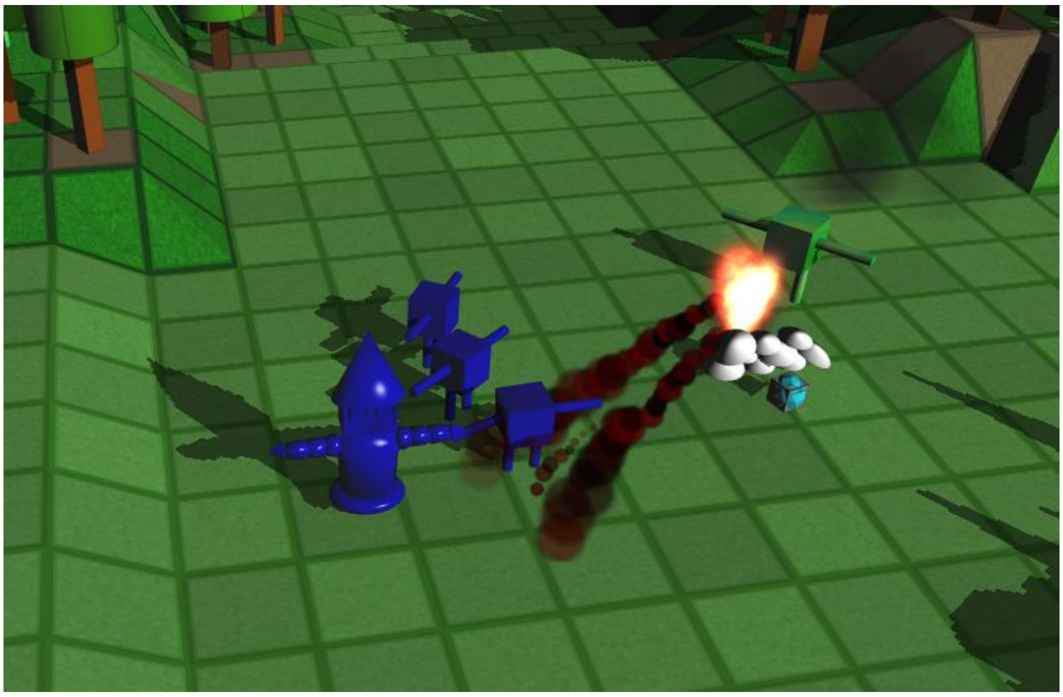


Developing a Real-Time Strategy game intended for the Xbox 360 gamepad

Bachelor´s Thesis

Computer Science and Engineering Programme

Joakim Lind Hasselskog Mattias Majetic

Pouya Mobarrez Naghsh Jacob Rippe

Jonas Wickerström

Institution for Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2011
Kandidatarbete/rapport nr 2011:28

Abstract

This report describes the details of developing a real-time strategy (RTS) game controlled with the Xbox 360 console gamepad. The main purpose for developing the game was to investigate the possibility of creating a strategy game playable on a console. In addition, the game will hopefully be released commercially after this thesis project. The report also describes aspects of game development such as graphics, artificial intelligence, and details on using Scrum for game development. The game was developed over the course of one semester using the Microsoft XNA Game Studio framework and the C♯ programming language. Microsoft XNA Game Studio is a tool for developing games for the Xbox 360, Windows, and Windows Phone. While the game was not finished, the process illustrated many challenges in developing a game, especially managing the art, game design, and technology aspects. Unfortunately, because of time constraints the control scheme could not be designed and tested properly.

Sammandrag

I denna rapport beskrivs i detalj utvecklingen av ett realtidsstrategispel (RTS) som kontrolleras med Xbox 360-konsolens handkontroll. Det främsta syftet för utvecklingen av spelet är att undersöka möjligheten att skapa ett strategispel spelbart på en konsol. Dessutom kommer spelet förhoppningsvis släppas kommersiellt efter att detta kandidatarbete är färdigt. Rapporten beskriver också aspekter av spelutveckling, till exempel grafik, artificiell intelligens, och hur Scrum kan användas för spelutveckling. Spelet utvecklades under en termin genom att använda ramverket Microsoft XNA Game Studio och programmeringsspråket C♯. Microsoft XNA Game Studio är ett verktyg för att utveckla spel till Xbox 360, Windows och Windows Phone. Även om spelet inte blev färdigt, visade arbetet på många utmaningar i att utveckla spel, i synnerhet de tekniska, artistiska och designmässiga aspekterna. Tyvärr kunde inte kontrollmetoden testas och designas ordentligt på grund av tidsbrist.

Acknowledgements
We would like to thank our supervisor Ulf Assarson, for helping us throughout the project. His expertise on 3D graphics rendering was very useful. In addition, Ulf helped us improving this report. We would also like to thank Sten Ludvigsson for helping us performing business analysis to see how commercially viable our game could be.

List of abbreviations

AI - Artificial Intelligence
API - Application Programming Interface
Cg - C for Graphics
CPU - Central Processing Unit
D-pad - Directional pad
fps - frames per second
GLSL - OpenGL Shading Language
GPU - Graphics Processing Unit
HLSL - High Level Shader Language
RTS - Real-time strategy game
TBS - Turn based strategy game

Table of Contents

Chapter 1

Introduction

The main goal of this project is to develop a real-time strategy (RTS) game controlled by a standard console gamepad, namely the Xbox 360 gamepad. As a side goal, the game is hopefully going to be playable on the Xbox 360 in the future. In addition, the other main reason for this project is to explore the viability of the Microsoft XNA Game Studio framework for developing more complex games.

1.1 Background

The gaming industry today has grown considerably since its humble beginnings in computer labs in the middle of the 20th century. For the year 2010, retail revenue in the US for all aspects of gaming (hardware, software, and accessories) reached approximately 18.5 billion USD (Matthews, 2010). While the console market has grown, there has been a decreasing focus on the PC market for more complex games. The PC platform has traditionally been associated with complex strategy and simulation games such as Age of Empires (Ensemble Studios, 1997), Civilization (Microprose, 1991), and Europa Universalis (Paradox Interactive, 2000). These kinds of games are therefore becoming much more of a niche. The problem with RTS games is that they traditionally require a large number of keys and a mouse to function properly. To develop such a game for a console is therefore a great challenge.

Developing games for a console has long been reserved for professional developers, but when Microsoft released their game development framework Microsoft XNA Game Studio, Xbox 360 development was opened for hobby developers as well as smaller development studios.

1.2 Previous work

While the real-time strategy genre is not well represented on consoles today, Herzog Zwei for the Sega Megadrive (Sega Genesis in the United States) is often considered the first RTS game. Since then, the majority of real-time strategy games released for consoles have been direct ports or adaptions of PC games. Some notable exceptions are Brütal Legend (Double Fine, 2009) and Halo Wars (Ensemble Studios, 2009).

1.3 Purpose

The main purpose of this thesis work is to develop a real-time strategy game controlled by the Xbox 360-gamepad. The basic goal is to run the game on PC, but hopefully, the game could also be run on the Xbox 360 console. The great challenge consists of developing a real-time strategy game, a genre that is uncommon outside the PC platform. Another goal is to learn how a game is developed, from programming to asset creation to work process. One large part of this was to investigate how working according to an established agile process, Scrum, would work in a school environment. The experiences acquired during the development of this game are also compiled in this report. If the game is developed further, it could be released through the digital distribution service of the Xbox 360.

The results of this project may also be of use to the gaming industry, as we can develop our game without financial risk, since larger studios generally avoid more experimental game designs because of financial reasons. However, it is more likely that this report can aid other students when they develop a similar game, or when developing with the XNA framework.

1.4 Delimitations

The main design delimitation was decided early: the game should not try to emulate a traditional PC-oriented real-time strategy control method. This means that a cursor should not be present. The main principle was decided as: 'The player should never wish that he or she had a mouse or a keyboard'.

The development of an advanced AI playing according to the same rules as a human player was decided to be too time-consuming for this project. Also, it was decided that the game should not have a network component because of time constraints. The opposition was therefore decided to be composed of either enemies with simple AI or another player. When playing against another player, a split-screen mode was implemented, since real-time strategy games have such large playing fields that containing the entire game world on one screen was not deemed feasible.

Because the group had very little to no experience in modeling in 3D, it was decided to focus on simple models based on primitive geometrical shapes. It was also hoped that this approach would create a distinct visual style for the game.

Another delimitation was to make sure that the game is runnable on the Xbox 360. While running the game on Xbox 360 was not of great importance, it was decided to keep the option available.

1.5 Method

The game was developed over the course of one semester in the spring of 2011. This report was written at the end of this period.

Because of the current state of game development, it is often hard to find scientific articles to support certain claims. Often, it is more prestigious for a game developer or researcher to share their knowledge on game industry sites, blogs, or exclusive conferences. Therefore, scientific articles have been used as sources where applicable. Where an example from a published game is discussed, a reference is provided to that game. The references to these games are provided in a separate Game References Section.

The work was divided according to disciplines such as programming and modeling. Prior to the start of the semester, all group members familiarized themselves with their respective development tools. The tools used included Microsoft Visual Studio 2010 Professional, a Subversion repository server hosted by assembla.net, 3ds Max, Maya, Paint.NET, Paint, and Google docs.

1.6 Gender statement
In this report, most instances of genderless or gender ambiguous words such as ‘player’ and ‘character’ will use the masculine pronoun. This is simply to improve readability and avoid awkward constructs such as ‘he or she’ and ‘his or her’.

1.7 Report structure
Because of the many different aspects of game development, this report is divided into many subsections. These subsections are in many ways self-contained, and generally follow the structure: Introduction, Method, Results and Discussion. In some cases this structure is not applied, such as the section on C♯ and XNA, as no method description is included. After these subsections, the results and discussion for the project as a whole is presented.

Chapter 2

Designing a real-time strategy game for a modern console

2.1 Concerning strategy games on consoles

A strategy game is, much like the name implies, a game where the player's ability to apply strategical thinking to his play-style is the most important factor that determines the outcome of the game. A well-known example would be the classic board game chess. One task many strategy games have in common is the management of resources. The one who is the best at handling the resources is usually the winner. Other common important elements of strategy games are maneuvering armies and building structures.

Speaking in video game terms, chess would be classified as a turn-based strategy (TBS) game, since players take turns and remain inactive during the other players turn. Strategy games are usually categorizes as either turn-based or real-time strategy (RTS), which are the two biggest sub-genres of strategy. Even though the main difference between the two kinds is the way that time progresses, RTS and TBS games are usually very different both in design and gameplay. Generally, the scope is larger in TBS games, such as commanding nations or empires, while RTS games have a narrower scope, such as managing units in the battlefield.

Evaluating some popular modern RTS games such as StarCraft II (Blizzard, 2010), Command & Conquer: Red Alert 3 (EA Los Angeles, 2008), and Supreme Commander 2 (Gas Powered Games, 2010), certain trends in design can be observed, especially regarding the control scheme.

A player usually controls a cursor with the mouse to select units or buildings, and to issue orders. Many RTS games can be played using only a mouse, but they often make good use of the keyboard. The large amount of keys helps simplify the performance of more complex operations. An effect of this is that one of the most important aspects of player performance is the ability to simultaneously manage his economy, structures, and army.

Since large-scale game development is expensive, it is understandable that big game companies do not want to take large economical risks. As a result of this, many of the RTS games available for consoles are re-released versions of PC strategy games. For the developer, this is much cheaper than creating an entire new game. The consequences of this is that even if the control scheme is reworked, it will still be centered around a cursor interface unless changes are made to how the game plays, and this is almost never done.

As an example, a common feature in RTS games for the PC is the ability to select multiple controllable units using the so called drag select technique. By clicking and dragging the mouse, a rectangle is formed between the clicked position and the mouse position, as shown in Figure 1. When releasing the mouse button, the units inside the rectangle are selected by the player. This is a trivial task when a mouse is available, but has no natural counterpart when using a gamepad.

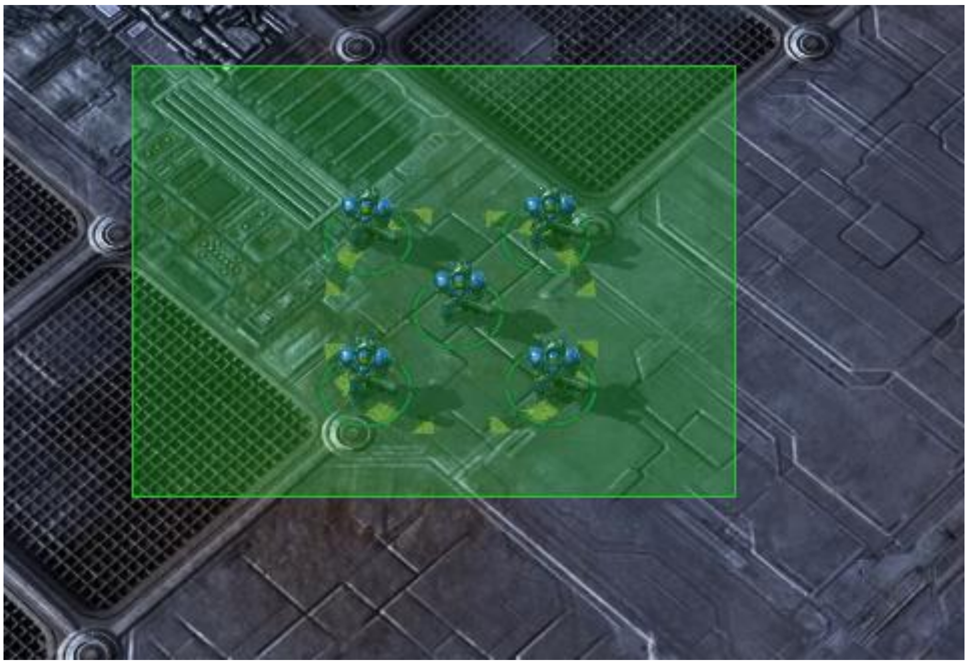


Figure 1. *Selecting multiple units in the RTS game StarCraft II (Blizzard, 2010)*

Even though what could be considered one of the very first RTS games, Herzog Zwei (Technosoft, 1989), was a console game, there are few successful RTS games that were developed specifically for consoles. One of the reasons for this could arguably be that a gamepad lacks the agility, precision and versatility that the combination of a mouse and a keyboard provides the user.
Since the early 2000's, the standard components of a console gamepad are two analog sticks, a directional-pad (D-pad), four buttons used by the thumbs, and up to four shoulder buttons used by the index- and middle finger of each hand. See Figure 2 for a depiction of the Xbox 360 gamepad.

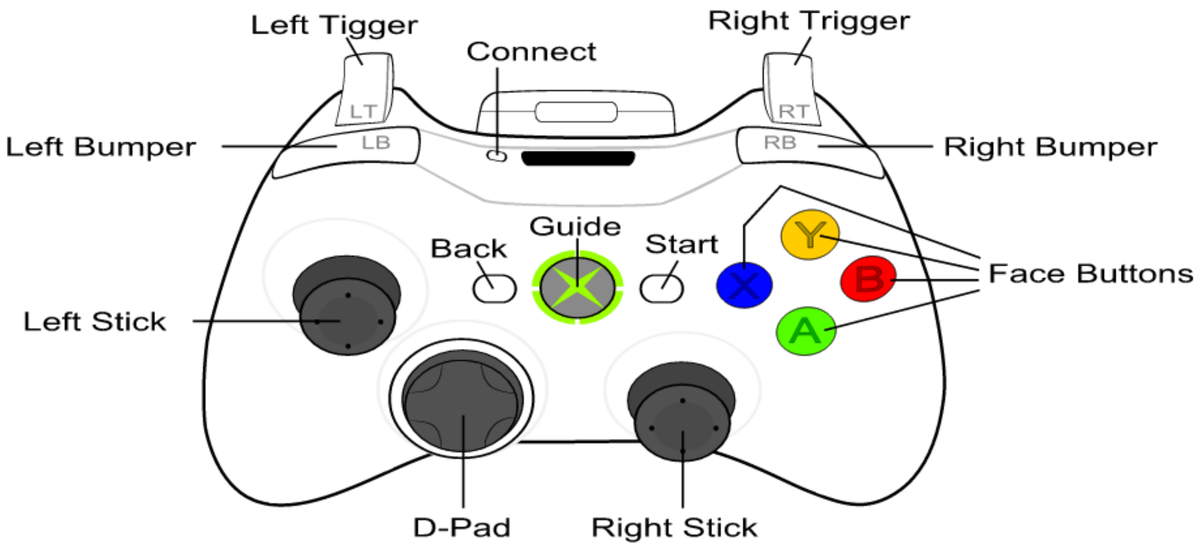


Figure 2. Xbox *360 gamepad button layout (Wikipedia 2011c)*

There are however a few rare cases of successful console RTS games, most notably Pikmin (Nintendo, 2001), Halo Wars (Ensemble Studios, 2009), and Brütal Legend (Double Fine, 2009). These games are some of the most commercially successful console RTS games and each has its own way of adapting the genre to the format and the gamepad. Zenko (2009) describes Halo Wars as a great introduction to the RTS genre for console players. In an interview, two of developers of Halo Wars describe a long process of prototyping and playtesting in order to appeal to console players (Nutt, 2008).

It should be mentioned that before the release of the first Halo (Bungie, 2001) game, the first-person shooter genre was in a similar situation. It was believed that first-person shooters were a game genre that could only be controlled properly on a PC, similar to how nowadays RTS games are thought to be best played on a PC. According to Gamespot (2005), people would argue that the precision of a mouse was a must for in-game aiming, and so, there were few critically successful console first-person shooter games. Halo is often mentioned for having revolutionized the genre for consoles, setting a new standard for controls and design. Halo also popularized the control scheme where the player makes use of both analog sticks, one for character movement, and one for weapon aiming.

Nowadays first-person shooter games for consoles are very common. Call of Duty: Black Ops (Treyarch, 2010) is one of the bestselling video games ever released, and has sold more than 23 million copies. 1.1 million of those copies were sold for the PC, while over 21 million of the remaining sales were for consoles, and the rest were for handheld devices(VGChartz, 2011). During the two first months following the release of the game, the game reached a collective combined playtime of 600 million hours, according to the publisher Activision (Albanesius, 2010).

This would suggest that first-person shooter games for consoles are indeed popular, and that a game with good design can be successful even if it belongs to a genre that previously was seen as inappropriate for the platform. While we have no false hopes that our project will have such an impact on the gaming market, we want to prove that with the right design choices, even an RTS game can feel natural being played with a gamepad.

2.2 Design goals

One of the great problems with controlling a traditional RTS game with a console gamepad is the cursor. A cursor is designed for a mouse, a device which position is relative to the cursor's position on the screen. The analog stick on a gamepad, however, is different in that it resets to its default center position when it is released. Therefore, the analog stick is better suited for movement that is relative to the velocity of the controlled object, not its position. With this in mind, it was decided that the game should be focused around an avatar, since the analog stick on a gamepad is much more suited for controlling this kind of object. That being the case, care had to be taken so that the avatar was not simply a cursor that looked like an avatar. Therefore, the player avatar moves around the map, but can be hit with projectiles, and has to deal with obstructing trees and changes in elevation. In this way, the player avatar is a physical object in the game world, and not simply a cursor.

Since the game would serve as an introduction for console players to RTS gameplay, the game has to be easy to control. Not requiring complex button combinations and moving the thumbs and fingers excessively was also decided as important. In addition, context sensitive actions were deemed important to minimize the amount of buttons needed. A context sensitive action was defined as the way a button would correspond to a different action depending on the context. For example, in a game that has an ‘use’ button, it could open a door if the player character is next to a door, or initiate conversation if the player is close to a computer controlled character.

In order to avoid the problem of controlling a large number of units, the concept of *minions* was introduced. Minions could be influenced by the player, such as making them follow his avatar. However, minions also have a mind of their own. The amount of control the player has over these minions was frequently debated. The reason for this limited control over minions is to ensure that the player is not overwhelmed trying to micromanage a large amount of units, while at the same time providing interesting emergent behavior from the minions acting on their own.

Another aspect of the design that was decided was to keep a standard game match to a time around 20 minutes. This is because console gamers are, as stated, not as used to RTS gameplay, and keeping the matches short creates a lot more tension and action. That the game should focus a fair bit of action was also decided, in order to make the console player comfortable. Still, a player with a good strategy should have a distinct advantage over one with good reflexes and hand-eye coordination.

2.3 Description of gameplay

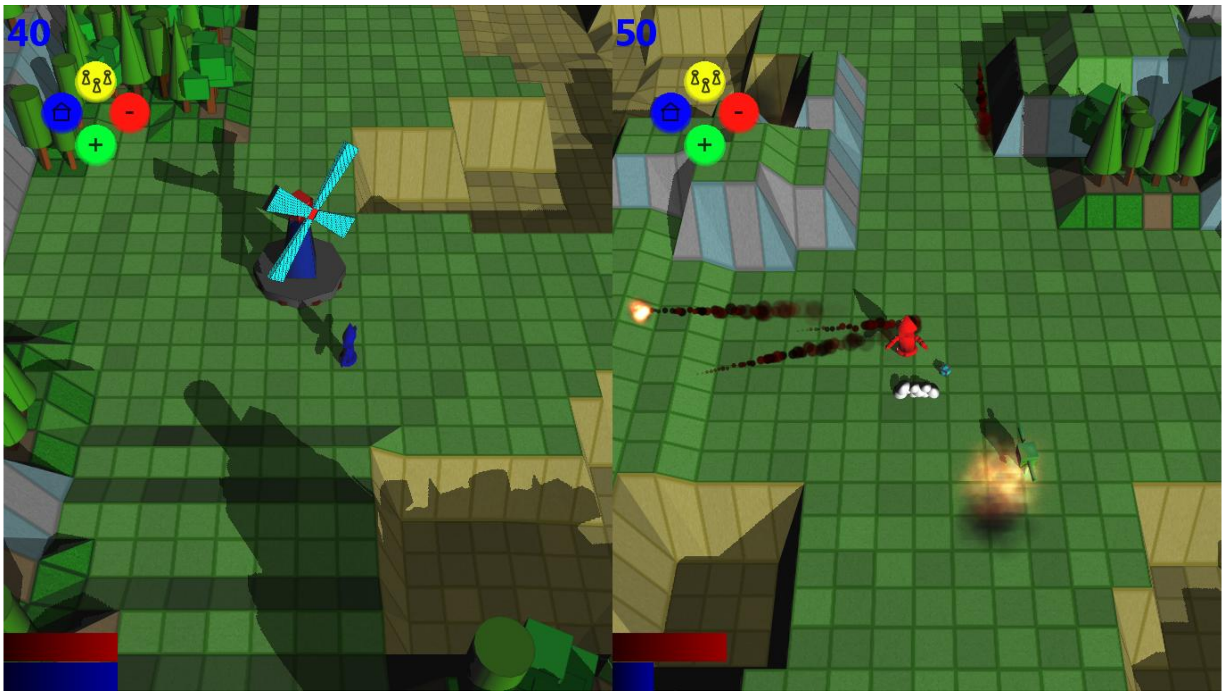


Figure 3. *Two players playing the game*

The game we developed, with the working title ‘A wizard did it... with science!’, tells the story of two robotic wizards locked in eternal struggle for supremacy. The game is played split screen (see Figure 3), and each player takes on the role of one of the two wizards. A wizard can hover around the game environment: a mechanical forest populated by aggressive monster robots. In addition, wizards have powerful spells such as the ability to cast fireballs. Casting these spells depletes the player’s energy. Two spells are shown in Figure 4 and Figure 5. The forest also contains small deposits of metal that the wizard can pick up. Objects that the player creates drop metal when they are destroyed, allowing for reuse of these resources, but this metal may also be stolen by the other player.

