, there are many different programs that can be used such as 3Ds Max (3ds Max, 2010), Blender (Blender, 2010) and Maya (Maya, 2010). The only requirement is that the software has to support at least one of the few file formats XNA has native support for.

When importing the model into XNA, it will pass what is called the content pipeline. The XNA Content Pipeline is a way of processing and preparing content so it can be used in the game at run-time. The model is first being imported to the Content DOM where it is saved in a well-known format to the XNA pipeline processor. Then, the model is being processed and an object is created that can be used at run-time (Klucher 2006) (see figure 7).

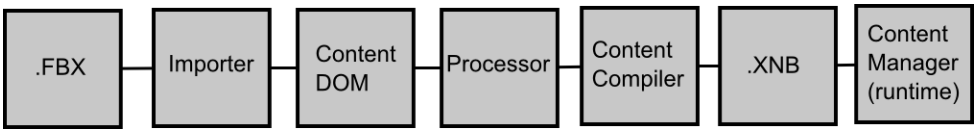


Figure 7: *The model, being the .fbx file, passes the XNA Content Pipeline*

3.4.2 Terrain Generation

One way to generate the terrain in computer games is by generating a mesh based on a two-dimensional heightmap. The heightmap is often a grayscale image where the value of a texel represents the height of that point in the generated terrain. A completely white area would thus be a flat square located at a high position or a low position, depending on the implementation's interpretation.

3.4.3 Texturing the Terrain

There are a few different methods to texture the geometry that is generated from a heightmap. One way of doing this is to just create a large texture and stretch it across the whole terrain (Dexter, 2005). This, however, will cause the need for extremely large texture resolutions if the terrain covers a great area and also should be detailed when viewed from a close range. Other methods use different smaller textures for each ground type and repeat them across the appropriate areas.

There are two basic ways to determine the areas where each ground type is. Either the data is manually created by the user, by setting the ground type of each vertex (Dexter, 2005). This gives the user great flexibility. However, since the user has to create a map of ground textures manually, there is some extra work that has to be done when using this method. Another simpler way to do it is to use the data that is already in the heightmap. The ground types are instead determined by the height of the terrain (Grootjans, 2008). For example, lower parts have a grass texture where higher parts use a mountain texture.

One problem with using different ground types is that the seams between the different textures can be very obvious. The most common way to deal with this is to make the two textures overlap at the seams and blend them together.

3.4.4 Results (Content)

The game's 2D and 3D assets were created for the game, and no pre-made material was downloaded. All 3D-models where created in 3D Studio Max, since this was a tool that was available to use for free. It was also a program the team had previous experience with.

The method for generating terrain used in the game was the terrain generation based on heightmaps mentioned in section 3.5.2. The heightmaps used were initially drawn purely in Photoshop (Adobe Photoshop 2010). Later into the development, the scenery rendering tool Terragen (Planetside 2010) was discovered that made it considerably easier to generate heightmaps and gave convincing and appealing results. Terragen provides a terrain-generating feature, so that the user can have the heightmap ready by simply entering a number of parameters. In the study, this removed some control from the team, but instead enabled them to create the terrain part of levels in just a few minutes. It is also possible to view the heightmaps as real

terrain instead of just in grayscale, and this makes editing a much easier task.

The texturing of the terrain in the game was done in the simplest possible way, which is to determine the different ground types by the height of the terrain. Four different textures were used, each having a specified height and a height interval to cover. The textures were overlapped with each other and blended together at the intersection points.

3.5 Discussion

A huge amount of time was saved using pre-made shaders and particle systems. Although they did not work as well as initially hoped, they were still good enough to add a sense of realism.

When it comes to the creation of content, finding a tool like Terragen earlier in the process would have saved a significant amount of development time. Drawing a heightmap by hand was very time consuming and tedious, compared to using Terragen. This clearly showed that working with the right graphical tools is very important when working within a small time frame.

Shadows are often considered a central part of computer graphics. While considered initially, the team decided to prioritize particle effects and water reflections. Besides being aesthetically pleasing, the rendering of an object's shadow gives the viewer an improved sense of its location in a scene. In our computer game, this information is already provided by reflections in water. Even without these reflections one could argue that such information would not be vital to the gaming experience, since the gameplay is two-dimensional to a large extent.

Beyond technically complex methods of improving the graphical quality of the scene, several simplifications were employed. Simple graphical details were shown to give good results, given the limited time spent on implementing them. As an example, it was decided to draw and use a very detailed skybox that was filled with rainbows, moons and effects. Motivational signs litter the scene and spin to ensure a dynamic screen even when the racer is stationary. Objects floating in the water are also animated accordingly, including the player's boat when its speed is low.

The advice to similar project groups is to make the graphical aspect of the game as good as possible, when the schedule allows it. Game play is what basically makes a game entertaining, and should be the first priority. Nice graphics is important for the feeling of realism and immersion, but cannot substitute game play and content.

4. Network

A common feature that often is expected in modern games is the multiplayer mode. This is something that was desired in the game in order to enable optimal game experience. An American study conducted by Ducheneaut and Moore showed that certain designs of multiplayer modes enhance social activities (Ducheneaut, N., Moore, R.J., 2004). Furthermore, multiplayer is the most popular way of playing games today. As much as 65% of teenagers play games with others in the same room (McEntegart, 2008).

The following sections will cover the different choices available when making a multiplayer game in XNA. Section 4.1 deals with network

architectures, which describe how the computers are connected to each other.

Section 4.2 explains the different topologies one can chose from when designing a network structure, and 4.3 to 4.4 covers the two possibilities of implementing a network mode in XNA respectively: XNA:s built-in network library or Lidgren's network library, respectively.

4.1 Network Architectures

The two basic systems of network architecture are called Peer to Peer- and Client/Server systems. When choosing between the two, a certain amount of aspects have to be taken into consideration, and this amount depends on what purpose the network serves. Some of these aspects include whether or not a dedicated server for the networking can be provided, how much bandwidth there is available for disposal, how security-dependant your application is and how many client computers that should be able to interact through the network.

The major difference between Peer to Peer (P2P) and Client/Server networks is that Peer to Peer networks have no notion of a central server. Each computer in the network interacts with the others directly in a P2P network, while in a Client/Server network it is the server that in the end decides what information that should be communicated between the computers (Maly, R. J. 2003).

4.1.1 Peer to Peer

As stated in 4.1, a P2P network handles the communication between clients (or peers) with a direct connection between the computers. There is no server type to relay the information as in the Client/Server architecture. The amount of data that needs to be sent when employing a P2P network architecture will logically increase when adding peers. However, since every peer shares his resources with the other peers, this becomes manageable in smaller constellations (home- or small business networks) (Maly, R. J. 2003).

The information which flows in a P2P network can be hard to secure because of the open nature of P2P applications (Doucer 2002).

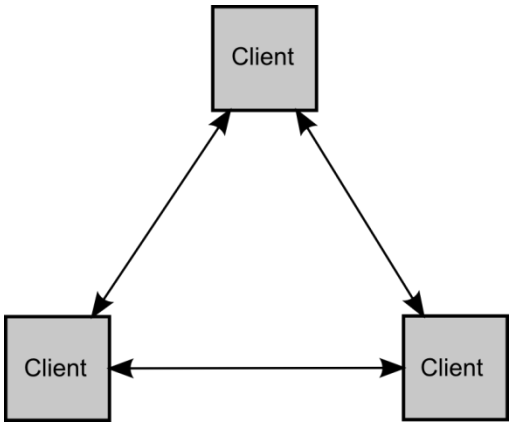


Figure 8: *Simple example of a Peer to Peer architecture. Each Client is connected to all the other clients with a single connection.*

4.1.2 Client/Server

In a Client/Server network, the server plays a central role in the exchange of information between the clients. Its task is to collect the information and the addresses of the clients to whom it is to be delivered, and thereafter relay it to them.

A networking model of the Client/Server type is more likely to get overwhelmed by data as the network grows (Maly, R. J., 2003). The need for expansion and costs that come with it will then increase. For a single server to be able to cope with the pressure of the entire network, it must be of limited size, such as a home network, or a small business network. An application which uses the Client/Server

model needs to be optimized towards sending smaller packets of data to be able to function as well as it can.

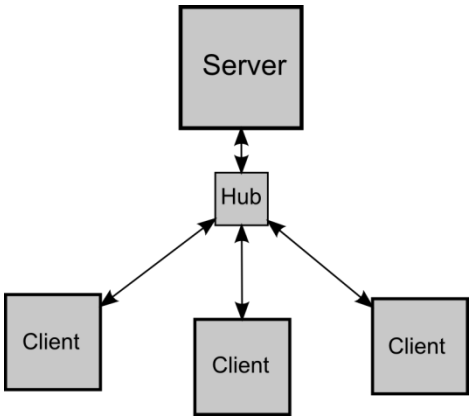


Figure 9: *Simple example of the Client/Server architecture. All the Clients are connected to each other through a Hub, which in turn is connected to the Server. The Server distributes the information through the Hub to the Clients.*

4.2 Network Topologies

There are six basic topologies in networking. For each topology, there are several modifications and variants of these. These all have different applications. These basic topologies are as follows:

- Bus topology - The computers are connected one after another.
- Star topology - All computers are connected to a central server, which forwards the communication between them.
- Ring topology - Each computer is connected to exactly two other computers.
- Tree topology - The network is formed in a tree structure, which means that if each computer is represented as a node, each node has at most one parent node and two child nodes.
- Extended Star topology - As the name implies, this is basically a star topology being less restricted; every computer may act as a server itself for other computers.
- Mesh topology - There are at least two computers with two or more paths between them.

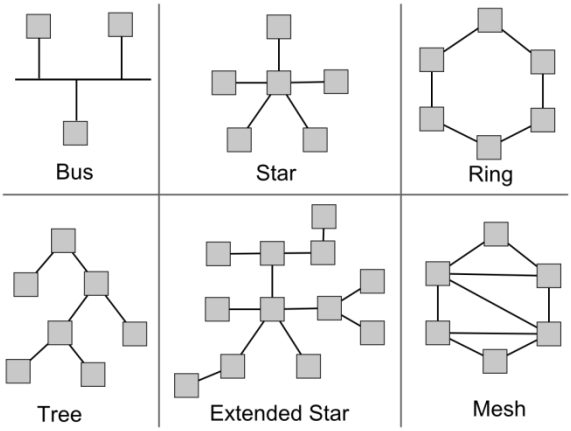


Figure 10: *The six main network topologies. Gray squares represent nodes and black lines represent connections.*

These topologies can be seen as virtual shapes of the network. They all have individual advantages and disadvantages. The bus topology has an obvious disadvantage; if the connection is broken at one point, the whole network is disrupted. The star topology looks as simple as the bus topology. However, it is more redundant. Actually, the star topology is nothing else than a more redundant version of the bus topology and is often used for Local Area Networks (LAN) and resembles the Client/Server network to a large extent. However, the central server has to be stable, since this ties the whole network together. The tree network has a similar weakness to the one of the star network, being that the top computer ties the network together. This is not used as often as the star network though, since two or more computers may be disconnected if any computer disconnects and

thus can often be considered more unstable. The mesh topology is stable and redundant, because of the many connections between the nodes. The main drawback, however, is the number of connections and cables that are required to create such a connection (Atis Telecom, 2007).

4.3 Lidgren Networking

The Lidgren networking library is currently one of the main libraries that can be used when implementing networking in an XNA game. It uses a single UDP socket and delivers an API for sending and receiving messages over a network. It was developed by an amateur as a hobby project, and updated and documented thereafter. Currently, three commercial games (and several non-commercial games/projects) use the library in their network implementations (Lidgren, 2009). Amongst them is a commercial game called Plain Sight, which is a third person shooter/adventure game in which one battles the opponents through the network (Beatnick Games, 2010). The other commercial games are Sacraboar (Makivision Games, 2009) and AI War (Arcen Games, 2010).

The methods of delivering a message with the Lidgren library can be customized to suit the application in the best way. You can decide whether to send a message *reliably/unreliably* and/or *ordered/unordered.* This determines what level of importance the packets of data have. Reliability refers to how often the message actually arrives, and order refers to whether the order of the messages is important for the communication.

4.4 XNA Networking

Using the built-in network framework is the most straightforward approach when implementing networking for a game in XNA. The features and techniques used are similar to the techniques used in the Lidgren library, with the ordering/reliability deliveries described in section 4.3. It has support for Xbox LIVE features and connection between PC and Xbox-machines. Features like in-game invites, cross-platform compatibility between PC, Xbox and Zune are parts of the framework. There is also the ability to use in-game avatars and voice communication over the same protocol as the original network communication.

When implementing a network game with this framework there are limitations, especially when the game is designed to be run between two computers. A Creators Club membership is needed in order to be able to connect with computers outside the local network, and this is a non-free membership. When a connection between clients on a local network is to be made, one can instead use a *Local profile* and create the network session on a *System link* (Klucher, 2007). This is because the framework was originally supposed to be used only for Xbox and Zune, and not for connections between computers. The main reason that connecting between several computers works at all, is that testing had to be done during development of a game for one of the other supported consoles.

4.5 Results

When implementing the network functionality for the game, the Lidgren network library, described in section 4.5, was first tried out. After constructing a few simple examples outside the XNA framework, such as a chat client and a simple send/receive data test, the library initially seemed like a good library to use for game networking. A more extensive test was made, which incorporated the XNA

framework to see how well it would work and how easy it was to implement. This turned out to be harder than initially thought, and the main problem was the lack of both documentation as well as examples on how to do similar implementations. As none of the group members had any previous experience in network programming and strived for a simple solution, it was decided to omit the Lidgren network.

As the first idea of using Lidgren did not work easily enough, the search for other libraries began. In the end it was decided to use XNAs built-in library for networking, mainly because of the better documentation and the good examples. Another reason for using this built-in framework is that it enabled usage of all the features of Windows LIVE gaming, such as player profiles and voice communication.

To allow perfect synchronization between remote players, the physics would have to be calculated on a server that sent info to the clients about the game status. However, in the game a simpler method was chosen. In each frame, all players send their positions, rotations and velocities, and then each machine calculates the physics locally. This could possibly cause different behavior on different machines. However, since the restriction of only using LAN was already present from the start, the issue of slow connections would not be a problem as long as the transmission was wired. The players also send data when they use power-ups. When the game ends for one of the players (the player finishes the race or the player is disqualified), the player sends information about this.

In the menus, the players send information when they change their selections of boats, so the remote player can see which boat his opponent has selected. When the remote player chooses a boat, he tells the host that he is ready. When all players are ready, both players advance to the course select menu. Here, only the player who hosts the game can control the menu. The remote player can see the changes the host makes and when the host is done the game starts.

4.6 Discussion

The Lidgren network showed to be very complex and requires further documentation and examples if it is to be used widely in practice in the future, in this kind of project. However, XNA's built in library was considerably easier to use and eventually resulted in a working multi-player mode. A peer-to-peer network in a mesh topology provided a stable solution and was chosen since the network would go on irrespectively of any computers disconnecting.

One problem with Windows LIVE networking was that users would have to pay a fee to be able to play online. When using System Link (connection over Local Area Network) this is not an issue (Klucher, 2007). However, this makes it more difficult to test the network's functionality, since a local network has to be created every time a test is to be run.

5. Sound

To give a complete game experience, sound is usually implemented in modern games to simulate real audio perception. A study has shown that the choice of sounds in a game is crucial for the game experience. Even one sound can change the game experience dramatically according to Norlinger (Norlinder 2007). Therefore, sound was implemented in the project. The presence of sound in a game

is not only important for the game experience factor. In modern games, since sound is often implemented with three-dimensional techniques, it is also a navigation tool. As an example in the racing genre, motor sounds can be used to give the player a sense of an opponent approaching him from behind.

Two approaches for programming the sounds in XNA will be presented. Both the built-in library for sounds and Microsoft's audio system called XACT can be used. Some of the features of these will covered in section 5.1 and 5.2, respectively.

As well as for implementing the sounds, ways of acquiring effects and music for the game will also be covered in section 5.3. Next the approach of Water Racing is presented in 5.4, and the chapter is concluded by a discussion in 5.5.