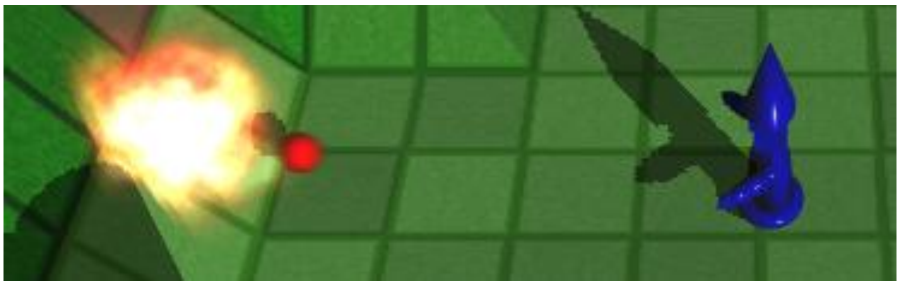


Figure 4. *The blue wizard firing a projectile into an innocent slope. To illustrate that the red projectile is moving, a motion blur effect was added to the screenshot. This may be rendered in-game in a later version*

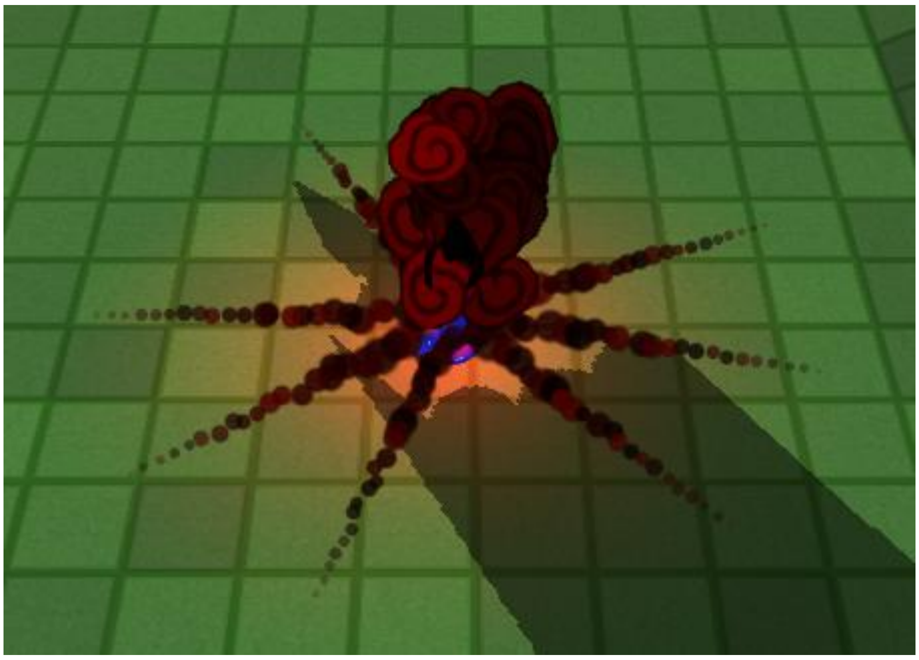


Figure 5. *A wizard unleashes his special attack, which creates an explosion and fires projectiles in many directions*

The player can also use his wizard to construct buildings at designated *building nodes*. These buildings have a wide variety of functions depending on their type, and all cost metal to construct. The first type of building is the *minion factory*. The player can use the minion factory to construct *minions*, which are robotic servants and warriors. Minions play an important role, as they can follow their wizard around, or stay close to buildings in order to increase the efficiency of that building. The minions are shown in Figure 6. Another kind of building is the defensive *tower.* The tower fires projectiles at approaching enemies. The third kind of building is the *windmill*, which generates energy for the player. The *healing shrine* is the last building. The healing shrine restores health to all friendly units around it, at a cost of energy for the player. The shrine can also be used by the wizard to teleport to another shrine on the map, and will also rebuild the wizard, should he be destroyed. If all healing shrines are destroyed, the wizard cannot be rebuilt and will lose the game. All buildings are displayed in Figure 7.

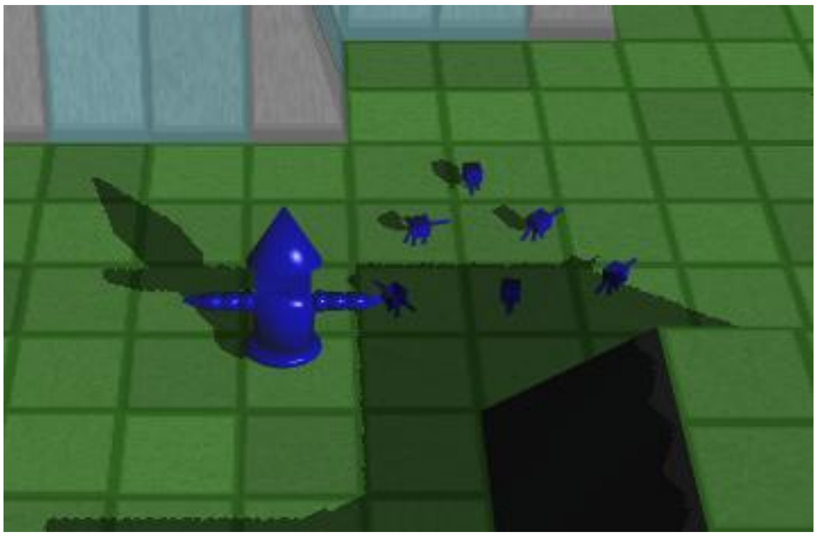


Figure 6. *The wizard followed by his loyal minions*

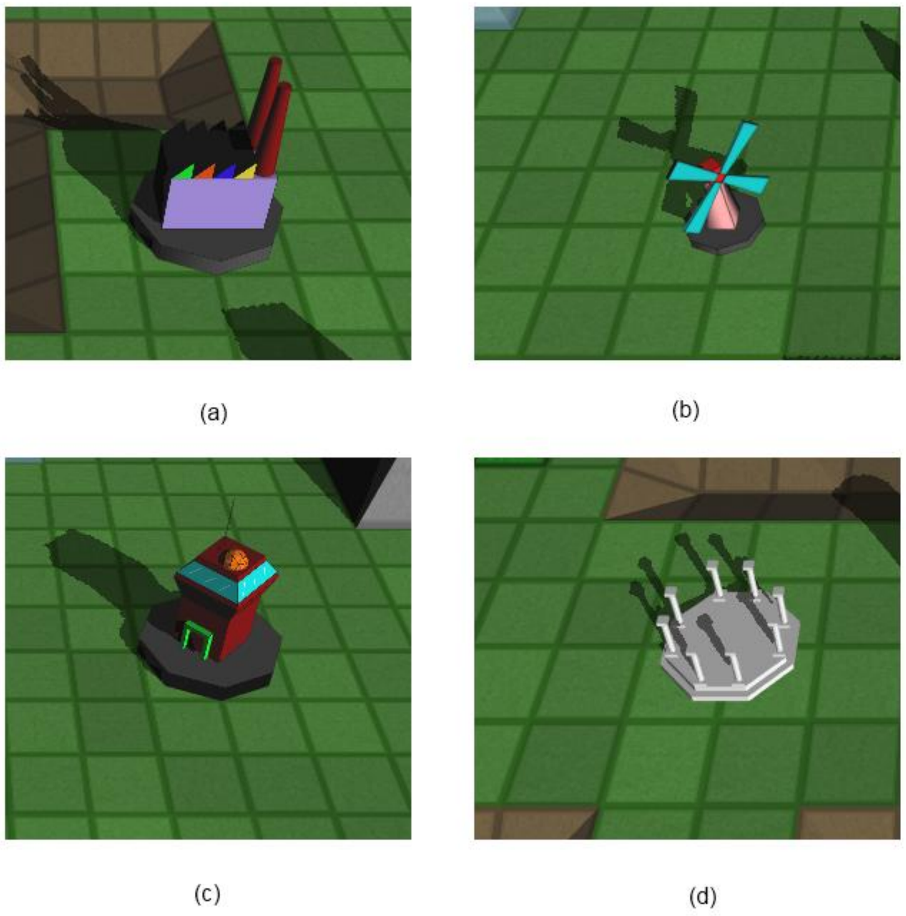


Figure 7. *The buildings in the game: Minion Factory (a), Windmill (b), Tower (c), and Healing Shrine (d)*

The player who braves the dangers of the forest and reaches the opposing player's base and destroys it, thus denying the other player the ability to respawn, will be victorious.

2.4 Description of the current prototype

In the end, the technical side of game development took a lot of time from the implementation of the game design. While basic functionality such as constructing buildings, controlling minions, and shooting projectiles exist, there was not enough time to try different designs and conduct user tests. As such, the game is in a very rudimentary state. The role the minion will play in the final game design is still vague. In the prototype, minions can fire projectiles and boost the efficiency of the windmill as well as the tower by standing next to them.

At the start of the project, an important goal was to evaluate different control methods. Since the development time was short, we did not have the full time to perform focus tests. However, a proposed control layout was put forth. It utilizes a function called the build menu, which is opened by the press of a button. This build menu changes the functions of the four face buttons. When the build menu is closed, the face buttons control buildings and minions. The face buttons correspond to the four different buildings when the build menu is opened.

The player can use the controller to construct buildings, but also to control them as well as minions. Minions can be added to or removed from a group which follows the wizard. The formation the minions move in can also be changed by the press of a button. The wizard also has two offensive abilities: firing projectiles and unleashing a special attack. Both cost energy to perform, which can be charged by pressing and holding the trigger button for a longer time. The entire proposed control layout is detailed in Figure 8.

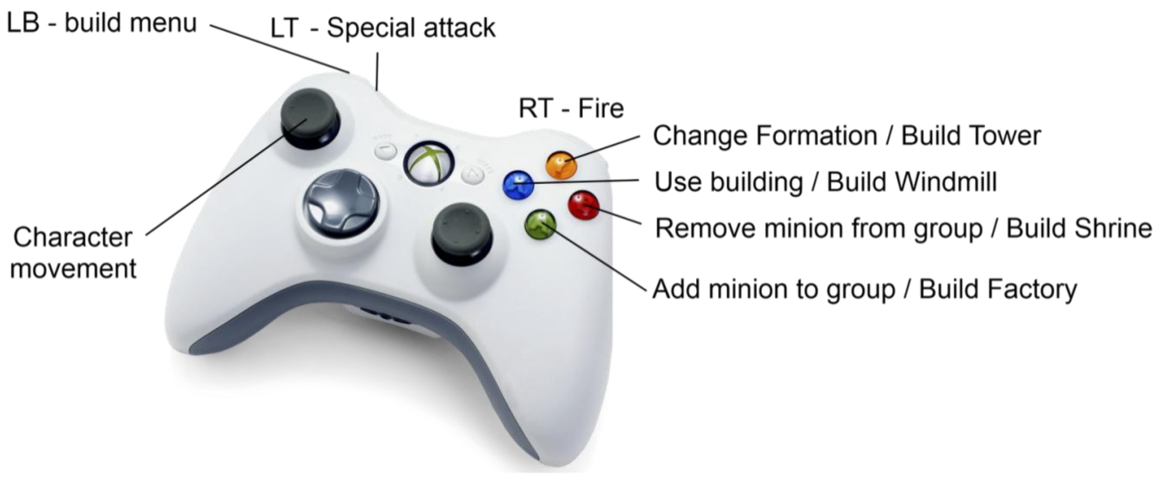

LB - build menu
LT - Special attack
RT - Fire
Change Formation / Build Tower
Use building / Build Windmill
Remove minion from group / Build Shrine
Character movement
Add minion to group / Build Factory

Figure 8. *Proposed control layout. A modified picture based on a controller image provided by Wikipedia (2011c)*

2.5 The challenges of designing a good game
Designing a game alone can be a tremendous challenge. Designing it in a group can be significantly harder, especially if the group is inexperienced. Much time was spent discussing the topic of game design, and every group member had a different opinion on many topics. This lead to confusion and indecision which features that where really important. In this way, a technical feature was easier to decide to develop compared to a new gameplay feature that did not correspond with half of the opinions in the group. If the main focus of the project is game design and not also learning how to create 3D games, it is important that the team can decide on what features are important for the game.

One of the main discussion points was the focus on either action or strategy. Some team members were perhaps surprised of the significant amount of action elements in the current prototype. Since few RTS games developed for console exist, it was difficult to assess what mixture of reflexes and logical thinking would be successful on a console. Some may point to the success of linear first-person shooter games such as Call Of Duty: Black Ops (Treyarch 2010) success on consoles as an indication that console gamers do not wish for a more cerebral experience. Others may see this as an excellent opportunity to introduce these players to a fresh gameplay experience. By gathering facts, more knowledge about the preferences of a potential audience could be gained. In the future, finding traditional console players and letting them test the game would lead to more data on the subject. A study could even be made using already existing games such as Herzog Zwei (Technosoft, 1989) or Halo Wars (Ensemble Studios, 2009).

A common action in RTS games, selecting multiple units, is handled in the current prototype by adding a minion to a group of followers. While there may exist a more elegant solution for the player to control his units, this focus on the avatar creates interesting gameplay dynamics. In contrast with tradition RTS games, the player cannot be omnipresent and near omniscient. In the current prototype, the player knows remarkably little about the state of the game world. This could very well lead to frustration, but as is well known, challenges and restrictions are an integral part of game design. That the player must be physically present in the areas of the map where the most important action takes place can create a sense of urgency and empathy with the minions that is rare in traditional RTS games.

Good controller design is also a topic that will be explored should the development of the game continue. The control layout in the prototype is simply a proposal. The possibility of firing the projectiles with the right analog stick was discussed, but this option was rejected for the prototype. The problem with this approach was that it could cause a lot of thumb movement from the face buttons to the right analog stick, which would mean that the entire control scheme would have to be reworked. Still, it is important to try all different approaches to such a complicated problem as control design. In the future, tests with different control methods will be made.

In conclusion, designing a game is very difficult, and to come as far as a prototype can be an achievement in itself. According to our experience, in order to design a good game data needs to be collected in order for all group decisions to be well informed, or to have a game designer with great intuition. Since we were lacking game design experience, we would probably have to rely on trial and error together with perseverance to create a great game.

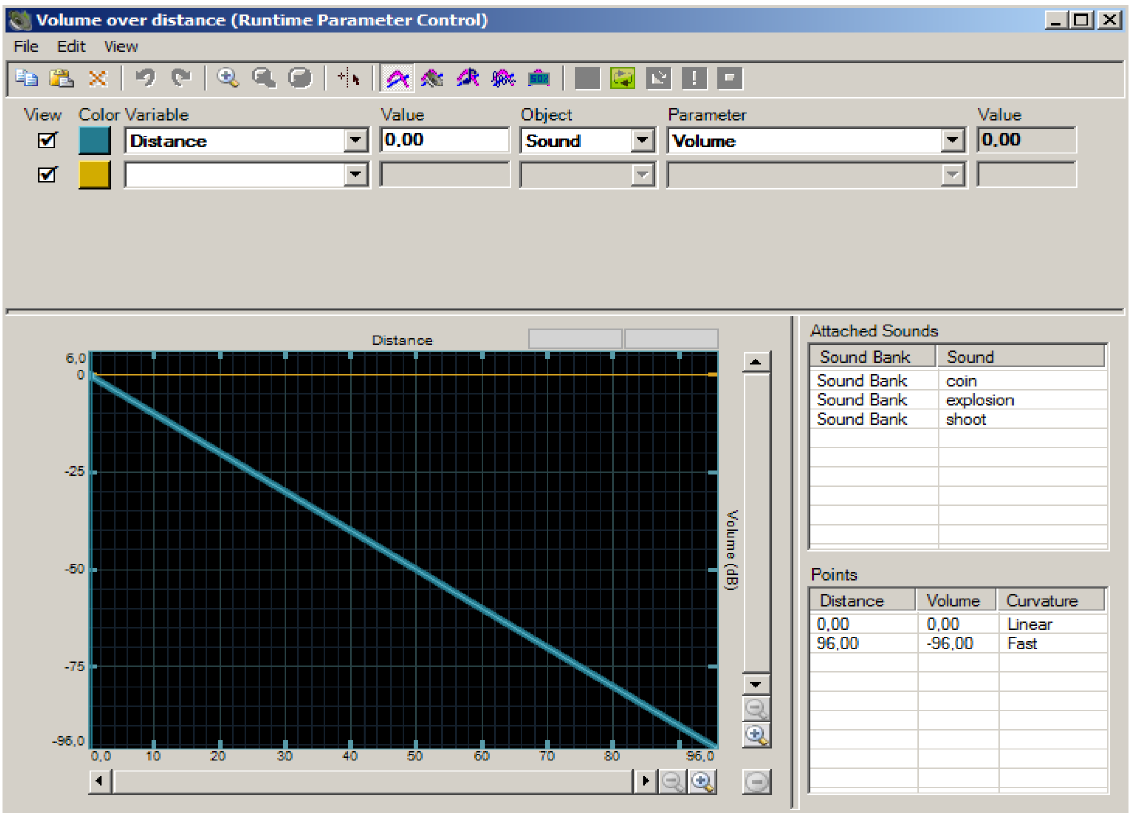
Chapter 3

C♯ and the XNA framework

3.1 About C♯ and XNA

For developing our game, we used the Microsoft XNA Game Studio framework, often simply called XNA. XNA is a framework developed by Microsoft to simplify game development on their three main platforms: Windows computers, Xbox 360, and Windows Phone. According to Microsoft (2006), XNA is intended for smaller developers releasing games through digital download, in contrast with bigger developers who supply their games on physical copies on discs. XNA was built with ease of use in mind and the philosophy that each line of code should do something in the game. In this way, much of the coding that is not directly related to the game is eliminated. In addition, XNA should simplify game development and provide a common ground for developers in order to improve software quality (Microsoft, 2004).

Sound management and playback was also simplified by the Cross-platform Audio Creation Tool (XACT), which has many useful features including 3D sound and sound modifications. The main advantage of using XACT, as explained by the Microsoft Developers Network (2011b), is that it separates the role of the programmer and the sound designer. The 3D sound supported in XACT is not very advanced. XACT simply distributes the sound volume to the speakers of the device depending on where in the game world the origin of the sound is located. It also supports sound property modifications based on external variables such as distance, illustrated in Figure 9. Commonly, volume is decreased as distance increases.



View	Color	Variable	Value	Object	Parameter	Value
☑	🟦	Distance	0,00	Sound	Volume	0,00
☑	🟨					

Attached Sounds

Sound Bank	Sound
Sound Bank	coin
Sound Bank	explosion
Sound Bank	shoot

Points

Distance	Volume	Curvature
0,00	0,00	Linear
96,00	-96,00	Fast

Figure 9. *One important part of XACT: The RPC (Runtime parameter control). The RPC takes a runtime parameter (in this case distance) and modifies the sound that is played. In*

this RPC, sound volume is decreased as distance is increased. As the decibel scale is logarithmic, the decrease in volume is logarithmic

XNA is most commonly programmed using the language C♯. C♯ (pronounced C Sharp) is an object-oriented programming language, mainly used to code for the .NET platform. The .NET platform is a common platform that makes it easy to distribute programs on all Windows operating systems. The language syntax is very similar to Java, and also shares some similarities with C and C++, on which it is based.

There are of course a multitude of alternatives to XNA and C♯ for making games, such as C++, Adobe Flash, Unity, DirectX, and OpenGL. Yet, the main goal of the project was to develop a game for a modern console, and for a small developer, XNA and C♯ is the easiest option. The barrier of entry for developing games for the Nintendo Wii or the Playstation 3 is higher. For the Playstation 3, a special development kit is required (Boyer, 2008), and for the Nintendo Wii, the developer has to be licensed by Nintendo (Bozon, 2008). With XNA, it is easy to develop for Xbox 360, though releasing a game commercially for the Xbox Live Marketplace requires the developer to go through a validation process according to Perry (2006). Xbox Live Indie Games is an alternative way to distribute the game on the Xbox 360, with a lower price point and a peer review process (Hawkins, 2008)

3.2 Loading assets: XNA content pipeline

Klucher (2006) describes the content pipeline as "an extensible content processing framework". When importing a 3D model as an asset, it needs to be exported as either the .FBX file format or the .X file format, which are the only formats XNA supports for using 3D files (Microsoft Developer Network, 2011d).

The purpose of the content pipeline is to process and prepare content in order to access it in the game. All of the content is managed inside Visual Studio and after importing an asset, for example a 3D model, an importer will take the file and normalize it. This means that it takes care of, for instance, the direction in which the model is facing and finally it will import the content into Visual Studio. The model in this example is then imported to the content DOM, which is a term used to represent a collection of classes, where the model is saved as a known format to the XNA pipeline processor. This means that the original file format of the asset does not matter because they are all represented in the same way. Afterwards, the processor takes the data from the content DOM and creates an object that can be used in the game, according to Klucher (2006).

The main reason the XNA content pipeline exists is to make the game run fast. If it did not exist, all the assets would have to be built in their original file format. When loading the assets, the game would need to decide their format and convert them. This would make the game slower compared to using a content pipeline (Microsoft Developer Network, 2011c).