5.1 Sound Techniques in XNA

By using XNA's built-in audio library, the programmer manually codes the loading of the sounds. Several possibilities of altering the sounds during run-time exist. There are mainly three variables to consider when doing this. The *volume* variable refers to the volume of the sound. The *pan* variable is used to give an impression of sound being sent from different sources; in other words that the sound moves between the speakers. The *pitch* variable sets the frequency of the sound.

A method called set3D can also be called. By doing this, XNA automatically handles the volume, pan and pitch variables, which vary depending of the position of the listening object and the emitting objects.

5.2 XACT

One way to load the sounds into the game is to manually code sound effect instances, e.g., specifying the file path, as with XNA's built-in library. However, an interface that is called XACT can also be used to handle all the sounds. This is an external audio system that is included in the installation of XNA, and its project files can be imported in XNA projects. With XACT, Cue instances can be created for each sound stored in a sound bank. When having this sound bank, the loading of the sounds can be done just by drag-and-drop in the interface. Also, by creating Cue instances every time a new sound is to be played, one will not experience the problem of different sources playing the same sound effect. Thus XACT provides a very efficient way of implementing game sounds.

Apart from loading and creating sound files, XACT can also be used to put effects on sounds. *Reverb* is such an effect, simulating the acoustic response of a room. As in the XNA sound library, pitch, volume and panning are variables that are possible to modify in order to give an impression of three-dimensional sound.

An effect to take into consideration particularly in racing games is the so-called Doppler Effect. The Doppler Effect is when the frequency of the perceived sound is changed relative to the actual frequency and the relative speeds of the source, observer, and the speed of the waves in the medium (Russell 2009). This is something that can be experienced in reality when a car goes by at a high velocity; the frequency appears to gradually increase while approaching and decrease after bypassing.

5.3 Sound Effects and Music

The fastest way to obtain quality sounds is to download them from a homepage where they are free (assuming that they are being used for non-commercial purposes). A1 Free Sound Effects (A1 Free Sound Effects 2010) and PacDV (PacDV 2010) are examples of two sites which provide free sound effects. These sites contain a large amount of sounds of different categories.

If the sounds are to be used for commercial purposes, royalty free sounds have to be acquired, which can be downloaded or purchased at sites like PartnersInRhyme (PartnersInRhyme 2010). Another option is to record sounds, something which allows more customized sounds to be used. Sound editing programs may be used to add effects to sounds or equalize them. Audacity (Audacity 2010) is an example of a program that may be used for this.

The ways of acquiring music are basically the same as for sound effects; one can either download royalty free music, or compose own songs and record them. The role of music as a motivating factor generally holds for most game genres, according to PhD video game researcher Zach Whalen (Whalen 2004). Therefore, game music can raise the entertainment value of the game.

5.4 Results

The initial plan was to use XNA:s built-in library exclusively. However, this plan was changed during the project, on one hand knowing that a certain amount of code would have to be discarded. On the other hand though, the use of XACT would save much time. So eventually XACT was used almost exclusively for handling sounds.

Not every sound could be automated, though. The motor sounds in the game were created by changing the pitch of just one sound, by an amount depending on the speed of the boat. Also the volume was adjusted, going from low to high, and this amount also depended on the speed.

Since there was not much free music available that fit into the game context, the choice was made to compose own songs for the game. Composing was an easier way of getting music that fit into the game, since the compositions can be made with the scenery in mind. Surf rock music was thought to suit well to a water racing game. The team listened to some old records by The Ventures and Dick Dale and came up with their own interpretations. The program Guitar Pro (Guitar Pro 2010) was used to create the drum and bass parts, and the guitar parts were recorded through a USB interface with the music production tool Tracktion (Tracktion 2010). Only the guitar parts were recorded for real; the other parts were digital emulations. All in all, the music creation was a very low-budget approach that was suitable for the conditions of the project.

5.5 Discussion

XACT was shown to be an efficient way of handling sounds. It makes sense to automate the loading of sounds, instead of hard-coding it. Worth pointing out is that XACT is much more adapted to be used when developing a complex game than XNA's built-in library. With XNA's sound class, sounds cannot be played more than once simultaneously. This means that if two objects will try to play the same sound at once, they have to play separate sound files. This is out of question in larger projects due to unnecessary use of memory space, as files have to be duplicated. The use of Cues, which XACT handle, solves

this problem as instances are created in real time.

The downloading of sounds showed to be an effective way of acquiring sound effects. The sites mentioned in section 5.3 provided a wide range of sounds that could be used for many game situations. The recording of game music should preferably be done with more professional equipment. While the music was regarded as good, one could hear that the recording equipment was not optimal. Thus, game music could preferably be recorded in a studio.

Recording was nevertheless an efficient approach of acquiring game voices, as this could be done with the laptop's built-in microphone. While again not being the optimal recording equipment, the quality of voice recordings showed to be of less importance than the quality of music recordings.

6 Game Physics

The limited amount of resources available in an interactive video game forces the programmer to approximate the physical reality. While technically inaccurate compared to the laws of nature, the behavior of objects can still be implemented to appear realistic. Contrary to what one might think, it is not necessary to make a perfect simulation of the reality in order to get an entertaining simulation. Escaping reality is desirable when playing a video game, so implementations of physics that deviates from reality should not in itself present a problem. However, consistency is an important aspect of game physics (Hecker 2000). The player wants to be able to predict how the physics will affect his actions, after learning how the game works.

This chapter mainly focuses on the detection of collisions as this is the central problem to solve, when implementing game physics. The chapter begins with an overview of different external physics libraries that can be used for game development purposes. In 6.2 the fundamentals of collision detection is examined, followed by methods to enhance collision detection performance by spatial partitioning in 6.3. The results and a discussion of the implemented game physics are found in sections 6.4 and 6.5 respectively.

6.1 Libraries

For simulating physics, there are a large number of tools that can be used to avoid the issue of having to write advanced physics calculations. Tools that have been created to handle physics in a 3D-environment include PhysX (PhysX 2010), Havok (Havok 2010), Bullet (Bullet 2010) and JigLib (JigLib 2007). Ports have been made for the two aforementioned tools to make them compatible to use in C# (BulletX 2007), (JigLibX 2010). These tools could handle all the physics calculations that would be needed; however, the open source tools that were considered, mainly JigLibX, proved to have a lack of full documentation.

6.2 Collision Detection

Collision detection - what happens when two objects touch each other - is a central problem to solve in most games when it comes to making the physics work. By making the decision of developing a racing game on water - a flat surface - with a heightmap for terrain, there were many problems with collision detection in a 3D game that had been avoided, since collision detection against a terrain based on a 3D model would require a more advanced approach.

The usual approach to collision detection is letting each object be represented by geometrical figures. One possible choice is to use spheres (Palmer 2005). A sphere is optimal for fast collision detection - the computer simply compares the sphere's location and the target's location and determines whether the sphere's radius is larger than the distance or not. Objects, unfortunately, tend to have more complex shapes than spheres. When comparing an object with an arbitrary shape, represented by a collection of triangles, to another object, one must perform triangle-to-triangle tests. This naive way of finding collisions forces checks for collision between each triangle in object A to every triangle in object B. This gives a time complexity of N^2, which is unsuitable for the dynamic, high-polygon scenes usually found in games.

6.3 Spatial Partitioning

Using spatial partitioning with the help of a data structure is a good way of improving the speed of the rendering, and is often used in game development. The goal of it is to speed up both the real-time rendering, as well as the intersection- and collision detection.

When implementing spatial partitioning, the geometrical objects are organized according to some data structure, for example a tree structure. A tree search is considerably less complex than a search in a list, or just a random search (Heger 2004). This means more calculations and searches can be made, and a higher graphical standard can be achieved.

There are several options available for lowering the search complexity, when choosing a spatial partitioning technique. One of the most common is the Octree, in which the space is first divided into eight parts, or octants. Thereafter, each of these octants is recursively subdivided into eight smaller octants. The recursion is repeated until a pre-defined maximum depth is reached. Each octant is connected to the parent octant in a tree structure (see Figure 11).

Binary Space Partitioning Trees (BSP Trees) is another common type of spatial partitioning. This technique is mainly found in two different variants, namely *Axis aligned*- and *Polygon aligned* BSP trees. The trees are created by partitioning the world with a plane, and then sorting the geometrical objects in these spaces recursively (Chin 1995).