Figure 10 shows the whole process of the XNA content pipeline. It also illustrates all the file formats the pipeline supports, as well as which format is used when exporting.

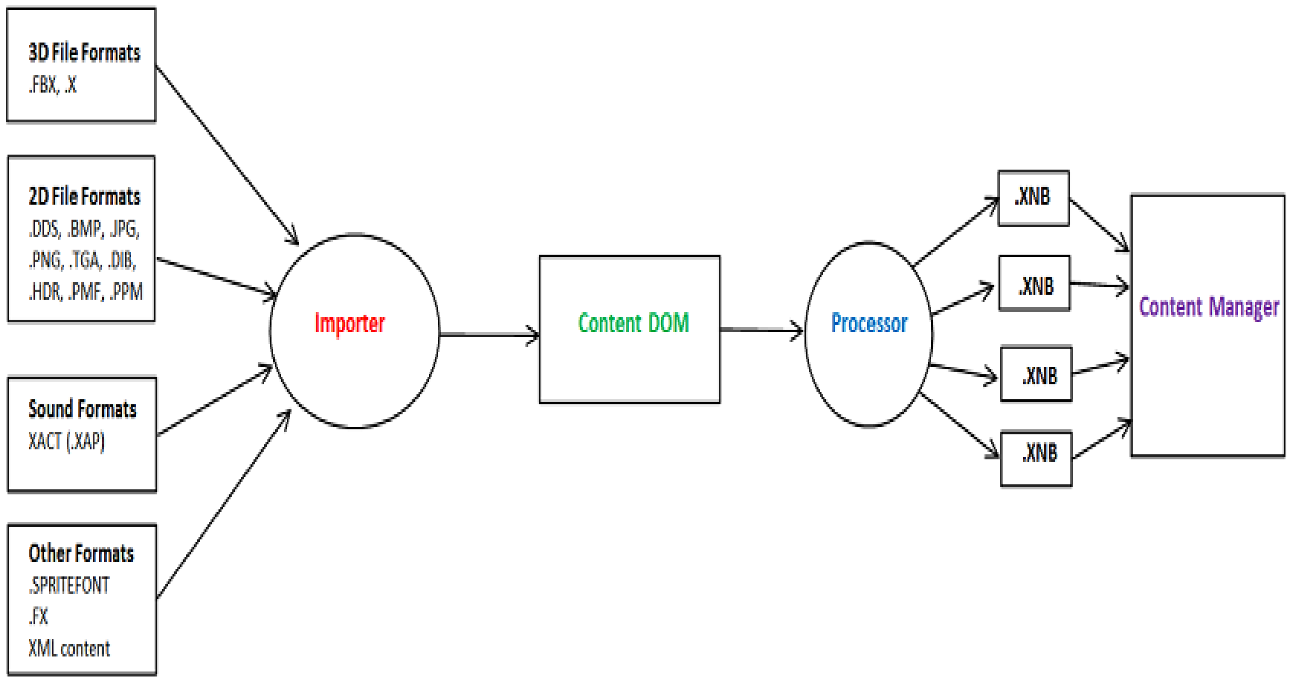


Figure 10. *XNA Content Pipeline*

3.3 Our experience with C♯ and XNA

The XNA framework assisted greatly in developing our game. Using a framework sped up the development of the game, especially the development of the graphics rendering. The XNA framework also had ready-made modules for player input and matrix and vector calculations that saved us much time. However, it is likely that similar modules are provided for the majority of popular programming languages. Even so, XNA provided a good framework that combined all these tools into one package. The possibility to release the game for Xbox 360 was also a major advantage of using the XNA framework.

In addition, because all members of our group were familiar with Java, the step to C♯ was a minor one, and the similarity in syntax sped up the process of learning a new programming language. C♯ also has some rather useful functionality such as properties that were a welcome addition to our programmers, who had previous experience with Java.

XACT was also a great feature of XNA. It made it possible to implement 3D sound without much effort. It was also used to randomize the pitch of certain sounds. This way, the sound of firing a projectile was not as monotonous as repeatedly playing the same sound at the exact same pitch. Working with a graphical user interface to handle sound made it easy to perform relatively complex sound modifications.

That is not to say that there were only advantages to developing using C♯ and XNA. The major disadvantage we found using C♯ was that it enforces garbage collection. According to Richard (1996), garbage collection means that the program will automatically scan all objects created by the program, and remove those which there are no references to. In this way, objects which are no longer used are deleted without the programmer having to manage the memory of the application. Garbage collection can, however, be a problem for real-time critical systems such as games. The scan can slow down the game, leading to a less

enjoyable experience for the player.
According to Beneux's blog (2011), the Xbox 360s garbage collection scans are far less efficient than those on the PC, causing them to be even more problematic. Hargreaves, one of the XNA developers, describes two ways to remedy this for the Xbox 360 (Hargreaves, 2007). The first way is to keep the number of scans to a minimum by not creating more objects. This is because after allocating a certain amount of memory, the garbage collection of Xbox 360 is triggered. By minimizing the amount of objects created, garbage collection scans become more infrequent. The second method is to have as few object references as possible, thus making the garbage collection scans finish faster.

3.4 Evaluation of C♯ and XNA

Overall, working with XNA was the right decision for our project. Since development for a console was one of our main project goals, we really did not have much of a choice. However, if we look beyond this requirement, XNA still is an excellent way to develop games. XNA is especially useful for small projects. At the time of writing, there are almost 1800 games approved for distribution on Xbox Live Indie games (Xbox Live Marketplace: Indie games, 2011). The games are of course of varying quality, but it is apparent that the barrier for entry is much lower than the more professional-oriented Xbox Live Arcade. It may be argued that XNA does not give as much functionality or freedom compared to developing the game from the bottom up. This is very true, yet the ease of use in XNA can really make the team feel motivated and focus on the code that runs the game, and not the code that manages windows or makes sure sound output works properly. In addition, many of the features of XNA are optional.

As mentioned, the garbage collection inherent to C♯ can lead to performance issues. This would however only be true for more complex games with a lot of objects, and as such is not a concern for smaller projects. For our game, we did not run into any garbage collection related issues on the PC, but the team members' PCs are much more powerful than the over five years old Xbox 360.

Another problem for more complex games is that XNA lacks any native support for animation. In our case, a modified animation example provided by Microsoft (Microsoft Developer Network, 2007) was used for our animation system. However, it was a complex task to implement the animation system properly, and something that would not be trivial for an inexperienced game developer to program.

In conclusion, we believe XNA is a good way to develop games, and an excellent way for smaller teams to release their game on a commercial platform, as evidenced by the great number of titles on Xbox Live Indie Games. In the future, the game developed in this project will hopefully be released for Xbox Live Indie Games.

Chapter 4

Scrum and agile software development

4.1 Scrum and agile: an introduction

For our game project, the goal was to work according to Scrum, which is a framework for project management. As a project management framework, Scrum gives guidelines on how a project is planned and run. Scrum is usually used for agile software development projects. Agile software development is a way of developing software iteratively and incrementally, instead of following a large and detailed plan. However, agile is often described as more than a technique to develop software, as it has some characteristics of a work philosophy. Agile is based on four principles, presented in the Agile Manifesto (Beck et al., 2001):

- "Individuals and interactions over processes and tools"
- "Working software over comprehensive documentation"
- "Customer collaboration over contract negotiation"
- "Responding to change over following a plan"

It is necessary to understand that although working software is more important than comprehensive documentation, it should not be ignored completely (AgileCollab, 2008). This is analogous for all four principles.

Scrum is often used in an agile process, but it has some principles of its own. Keith (2010a) explains five major principles of Scrum:

- Empiricism: Change conditions and work process in real-time according to actual data.
- Emergence: Not everything can be known from the start. Do not prevent features being developed up front to determine viability.
- Timeboxing: Meetings should be of a fixed length of time
- Prioritization: Develop what is most valuable for the consumer first
- Self-organization: Small teams from multiple disciplines are encouraged to manage their process and create the best software the way they want.

Keith (2010a) goes on to state that these basic principles are reinforced by the three main parts of Scrum: the product backlog, sprints, and releases. The product backlog is a list of all features that could be implemented in the software project. The features in the product backlog are prioritized according to their value to the consumer. The second part of Scrum, the sprint, is a term for an iteration period where features are taken from the product backlog, distributed to the members of the team, and worked on for the duration of the sprint. If a task is deemed too time-consuming to finish in one sprint, it is split into sub-tasks and planned for coming sprints. The tasks of the sprint constitute the sprint goal, which is not to be changed while the sprint is being worked on. Each day, the team meets for a daily Scrum, where they briefly discuss what is going to be done that day. Releases are the final part of Scrum. After some sprints, releases are planned in. These can range all the way from basic functionality implemented to finished product. The purpose of releases is to focus on delivering a product, and not having half-finished features. The iterative process of Scrum is illustrated in Figure

11.

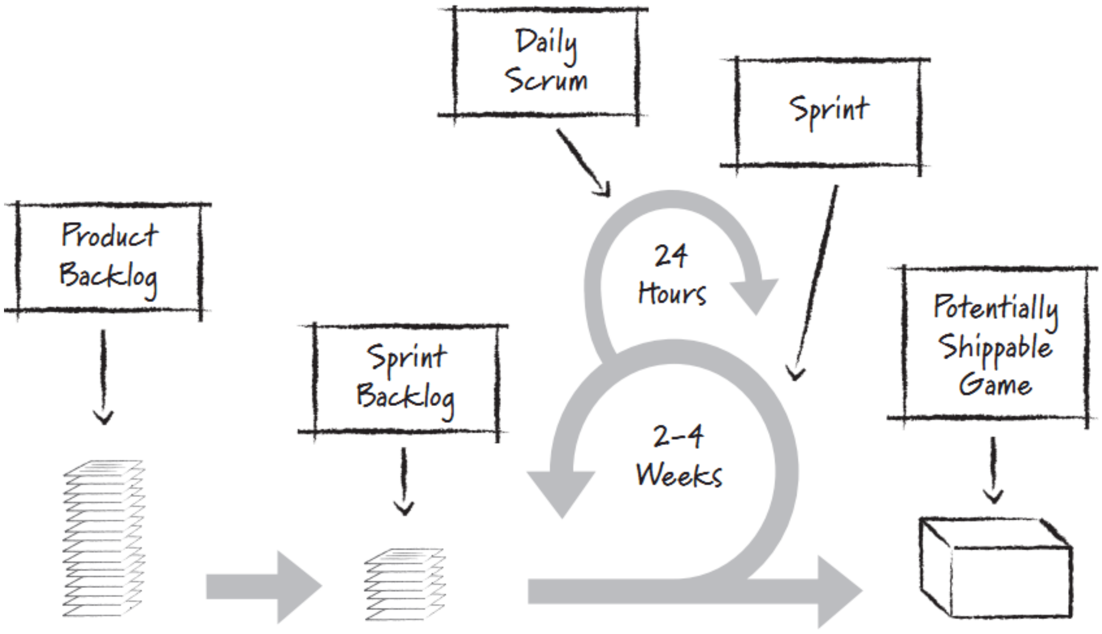


Figure 11. *An illustration of the iterative nature of Scrum. Features are selected from a product backlog and put into a smaller backlog, the sprint backlog. The team works on the features outlined in the sprint backlog and delivers a game that ideally has no unfinished features (Keith, 2010a)*

Schwaber and Sutherland (2010), two of the co-creators of Scrum, describe the three roles of Scrum as: the team, the product owner, and the Scrum master. The team are the group of people who work full time on the project, while the two other roles are mostly managerial. The product owner's role is to represent the view of the end customer, or the company that hires the software company, and as such is not a full member of a team. Not being a full member means that the product owner does not devote his entire work day to the project, and perhaps is product owner for multiple projects. The product owner prioritizes the product backlog so that features that are important to the consumer are always the ones that are worked on.

Like the product owner, the Scrum master is not a full member of the team, and may be Scrum master for multiple teams. The Scrum master's responsibility lies in making sure that the team follows the principles of Scrum. In addition, the Scrum master makes sure that impediments to progress are dealt with, that the team can meet deadlines, plans (but does not control) meetings, and maintains communication between the team and the users of the system. If the company is large, the entire development team is divided into Scrum teams of approximately ten people. The composition of a typical game development team using Scrum is shown in Figure 12.

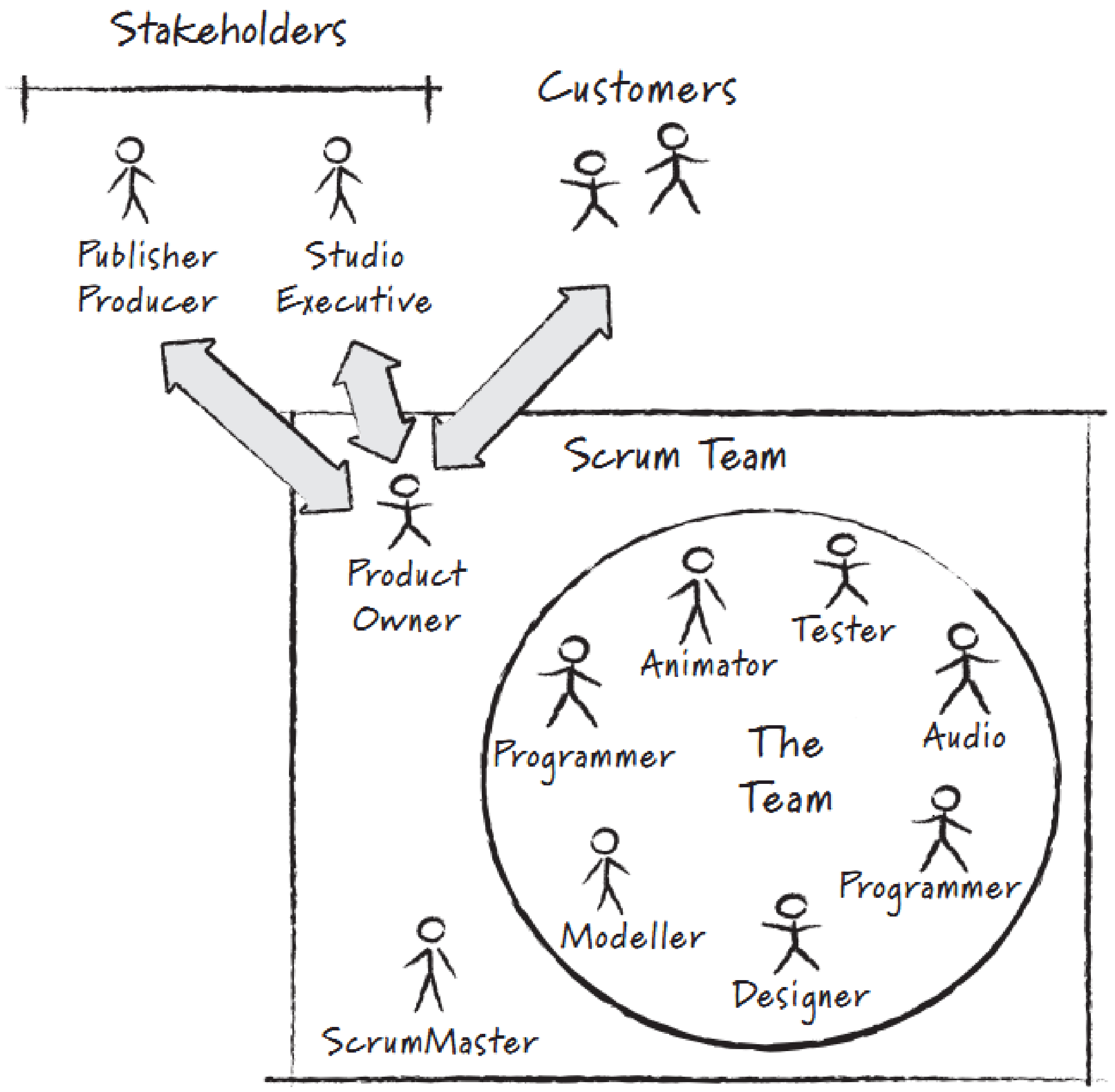


Figure 12. *A typical Scrum game development team. The product owner manages communication with stakeholders and players. The actual team (in the circle) is multi-disciplinary and inspected by the Scrum master (Keith, 2010a)*

4.2 Scrum in the software and game industry

Scrum has seen increased adoption in the software industry since its introduction. According to a survey made by the agile tool company VersionOne, Scrum and its variants are used by 78% of the software companies using agile processes (VersionOne, 2010). Increasing productivity and becoming more able to manage changing priorities were given as the main reasons for adopting Scrum. The situation in the game industry is more vague, and hard numbers are hard to find. However, Brütal Legend, a mainstream console RTS for the Xbox 360, was developed using Scrum (Esmurdoc, 2010).

Keith (2010a) writes that Scrum can decrease risk in game development, as developing a game incorporates a huge amount of risk-taking. In recent years, the cost and time for developing mass-market video games has risen steeply, while the price of video games have not risen much in comparison. As such, modern mass-market video games have to sell a larger amount of copies in order to make a profit. Yet it is hard to guarantee that a game is fun, says Keith. By developing iteratively, the game can reach a playable state faster, and as such the developer can determine which parts work and which do not. By using Scrum, features that are deemed valuable for the player are prioritized, minimizing risk.

However, it is not trivial to use Scrum and modifications to Scrum can often be detrimental to efficiency. Keith advises against modifying Scrum before the team has a good understanding of what ordinary, by-the-books Scrum means. On the other hand, forcing Scrum practices to be used instead of established best practices creates problems, according to Miller (2008). If the team is not careful, Scrum can lead to unorganized code and work not being done because team members do not organize themselves. For larger teams, it is hard to combine the idea of cross-disciplinary collaboration with the need for discussion between members of the same discipline. If the team moves to Scrum from more traditional development methods, it is easy to lock down many aspects of the planning, thus removing the positive iterative aspect of Scrum. In contrast, Schwaber and Sutherland describe Scrum as a collection of best practices evolved from software development. As such, it is ill-advised to ignore current best practices just to follow Scrum.

4.3 Our implementation of Scrum

Scrum was chosen because of its previous use by familiar game developers, as well as its iterative nature being a perfect fit for a more exploratory project, as our project was.

However, we did not follow Scrum strictly. Alterations had to be made because this project was a school project, and as such exams, other courses and extracurricular activities made it hard to realize all aspects of Scrum. Scrum also incorporates parts such as daily meetings and a common room for discussion and planning. This was very hard to accomplish in a school environment. Because of this, most of the communication had to be done through e-mail. In addition, the exam periods and breaks made the sprint length variable and hard to plan correctly.

One thing we changed that we thought was useful was that we prioritized not only according to value for our consumer, but for the amount of information completing a feature would give. For example, how the world was represented in code was important for many other features, and as such was highly prioritized, even though this had no direct value for the consumer. A further example would be how the animation system would work. If the way animations were integrated into the game was not decided, it would be hard for our animators to know how to export and work with their animations.

A major difference between Scrum and the way we worked was that we did not have a Scrum master or a product owner. Having no product owner is perhaps not surprising since we were not working directly towards a consumer. The role of the Scrum master was not explicitly realized, but we did have a team leader who assumed most of the Scrum master’s responsibilities. In addition to this, we deviated from Scrum in that we had no releases. Even so, we did set up vague goals for each sprint, although they were often too optimistic. A large part of the development time was spent on fundamental work such as animation and game engine design while only little time was left to work on the actual game design or control method design.

4.4 Results of using Scrum

While we did work iteratively with the project, it is questionable how similar the end result was Scrum. Because most of the communication was done via e-mail, there were some communication problems. One member of the team could start working with a feature and encounter a problem, only to find out later that another team member already had worked on the feature and had the same problem. Also, sometimes code was worked on without the original writer of the code being consulted, resulting in errors and reduced productivity. These problems would all have been prevented with better communication.

In addition, the iterative nature of the development of the code led to poor documentation and code comments, and sometimes features were added that were hard to expand on because they were coded from the bottom-up. This is a risk in all iterative processes, and time had to be spent going back and commenting and documenting code that had been written weeks ago.

Another problem was that the time it took to complete a task was not recorded properly. As such, it was hard to plan sprints since we did not know how much time it would take to implement a feature. For most of the project, we had nothing similar to a product owner. This turned into a problem when the game design had to be realized. Some game ideas needed some features implemented, while other game ideas were based on wildly different mechanics. In the end, we decided that one person should determine what gameplay features and functionality are important, in a way acting as a product owner.

4.5 Thoughts on Scrum and game development

The adoption of an agile work process helped greatly in developing our game, especially as the team had little experience with XNA, 3D rendering, and 3D game programming. The work was however hindered by limited time for face-to-face communication, an important aspect of agile software development. While the agile manifesto advocates working software over comprehensive documentation, it became clear that at least some documentation would be very useful, especially as new developers are brought in.

A very important lesson learned during the project was that while Scrum helps in deciding what should be developed by prioritizing from the product backlog, it is the product owner (in our case the team) that decides what features are important. If the team cannot decide which gameplay feature is important, the risk is high that the technical features of game development get more attention, since the value of these features are easier to demonstrate. Scrum does encourage developing features with questionable value, but during a constrained time period, teams often decide to play it safe. This can be seen in movie-licensed games, where gameplay often mimics other popular games. A stern product owner may alleviate this problem, but may on the other hand decrease the sense of ownership that is often an important advantage of Scrum. Another solution which is planned to be introduced if the development of the game continues is user tests, since they can show what gameplay features are important.

One aspect that is worth highlighting is that different people on the team may have different opinions on the ‘definition of done’, i.e. when the implementation of a feature was finished. Perhaps a module is fully implemented and tested, but is very poorly optimized. Some may regard the module as done, while others may not. In game development this can be a common problem, because of the difficulty to determine when a gameplay feature or art asset is done.

Modifying Scrum is a risky prospect. However, we could not work strictly according to Scrum as the game was developed in a school environment. Scrum is better suited if the team has regular work-days and a reliable schedule, since daily meetings are possible and planning is easier. Teams working on a school or hobby project should therefore be careful when using Scrum. The elements of Scrum that were implemented, such as iterative development using a product backlog and continual improvement of the process, were however very useful. The designated team leader of the project acted as a Scrum master, even though Keith (2010a) advises against this practice, since a Scrum master needs to be separate from the work in order to monitor it without bias. Had we known the problems of combining Scrum and school work hours, we would have made sure that all group members had a firm understanding of the underlying principles of Scrum and agile development to better adapt to changing circumstances.
However, the largest problem with this was that the team leader was inexperienced in Scrum, and as such, Scrum was not followed strictly. A small survey conducted by Keith in 2010 (Keith 2010b) shows that most failures with Scrum come from inexperienced or unwilling managers and teams, who did not implement Scrum correctly. One respondent used the term “ScrumBut”, as in the phrase ‘it’s Scrum, but...’. Clearly, using Scrum simply because it is a buzzword is very dangerous for large-scale game development. Also, according to Keith’s survey it took months for a team of three to find a Scrum-based process that fit their needs.

Empiricism, the continual improvement of our process, was employed to a certain extent. However, much more could have been done in the collection of data to make the decision more informed. During the project, there was also discussion about splitting the project into separate game ideas in order to experiment with many approaches to our main project goal. This was never realized because of time constraints, but will hopefully become reality if the game is further developed. If the game could be developed at a more leisurely pace or without distractions from other school-related work, more time could be spent working together, and perhaps a work process more faithful to Scrum could be adopted. It remains to be seen whether this would help game development. In conclusion, Scrum is difficult to implement, especially in a school environment, and therefore it is important that the team has full understanding of the basic principles of agile and Scrum. This way, the team can adopt a work process that suits their needs and variable schedule.

Chapter 5

Game engine

5.1 About game engines

A game consists of many parts, and the game engine is responsible for all these parts to work together. As the game engine is such an important part of the game, it is tightly connected to the game design, graphics rendering, artificial intelligence system, and so forth. The game engine is responsible for the updating and drawing of all objects in the game world, such as characters, environments, and cameras. This is done by an 'update-draw'-loop (Reed, 2008). Each iteration of this loop is often called a frame. In its most simple form, the pseudo code could look as shown below:

```
...
List<GameObject> gameWorld;
...
while(gameIsRunning){
elapsedTime = 1000/frameRate; //Elapsed time in milliseconds
input.Update();
      foreach(GameObject gameObject in gameWorld){
      gameObject.Update(elapsedTime);
      }
      screen.Clear();
      foreach(GameObject gameObject in gameWorld){
      gameObject.Draw(elapsedTime);
      }
      wait(1000/frameRate);
}
```

For a simplistic game such as Pong (Atari 1972), this would be enough. The input state would be updated, and then the position of the paddle would be updated according to what buttons where pressed. The position of the ball would update regardless of player input, and would change direction if it collided with a paddle. After the new positions are calculated, all objects in the game world are drawn. The paddle would be drawn as a rectangle, and the ball as a circle. To prevent the game from running extremely fast on more powerful computers, the program is told to wait a certain amount of milliseconds at the end of the loop.

According to Bishop (1998), the game engine has high demands for performance, because while the game loop itself does little work, it has to make sure that the methods are called correctly for optimal performance. An example of this would be to not draw game objects which are not seen from any active viewpoints in the game. Another way would be to only update objects that are close to the player, essentially freezing the game world in parts that are far away from the player. While Bishop admits that the speed of a game engine is important, it also has to be easy to modify in order to bring development costs down. This is especially true for smaller games that do not require the full performance capabilities of modern hardware. Because of the high amount of reuse in game engines as well as their complexity to program, there are many proprietary game engines available, such as the

Unreal Engine (Epic Games 2011) or Source (Valve 2007).

In XNA, the programmer is provided with the Game class. It has Update and Draw methods that are called automatically a certain amount of times per second (usually 60). Draw, in contrast with Update, is not called every iteration of the game loop. If the Update stage takes too much time, which would result in the game loop taking more time than it is allotted, the Draw call is entirely skipped. While this keeps the game from slowing down, it will make the game stutter, as it skips drawing certain frames. In addition to this, XNA provides a GameComponent class. If added to the game, its Draw and Update methods will be called automatically. This can be used for game objects, but also other components that need updating, such as sound engine or a collision manager (Reed, 2008)

However, having one Update method per game object to do everything is often too simplistic for modern game engines. In order to optimize performance, batched updating is used. Batched updating is the organizational style of making one kind of update for all objects at once, instead of doing a full update of each object one at a time. Gregory (2009) explains that it is more efficient to, for example, update all animations in a row rather than spread them out. This is because data used for animating one object may be needed for another object. Keeping this data close in the memory cache increases performance. Dividing the Update functions into separate parts also has the advantage that the game can utilize parallel processing. One thread or processor could calculate one animation while another animation is calculated in another process. Alternatively, one thread could be responsible for animation while another is responsible for artificial intelligence.

In the above example, all the game objects were stored in one list. Gregory (2009) does not recommend this approach for more complex games. If some object depends on another object to be fully updated, the game objects need to be organized in a tree-like structure in order to update properly. Bishop (1998) also explains that rendering speed can be greatly increased by organizing the game objects in a so called scene graph, a tree-like structure. If a node in this graph is deemed unnecessary to draw or update, its child node will not be drawn or updated.

5.2 Our choice of game engine implementation

Our game engine is rather simplistic compared to commercial ones, yet more complicated than the one described in the very beginning of this Chapter. At the heart of the game engine are two classes, the GameState and the DrawManager. The GameState is called to update all game objects in the world by the Update method of the XNA Game class. Similarly, the DrawManager is called by the Draw method in the Game class.

The GameState consists of a multitude of lists, each containing different kinds of game objects. For example, if a game object is animated, it is placed in a separate list and its animation is updated each frame. In this way, the GameState updates the game world in a batched manner, increasing performance. The primary reason for batched updating was however to ensure that utilizing parallel processing in a future version of the game. The game state also updates the position of cameras, user interfaces, a sound manager, a gamepad vibration manager, and the collision manager responsible for detecting and resolving collisions between game objects.

The DrawManager also contains a collection of lists, such as a list for 3D models and a list for 2D overlays such as game user interfaces. Since the game is played split screen, the DrawManager has to draw two separate views of the game world.

5.3 Evaluation of our game engine

This reliance on flat lists means that the game engine that was implemented may be inefficient compared to professional ones. However, it is worthwhile to note that no performance issues were noticeable in the game, most probably because of the simplistic graphics. Also, the game engine does update objects that are far away from the player avatar, since in a strategy game the entire world should be updated at all times.

5.4 Thoughts on our game engine

Premature optimization can often lead to more complex and less readable code. As this was first most complex game anyone of the team members had developed, it was decided that an extremely complex engine would only create problems with testing and design. In the future, more optimizations could be made, such as not updating animations unless the game object is visible. Also, the DrawManager and GameState are quite large classes and are hard to get an overview of. Separating these two classes into smaller components would improve modularity and understandability. While we did not use many complete libraries, it would probably have improved work speed but at the same time, we would not have learned as much. As learning was a side goal for this project, creating game components such as animation and particle systems was useful.

Chapter 6

Artificial Intelligence

6.1 Artificial Intelligence in games

Artificial intelligence (AI) is a large topic, and is used in a wide variety of areas, such as robotics, natural language processing, speech recognition and video games. The term artificial intelligence was first defined by John McCarthy (Skillings, 2006), and later he described AI as *"...the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable"* (McCarthy, 2007). Even so, there are many definitions of AI, especially in the game industry (Bourg and Seemann, 2004). An entity which is controlled by AI is often called an *agent*.

In games, artificial intelligence is responsible for directing so called non-player characters, or controlling an organized group of agents. Game AI can either stand in for a human, such as playing the opposing side in chess, or play using completely different rules, e.g. controlling the ghosts in Pac-Man (1980). As AI plays the opposition or aids the player in a game, the design of how the AI behaves is an important part of the game design. According to Bourg and Seemann(2004), there are two main challenges with game AI: appearing lifelike and not being computationally expensive. The factors that determine if the opponents appear lifelike include other aspects such as animation and sound. That being the case, lifelike AI can be summed up in two rules: 'appear smart' and 'never appear stupid'. Therefore, the appearance of intelligence is important, not the actual presence of intelligence. This is important — since while it may be possible to construct very advanced AI that can do a multitude of things — the game engine must be able to execute the AI operations on a restricted amount of time.

6.2 Common AI techniques and algorithms

Game AI is a highly varied field, but there are some common problems that need to be solved fairly often.

6.2.1 Pathfinding

Pathfinding is perhaps one of the most common problems in the game AI field. The problem consists of finding a path through an area with obstructions or rough terrain. By rough terrain, it is meant that some areas of the map take more time to traverse. The calculated path would ideally be the fastest, but often the path that is reasonably fast, but also quick to calculate and easy to smooth out, is more desirable.

The most common path-finding technique is the A* search algorithm (where A* is pronounced A star), a modified version of Dijkstra's algorithm (Higgins 2002, Dijkstra 1959). Dijkstra's algorithm calculates the shortest path from all nodes in a graph to one starting node. The A* algorithm is only concerned about the path from the start node to one other node, the goal node. While Dijkstra's algorithm is fully functional, A* is faster because it takes into consideration a heuristic. This heuristic is an approximation of the distance from the goal to a node being considered for the path. A node that is closer to the goal is favored, and as such, only a small part of the graph is ever considered for the path. For an example of the A* algorithm, see Figure 13. This means that the calculation time is shorter than that of Dijkstra's algorithm.

The A* algorithm calculates the fastest path through a graph, which means that the playing field of the game must be converted into graph form (Patel, n.d.). The simplest way of doing this is to represent the game world as a grid, considering each square as a node, with edges going to adjacent squares. Another way of representing the game world is a so called navigation mesh, which is a mesh of edges where the nodes are located at the corners and sides of the obstructions.

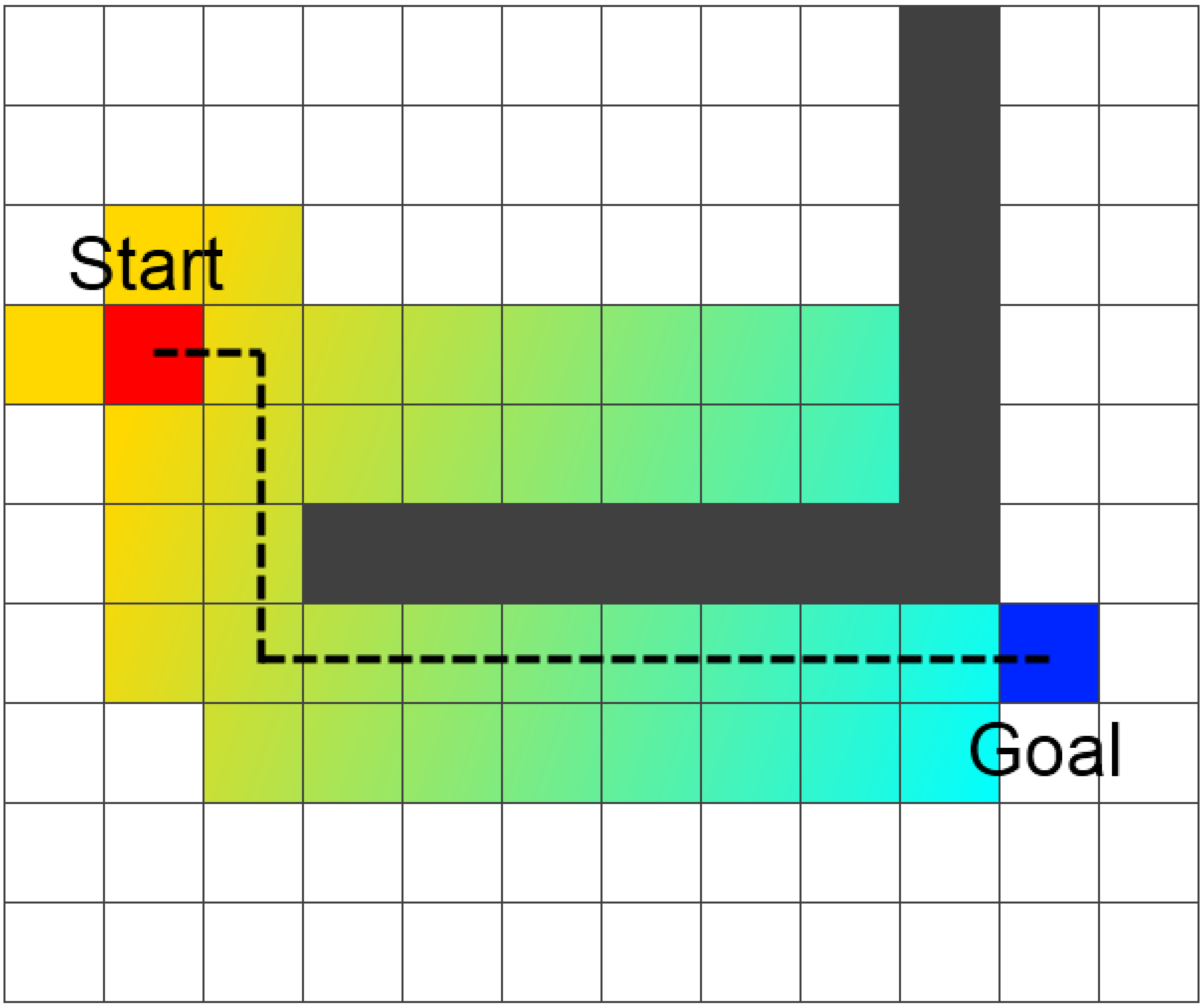


Figure 13. *Pathfinding through a grid-based environment. Note that in order for this pathfinding to work properly, the game environment has to be converted into grid form. The dark gray squares in an L-shape are impassable squares. The dashed line is the path found, and the other colored squares are the squares the algorithm considered while calculating the path. The shade is used to show which squares were determined by the heuristic to be more desirable because of proximity to the goal square*

6.2.2 Line of sight
Another problem that needs solving is determining which other agents or objects an agent is aware of. Simply including all objects in the game will often lead to unpredictable behavior, as agents will be omniscient. In addition, having to consider all other objects will likely diminish performance. Filtering out objects within a certain area is often necessary. This area could be a simple circle around the agent, or ray casting could be used. Ray casting (sometimes called ray tracing) is the process of sending out an imaginary ray from one point to another (Shirley and Morley, 2003). This ray could go from the agent's eyes to an object that it could possibly see. If the ray reaches this object, the object can be seen by the agent. However, ray casting is more computationally expensive than simply considering an area around the agent.

6.2.3 Scripting
A very common technique to create the illusion of intelligence is scripting. Bourg and Seemann (2004) describes this as writing a script for AI controlled characters to follow. As the behavior is written beforehand, scripting relies on certain events to have happened before to trigger the scripted event. When a script is being followed, the agent is usually set on its behavior and will try to carry out its script regardless of other factors. This can make the agent look unintelligent, but on the other hand, scripting allows for complex behavior that would be extremely complex to program any other way. Another kind of scripting is allowing game designers to modify the AI of the game via a scripting language, to change parameters and conditions for AI behavior without having to edit the actual AI code.

6.2.4 Flocking
Flocking is another AI technique, pioneered by Craig Reynolds in 1986 with his "Boids", a program modeling real life flocks of birds (Reynolds 1986). Boids were the name given to the simulated creature, which flew in flocks in a computer-generated environment. The program imitates flock behavior by following three simple rules:

- "avoid collisions with nearby flock-mates" (in more advanced simulations, also avoid obstacles in the environment)
- "attempt to match velocity with nearby flock-mates"
- "attempt to stay close to nearby flock-mates"

These simple rules are surprisingly effective, and the end result is more complex than would be expected from the simple rules. Other rules were also used, such as maintaining the same flying angle as surrounding boids. One notable early example of flocking in computer animation is the short film Stella and Stanley, Breaking the Ice (Symbolics Graphics Division 2009)

6.2.5 AI behavior selection
There are many ways to structure AI. Yet, what the problem essentially comes down to is picking one behavior or strategy from a collection of many, based on what the AI knows about the game world. In order to pick this behavior, a wide array of techniques can be used. According to Bourg and Seemann(2004), the most common are Finite State Machines, which are well known in the field of computer science. The Finite State Machine approach basically consists of agents being in a certain state or mood, such as 'scared', 'searching' or 'resting'. Certain events would then make the agent transition between these states. Other AI behavior

selection techniques include expert systems, Bayesian networks, behavior trees and neural networks.

6.3 Our AI implementation

Our approach to AI was rather simplistic. The agents in our game each have a separate AI manager. This manager is responsible for checking all objects within a certain radius around the agent and selecting an appropriate behavior. A behavior can be anything from moving towards a point to shooting at an enemy. The behaviors can also be very complex, and greatly customized. The selection process consists of checking each behavior against all objects in the surrounding area and selecting the one with the highest priority.

The AI system is very versatile because the programmer can define three different parts to a behavior: a triggering condition, a priority function, and an action. These three behaviors are illustrated in Figure 14. The triggering condition is a logical condition that must be met for the behavior to even be considered. This could be that an agent cannot fire a weapon unless he has enough bullets, or that an agent should not attack unless he has at least one friendly agent close to him. The triggering condition functionality also means that the AI system can be used with state machines. A condition could simply be that a behavior should only be evaluated for certain states. The second part of the AI behavior is the priority function. This is a separate object that is linked to the behavior that represents a certain mathematical function. The priority function can be based on simple parameters like distance or more complex criteria like amount of energy the agent has left. It could be that the priority given by the function is equal to the square root of the distance to the other object in the game world. After the AI manager has picked the highest priority behavior with a satisfied trigger condition, that behavior is then executed. The execution of the behavior can manipulate both the agent and the other object that is being considered. An example of an execution implementation would be that the agent fires a bullet towards the other object, presumably an enemy.

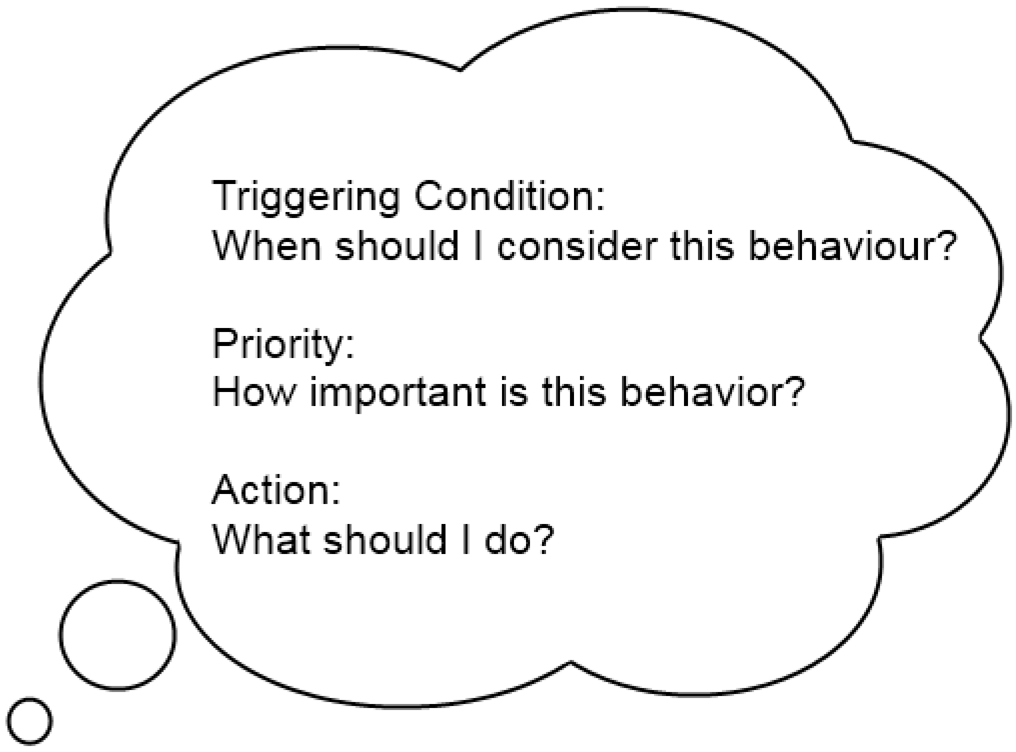

Triggering Condition:
When should I consider this behaviour?
Priority:
How important is this behavior?
Action:
What should I do?

Figure 14. *The components of a behavior*

We did not implement any pathfinding algorithm, mostly because of time constraints. Even so, the game map is divided into a large grid and also a smaller, fine-grade grid. This approach was chosen to make the implementation of pathfinding easier.

6.4 Evaluation of our AI implementation
The AI that was implemented for the game is not very complex, yet it is very flexible. A more complex AI was not developed for several reasons. The primary reason for keeping AI simple was that agent behavior needed to be easily understood by the player. There was also a desire to be able to handle a great amount of agents in the world, and so, complex AI was avoided in order to not decrease performance. Finally, the team simply did not have much experience programming AI, and therefore it was decided to write an AI system which was easy to understand, but most importantly, one that was easy to test. The AI approach used is mostly based on the idea of flocking: a few simple rules that create complex behavior.

One major advantage of the AI system is the loose coupling between agent and AI manager. This fact could prove very useful if a different AI structure had been implemented, such as finite state machines or behavior trees. Pathfinding was not implemented in our AI solution, mostly because of time constraints. Even though pathfinding was not implemented, agents in the world were made capable of sliding alongside obstacles on the way to their goal. Other functionality, such as line of sight and scripting, was also not implemented, mainly because their value was not deemed high enough to pursue in the limited time developing the game. In a way, it was unnecessary to implement line of sight. This is because the player can see a very large area, and it would be confusing for the player if one of his allied units could not see an enemy that was clearly visible to the player.

Yet the simple AI system used in this project has at least one disadvantage: it cannot handle AI behavior spanning multiple frames. For instance, each time the agent fires a projectile a firing animation is played. In order to ensure that this animation is played to completion, no new behavior should be selected until the animation is finished. This problem could be remedied by pausing the AI manager for the duration of the animation. However, this functionality was not implemented.

6.5 Selecting the right kind of AI

A simple AI has advantages beyond being easier to implement. Since our game was an RTS, the AI system must control a larger amount of agents compared to an action game. In a game such as StarCraft (Blizzard 1997) or Age of Empires, hundreds of units are active. These units are usually quite unintelligent. In the case of StarCraft, units will fire at enemies within range, and move away from an attacking unit that it cannot fire back at. All other actions are completely decided by the player. In contrast, in a modern first-person shooter games such as Halo, agents can take cover, run away, and flank the player. Also, this AI is hard for the player to predict, which is a great quality for an enemy in an action game, but this unpredictability can actually be seen as a disadvantage for an allied unit in a strategy game. Since the game design was not fully completed, it remains to be seen what kind of AI will be deemed important. In conclusion, it is important to consider what kind of gameplay that is required, as AI is an integral part of game design.