These techniques are used to cull large portions of the space and geometry from the view. They should mostly be used on static world objects, since computations of this sort in run-time can be very expensive (Lengyel, 2004).

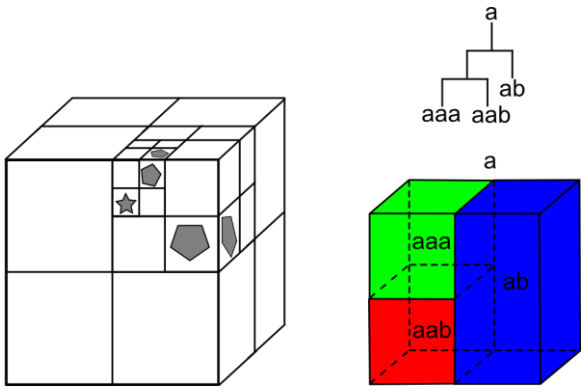


Figure 11: *An Octree and a BSP tree, respectively.*

6.4 Results

Due to the simple 2D-nature of the gameplay, all physics was written without any use of external tools.

When collision checking between boats was carried out in the game, the boats were

represented by spheres, due to the simplicity. The same representation was also used when checking for collisions between rockets, mines and boats. A force was applied on the boat when colliding with rockets or mines, which threw the boat up in the air. The direction of the throw was calculated from the position of the boat in relation to the mine or missile.

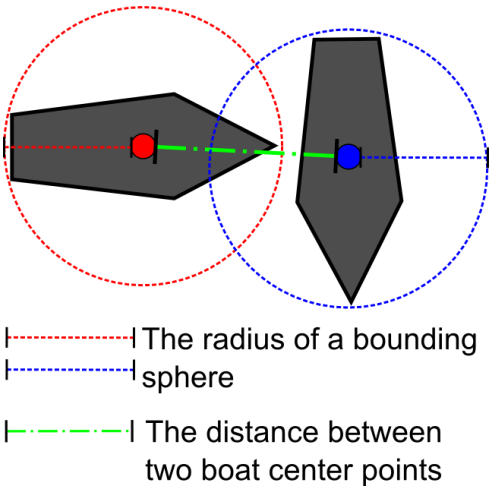


Figure 12: *Collision between boats with bounding spheres.*

As the water was represented by a plane of a specific height, and a heightmap for the terrain provided a height value for each two-dimensional position within the heightmap, checks for collision against the terrain were easy to implement. If the boat is placed at the same or a lower height than the corresponding point on the heightmap, there is a collision against the terrain. A similar check against the water level indicated if the boat touched the water.

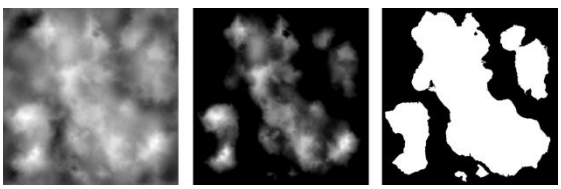


Figure 13: *The original heightmap used in the game (left), the heightmap without areas below the water level (center), and a slice of the heightmap at the water level that shows areas resulting in a collision as white (right).*

A check for collision between a boat and the terrain, with only one point in space representing the boat, would show only if a specific point of the boat hull collided. Therefore, the game used six points that outlined the shape of the boat in order to detect the collisions in a better way.

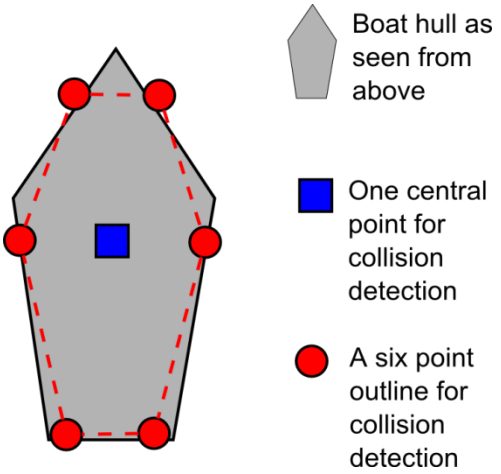


Figure 14: *The six point pattern of point collision against terrain.*

At collisions between the terrain and the boat, the speed was heavily decreased and the direction of moment was changed to the direction of an arrow from the collision point (one of the six possible) and to the center of the boat.

6.5 Discussion

In the study, the techniques used to represent the level made it possible to simplify the simulations of reality considerably.

One of the particular advantages of making a water racing game was that the physics would be easy to handle. There were initial problems with making collision detection work against a 3D-terrain, and external tools would probably be beneficial - tools that would not be

available for free and that would also require additional study time. The team settled for a simple, yet effective, solution by making a race on water - a flat surface - to avoid the problem of collision detection against a 3D-terrain.

7 Artificial Intelligence

To be able to play a game in single player mode and still compete with other racers, some form of Artificial Intelligence (AI) is required in order to provide believable and competent opponents. This might be the most advanced area when developing a game if an optimal, complete and versatile solution is desired. An example of the complexity in the AI field is the game of chess. However, while a chess AI can beat a good human player, this high level of AI intelligence is limited to only this specific domain (Thomas 2004).

For a long time, AI behavior failed to simulate complex behaviors. It was first when the computer game Half Life (Valve 1998) was released that Artificial Intelligence in video games advanced to achieve fairly realistic results.

Since this is an area of high complexity, a relatively simple solution used in the study will be presented in section 7.2. This was partially inspired by the techniques described in section 7.1.

7.1 Relevant Forms of AI

There are many fields that could be classified as AI. As each game may be unique in the aspect of which AI it requires, it is important to analyze what kind of algorithm that a specific game needs (Tozour 2002).

In a typical car race, the drivers attempt to reach the goal as fast as possible. They strive for finding the shortest path around the course based on the best positioning on the road. Relevant forms of AI would therefore need to simulate this behavior.

7.1.1 A* Algorithm

The purpose of the A* algorithm is to find the shortest path between the nodes in a graph. The algorithm is versatile and can be used for many different types of games (Matthews 2002). A* was described in the year 1968 by Hart, Nilsson and Raphael (Hart, Nilsson, Raphael 1968).

To use the A* in a game, a level needs to be defined as a graph using nodes connected by paths. Nodes could be divided into three categories, nodes already visited, nodes that may be visited and nodes that have not yet been found. As the algorithm iterates, nodes may change state in two ways: from *not found* to *may be visited*, and from *may be visited* to *already visited* (Stout 2000).

A function to estimate a cost value for each node is needed. This cost should reflect the length to reach this node and estimate the remaining length to reach the goal (Stout 2000).

A* keeps track of the current node while iterating. This is initially set to the start node. A simplified iteration goes as follows:

1. Each node that is connected directly with the current node changes state to indicate that it may be visited, unless it has been visited before.
2. A new current node is set. This should be the node with the lowest cost of those nodes that may be visited.

The algorithm iterates using this pattern until the goal node is set as current node, or when there is no node left that may be visited (Stout 2000).

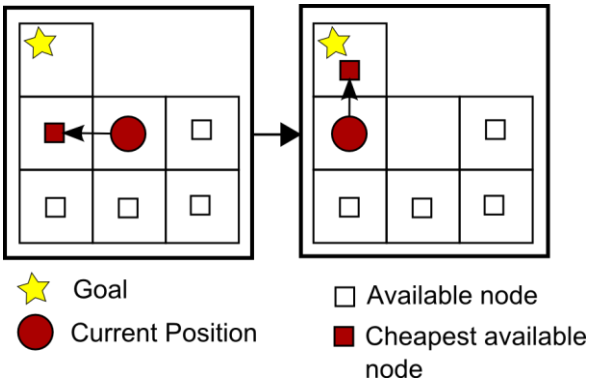


Figure 15: An available node in the illustration is the same as a node that may be visited.

To improve the performance while an application is running, a less detailed version of the level graph can be used to rapidly calculate an initial path. This path could be followed while a more precise path based on the full graph is calculated in the background. This enables an instant response which may be required in some games (Higgins 2002, Path finding Design Architecture).

If A* is to be used for path finding in a real-time game, the developers should know that they may need to spend time to optimize the algorithm to the specific conditions of the game (Higgins 2002, How to Achieve Lightning-Fast A*). This may be done by simplifying the path finding problem itself, or by excluding cases where A* might be replaced by less requiring methods (Cain 2002).

The A* algorithm could be used for other tasks than path finding. An example would be as a flooding algorithm, and this would be done by setting an unreachable goal (Higgins 2002, Generic A* Path finding).

7.1.2 Precomputed Paths

While the A* algorithm could be used to compute the shortest path each time a path is going to be used, the shortest path could also be precomputed and stored in a table. This approach requires that the shortest path is available for each valid position. It requires a larger amount of memory, but the benefit lies in the less amount of load on the CPU being used when running the game (Sterren 2004).

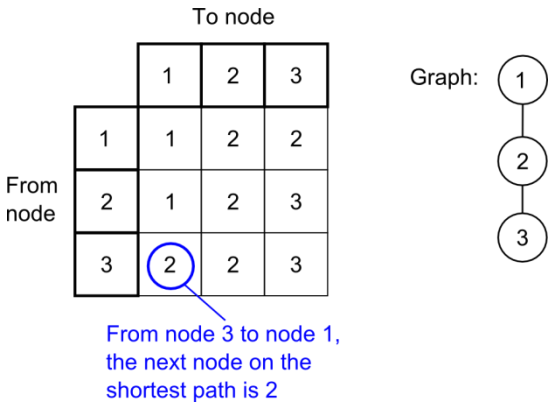


Figure 16: Finding the next step on the shortest path within the graph is just a simple table look up.

In a single large graph, the number of elements in the table would increase quadratically, based on the number of nodes. Dividing a larger graph into several smaller sub graphs that each has their own tables could reduce this effect (Dickheiser 2004).

7.1.3 Driving Lines

Another concept that is more specific to racing games is the use of driving lines. Like precomputed paths, this concept is based on the fact that the best path is already known. Unlike precomputed paths, where many paths are based on the variables current position and static goal, driving lines define only a few paths based on the static race track.

The AI for a racing game could use driving lines as a method to navigate through the

game. In a car game, a segment of the road could be described as a left edge of the road and a right edge of the road. Between these limiting edges is the road area that a car may drive on. A line could be drawn on this area between the start and the end of the road segment to define a driving line. This is a line upon which the AI should aim to drive. For a whole continuous race track, the driving line could be represented by a series of nodes combined with edges (Biasillo 2002, Representing a Racetrack for the AI).

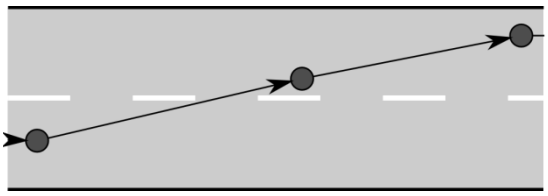


Figure 17: *The drive line should provide the AI with an ideal path to follow.*

It is not necessary to develop special tools or to create the drive lines by hand. A relatively simple way of acquiring a drive line is by recording the path taken by a humanly controlled vehicle. This path could then be used as a drive line (Biasillo 2002, Training an AI to Race).

An alternative way of creating driving lines is by generating them from the track structure. This could be useful in games were the drive line cannot be predicted such as with randomly generated maps. One such method is to place the edges, that combined represents the drive line, along the center line of the road. Then, these points would be moved step by step in such a direction that the angle between any two edges is minimized while keeping the points on the road (Manslow 2004).

To make behavior appear as more realistic, the AI could set an aim at the road further ahead. This makes it easier to predict situations (Biasillo 2002, Racing AI Logic). Another improvement is to have several alternative drive lines, such as one for entering the pit lane for a pit stop (Biasillo, 2002, Representing a Racetrack for the AI).

7.1.4 AI Behavior

A driver might need to take other things into consideration besides the optimal path through the track. This may for example be avoiding collisions with other drivers, making a pit stop or handling specific situations, such as when the wheels lose grip of the road.

One approach is to use a state machine for the AI. Several different states when the AI would act according to a specific situation are then defined, and the current state should be updated as situations change (Biasillo 2002, Racing AI Logic).

The commercial racing game Downforce used an approach based on layers when dealing with the different factors. The task of driving was divided into several sub-tasks. The best way to drive was calculated according to each sub-task and the result of these calculations where prioritized and combined into a final driving choice (Darby 2004).

7.2 Results

The game developed in the project used driving lines in order to navigate. This made it possible to easily implement a recording feature that recorded a human player racing a lap. No further logic besides trying to follow this path was implemented, as this alone worked well in the game.

Each AI controlled boat had an individual driving line. Each of these lines was defined as an ordered list of points within the game.

Each point had only two-dimensional coordinates. Since the water level was a flat plane, the height was not needed as a coordinate. By connecting the last point with the first, this list of points can be seen as a directed graph where each node is connected with edges to two other nodes.

The AI was limited to the same input options as a human player. On one hand being a limitation, it also made the AI behave more natural.

In order for the AI to calculate a desired direction, the most recently passed node was being saved. Both the distance to this node and the distance to the next node were calculated. The distance from the last passed node was divided by the sum of the two distances. This result can be considered a percentage of how far the boat has reached on the ideal edge between the most recently passed node and the next node to reach, and can be used to acquire a point on the edge between the two nodes. This point indicated where the AI controlled boat would ideally be. An offset was then added, since the AI would aim beyond the ideal point.

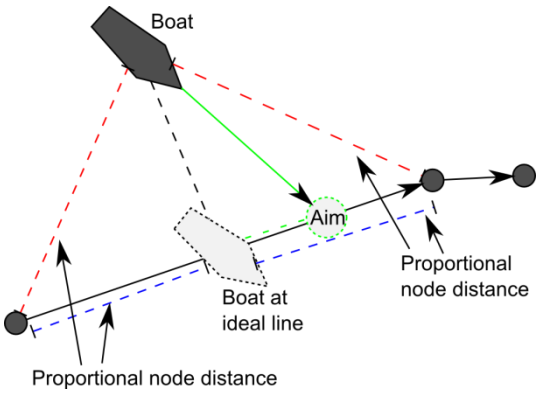


Figure 18: The point for the AI to aim for is calculated as a short distance ahead on the driving line.

This system needed to check when the next node had been passed. This was done by calculating the distance between the most recently passed node and the next node, which then was compared to the distance between the most recently passed node and the boat. If the boat had reached further away than the next node, the boat was considered to have passed it.

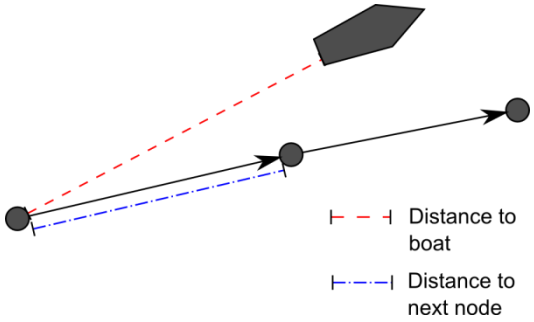


Figure 19: The boat is considered to have passed a new node if the distance between the last node and the boat is larger than the distance between the last node and the next node.

7.3 Discussion

The use of driving lines worked well for the game. While an A* approach based on reaching the next checkpoint would also be possible, it may also have required more development time if optimizations would have to be made. Precomputed paths would have needed optimizations as to minimize the size of the data representation of the game world.

By recording driving lines from human players, the creation of new driving lines was a fast process. While the human player might not have taken an optimal path, this would also add a human factor to the paths taken. As the AI players would use the same path every time the game was played, a human player would be able to predict how the AI

would drive. Due to the fact that collisions and explosions put the AI off track, this effect might be less noticeable.

Further functionality could be added to the AI such as intelligent use of power ups. However, this has had a low priority compared to other areas of the game.

8 User Interface

A complete gaming experience, according to today's standards, includes ways of actually starting the game, defining game settings, choosing to play in a specific mode, quitting the game and so forth. During the game, there are menus present showing the actual state of the game, for example the heads up display showing the player's current position.

A user interface is the interface between the user and the machine. Its purpose is to help the user in his interaction with the system, and to make the interaction occur on the user's terms rather than the machine's terms. The user interface is responsible for all input from the user, as well as the feedback that is sent to the user about the performance.