Chapter 7

Graphics rendering

7.1 Rendering 3D games in real-time

The problem with 3D games is that, unlike movies or video clips, they need to be rendered to the screen in real-time, as the scene is constantly changing depending on user input. With real-time means displaying a large enough number of images per second (frames per second - abbreviated fps) to make the media appear smooth. This display rate is intimately tied to the framerate of the game engine. If the game engine runs slowly, the fps of the game will decrease. Many display devices, such as LCD-screens, are set to 60 Hz. This means they will output a maximum of 60 frames per seconds. Research conducted by Eurogamer (2009) has shown that, not only is a high fps important for the eye, but it also reduces response time from user input. While a computer monitor may be limited to displaying 60 fps, a higher number than that will still benefit the response time, and therefore it is important for the game to run as fast as possible (Akenine-Möller, Haines and Hoffman, 2008).

7.2 The graphics pipeline

The heart of real-time graphics is the graphics pipeline. It is responsible for taking a 3D scene and render it as a 2D image to be displayed on the monitor. A scene is made up of points, lines and triangles. Usually these drawing primitives are combined to create a model or an object, such as terrain or a car. A virtual camera is used to define the scene, and the appearance of an object is affected by its material, the scene lighting, textures, and any provided special effects. Just like any pipeline, the graphics pipeline consists of multiple stages which execute in parallel. Generally, the three different stages are: application, geometry, and rasterizer, as shown in Figure 15. These stages could be (and normally are) pipelines in themselves. Optimization is important because the render speed is not faster than the slowest stage of the pipeline (Ashida, 2004). This stage is called the bottleneck. When developing a game, finding and removing the bottleneck is a very important part of the development process as the frame rate is starting to drop.

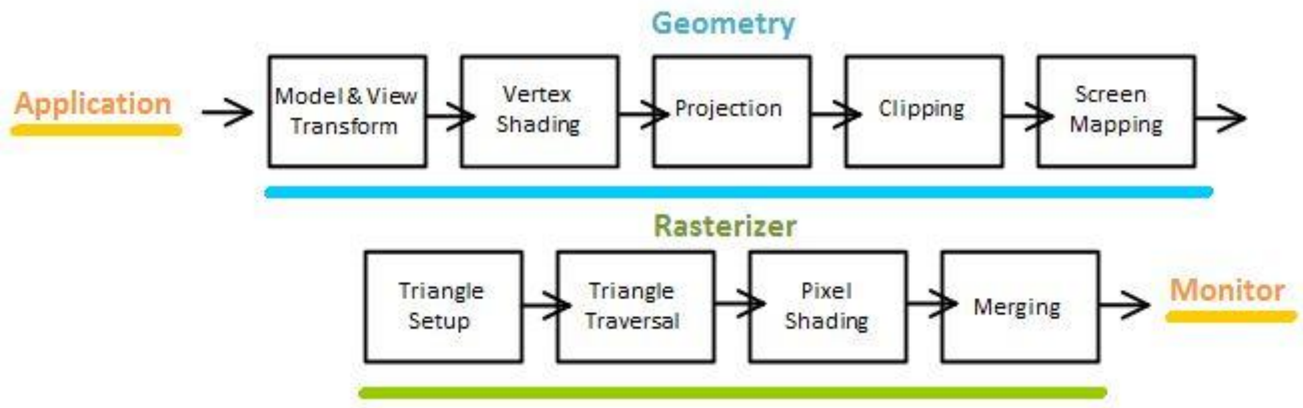


Figure 15. *The Graphics Pipeline, showing the process of rendering a scene to the computer monitor*

7.2.1 Application stage

In the application stage, the developer has full control of the events happening. Everything in this stage is executed on the CPU, which means that one of the biggest problems is to spread the workload into multiple threads that run in parallel on the CPU cores. Usually, when developing a game for multiple systems, heavy modifications have to be made to optimize the workload for each system hardware, since many CPUs work differently. Examples of tasks that traditionally execute in the application stage are: collision-detection, animation, processing of input, and AI (Akenine-Möller, Haines and Hoffman, 2008, pp.14). These tasks are managed by the game engine, as described in a previous Chapter. The drawing primitives are stored in a data structure (i.e. vertex buffer) and are finally sent further down the pipeline for rendering. From there on, all the operations are performed on the graphics processing unit (GPU). This has been the case since 1999 when GeForce 256 was released, which introduced full hardware transform and lighting (NVIDIA, 2011x)

7.2.2 Geometry stage

The geometry stage is responsible for transforming all the vertices into a common coordinate system (also called space). For example, models formed in a modeling program are residing in their own model space. By model space, it is meant that the model has its own coordinate system and scaling. All models used in the scene need to use a common coordinate system and be scaled appropriately. They are therefore transformed into world space, where all objects reside together. Additional transforms are also made to be able to simulate the camera view. Aside from transformations, the models are shaded according to their material and lighting sources. This may be performed in either of the geometry (per-vertex lighting) and rasterizer (per-pixel lighting) stages, or both. The vertices may store properties such as position, color and normal to be used by the shading equation. A normal is simply a direction orthogonal to the face it emerges from, used for various calculations including lighting. After shading, the models are projected from 3D to 2D, where the z-coordinate is placed in a special buffer. Next, clipping is performed, which means filtering the objects so that only those that are visible to the viewer are rendered. Finally, the x- and y-coordinates are transformed from 3D to screen coordinates (screen mapping) and passed on to the rasterizer stage.

7.2.3 Rasterizer stage

The goal of the rasterizer is to simply compute and color each pixel on the screen. It does so by first setting up the triangles consisting of three vertices each, finding which pixels are located inside each triangle and computing the pixel properties (fragments) by interpolating the data (like depth and shading values received from the geometry stage) from the three triangle vertices. After that, any per-pixel shading computations are performed followed by different tests to determine the final color of the pixel, which is then placed in a color buffer - an array of pixels to be displayed on the screen. A test that is executed automatically is the depth test, usually performed by the Z-buffer algorithm (Newman, Sproull, 1979). For each pixel, the depth value of the currently closest primitive to the camera is stored in a depth buffer, and any new depth value of the primitive is tested against this value. If the new value is closer to the camera than the currently stored value, the depth value for this pixel is updated in the depth buffer, as well as the color in the color buffer. To prevent the player from seeing the whole rasterization progress on the screen as it happens, double buffering is used. This means that the scene, which is currently stored in the color buffer, is rendered to an off screen image called the back buffer. When rendering is finished, the back buffer is swapped with the image that previously was displayed on the screen, called the front buffer.

7.3 Shaders

For a long time, the graphics pipeline was fixed, meaning there was no way for developers to program their own graphics functionality. Instead, developers had to rely on the graphics API and use its set of functions. However, the fixed-function pipeline has been replaced by a more flexible one in modern days, although it is still used in the Nintendo Wii (Akenine-Möller, Haines and Hoffman, 2008). The largest steps towards this flexible pipeline were the introduction of programmable vertex and pixel shaders, which are executable programs. In 2001 the very first programmable vertex shader was introduced with the release of NVIDIA's GeForce 3 (NVIDIA, 2011y) together with the DirectX 8 interface, but it was very cumbersome to use; developers had to write their code in assembly and many features were missing such as the ability to use conditional statements to control the execution flow (Akenine-Möller, Haines and Hoffman, 2008, pp.34).

It was not until the following year with DirectX 9 and Shader Model 2.0 that both vertex and pixel shaders became truly usable. A new high level programming language was included with the updated API, HLSL, which was easier to program (Fosner, 2003). Today, HLSL is still the shading language used in DirectX as well as in XNA. Other popular shading languages include GLSL (OpenGL) and Cg (DirectX and OpenGL). New shader models have emerged throughout the years, each containing additional functionality and increasing the resource limits (such as the number of arithmetic instructions and constant registers), and with Shader Model 4.0 even introducing a whole new shader called the geometry shader. As of now, the latest shader model is Shader Model 5.0 included in DirectX 11 (Microsoft Developer Network, 2010). Although surveys from Steam (2011) and Unity (2011) show that most people who play games on the PC are using graphics cards that support Shader Model 4.0, the list of games using higher shader models than 3.0 is still very short (Wikipedia, 2011b). This is a result of the recent dominance of Xbox 360 and PlayStation 3, which both are equipped with a Shader Model 3.0-level GPU (Akenine-Möller, Haines and Hoffman, 2008, pp.35).

7.4 Lighting

Lighting is a very important factor in determining the overall graphical quality of a scene. Without proper lighting, it is easy for a game to become dull looking and uninteresting to the player. In the real world, light is emitted from different light sources such as the sun, lamp posts, fire or a flashlight. As light travels and hits objects, part of it scatters and part of it is absorbed. Scattering means the light will change direction, for instance by being reflected or refracted by the surface, while absorption means the light will be transformed to other types of energy. The amount of light being scattered and absorbed depends on the surface material. Of course, light is an extremely complex phenomenon, which has forced developers to implement numerous approximation algorithms and optimizations to make it more suitable for graphics rendering.

7.4.1 Light sources

Light sources often appear in three different forms: directional light, point light, and spot light (Selman, 2002).

Directional light is the simplest type of light source, only being made up of one directional vector that hits all the objects from the same direction. On Earth, such light is received from the sun, since the sun is so far away that all light rays can be considered parallel.
A more complex light source is the point light (Selman, 2002), as shown in Figure 16. A point light radiates light in all directions equally from a specific point in space, with a certain power. Light bulbs, torch lights, and light emitted from explosions are all examples of point lights. The light radiating from a point light has an attenuation property that defines how the intensity diminishes with distance. In the real world, it is considered quadratic (Wikipedia, 2011a), but Daumann (2011) suggests adjusting the attenuation equation to better suit the type of game developed. Although light never falls off completely in reality, it is simplified to do so in computer graphics with a maximum range value.

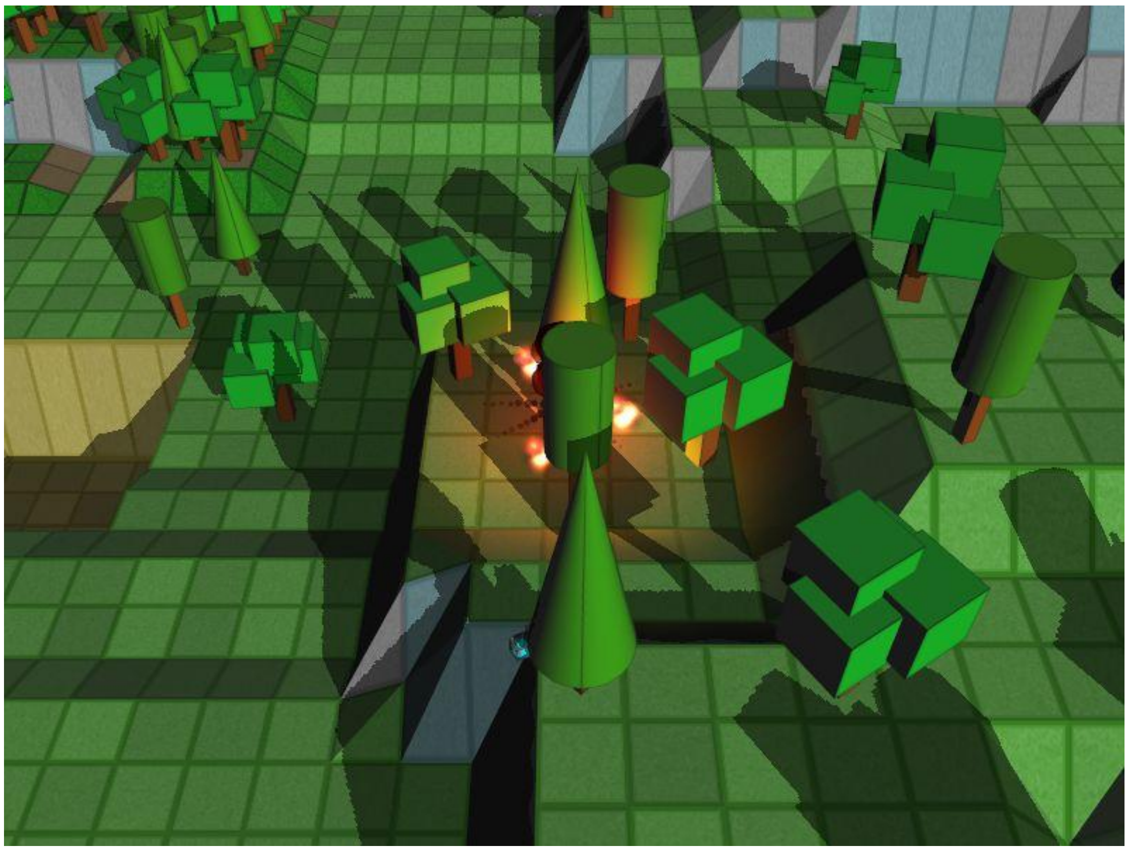


Figure 16. *Yellow and red point lights from explosions and the special attack lighting up the trees*

The third and most complex light source is the spot light (Selman, 2002). Similarly to a point light, a spot light has a position and attenuation, but is instead directional with an angle to determine the lit area. The light is cone shaped, divided into an inner and an outer cone to distinguish the light intensity. Typical spot lights include flashlights, car headlights, and desk lamps.

7.4.2 Types of lighting

In computer graphics, the three most important types of lights are ambient, diffuse, and specular. Ambient light is everywhere; it is the result of the light rays scattering around the world and thus lights objects which are not even directly lit. In computer graphics, ambient lighting is simply a constant of intensity multiplied with color and applies to every object in the scene (Fosner, 2003). It is important since without it, surfaces that are not lit by any light sources would be completely black, as shown in Figure 17.

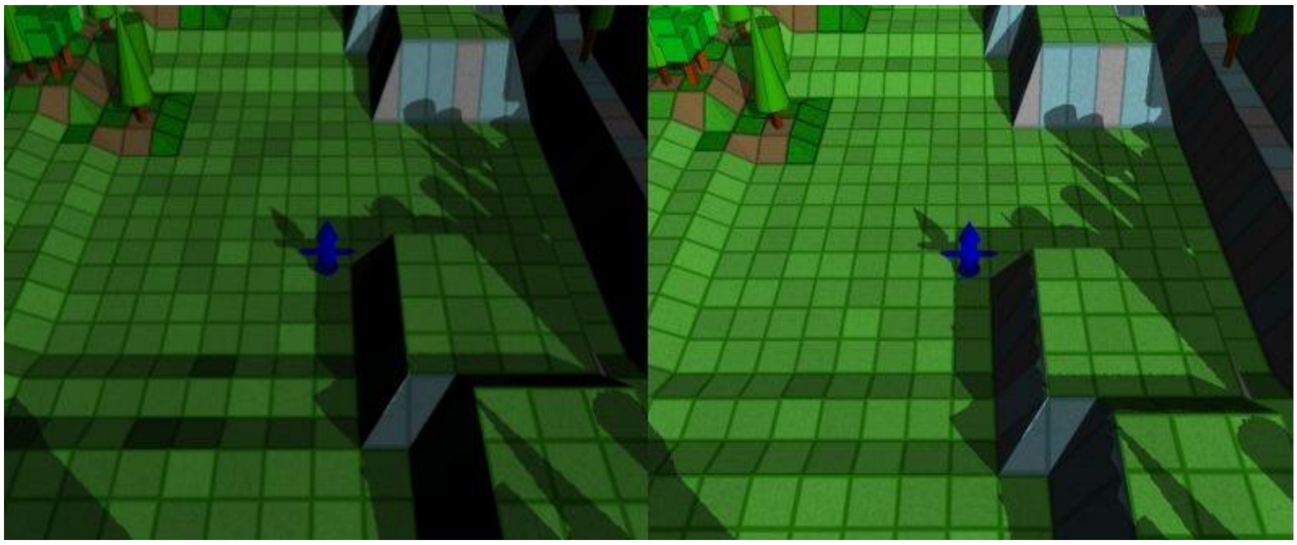


Figure 17. *Difference between ambient lighting off (left) and on (right). Besides brighter overall, non-lit areas are not completely dark with ambient lighting*

Diffuse lighting defines how much a surface is lit up from a light source (Fosner, 2003). Comparing the angle between the light direction and the normal of the surface gives the amount of diffuse lighting for the specific surface. If both the light and the normal point in the exact same direction, full lighting is applied, while if they are perpendicular to each other no lighting is received at all. In the shader, this is calculated with the help of the dot product. The dot product compares the angle between two vectors and returns a decimal value: 1.0 if the angle is 0°, 0.0 if it is 90°, -1.0 if it is 180° and any value between 1.0 and -1.0 depending on the angle. This only applies if the vectors are in unit lengths. Otherwise, the result of the dot product will vary greatly and will not be intelligible. Thus, it is important to normalize the vectors before using the dot product.

Specular lighting defines how reflective a material is (Neider, 1994). Smooth materials like metal have a high specular reflection while rough materials like rock have very low. This is because if the material has perfect reflection, such as in a mirror, all the normals of the surface are pointing in the same direction, and the light rays striking the mirror bounces off at the same angle as they hit with respect to the surface normal. Thus, the light rays are still parallel to each other and the image will be intact as it is reflected to the eye. However, if the surface is rough, the light will scatter in many directions as they reflect on the surface, and no mirror image can be seen.

To simulate specular lighting in computer graphics, Fosner (2003) provides an implementation where four components are needed: the direction of the light, the surface normal, the camera view vector and the half angle. The camera view vector is easily found by subtracting the camera location with the vertex position in world space. The half angle is the vector splitting the light direction and the camera view vector in the middle, and it is computed by adding these both vectors. To calculate the amount of specular reflection, a dot product of the half angle and the normal is performed. The closer the result is to 1, the closer the camera is located to the predicted reflected light direction and the higher the specular reflection becomes. Finally, the value is raised to a shininess factor which is based on the surface material – a lower factor means the material reflects more light. Figure 18 and Figure 19 shows examples of the different lighting.

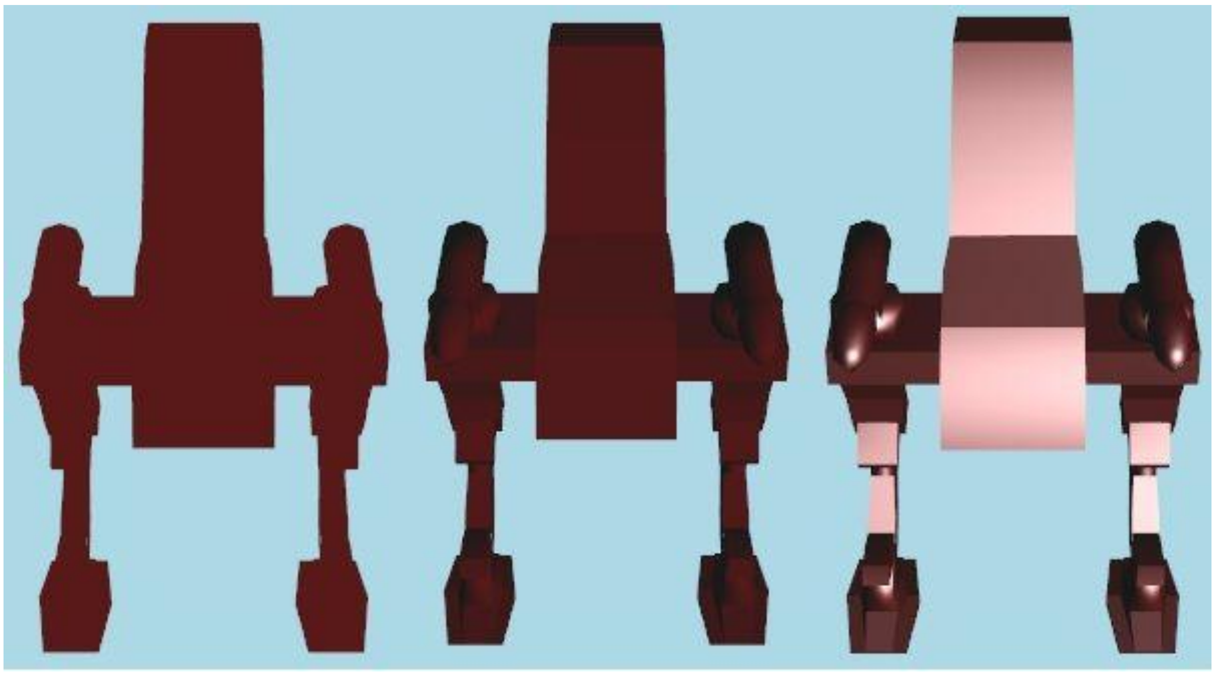


Figure 18. *A robot lit up with ambient, diffuse, and specular lighting*

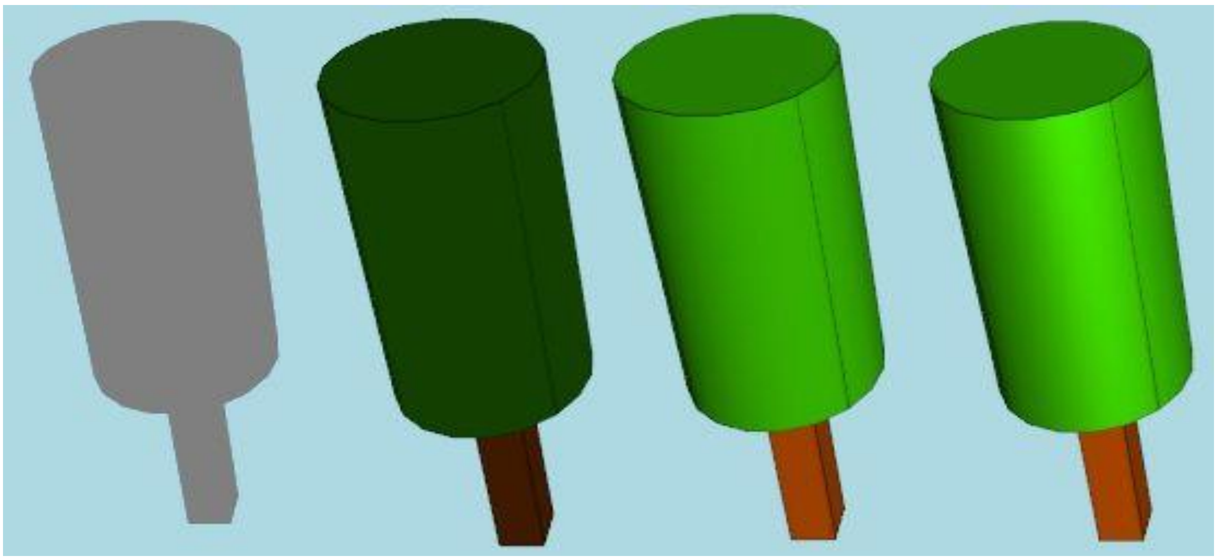


Figure 19. *Showing ambient only lighting, textured with ambient, diffuse, and specular lighting*

7.4.3 Rendering techniques

There are a few common lighting techniques used in computer graphics that each has their advantages and downsides.