The science of user interaction consists in its most basic form of user input and system response. In the following sections, an additional split has been performed in order to present the subject more clearly (Dix 2004).

8.1 Input

When designing the input logic for a computer game, it is important to see it from the user's perspective. A far too complex system will result in the user not feeling comfortable with it, and will in the end affect the gaming experience. It is therefore important to construct an input system which has a learning curve that is as easy as possible, and a recognition factor that is as high as possible. A good approach to this is to implement simple controls that are natural to the average user, since they will relate to similar products the user has used before (Federoff 2002).

One way to achieve a high degree of immersion is the use of pseudo real input control components. In a racing game, this can be done with the use of steering wheels and pedals, or with a joystick in a flying game. In order to create a feeling of immersion for the user, controls with haptic feedback, such as Force Feedback, can be used (Edwards; Barfield; Nussbaum 2004).

8.2 Graphical feedback

The graphical part of the feedback to the user is often the most emphasized and deterministic part of the UI. This is due to the fact that it will deliver the majority of the information to the user in most systems.

In order to enhance the user's ability to navigate in the game world, information about the player's location is often good to have available. A way of achieving this is to use an in-game mini-map or to have such information reachable through a menu or a separate screen.

Having the mini-map constantly rendered on the screen, ensures that the user receives information without having to give any input to the system. A drawback of this method is that the mini-map, for some users or in some situations, remains unused most of the time. According to Gregory Wilson, it can also interrupt the player's immersion in the game (Wilson 2006).

A mini-map accessed by a menu avoids the problem of taking up precious screen space, but can be tedious to access if often needed. This could result in lower usability due to lower efficiency.

8.3 Sound feedback

Sound feedback can also be used to greatly enhance the feeling of immersion (Edwards; Barfield; Nussbaum 2004). This is achieved by letting the user receive relevant information about the game and the user's performance in it by sound effects and/or a speaker voice. Studies have shown that sound feedback about performance increases both the player's feeling of immersion and the level of efficiency of the interaction with the game (Edwards; Barfield; Nussbaum 2004).

8.4 Results

Sound feedback was used extensively in the game to inform the player about how he performed. For example, if a player missed a gate, a sympathetic voice exclaimed the player's failure.

A mini-map was also implemented in order to ensure that the player would not get lost or fail to follow the race course. This feature also enabled the player to receive information about the position of the opponents and how well the user performed in relation to them.

The first version of the mini-map was a fairly crude construction which basically rendered a smaller version of the heightmap, with different colors representing geometry above and below the sea level respectively. In a later version of the project, large sections of the map consisted of water. Thus, the mini-map being rendered was filled with much information that was of little use. The results of the hand drawn map looked more impressive, and also provided more useful information. This final version of the mini-map showed the map's intended race path, as well as race checkpoints and the opposing boats.

Visual feedback about the user's performance in the race was present as well. In the mode where the player raced against the AI, the player's position relative to his opponents was shown. In the time trial mode, the time was visible.

Initially, support of the rebinding of keys was implemented. The purpose of this was to allow the user to set his preferred keys before playing, tailoring the gaming experience for individual users. Later on, a decision was made to make the controls simple enough that rebinding was considered unnecessary.

8.5 Discussion

The decision of minimizing the number of keys used for controlling the game gave the game a simple control system. One could argue that this lowered the challenge of the game by being too simple. However, in relation to the low amount of content that was present in the game, this did not present any major problems. Furthermore, a new user could quickly master the controls of the game with the simplified control scheme and focus on other aspects of the game, rather than the control system.

The possibility of implementing pseudo real game controls was considered, which could have used Force Feedback. The original plan was to implement an interface between the controls and the game core software to handle the extra logic involved. However, due to the change of game concept from car racing to boat racing, normal feedback systems such as steering wheels, pedals and gearboxes did not seem appropriate anymore.

The extensive use of sound feedback was considered by the development team to be a natural way of delivering information, as it reduced the need of explaining game rules to the player before the race began. However, it could have been used even more in order to minimize the need for visual information on the screen. In this way it would also contribute even more to player immersion. Sound feedback was not prioritized high enough, though.

9 Results of implementation

One result of the study was a fully playable racing game with a single player mode for playing against a computer opponent, as well as a multiplayer mode against another human opponent. This chapter will describe the game on the whole, to proceed with a discussion about these results in chapter 10.

When starting the game, the menu shows up. The user navigates the application's menu scenes to reach his preferred game mode. When choosing either "Time Trial Mode", "Race Against AI" or "Multiplayer" the user will come to the next scene. This scene prompts the user to select his vehicle and map, and it works very similar in the different game modes. The player or players selects their boats by using the arrow keys and then accepts their current alternative to start the game.

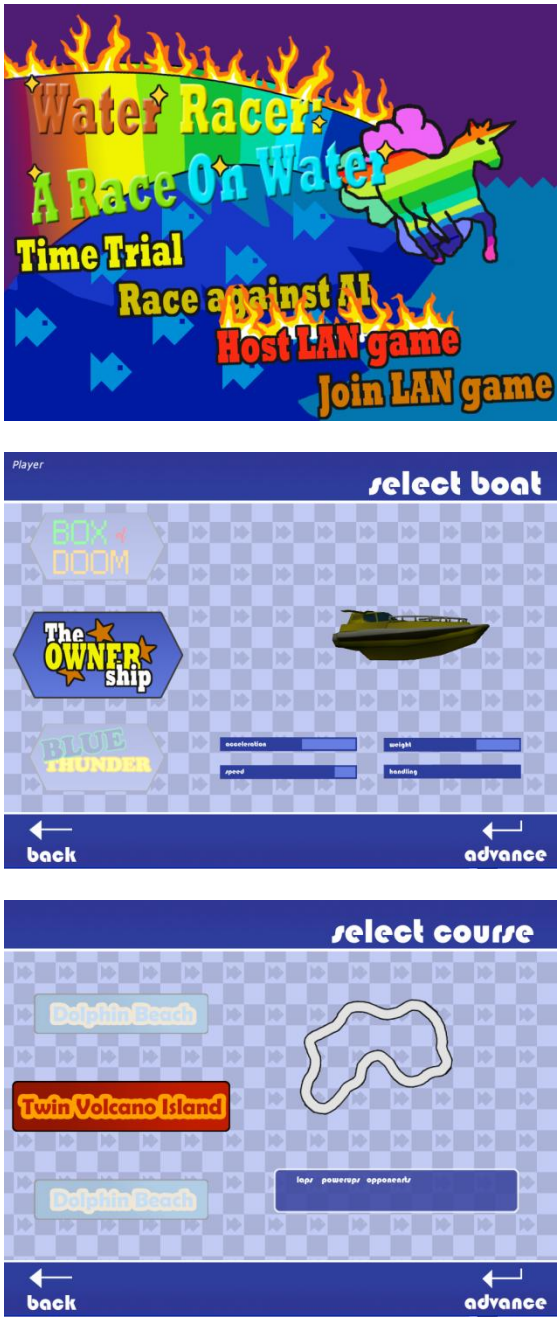


Figure 20: *Various stages in the game menu.*

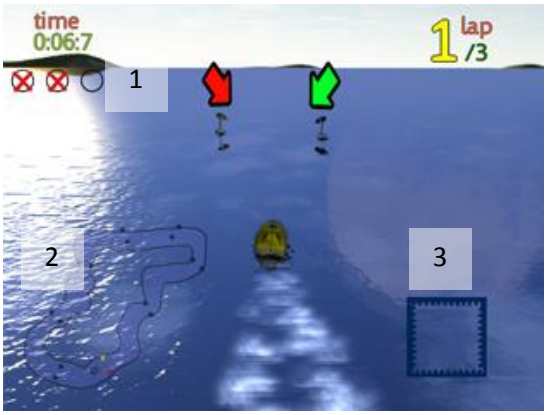


Figure 21: *The final result of Water Racing has a graphical user interface contains a counter tracking number of missed gates (1), a mini map (2) and a power up container (3) as well as a few additional gadgets.*

In the game, the user is prompted to race through the map by passing gates. The first boat that crosses the finish line wins the race, and depending on the type of race two things may happen. Either a local highscore is presented with the player's score highlighted, or a list of the various racers is shown, placing the winner on top. If a boat fails to pass a certain number of gates, he will be disqualified and has no longer any chance of winning. These gates were implemented to disable the player from taking shortcuts. The presence of the opponent is different depending on what game mode that was chosen. In "Time Trial", the user has no opponent. In "Race against AI", the opposing boat is steered by the computer. In the "Multiplayer" mode, there is another human that controls the opposing boat, playing from another computer.