In single pass lighting, all lights are applied to every object which is rendered (Hargreaves, n.d.). It is a simple method that is easy to implement, and it is good for scenes with a small number of lights. However, single pass rendering is difficult to organize if there are many lights, and single pass rendering easily overflows shader resource limitations, since every light has to be stored and calculated in the shader although not all of them may actually affect objects.

Multipass lighting means splitting up the rendering into multiple passes, instead of doing everything at once like in the single pass technique. For instance, the diffuse and specular reflection from a directional light could be rendered in the first pass, followed by the lighting reflections from a spot light in the next pass, and so on. The advantage is that the lighting calculations only have to be evaluated for those lights that affect an object, and because this approach is modular, the developer has better control over the rendering process (Akenine-Möller, Haines and Hoffman, 2008, pp.278). However, an object needs to be processed multiple times by the vertex shader, and the technique requires a lot of memory bandwidth.

The third and most recent rendering technique is deferred shading. Essentially, deferred shading separates the lighting rendering from the geometry rendering. This means that all geometry is rendered first and all information required from the geometry is saved in different buffers (Filion, McNaughton, 2008). The lighting can then be rendered in a separate pass, with all the required geometry information obtained from the buffers. Deferred shading is able to render many lights without much performance impact, and it works well with post-process effects. Some of the downsides of deferred shading include the difficulty to render transparent objects, and high performance and memory requirements

7.5 Shadow mapping

Shadow mapping is a technique used to simulate shadows cast by objects. The shadows are drawn by first rendering a *depth map* of the scene from the light casters point of view. A depth map is an image where each pixel of the image contains *depth* information, that is, how far away the pixel is from the point of view. This depth map is then used when rendering the real geometry to decide whether a pixel is lit or not. Pixels not lit lies in shadow, and will thus have their color darkened (Brabec, Annen, Seidel, 2002).

7.6 Particle effects

Particle effects is the term used for phenomena which are very hard to reproduce using conventional rendering techniques. Fire, explosions, smoke, magic, rain, grass, and sparks are all examples of particle effects, as shown in Figure 20. These are often simulated as semi-transparent images which always face the camera, better known as billboards (Reeves, 1983). It is however not necessary for billboards to face the camera at all times, grass should for instance only face the camera along the x- and z-axises. It is a cheap and simple way of representing particle effects, and if done correctly, the player usually does not notice that the effects are simple 2D images.

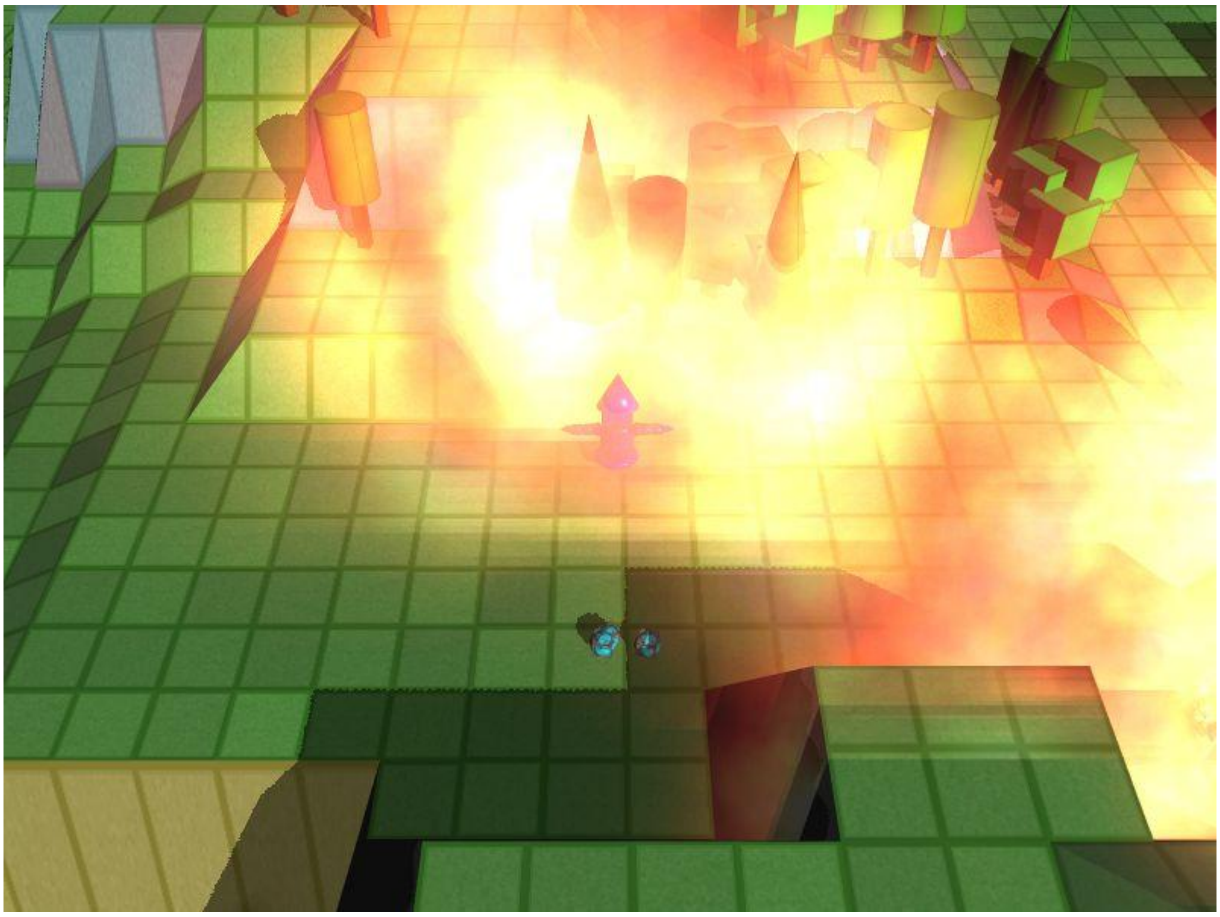


Figure 20. *Large explosions generated as billboards*

A particle system is normally responsible for updating and rendering particle effects. The position of the particle system in 3D space is referred to as the emitter. This is where the particles will emit from in this particular particle system, and it may have properties such as particles spawned per second, initial and ending velocity, lifetime, color, and so on. It is common to add a randomized value to these properties, in order to ensure that not every visual effect is predictable and looks the same. The particles are then transformed, scaled, and rotated according to the settings specified, until their duration has expired.

7.7 Our graphical approach

XNA includes a simple shader which provides basic functionality like ambient, diffuse, and specular lighting, as well as three directional lights. However, the shader is fixed, and therefore we decided to create a custom shader to be able to add any further shading techniques as well as to have more control over the shading code. Even if we had to rewrite the lighting calculations provided in the built-in shader, we gained theoretical experience in doing so. All transforms were processed by the vertex shader, while lighting was handled by the pixel shader. This is because per-pixel lighting is standard today, as modern graphics cards are powerful enough to easily handle it, and it also results in better looking shading.

One directional light was implemented. Since our game was taking place outside, it was acting as the sun. Because we used a fixed camera, the light was set to point in the same direction as the camera (although tilted to the left), lighting up all objects facing the camera.

Aside from this directional light, several point lights were implemented. When adding a point light to the game, the programmer can specify properties such as position, range, power, color, and duration. Linear attenuation was used in the lighting calculation, since this is a common attenuation used in games and the team decided it looked good in the game. The maximum amounts of point lights used simultaneously was limited to three, because of shader resource limitations in Shader Model 2.0. Therefore, point lights were only used in explosions, acting as a quick, yellow flash, and for the special attack, as a longer lasting, red light. This made it possible to still have point light flashes for every explosion, as it was unlikely that four explosions would trigger at the same time.

In order to render shadows in our game, we implemented the shadow mapping technique. Only the sunlight was used to cast shadows, as it would be very computationally expensive to have multiple lights casting shadows, and we felt that the shadows cast from the sun would be more noticeable than other shadows in the environment.

Ambient, diffuse, and specular lighting were implemented. The ambient lighting was a single intensity value specified in the application, affecting every object in the scene, and the color was fixed to white. Diffuse lighting was calculated for all objects and every light affecting them, while specular was only calculated for the sunlight. Although the shininess of the object and the specular intensity could be specified in the application, only two models were hard coded to use specular lighting: the wizard, and the mechanical walking robot. This was because the specular lighting would not be visible for most objects, but also because specular lighting was implemented very late in the development process, as it was never really prioritized. However, since the wizard was always visible on the screen, and the mechanical walking robot was huge compared to the other objects, they were given a bit of specular lighting.

Textures were also used, although it was made optional in the shader, as not every model had textures applied to them. Support for skeletal animations was implemented in the shader using a separate shader technique because of the extra animation information required as input to the shader. The rendering technique used was single pass lighting, with all the lights summed in the shader to receive the final lighting color. Although deferred shading is becoming more and more popular, we did not plan to use that many lights to make it worth implementing. Also, by working with the most basic technique, we gained fundamental theoretical knowledge, since the techniques are not that similar.

A particle system was implemented to handle all the particle effects. The particle effects were crafted by hand, although it was possible to customize a few parameters in the application, such as scale and velocity. To manage all these types of particle effects, a particle manager was built, to handle the general additions, removals, and updates of any particle effects. For one time effects, like an explosion, 30 particles were simply added at once at the specified position, while for moving objects, such as a projectile, a particle emitter was bound that output a certain amount of particles per second. As for the particle system itself, it was implemented mostly in a dedicated shader. The application was responsible for adding new particles, keeping track of active particles, and freeing particles not used anymore. A few settings were specified for the particle effect, such as direction, rotation, velocity, texture, start and ending sizes, and then any active particles were sent to the GPU. All the transformation, rotation, and scaling calculations during the lifetime of the particles were therefore handled by the GPU. Since our game was supposed to be an RTS game with many units and AI potentially taking up a lot of CPU resources, we decided not to use a CPU based particle system.

7.8 Visual results

The overall visual result looked good without many problems encountered. Developing the graphical part of the game consisted mostly of theoretical reading and research, since the team was completely new to 3D graphics. Although only well-known methods and techniques were used, a basic understanding was needed to put them all together. However, one problem was that most sources and examples were based on XNA 3.0, since XNA 4.0 was fairly new when the development started.

Because of time constraints and little prior knowledge of graphics programming, there were not many advanced graphics techniques used. No real textures were really made for the objects, instead they were using solid colors, which rendered advanced texturing techniques pointless. A few particle effects were created, including explosions, explosion smoke, factory smoke, and special attack effects. However, it was hard to create additional ones since it was not known what kind of effects would be useful and how they should look. The same could be said about point lights; although they were implemented into the game, their usefulness is doubtful. The positive side of the simple graphics is that the game runs very smooth without any frame rate issues.

7.9 About the look of our game

Most of the problems explained above were caused by the lack of proper visual design; we never seemed to come to an agreement on the visual style, plus there was not enough time to build the game we wanted gameplay wise. Without a sound gameplay design it is also very hard to create a good-looking visual style. Although the game ran well on a desktop PC, we never tested the performance on the Xbox 360. The impact of running a game on different graphical hardware was thus never seen, and any alternative graphical implementations or attempts of hardware optimization were never made or tested.

For a better visual result, more people would have had to be involved in the graphical part, but it is hard to accomplish this within a small development group. After all, a game needs to be made too, and a 3D RTS game requires a lot of effort put into fields like AI and collision detection as well.

Chapter 8

Modeling and animating

8.1 The creation of all objects

Drawing an object, such as a monster, can be done in XNA, but it would be very difficult and time-consuming. This is mainly because each single point (vertex) would have to be specified in the code, and perhaps thousands of triangles would be needed to shape the monster (Reed, 2008). The solution to this problem is to use a 3D model, and there are a lot of different modeling tools available for creating and animating 3D models. Reed (2008) mentions that some of the most common applications for developing 3D models and animations for video games are: Autodesk Maya (Autodesk Maya, 2011), Autodesk 3ds Max (Autodesk 3ds Max Products, 2011) and Blender (Blender, 2011).

Alexandre Lobão (2009) describes a 3D model as a hierarchy of meshes that can be rendered independently. A mesh is a collection of points, edges and faces that specify how an object in 3D computer graphics should look like. Models are, as mentioned earlier, often created outside of XNA in a third-party modeling application and can also store extra information such as textures, colors, and animations (Reed 2008). By using these third-party applications, more advanced 3D models were able to be used than if the models were created inside XNA (Lobão, 2009). Figure 21 shows a 3D model created in Maya by the group members. The model is a character used in the game, and is called the minion.

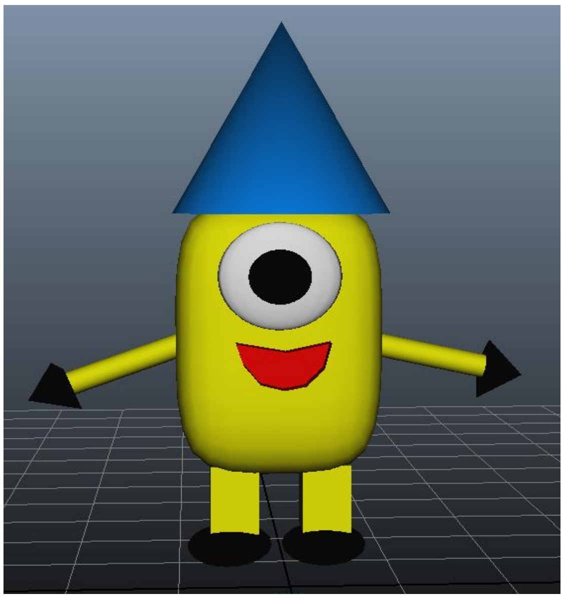


Figure 21. *The final version of the minion used in this project*

8.2 The movement of characters

There are several ways to make the objects in a game move, and one useful approach is to animate them. Animations can be done inside XNA by, for example, rotating part of a mesh over its axis. When it comes to more complex animations, such as moving a character and making it jump, the process becomes more difficult (Lobão, 2009). For achieving these types of animations, Maya was used.

An animation is composed of different frames, where each frame represents a specific pose of the model. Each frame has a time offset that decides when the model changes its pose. Two main types of animation techniques will be described, which are keyframed animation and skeletal animation (Lobão, 2009).

Figure 22 illustrates a shooting animation of the main character in the game. The animation sequence consists of several frames, where each frame has a different set of configurations. The last frame has the same pose as the first one, which makes the character's animation loop.

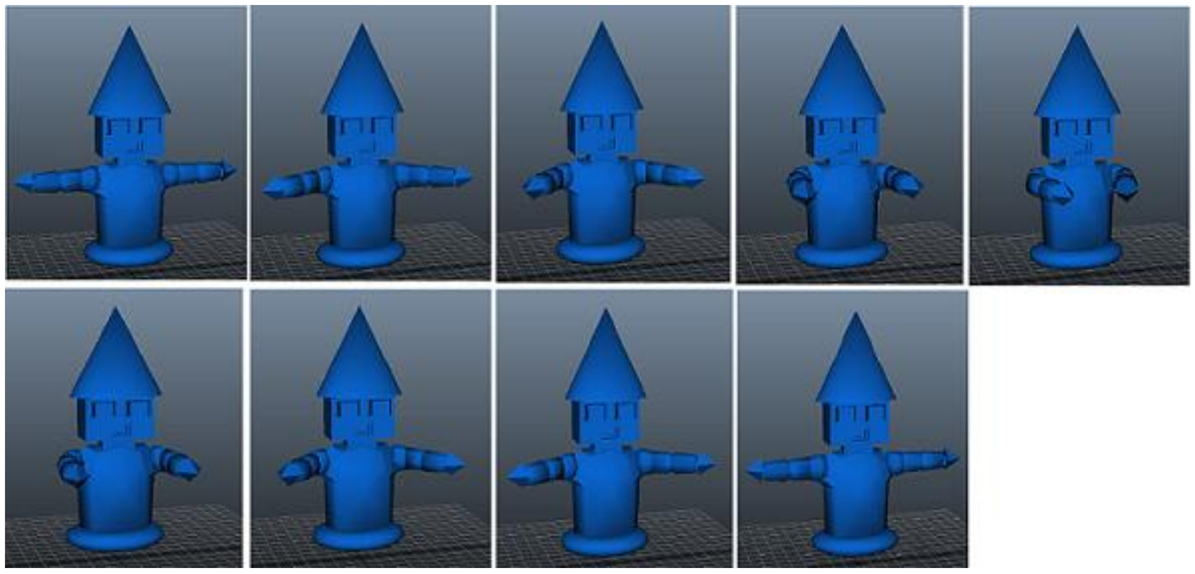


Figure 22. *Main character with a shooting animation*

8.2.1 Keyframed animation

Keyframed animation is a technique where the animator changes the position of an object on the screen and takes a keyframe, which resembles taking a snapshot, of every significant moment. The software used for animating the object usually interpolates the frames between these keyframes and creates an animation sequence.

According to Sean James (2010), one of the advantages of using keyframed animation is that it is a fast way for creating complex animations because each frame does not need to be animated.

The best way to demonstrate this technique is by showing an example. Figure 23 displays a simple animation sequence where a ball bounces once while it changes its position. The ball has three keyframes and each one is set on a different time and has a different position.

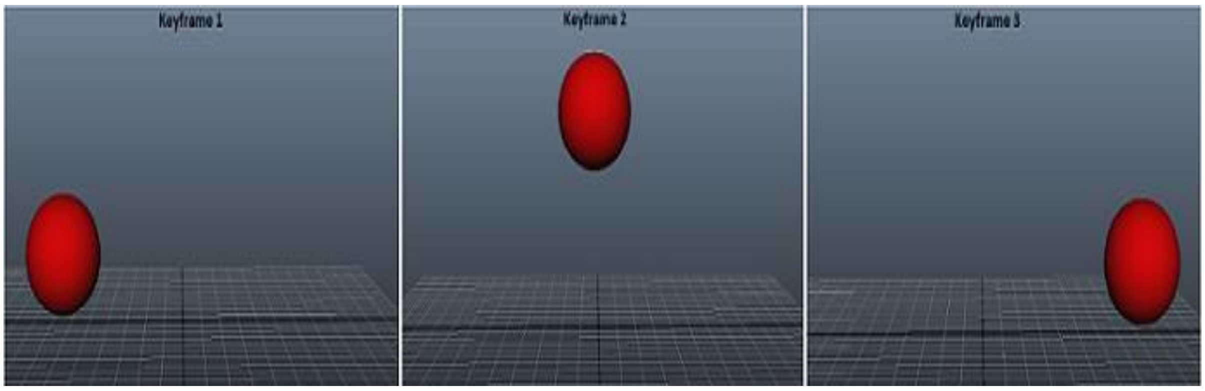


Figure 23. *Ball animation with three keyframes*

8.2.2 Skeletal animation

In this technique, the 3D model is bound to a skeleton. The skeleton is a set of bones, one bone for each movable part of the model, which are connected to a root bone (Lobão, 2009).

Skeletal animation usually uses keyframed animation for actually animating the model. This is used by defining different positions and configurations of each bone at specific frames and taking a snapshot of them. The skeleton of a model are generally built as a hierarchy of pieces which means that when changes are made to one part they will also be reflected in its child bones. For example, moving a characters arm will move its hand and fingers as well (James, 2010).

Lobão, A.S (2009) states that skeletal animation has many advantages compared to keyframed animation. According to Sean James (2010) skeletal animation is useful when animating, for example, characters because it makes it easy to simulate movement of the skin in a more smooth and flowing way.

An example of how this method is used is shown in Figure 24 which displays a robot created for our game. The Figure shows the same robot in two different states. The first one illustrates the robot without any skeleton, whereas the second one shows the model with its skeleton included. As noticed, all the bones are connected to a root bone and have a hierarchical structure.

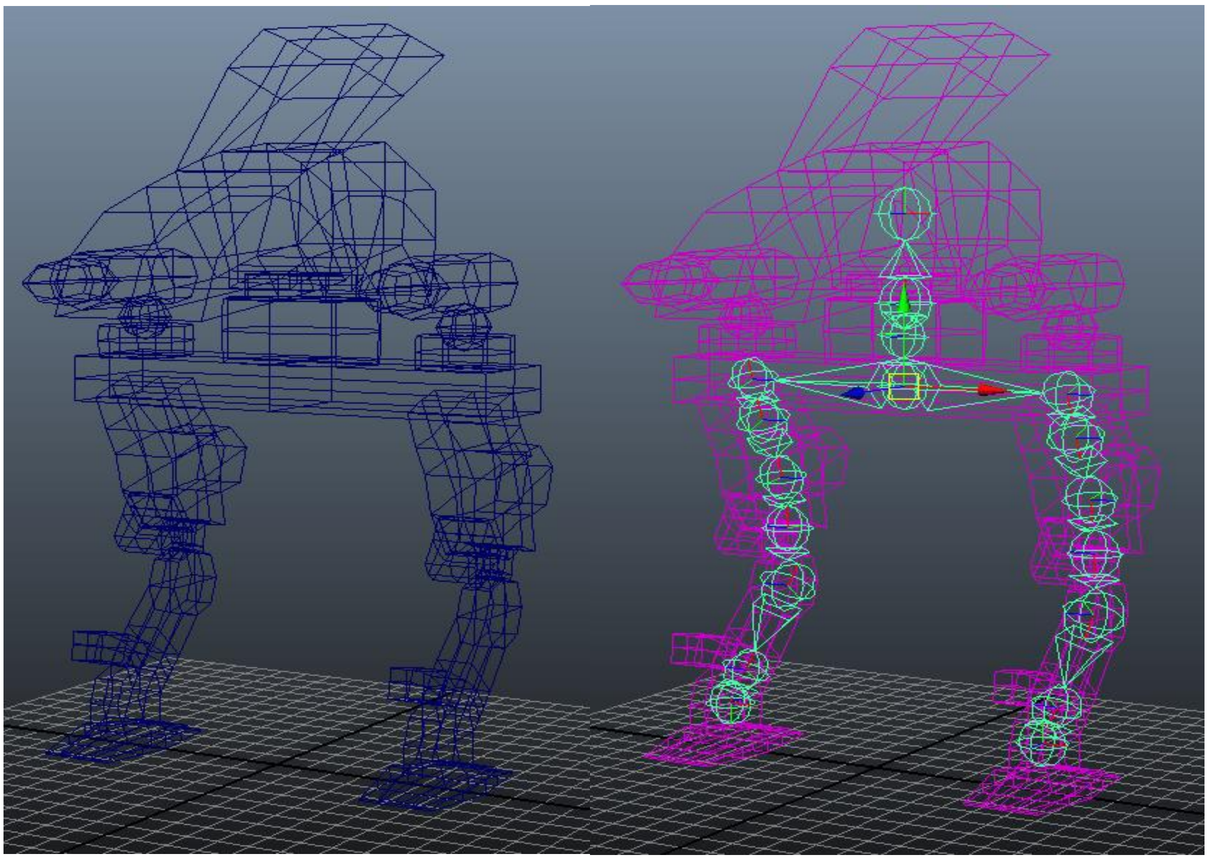


Figure 24. *Comparison of the robot with, and without bones*

8.3 Our approach for creating 3D models and animations

We used third-party modeling applications to be able to display and animate much of the graphical content in our game. There are, as mentioned earlier, various software products for modeling and animation available on the market. We decided to use two of the most common ones, which are Autodesk Maya and Autodesk 3ds Max. We chose these two because they are the ones that appeared most popular according to different Internet forums. Another important factor in our decision was that both Maya and 3ds Max did have a free student edition available, which we used.
By choosing to create the models ourselves we faced some advantages as well as some disadvantages. One advantage was that we had the flexibility to create the models as we desired. We could also easily make changes to the models and add new features to them. On the other hand, we had to spend a lot of time on learning how to use the software and creating the models and animations. This was mostly done by reading and watching many tutorials on different websites as well as doing a lot of trial and error while using the software.

It was decided in an early stage of development that the models and animations created should be quite basic and primitive. This decision was made mainly for two reasons: the first one was that no one in the group had previous experience in the field and the second reason was that the team decided to have a more simplistic graphical style for the game. Once some of the 3D models were created, such as characters, the animation process began. All animations were created in Maya and we mostly used skeletal animation as our main technique since we had an animation engine that could handle these types of animations.

8.4 Modeling and animation results

Most 3D models were created without any major problems, but this cannot be said for the animations. A lot of difficulties were encountered with the creation of the animations as well as the implementation process into the game. This was mostly the animation engines fault since it had some difficulties handling animated models.

XNA supports 3D models by default, but it does not support animated 3D models. This meant that an animation engine had to be created. While the animated characters were working as intended in the modeling software, they did not display and act correctly in the game. This meant that the animator had to adapt the whole animation process to the animation engine. By testing and altering the animation engine, the group managed to make it work in an adequate way, although not optimal. This meant that many test objects had to be created to see if the way the characters were animated would be shown correctly in the game.

The animating process in Maya was quite difficult as well, especially when trying to create more complex animations for more complicated models. An example of this is the robot in Figure 25, which was very difficult to animate since the model is more complex compared to other models created for the game. It has a lot of bones, which makes the animation more flexible but at the same time more complicated. Because of this, it is also more difficult to create a good-looking and smooth animation sequence. The final result of the animation was deemed adequate. More realistic movements would have been preferred, but this was too time-consuming. On the other hand, the 3D model of the walking robot was considered satisfying.

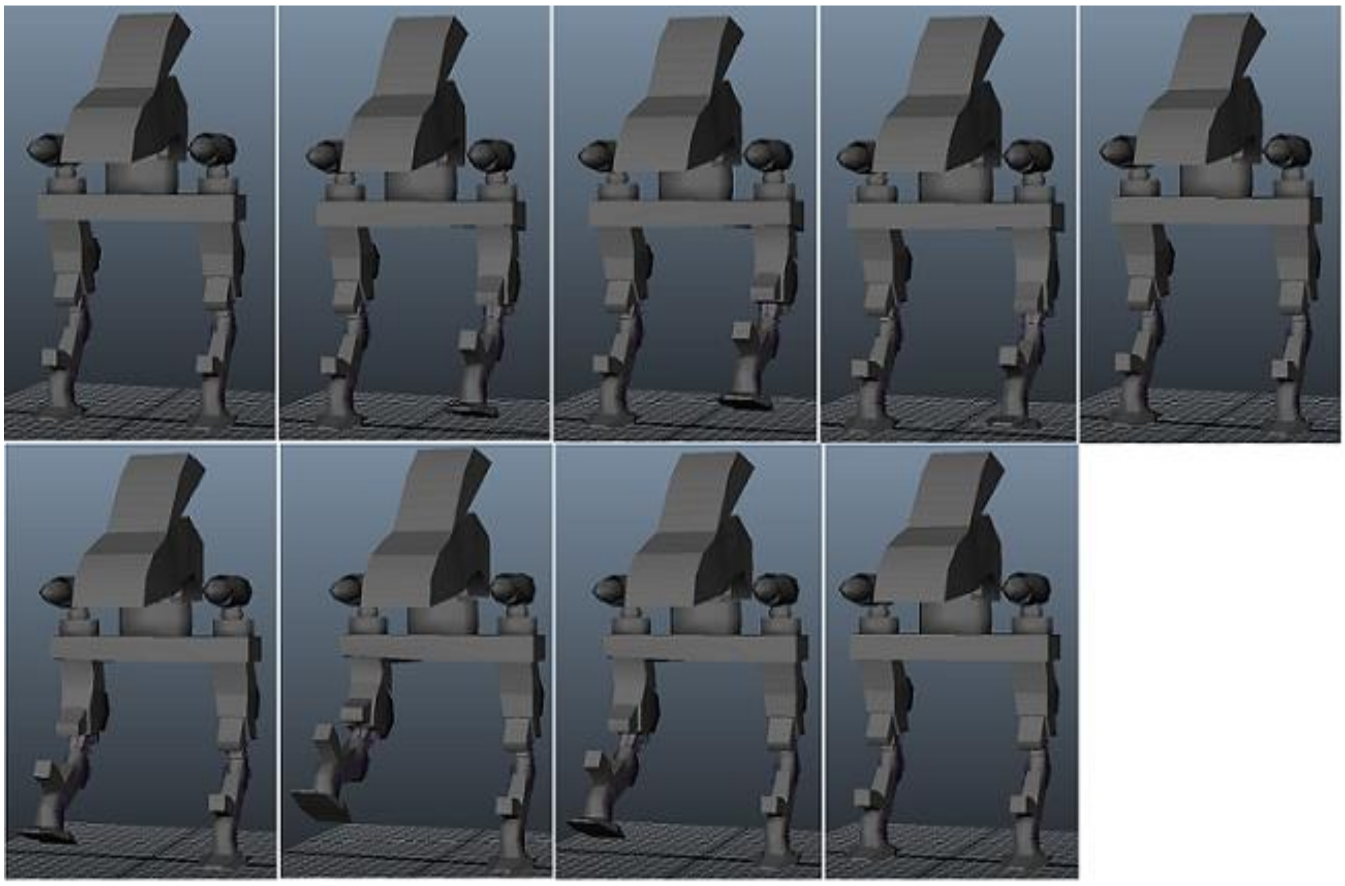


Figure 25. *Animation sequence of our robot*

Most models, such as the buildings shown in Figure 7, were created without any major difficulties and, since a more simplistic design was desired, the group was very satisfied with them.

The main character was hard to create because all group members had different opinions on how he should behave and look like. This lead to many draft versions until the group found one that suited their needs. Once the model was finished, the whole animation process began. Creating many different animation sequences, and a few complex ones, was desired but after a while it was realized that it would be too difficult and time-consuming. Finally, only three animations were made. The first one is a shooting animation, as shown in Figure 22. The second one is an animation where the main character does a special move in which he spins around with his arms wide open as illustrated in Figure 26. The final animation created is an animation where the wizard bounces up and down. This last animation sequence in this report is not shown since it is very hard to notice the bouncing effect on a still image. In addition, several other animations were made for the main character but they were not used because they were either unnecessary for the game or there was not enough time to implement and make use of them in a sensible way.

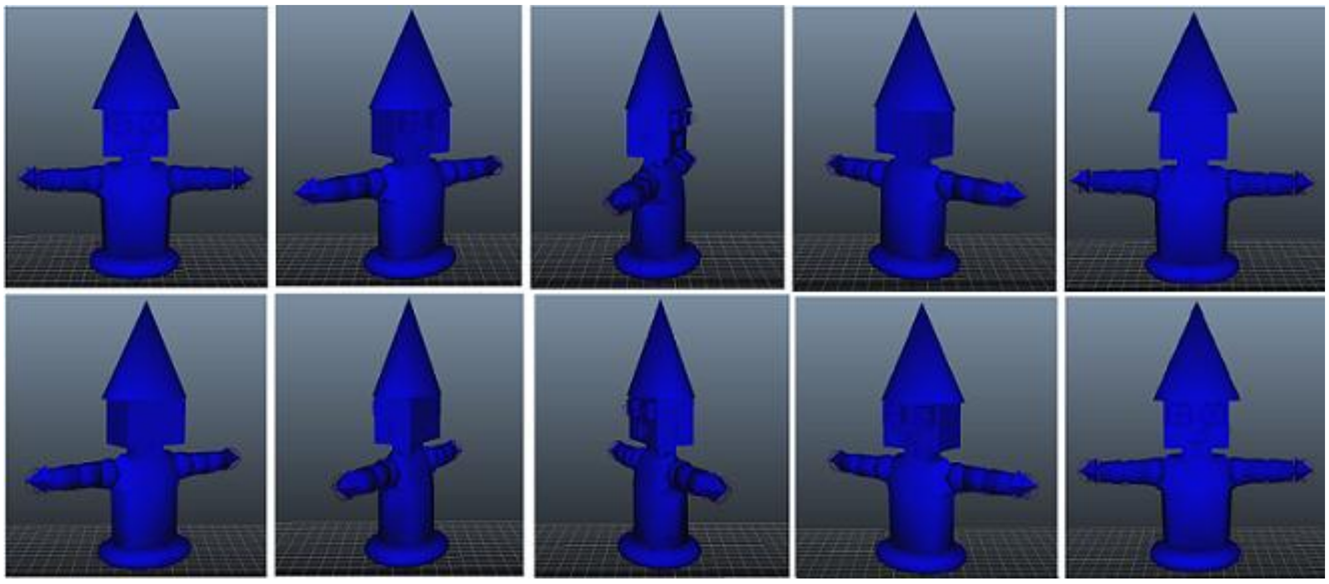


Figure 26. *Main character doing his special move, which is spinning around in a circle with his arms open*

One of the characters created for the game was the minion. This character is shown earlier in Figure 21 and the result of the model was very satisfying. Unfortunately, there was not enough time to create any animations for this character.

Since none of the characters or objects was designed before the actual development started, many 3D models were created in vain because they were not used in the game. A lot of time was spent creating these models and animations which could instead have been spent on other parts of the development.

Many of these points sound very negative, but the truth is that the group was pleased with most of the results, and even though a lot of problems were encountered on the way, a great deal was learned and the team now knows what works and what does not.

8.5 Discussion about the modeling and animation process

The overall experience with modeling and animating was positive, even though many difficulties were encountered. One problem that occurred during the entire project was that the modelers and animators did not know how, for example, a particular character should look like and behave. This made the modeling process quite problematic, since it is pretty hard to create something in 3D if there are no reference images to base the model on. Figure 27 shows four examples of models that were created but not used in our game.

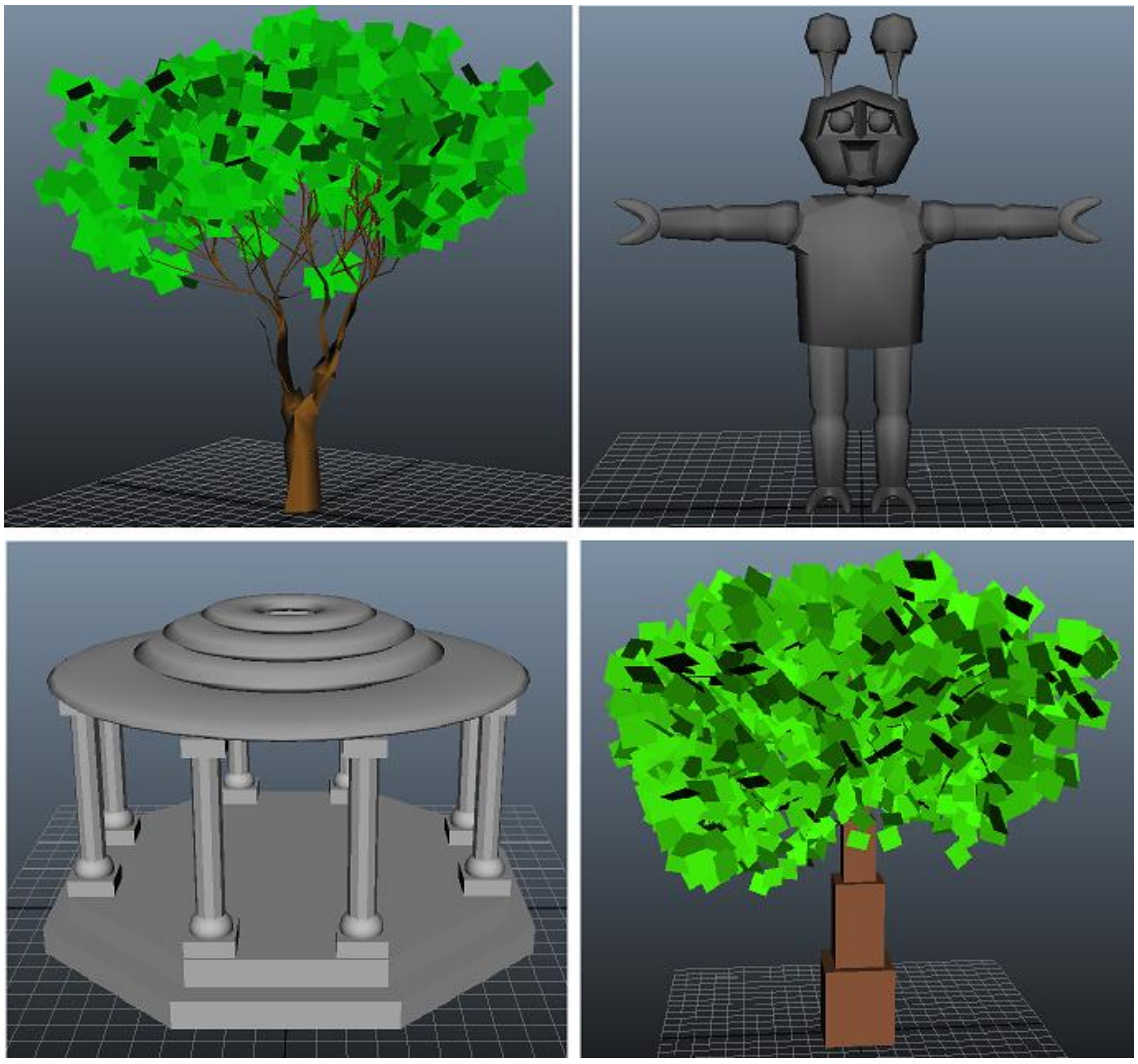


Figure 27. *Example of models that were not used in the game*

As mentioned earlier, the time it took to create these models could have been spent on other parts of the game or on improving models and animations already used in the project.

One solution to this problem would be to have a concept artist in our team. A concept artist would be responsible for drawing all the characters and objects, meaning that the modeler would have an image reference. The concept artist would not necessarily be very artistically gifted, as the purpose of the images created would be to create a holistic visual design. If the design of all the main objects had been decided in an early stage of the development process, the whole modeling procedure would have gone a lot smoother. This would have given the group something to refer to and compare with while creating the models.

Spending more time on tutorials on how to animate in Maya would have been preferable since this part was more difficult than creating the actual models. At the same time, this would probably have been too time-consuming.

The overall conclusion of modeling and animating, although many difficulties were encountered, was that the team gained a lot of knowledge and managed to develop their skills in this new field. In a future project the group now knows that planning, designing and communicating is an essential aspect of creating 3D content for a video game.

Chapter 9

Collision detection and collision handling

9.1 Problems of collision detection

Collision detection is the task of detecting when and where objects in a game are interacting with each other. When the collision detector has discovered an intersection between objects, the task of deciding what do with each object is called collision handling. Collision detection and handling is executed by the game engine.

There are two core problems that have to be solved when designing a collision detection system: first, there is the computational complexity of treating many objects. This will be referred to as the broad problem, and generally this is solved by something called the broad phase. Second, there is the task of making accurate detections. This will be referred to as the narrow problem, and the algorithm that performs this calculation is generally called the narrow phase (Hubbard, 1993).

9.2 The narrow problem

The basis of the narrow problem is to find out if two objects collide, and occasionally find additional information about the collision. An example of additional information that may be of interest would be the precise moment the collision occurred, another would be the exact location in space where the objects first touch. Neither of these is usually retrieved naturally from the detection test. This is because games operate on discrete time, with the consequence that when collision is detected, the precise moment and location of the collision has already happened (Bergen, 2004). Furthermore, game objects can be quite complicated as a result of attempting to represent a physical object, and thus it is a complicated and computationally heavy task of figuring out if they collide or not. Because of all of these demands, the narrow problem of collision detection is a difficult and slow process with much research dedicated to it (Mirtich, 1997).

9.3 The broad problem

An RTS game world usually contains a significant amount of objects. In order to know if some of these objects collide, an intersection check must be performed. This is done by comparing the objects in pairs. In the broad phase, potential collisions are identified and are further evaluated in the narrow phase. Thus, to know if a given object can move freely through game space, this object needs to be compared to every single other object in the game. Since this is required for all objects, the complexity for calling the narrow phase lies in $O(n^2)$ (Šinjur, 2001). The broad problem of collision detection is the task of reducing the number of calls to the detection algorithm (i.e. invoking the narrow phase).

9.4 Broad problem research

9.4.1 Sweep and prune

One solution to the broad problem is using a technique presented by Baraff (1992), called sweep and prune. All objects are placed in a list. For each collision detection pass, sort all the objects according to their in-game position along one axis. Then, the list is iterated, and for the current object the closest neighbor is retrieved. If the two objects intersect on the specified axis, they are passed on to the narrow phase. If they did not collide, then the next closest neighbor is fetched. Once the current object finds objects on either side along the axis that does not intersect on this axis, it does not need to check further as all other objects will be farther away (Terdiman 2007). According to Bergen (2004), sweep and prune like algorithms have a worst time complexity of O(n * log n).

9.4.2 Uniform grid partitioning

The idea behind grid partitioning is to divide the game space into a number of isolated distinct areas. Then, each object in the game should reside in those areas that it overlaps with. Optimally, the ratio between object size and area size is such that each object resides in as few areas as possible, yet each area contains as few objects as possible. The benefit of grid partitioning is that while the complexity within each area is $O(n^2)$, given n objects in that area, the overall complexity will only grow linearly for each new such area added (Šinjur, 2001).

9.5 Narrow problem research

The straightforward method is to take all the triangles of the model of the object, and for each triangle check for intersections against all the triangles in the model of the other object. This method is accurate, but has a high complexity of essentially $O(m \times n)$, where m is the number of triangles on one object and n is the number of triangles on the other object (Šinjur, 2001).

A simplified strategy is to define one or several invisible bounding volumes around the model. Then, these primitive bounding volumes provide faster, but less accurate, intersection checks (Konečný, 1998). If better accuracy is desired, the collision test can continue with either more detailed bounding volumes (Haverkort 2004), or doing the full model test.

9.6 Collision response research

Collision response is usually divided up into two distinct phases (Burns, Sheppard, n.d.). The first phase consists of separating the overlapping objects from each other. Common solutions to this are the *projection* method, the *binary search* method, or just using the position of the objects before the collision.
The *projection* method can only work on very simple shapes. It separates the objects by the same length as the intersection depth, which requires that this depth can be calculated. For more complex shapes where no such calculation can easily be done, the *binary search* method is usually preferred. It searches for the moment just before the collision, by using increasingly smaller timesteps (Yan, Chen, Pa, 2005).

The second phase concerns giving the objects new velocities. Even if the collision response is supposed to mimic physics as much as possible, the reaction still varies depending on what kind of objects that are colliding. One extreme reaction is the fully elastic reaction. In this case, no kinetic energy is intended to be lost, and the result is that the involved objects will bounce off each other. The opposing extreme is the completely inelastic reaction, in which all kinetic energy is lost, causing both objects to lose all velocity in the direction of the collision (Baker, 2010).

9.7 Our collision detection and response method

Because the intent of our game design was that there should be many objects interacting with the world at once, and because the camera was set far enough away from these objects that it becomes difficult to see finer details or correct physics, our collision detection focus was set almost exclusively on the broad problem. Since the world in which the game takes place is a large flat land, we decided to use mostly two-dimensional collision detection. Thus, we did not need to deal with such strange and unusual situations such as when one object is resting atop of another object.

Our choice for the broad problem was uniform grid partitioning. Initially we considered using sweep and prune, but decided against using it because of its higher complexity. In his 2004 book, Bergen argues that the algorithm performs almost linearly when there are few moving objects. However, since it was our intention to have a lot of moving objects, we did not feel that this benefit would have helped us much.
For doodads, the decorations of the world such as trees, rocks, and other terrain embellishments, a special function was implemented. Because doodads never move, and are represented as a part of the static game world, it is easy to find which doodads that are within range of any given location.

To tackle the narrow problem, we selected different approaches depending on what objects that were being compared. For all objects except those which were flying, a simple 2D range check was considered sufficient. For flying objects, such as projectiles, we used 3D bounding volume comparisons.

For the first phase of collision response, we used only the pre-collision positions. The main reason was simply that the camera distance was too great and the models so small that any positional improvement would hardly be noticeable. The associated performance load with Binary-Search made it an even more unfit solution.

As for the second phase of the collision response, we ignored trying to emulate regular physics, and designed a customized response that we felt would fit better in the game. Elastic behavior did not seem natural to our game at all, and the inelastic calculations become very complex when many objects cluster together tightly. Further, we wanted more specific control over how objects would try to move around arbitrary obstacles, and thus, a hand-tailored solution fit well. The strategy we took to handle this problem was to search for a new possible movement using increasingly angled versions of the original movement.