The race occurs between islands which are filled with objects such as trees and buildings to make the game look more interesting. Floating mines are placed in the water as obstacles, and thus enables more complex game play to make it more interesting to play the game.

Power-ups are spread throughout the level. By picking one of these up, the player is granted one of several possible powers, which then can be used to gain an advantage in the race. The power at disposal is displayed by a stylized icon in the user interface. Power-ups can be of either defensive or offensive nature, or sometimes both depending on how they are applied by the user. The set of power-ups includes various kinds of rockets and mines, as well as shields and abilities that alter the way the boat moves. These are jumping abilities and abilities that heighten the velocity.

There is no way for a boat to be permanently destroyed or damaged. An explosion simply propels it in a new direction. If it crashes upon land, it is replaced in the position where it was before the explosion.

10 Discussion

The overall belief of the team was that the game had good potential of being regarded as an entertaining racing game, according to modern standards. However, one of the main drawbacks of Water Racing was that only one complete level was implemented. Level design was something that was more demanding than initially expected. While being a fully playable game, the risk would be that the game becomes monotonous when playing the same map in every game.

One aspect that the team had in focus when striving for an entertaining game, was versatility. The networking- or music parts, for instance, could have been left out when they initially presented unexpected challenges, but the goal of wanting a result that was complete in all its aspects made the team continue work on these areas. Therefore, the overall belief of the team was that when developing a game

rapidly, it should not be done with the means of omitting parts of it. Instead solutions should be found for each area that allows a simple solution.

In this project, there are several tools that we acknowledge for having contributed considerably to the resulting game. The pre-made shaders used for the water saved much time that otherwise would have been spent on the graphical parts. XNA is a framework shown to be useful to develop a game of this kind, as much work is pre-made compared with just using C#.

Though XNA is a tool with potential, it is not without disadvantages. First and foremost XNA LIVE is not adapted for commercial multiplayer use. If the users have already paid for a game, it is not likely they will want to pay for playing online unless it becomes somewhat of a commercial success.

11 Conclusions

There are interesting techniques available at the time of writing this report that enables rapid development of games. Agile development is a process that makes it possible to focus on programming to a greater extent, rather than formalities. There are royalty free tools available for download to enable good looking visuals, and royalty free sounds and music can be acquired from a number of sites. By making an appropriate choice of the game concept one can make a relatively impressive game that avoids many physical and graphical challenges, in case these would be predicted to cause problems in the development phase. In this study the concept of Water Racing has resulted in a game with functional graphics, gameplay, a multiplayer mode, computer controlled opponents, simple physics, suitable game audio and working menus.

Something interesting for further studies could be to show a way of rapidly developing a multi-level game. This would then not only be considered a complete game, but also have entertainment value enough to be compared with commercial games. Also, comparing different development tools could be of interest. The team was more or less bound to the programming languages they had knowledge of. However, there are many languages available that could be used in game development.

12. References

3ds Max (2010) http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13567410 (2010-05-13).

A1FreeSoundEffects (2010) http://www.a1freesoundeffects.com/ (2010-05-04).

Adobe Photoshop (2010) http://www.adobe.com/products/photoshop/family/?promoid=DIODG (2010-04-10).

AgileCollab (2008) *Iterative and Incremental is not equal to Agile: Key Aspects of Agile*. http://www.agilecollab.com/iterative-and-incremental-is-not-equal-to-agile-key-aspects-of-agile (2010-05-05).

Arcen Games (2010) *AI War.* http://www.arcengames.com/ (2010-05-12).

Atis Telecom Glossary (2007) http://www.atis.org/glossary/definition.aspx?id=3516 (2010-05-04).

Audacity (2010) http://audacity.sourceforge.net/?lang=sv (2010-05-03).

Beatnick Games (2010) *Plain Sight.* Published by: Steam. http://www.plainsightgame.com/ (2010-05-12).

Biasillo, G (2002) Racing AI Logic. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 444-454. Hingham: CHARLES RIVER MEDIA.

Biasillo, G (2002) Representing a Racetrack for the AI. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 439-443. Hingham: CHARLES RIVER MEDIA.

Biasillo, G (2002) Training an AI to Race. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 455-459. Hingham: CHARLES RIVER MEDIA.

Blender (2010) http://www.blender.org/ (2010-05-13).

Boehm, B. (1988) A Spiral Model of Software Development and Enhancement, *Computer*, vol. 21, no. 5, pp. 61-72.

Brightman, J. (2010) *NPD: Video Game and PC Game Industry Totals $20.2 Billion in '09.*

Bullet (2010) http://bulletphysics.org/wordpress/ (2010-05-05).

BulletX (2007) http://xnadevru.codeplex.com/wikipage?title=Managed%20Bullet%20Physics%20Library&ProjectName=xnadevru (2010-05-05).

Cain, T (2002) How to Achieve Lightning-Fast A*. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 146-152. Hingham: CHARLES RIVER MEDIA.

Chin, N. (1995). A Walk Through BSP Trees. In *Graphics Gems V*, ed. A. Paeth., pp. 121-138. London: Academic Press.

Darby, A (2004) Racing Vehicle Control Using Insect Intelligence. In *AI Game Programming Wisdom* 2, ed. S. Rabin, pp. 469-484. Hingham: CHARLES RIVER MEDIA.

Dexter, J. (2005) Texturing Heightmaps. http://www.gamedev.net/reference/programming/features/texturingheightmaps/page2.asp. (2010-05-04).

Dickheiser, M (2004) Inexpensive Precomputed Pathfinding Using a Navigation Set Hierarchy. In *AI Game Programming Wisdom* 2, ed. S. Rabin, pp. 103-113. Hingham: CHARLES RIVER MEDIA.

Dix, A. et al. (2004) *Human-Computer Interaction.* Third edition. Harlow: Pearson Education Limited.

Doucer, J. R. (2002) *The Sybil attack,* First International Workshop on Peer-to-Peer Systems. http://www.iptps.org/papers-2002/101.pdf (2010-05-04).

Ducheneaut, N.; Moore, R. J. (2004) *The social side of gaming: a study of interaction patterns in a massively multiplayer online game*. *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*. November 6th-10th 2004, Chicago.

Edwards, G.W., Barfield, W., Nussbaum, M.A. (2004). The use of force feedback and auditory cues for performance of an assembly task in an immersive virtual environmen. *Virtual Reality,* vol. 7:2 2004.

Extreme Programming (2009) http://www.extremeprogramming.org/ (2010-03-17).

Federoff, M. (2002) *Heuristics and usability guidelines for the creation and evaluation of fun in video games.* Indiana: Indiana University. (Master of Science in the Department of Telecommunications). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.8294&rep=rep1&type=pdf (2010-05-13).

Gomaa, H., Kerschberg, L. (1995) *Domain Modeling of the Spiral Process Model.* http://mason.gmu.edu/~kersch/KBSE_folder/ESPM_folder/ESPM_DM.html (2010-03-16).

Grootjans, R. (2008) *XNA tutorial using* C# *and HLSL series 4 - Overview.* http://www.riemers.net/eng/Tutorials/XNA/Csharp/series4.php (2010-04-03).

Guitar Pro (2010) http://www.guitar-pro.com/en/index.php (2010-04-14).

Hart, P.E, Nilsson, N.J and Raphael, B. (1968) A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp.100-107. DOI: 10.1109/TSSC.1968.300136

Havok (2010) http://www.havok.com/ (2010-05-05).

Hecker, C. (2000) *Physics in computer games.* Communications of the ACM. Volume 43, Issue 7 (July 2000), pp: 34 - 39.

Higgins, D (2002) Generic A* Pathfinding. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 114-121. Hingham: CHARLES RIVER MEDIA.

Higgins, D (2002) How to Achieve Lightning-Fast A*. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 133-145. Hingham: CHARLES RIVER MEDIA.