To summarize, the *collision manager* was organized in the following way: every object has a position and a *destination*. The destination is the position which this object has intended to move to. The collision manager then moves one object at a time to its destination. If a collision occurs at this new destination, the collision handler is called. The collision handler moves the object to a new alternate destination, and the collision detector is called once again. If this new destination also causes a collision, the collision handler is called once again, and this kind of back and forth iteration goes on for a while. Eventually, either the collision handler finds a suitable destination which does not cause a collision, or it gives up and retracts the object back to its original position.

9.8 Results of our collision detection and response

Over the course of the development of the game, the design was continuously changed to contain decreasing number of objects. This cut out a significant portion of the usefulness of uniform grid partitioning over sweep and prune. Internal tests showed that uniform grid partitioning still performed much better than not having any algorithm to tackle the broad problem at all.

A problem that arose with uniform grid partitioning was that the memory required for each square depends on how many game objects that currently reside within each square. When executing the game, we found that most squares were almost or completely empty, while a few squares needed to keep references to a large number of objects. To avoid making new allocations during runtime, each square was required to have sufficient memory to be able to hold as many objects as could possibly occupy it. This meant that uniform grid partitioning required much more memory than we had initially expected, while most of this memory remained unused. This problem is not as severe on the PC as it is on the Xbox 360, as PCs have better support for garbage collection and consequently allows memory allocations without as much of a decrease in performance.

A mistake we did early in the process was that we planned on using the same collision detection pass for both solid collisions and for game logical interactions. As we developed the project further, it became clear that these two kinds of interactions were too different from each other.

The fact that the collision handler may be called several times for a single object was a reason that made merging with the logical interaction code unsuitable. The logical interaction code should never execute more than once per game cycle, and thus additional bookkeeping would be required. Furthermore, the collision handler would have required the ability to distinguish solid objects from logical objects and would also be required to dictate how the *collision manager* should behave.

9.9 Thoughts on our collision detection

If we had stayed with a set gameplay design, the collision detection algorithms we would have developed would not have lost effectiveness due to a changing specification. On the other hand, uniform grid partitioning is a flexible algorithm that worked well with our alternative gameplay design, and would also do so in case we wanted to go back to our original design. It is also possible to extend and optimize the algorithm for future needs.

Chapter 10

Results

Sadly, the game was not implemented to a degree that was planned. This was mostly due to time constraints, and to some extent to other events impeding progress. However, a lot of theoretical and technical experience was gathered, as presented in this report.

Still, the game reached a playable prototype stage, with much of the basic functionality implemented. It may be argued that the game is not feel much like an RTS game at this point. C♯ and XNA assisted greatly in creating the game and made many parts of development easier, such as managing sound with XACT. However, since we never optimized the memory management for the Xbox 360 garbage collector, this restricted the use of the prototype to the PC architecture. Releasing the game for the Xbox 360 will require some adjustments, and will therefore be a future goal. The built in game engine features of XNA were not used extensively, as they were deemed to simplistic. The resulting game engine was however not very advanced compared to commercial engines.

The components of the game were quite simple. The AI system was designed to be simple in order to not take up a large amount of computer resources, but the AI system is also very flexible. The rendering effects were also quite simple, but worked well, especially since the models are pretty basic. Because of the difficulty of learning how to animate models and the troubles with the implementation of an animation system, only a few animations were finished. There were also a lot of time spent on a collision management system that changed a lot throughout development. In the end, the collision management system was efficient and robust.

Because of time constraints and other circumstances around the way we worked, communication and documentation often suffered. Clear visual direction from a concept artist would have simplified the process of creating models, and a clear game design could have minimized the amount of discussions about details on the design. The ambiguity of the game design also hindered the development of a control scheme.

Chapter 11

Discussion

As the project is finished, we stand with a fairly stable game engine, and newly gained experience in developing games in XNA. Due to time constraints we prioritized the functionality of the game. The project as it is now is not as much of a game as it is a base for continued development. Should we choose to continue working on the game there should not be many problems in realizing our visions of the final product that we originally intended. The code for nearly all features and goals of the project exist, and has been tested, but not everything is implemented as it would be in the final product. As an example, we have full support for the Xbox 360 gamepad, but aside from the implementation of an avatar to adapt the RTS genre, we have not yet included some of the RTS-centric features such as issuing orders to minions.

As the group as a whole is both interested in and experienced with video games, we have a number of interesting ideas regarding game elements, and opinions of good design that we could try out in order to make a product that we are satisfied with, and hopefully release to the public in the future.

One of our original intents was to release the game on Microsoft's Xbox Live Indie Games, but since we focused more on functionality, and stability on PC, we did little work to port it to the Xbox 360. Even though code written in XNA is supposed to work on both platforms, the differences in hardware make it necessary to rework some parts of the code to better suit the Xbox 360; the garbage collection is one of these parts.

With more media focus on small-scale game development, as well as digital distribution becoming more common, there is also an option to release the game for PC. This could however lead to a redesign in control scheme, since all PC users do not necessarily own a gamepad that works for the PC, which would defeat the purpose of the gamepad-centered design.

Early in the project, when designing the game we worked very democratically, and discussed every aspect that was to be implemented. Since we all had unique opinions of what was important, most decisions took more time than anticipated. It was not until later that we agreed to disagree and appointed a lead designer. This simplified matters, and put an end to the time consuming design-meetings. Had we decided on this matter earlier it is possible that we would have had more time to implement the features we were discussing on including. It is now obvious to us why a designer is an important role in video game development teams. In future work, we will assign a designer in a much earlier state of development.

Chapter 12

Conclusion

It is a challenge to present a strategy game to a console audience in the right way. It is not enough to have easy controls or a good balance of strategy and action. The overall aesthetics such as sound and visual style must be of high quality. It might prove impossible to achieve this for a small team. Even so, larger teams cannot take the financial risk, and with XNA, new gameplay frontiers such as RTS games for the console may be explored by hobby and student developers. However, developing a game is also a great challenge in communication on topics of game design, programming, and artistic direction.

Game references

Atari (1972) *PONG* (Arcade game) http://www.atari.com/games/atari_arcade/pc-download

Blizzard (1997) *StarCraft* (Windows, Mac OS game) http://eu.blizzard.com/en-gb/games/sc/

Blizzard (2010) *StarCraft II* (Windows, Mac OS) http://eu.blizzard.com/en-gb/games/sc2/

Bungie (2001) *Halo: Combat Evolved* (Xbox 360 game) http://www.bungie.net/Projects/Halo/

EA Los Angeles (2008) *Command & Conquer : Red Alert 3* (Windows, Xbox 360,PlayStation 3, Mac OS game) http://www.commandandconquer.com/en/games/bygameid/ra3

Ensemble Studios (1997) *Age of Empires* (Windows game) http://www.microsoft.com/games/empires/

Ensemble Studios (2009) *Halo Wars* (Xbox 360 game) http://www.halowars.com/

Gas Powered Games (2010) *Supreme Commander 2* (Windows, Xbox 360, Mac OS X game) http://www.supremecommander2.com/

Microprose (1991) *Sid Meier's Civilization* (Windows game) http://www.civilization.com/

Namco (1980) *Pac-man (Pakkuman)* (Arcade game) http://www.uk.namcobandaigames.eu/product/pac-man-championship-edition-dx/playstation-3

Paradox Interactive (2000) *Europa Universalis* (Windows game) http://www.paradoxplaza.com/

Technosoft (1989) *Herzog Zwei* (Sega Genesis/Megadrive game)

Treyarch (2010) *Call of Duty: Black Ops* (Xbox 360, Playstation 3, PC game) http://www.callofduty.com/

References

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 Retrieved 3 May 2011.

Akenine-Möller, T., Haines, E., Hoffman, N. (2008) *Real-Time Rendering 3rd edition.* Third edition Wellesley: A K Peters, Ltd.

Albanesius, C. (2010) *'Call of Duty: Black Ops' Gamers Log 600M Hours of Play Time.* PCMag http://www.pcmag.com/article2/0,2817,2374762,00.asp Retrieved 12 May 2011.

Ashida, K. (2004) *Optimising the Graphics Pipeline.* http://www.nvidia.com/docs/IO/10878/ChinaJoy2004_OptimizationAndTools.pdf Retrieved 15 May 2011.

Autodesk 3ds Max Products (2011) http://usa.autodesk.com/3ds-max/ Retrieved 28 April 2011.

Autodesk Maya (2011) http://usa.autodesk.com/maya/ Retrieved 28 April 2011.

Baker M. J. (2010) *Physics - Dynamics - Collision response* http://www.euclideanspace.com/physics/dynamics/collision/index.htm Retrieved 15 May 2011.

Baraff, D. and Witkin, A (1992) Dynamic simulation of non-penetrating flexible bodies. In *Computer Graphics 26*, 2 July 1992, Chicago. pp 303-308.

Beck, K. et al. (2001) *Manifesto for Agile Software Development*: http://agilemanifesto.org/ Retrieved 2 May 2011.

Beneux, J. (2011) *Why we need better garbage collection on the Xbox* http://sharp-gamedev.blogspot.com/2011/02/why-we-need-better-garbage-collection.html Retrieved 1 June 2011.

Bergen, G.V.D. (2004) *Collision detection in interactive 3D environments*. San Francisco: Morgan Kaufmann Publishers.

Bishop, L.M. et al (1998) Designing a PC game engine. *Computer Graphics and Applications, IEEE,* Volume 18, Issue 1, pp 46-53.

Blender (2011) http://www.blender.org/ Retrieved 28 April 2011.

Bourg, D. and Seemann, G (2004) *AI for Game Developers.* Sebastopol, CA: O'Reilly Media.

Boyer, B (2008) *Making Games For PlayStation Network - The Facts* http://www.gamasutra.com/php-bin/news_index.php?story=17707 Retrieved 5 May 2011.

Bozon, M (2008) *GDC 2008: Wii Ware Interview*
http://wii.ign.com/articles/853/853752p1.html Retrieved 5 May 2011.

Brabec, S. Annen, T. and Seidel H. (2002) Shadow Mapping for Hemispherical and Omnidirectional Light Sources. In *Proceedings of Computer Graphics International*, pp 397-408.

Burns, R. and Sheppard, M. (n.d.) *Tutorial - Collision Detection and Response.*
http://www.metanetsoftware.com/technique/tutorialA.html Retrieved 14 May 2011.

Daumann, N (2011) *Natural light attenuation*
http://blog.slindev.com/2011/01/10/natural-light-attenuation/ Retrieved 15 May 2011.

Dijkstra, E.W (1959) A Note on Two Problems in Connexion with Graphs. In *Numerische Mathematik*, Volume 1, pp 269-271.

Epic Games (2011) *Unreal Tecnology* http://www.unrealengine.com/ Retrieved 16 May 2011.

Esmurdoc, C (2010) *Postmortem: Double Fine's Brutal Legend:*
http://www.gamasutra.com/view/feature/4308/postmortem_double_fines_brutal_.php?page=2 Retrieved 3 May 2011.

Eurogamer (2009) *Console Gaming: The Lag Factor*
http://www.eurogamer.net/articles/digitalfoundry-lag-factor-article Retrieved 15 May 2011.

Filion, D., McNaughton, R. (2008) *StarCraft II: Effects & Techniques*
http://developer.amd.com/documentation/presentations/legacy/Chapter05-Filion-StarCraftII.pdf Retrieved 15 May 2011.

Fosner, R. (2003) *Real-time shader programming: covering DirectX 9.0* [Electronic] San Francisco: Morgan Kaufmann Publishers.

GameSpot (2005), *Best Launch Titles* http://www.gamespot.com/features/6134761/p-2.html Retrieved 12 May 2011.

Gregory, J (2009) *Game Engine Architecture* Natick, MA: A K Peters.

Hargreaves, S. (n.d.) *Deferred Shading*
http://www.talula.demon.co.uk/DeferredShading.pdf Retrieved 15 May 2011.

Hargreaves, S (2007) *Twin paths to garbage collector nirvana*
http://blogs.msdn.com/b/shawnhar/archive/2007/07/02/twin-paths-to-garbage-collector-nirvana.aspx. Retrieved 5 May 2011.

Haverkort H.J. (2004) *Introduction to bounding volume hierarchies*. Utrecht: Utrecht University (PhD thesis, introduction to publications about bounding volume hierarchies).

Hawkins, D (2008) *Sponsored Feature: Democratizing Game Distribution: The Next Step* http://www.gamasutra.com/view/feature/3545/sponsored_feature_democratizing_.php Retrieved 5 May 2011.

Higgins, D (2002) *Generic A* Pathfinding.* AI Game Programming Wisdom, ed. S. Rabin, pp. 114-121. Hingham: Charles River Media.

Hubbard, P.M. (1993) Interactive Collision Detection. In *Proceedings of 1993 IEEE Research Properties in Virtual Reality Symposium*. 25-26 Oct. 1993, San Jose. pp 24-31.

James, S. (2010) *3D Graphics with XNA Game Studio 4.0* [Electronic] Birmingham, UK: Packt Publishing.

Keith, C (2010a) *Agile Game Development with Scrum*. United States: Addison-Wesley.

Keith, C (2010b) *State of Agile in the Game Industry* http://www.gamasutra.com/view/feature/4295/the_state_of_agile_in_the_game_.php Retrieved 3 May 2011.

Klucher, M. (2006) *The XNA framework Content Pipeline* http://blogs.msdn.com/b/xna/archive/2006/08/29/730168.aspx Retrieved 12 May 2011.

Konečný, P. (1998) *Bounding Volumes in Computer Graphics [Master thesis]* Masaryk University, Brno Czech.

Lobão, A.L. et al. (2009) *Beginning XNA 3.0 Game Programming From Novice to Professional.* [Electronic] New York: APRESS.

Matthews, M (2010) *NPD: Behind the Numbers, December 2010* http://www.gamasutra.com/view/feature/6258/npd_behind_the_numbers_december_.php Retrieved 11 May 2011.

McCarthy, J (2007) *WHAT IS ARTIFICIAL INTELLIGENCE?* http://www-formal.stanford.edu/jmc/whatisai/whatisai.html Retrieved 5 May 2011.

Miller, P (2008) *Top 10 Pitfalls Using Scrum Methodology for Video Game Development.* http://www.gamasutra.com/view/feature/3724/top_10_pitfalls_using_scrum_.php Retrieved 3 May 2011.

Microsoft (2006) *Microsoft Invites the World to Create Its Own* Xbox *360 Console Games for the First Time* (press release) http://www.microsoft.com/presspass/press/2006/aug06/08-13XNAGameStudioPR.mspx Retrieved 5 May 2011.

Microsoft (2004) *Microsoft: Next Generation of Games Starts With XNA* (press release) https://www.microsoft.com/presspass/press/2004/mar04/03-24xnalaunchpr.mspx Retrieved 5 May 2011.

Microsoft (2011) *Xbox Marketplace: Indie Games* http://marketplace.Xbox.com/en-US/Games/XboxIndieGames Retrieved 5 May 2011.

Microsoft Developer Network (2010) *Direct3D 11 Features* http://msdn.microsoft.com/en-us/library/ff476342%28VS.85%29.aspx Retrieved 15 May 2011.

Microsoft Developer Network (2011b) *Microsoft Cross-Platform Audio Creation Tool (XACT)* http://msdn.microsoft.com/en-us/library/bb174772.aspx Retrieved 5 May 2011.

Microsoft Developer Network (2007) *Skinned Model* (code sample) http://create.msdn.com/en-US/education/catalog/sample/skinned_model Retrieved 5 May 2011.

Microsoft Developer Network (2011c) *Standard Importers and Processors* http://msdn.microsoft.com/en-us/library/bb447762.aspx Retrieved 12 May 2011.

Microsoft Developer Network (2011d) *What is Content?* http://msdn.microsoft.com/en-us/library/bb447756.aspx Retrieved 12 May 2011.

Mirtich, B. (1997) *Efficient Algorithms for Two-Phase Collision Detection*. Mitsubishi Electric Research Laboratory.

Neider, J., Davis, T., Woo, M. (1994) *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1* Reading: Addison-Wesley Publishing Company http://glprogramming.com/red/Chapter05.html Retrieved 15 May 2011.

Newman, W. M., Sproull, R.F. (1979) *Principles of interactive computer graphics.* New York: McGraw-Hill.

Nutt, C (2008) *Q&A: How Ensemble Gets Halo Fans To Appreciate Halo Wars*. Gamasutra http://www.gamasutra.com/php-bin/news_index.php?story=20835 Retrieved 12 May 2011.

NVIDIA (2011x) *GeForce 256.* http://www.nvidia.com/page/geforce256.html Retrieved 15 May 2011.

NVIDIA (2011y) *GeForce 3* http://www.nvidia.com/page/geforce3.html Retrieved 15 May 2011.

Patel, A (n.d.) *Amit's A* pages*. http://theory.stanford.edu/~amitp/GameProgramming/ Retrieved 5 May 2011.

Perry, D (2006*) XBLA: How to Make an* Xbox *Live Arcade Game* http://Xboxlive.ign.com/articles/721/721843p1.html Retrieved 5 May 2011.

Reed, A. (2008) *Learning XNA 3.0* [Electronic] Sebastopol, CA: O'Reilly Media.

Reeves, W.T. (1983) Particles Systems - Technique for Modeling a Class of Fuzzy Objects In *Computer Graphics, 17 (SIGGRAPH '87 Conference Proceedings)* pp 359-375.

Reynolds, C. W. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model, in *Computer Graphics, 21 (SIGGRAPH '87 Conference Proceedings)* pp 25-34.

Richard, J. and Lins, R. (1996) *Garbage Collection: algorithms for automatic dynamic memory management.* Chichester, Wiley.

Selman, D. (2002) *Java 3D Programming* [Electronic] Greenwich: Manning Publications.

Šinjur S. (2001) Collision detection between moving objects using uniform space subdivision. Paper from the *CESCG conference* [No conference publication] 23-25 April 2001, Budmerice.

Schwaber, K. and Sutherland, J. (2010) *The Scrum Guide*: http://www.scrum.org/scrumguides/ Retrieved 2 May 2011.

Shirley, P and Morley, R (2003) *Realistic Ray Tracing*. Second Edition. Natick, Massachusetts : AK Peters.

Steam (2011) *Steam Hardware & Software Survey* http://store.steampowered.com/hwsurvey/ Retrieved 15 May 2011.

Symbolics Graphics Division (2009) *Stella & Stanley: Breaking the Ice (1987)* [youtube] http://www.youtube.com/watch?v=3bTqWsVqyzE Retrieved 12 May 2011.

Terdiman, P (2007) *Sweep and Prune* http://www.codercorner.com/ Retrieved 14 May 2011.

Unity (2011) *Web Player Hardware Statistics - 2011 Q1* http://unity3d.com/webplayer/hwstats/pages/web-2011Q1-shadergen.html Retrieved 15 May 2011.

Valve (2007) *SOURCE ENGINE* http://source.valvesoftware.com/ Retrieved 16 May 2011.

VersionOne (2010) *State of Agile Survey 2010.* http://www.versionone.com/pdf/2010_State_of_Agile_Development_Survey_Results.pdf Retrieved 3 May 2011.

Skillings, J (2006) Getting machines to think like us. *cnet News* http://news.cnet.com/Getting-machines-to-think-like-us/2008-11394_3-6090207.html Retrieved 1 June, 2011.

VGChartz (2011) *Call of Duty: Black Ops Sales (*Xbox*360).* VGChartz http://gamrreview.vgchartz.com/sales/44952/call-of-duty-black-ops/ Retrieved 12 May 2011.

Wikipedia (2011a) *Inverse-square law* http://en.wikipedia.org/w/index.php?title=Inverse-square_law&oldid=425552301 Retrieved 15 May 2011.

Wikipedia (2011b) *List of games with DirectX 10 support* http://en.wikipedia.org/w/index.php?title=List_of_games_with_DirectX_10_support&oldid=429099971 Retrieved 15 May 2011.

Wikipedia (2011c) Xbox *360 Controller*. http://en.wikipedia.org/w/index.php?title=Xbox_360_Controller&oldid=428577191 Retrieved 12 May 2011.

Yang B, Cheng X, Pan Z. (2005) A Real-time Collision Detection Algorithm for Mobile Billiards Game. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 15-17 June 2005, Valencia. pp. 294 - 297.

Zenko, D (2009) *Halo's smart design lures you in*. Toronto Star, March 14.

Appendix A

Ansvarsuppdelning

- Ansvarsområden
  - Planering

  Jonas ledde planering, men gruppmedlemmar planerade i stor del själva på sina egna arbetsuppgifter

  - Informationsinhämtning/inläsningsdel

  Varje gruppmedlem hämtade själv in information om sitt ämne och delade även med sig av information som var intressant för flera gruppmedlemmar

  - Metoder -- val/utveckling

  Alla arbetade gemensamt enligt Scrum. Vilken mjukvara som valdes var upp till varje ansvarig för ett område

  - Genomförande

  Joakim: Kollisionsdetektion och hantering. Terräng och världgenerering. Grundgrafikuppritning.
  Pouya: Modellering, animering och texturering.
  Mattias: Shader-programmering, partikelsystem.
  Jonas: Planering, AI, animationsmotor.
  Jacob: AI, spelsessionshantering, övrig programmering.

- Bidrag till problemlösning, syntes och analys
  - Problemlösning

  Alla gruppmedlemmar medverkade till att problem löstes

  - Kreativitet, idérikedom

  Alla gruppmedlemmar medverkade med idéer, lite för många kanske.

  - Skapande av modell
  - Analys av projektrelaterat material
  - Diskussionsbidrag

  Alla gruppmedlemmar diskuterade mycket tillsammans.

  - Slutsatser

  Varje medlem i gruppen gav sina synpunkter på slutsatser

- Huvudansvarig författare av avsnitt

- Författare av avsnitt:

Introduction: Jonas
Designing an RTS for a console: Jacob och Jonas
C# and the XNA framework: Jonas
Scrum and agile: Jonas
Game Engine: Jonas
AI: Jonas
Rendering: Mattias
Modeling and animating: Pouya
Loading assets: XNA content pipeline och struktur: Pouya
Collision Detection: Joakim
Results, Discussion and Conclusion: Alla

- Redaktionell ansvarsfördelning

Jonas och Mattias hade det största redaktionella ansvaret, men alla i gruppen bidrog med kommentarer och korrekturläsningar