Higgins, D (2002) Pathfinding Design Architecture. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 122-132. Hingham: CHARLES RIVER MEDIA.

http://www.industrygamers.com/news/npd-video-game-and-pc-game-industry-totals-202-billion-in-09/ (2010-04-03).

Hung, T. (2007) *Software development process*. http://cnx.org/content/m14619/latest/ (2010-03-19).

Irrational Games (2007). *BioShock*. Published by: 2K Games.

Jeffries, J. (2009) *Absence of Errors.* http://xprogramming.com/articles/qa/xp_q_and_a_absence/ (2010-03-18).

JigLib (2007) http://www.rowlhouse.co.uk/jiglib/index.html (2010-05-05).

JigLibX (2010) http://jiglibx.codeplex.com/ (2010-05-05).

Klein, F. (2008) *The Waterfall Model of Software Development*. http://www.relativitycorp.com/projectmanagement/article10.html (2010-05-10).

Klucher, M. (2006) *XNA Game Studio Team Blog.* http://blogs.msdn.com/xna/archive/2006/08/29/730168.aspx (2010-04-09).

Klucher, M. (2007) *XNA Framework Networking and LIVE Requirements.* http://blogs.msdn.com/xna/archive/2007/11/16/xna-framework-networking-and-live-requirements.aspx (2010-05-04).

Kumar, S.; Manocha, D.; Garrett, B.; Lin, M. (1996) Hierarchical Back-face Culling. In *7th Eurographics Workshop on Rendering*, pp. 231-240.

Lander, J. (1998) The Ocean Spray in Your Face. *Game Developer. Game Developer,* July 1998. http://double.co.nz/dust/col0798.pdf (2009-05-03).

Larman, C. (2003) *Agile/iterative methods: From business case to successful implementation*. Addison Wesley.

Lengyel, E. (2004) *Mathematics for 3D Game Programming and Computer Graphics.* Second edition. Massachusetts: Charles River Media Inc..

Lidgren, M. (2009) *Featured Projects*. http://code.google.com/p/lidgren-network/wiki/FeaturedProjects (2010-05-04).

Lidgren, M. (2009) *Networking Library*. http://code.google.com/p/lidgren-network/ (2010-05-04).

Lombard, Y. (2004) *Realistic Natural Effect Rendering: Water I.* http://www.gamedev.net/reference/articles/article2138.asp (2010-04-07).

Makivision Games (2009) *Sacraboar.* Published by: Steam. http://www.sacraboar.com/ (2010-05-12).

Maly, R. (2003) *Comparison of Centralized (Client-Server) and Decentralized (Peer-to-Peer) Networking.* ETH Zurich. ftp://ftp.tik.ee.ethz.ch/pub/students/2002-2003-Wi/SA-2003-16.pdf (2010-05-04).

Manslow, J (2004) Fast and Efficient Approximation of Racing Lines. In *AI Game Programming Wisdom* 2, ed. S. Rabin, pp. 485-488. Hingham: CHARLES RIVER MEDIA.

Matthews, J (2002) Basic A* Pathfinding Made Simple. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 105-113. Hingham: CHARLES RIVER MEDIA.

Maya (2010) http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13577897 (2010-05-13).

McEntegart, J. (2008) *Yet Another Study Finds Videogames a Social Experience*. http://www.tomsguide.com/us/Video-Games-Children-Pew,news-2607.html (2010-05-04).

MindTools (2010) http://www.mindtools.com/pages/article/newPPM_03.htm (2010-03-17)

Norlinder, M. (2007) *Lair of Beowulf.* http://www.gamessound.com/texts/3dPosi%20Audio%20in%20game.pdf (2010-05-04).

NST (2001). *Wave Race: Blue Storm*. Published by: Nintendo.

nVidia (1999) *Cube Map OpenGL Tutorial.* http://developer.nvidia.com/object/cube_map_ogl_tutorial.html (2010-04-07).

nVidia (2010) *PhysX*. http://www.nvidia.com/object/physx_new.html (2010-05-05).

OpenGL (2001) *Planar Reflections and Refractions using the Stencil Buffer.* http://www.bluevoid.com/opengl/sig00/advanced00/notes/node165.html (2010-04-07).

PacDV (2010) http://www.pacdv.com/sounds/ (2010-05-04).

PartnersInRhyme (2010) http://www.partnersinrhyme.com/ (2010-05-03).

Physics Engine (2008) *Second Life*. http://wiki.secondlife.com/wiki/Physics_engine (2010-05-05).

Plamer, G. (2005) *Physics for Game Programmers*. New York: Springer-Verlag.

Planetside (2010) http://www.planetside.co.uk/ (2010-04-11).

Russell, D. (2009) *The Doppler Effect and Sonic Booms.* http://paws.kettering.edu/~drussell/Demos/doppler/doppler.html (2010-05-04).

Scacchi, W. (2001) *Process Models in Software Engineering*. In *Encyclopedia of Software Engineering*. 2nd Edition, by Marciniak, J.J. New York: John Wiley and Sons, Inc..

Schell, J. (2008) The Game Improves Through Iteration. In *The Art of Game Design,* pp. 79-95. . Burlington: Elsevier.

Schreiner, T (2003) *Artificial Intelligence in Game Design.* http://ai-depot.com/GameAI/Design.html (2010-03-22).

Sekulic, D. (2004) *Efficient Occlusion Culling*. http://http.developer.nvidia.com/GPUGems/gpugems_ch29.html (2010-04-07).

Simpson, J. (2010) *How Important Are Graphics to Games?*. http://www.gamespot.com/features/2693475/p-2.html (2010-03-25).

Soft Particles (2009) http://www.gamerendering.com/2009/09/16/soft-particles (2009-05-03).

Sommerville, I. (2010) *Software Engineering.* 10th edition. Harlow: Addison Wesley.

Stam J. (2003) *Real-Time Fluid Dynamics for Games.* http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf (2010-04-07).

Sterren, W (2004) Path Look-Up Tables-Small Is Beautiful. In *AI Game Programming Wisdom* 2, ed. S. Rabin, pp. 115-129. Hingham: CHARLES RIVER MEDIA.

Stout, B (2000) The Basics of A* for Path Planning. In *Game Programmering Gems*, ed. M. DeLoura, pp. 254-263. Hingham: CHARLES RIVER MEDIA.

The 3D Studio (2010) http://www.the3dstudio.com/ (2010-04-07).

Thomas, D (2004) New Paradigms in Artificial Intelligence. In *AI Game Programming Wisdom* 2, ed. S. Rabin, pp. 29-39. Hingham: CHARLES RIVER MEDIA.

Toman, W. (2009) *Rendering Water as a Post-process Effect.* ttp://www.gamedev.net/reference/programming/features/ppWaterRender/ (2010-04-07).

Tozour, P (2002) The Evolution of Game AI. In *AI Game Programming Wisdom*, ed. S. Rabin, pp. 3-15. Hingham: CHARLES RIVER MEDIA.

Tracktion (2010) http://www.mackie.com/products/tracktion3/ (2010-04-12).

Ubisoft Romania (2005). *Silent Hunter III*. Published by: Ubisoft.

Valve Software (2004). *Half-Life 2*. Published by: Sierra Entertainment.

Westin, S. (2007) *Fresnel Reflectance.* http://www.graphics.cornell.edu/~westin/misc/fresnel.html (2010-04-07).

Whalen, Z (2004) *Play Along - An Approach to Video Game Music.* http://www.gamestudies.org/0401/whalen/ (2010-05-04).

Wilson, G. (2006) Off With Their HUDs!: Rethinking the Heads-Up Display in Console Game Design. *Gamasutra.* http://www.gamasutra.com/view/feature/2538/off_with_their_huds_rethinking_.php?page=2 (2010-05-13).

VTT Electronics (2002) *Agile software development methods.* http://www.pss-europe.com/P478.pdf (2007-03-20).

XNA (2007) http://www.xna.com/ (2010-03-21).

XNA Creators Club Online (2010) - http://creators.xna.com/en-us/sample/particle3d (2010-03-27).

XNA Game Studio 3.1 - Gamer Services (2009) http://msdn.microsoft.com/en-us/library/dd254747.aspx (2010-05-04).

Zhang, H. (1998) - *Effective Occlusion Culling for the Interactive Display of Arbitrary Models.* Chapel Hill: University of North Carolina (Doctoral Thesis).