NTNU
Norwegian University of
Science and Technology

Evolutionary Game Prototyping using the Unreal Development Kit

Kjetil Guldbrandsen
Kjell Ivar Bekkerhus Storstein

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Alf Inge Wang, IDI
Co-supervisor: Meng Zhu, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project will use the Unreal Development Kit to implement a prototype game. The Unreal Development Kit be evaluated according to its practical use for game prototyping in a small team environment.

Assignment given: 15. January 2010
Supervisor: Alf Inge Wang, IDI

Abstract

The goal of this thesis was to evaluate the *Unreal Development Kit (UDK)* as an evolutionary game prototyping tool. To conduct this evaluation in a realistic setting, a prototype of a game concept was to be implemented using this tool.

To aid the prototyping process, extensive research was done into existing theory on game prototyping, as well as how traditional prototyping techniques can be utilized in a game prototyping environment.

The project team created their own prototyping process tailored for evolutionary game prototyping, based on the theoretical insight gained through the research on general prototyping processes. Due to time constraints, the team was unable to test this process extensively. This is work that remains before the process can be fully recommended for further use.

During the evaluation process, the team identified key criteria for evaluating a game prototyping tool and compiled this into an evaluation framework. The key points identified in the evaluation was that the UDK offers low-risk licensing terms for a game engine suite with an outstanding track-record of successful game titles. To properly utilize the speed gains that can be achieved through the UDK, a deep understanding is needed of its feature set.

The main challenge in this project was the balancing act of two somewhat conflicting goals: Acquiring knowledge of the UDK, thus covering the breadth of its features, while at the same time following narrowly focused prototyping techniques.

From this work, the project team has gained deep insight into one of the game industry's most widely used engines and how it can be used as an evolutionary prototyping tool. The team is particularly satisfied with the evaluation framework and the evaluation itself, as these will provide useful information for anyone considering using the UDK, both professionally and academically. Engine developers will also benefit from a novice user's point of view.

Contents

Part I

Introduction

Chapter 1

Project Background

"Now it is beginning of a fantastic story!! Let's make a journey to the cave of monsters! Good luck!"
- Opening Screen, Bubble Bobble (1986)

This chapter aims to describe the background of this master thesis. The first section will explain the context of the project, followed by a section on the motivation behind the project. The last section will describe the project's goal.

1.1 Project Context

This project is the master thesis of the authors at the *Norwegian University of Science and Technology(NTNU)*. It is a part of the game technology research program at the *Department of Computer and Information Science(IDI)* under the *Faculty of Information Technology, Mathematics and Electrical Engineering(IME)*. The project loosely follows the depth project "Apocalypse Engine - A Study of Software Architecture and Conventions in Modern Game Engines" conducted the fall 2009 by the authors [7].

1.2 Motivation

This section describes the motivation behind this project. The first subsection aims to explain current trends in the video games industry that relate to the project. The second subsection outlines the personal motivation behind the thesis.

1.2.1 Trends in the Video Game Industry

Video games have over the last decade become huge and complex development efforts [8]. Team sizes, budgets and development hours have consequently risen to a level comparable with Hollywood blockbusters. As an example Grand Theft Auto IV by Rockstar Games had an estimated budget of roughly $100 million with over a thousand people involved [9]. Since millions of dollars are involved, the large studios tend to stick to tried and tested game concepts to minimize risk.

Another emerging trend in the world of video games is that games are no longer only sold over the counter, but also in a downloadable digital form. Services such as Steam for PC and Mac, Xbox Live Marketplace for the Xbox 360 and Playstation Store for the Playstation 3 give gamers the chance to buy and download games digitally, in the comfort of their own homes. For developers, these services provide some major benefits: Traditionally developers want to maintain creative control over their intellectual property. This has often interfered with the distributor's need to minimize risk. With digital distribution this nuisance can be overlooked as there is no unit production cost, and only a royalty per sold game need to be paid to the digital distributor. Another benefit is instant access to the worldwide market, since no local distributors are necessary.

The lack of innovation among the larger game studios, combined with today's ease of digital distribution has spawned the Independent Game Developer movement [51]. Indie game developers tend to emphasize innovative game concepts or a fresh look upon old classics. There are numerous success stories when it comes to indie games. Among the most successful ones is the critically acclaimed *Braid*, developed by only two people selling over 250,000 units on the Xbox Live Marketplace alone. Another huge success is *Castle Crashers* created by a core team of five which has sold over 520,000 units on the Xbox Live Marketplace [63]. Starting out as an indie developer, however, is no easy task. With no major financial backup, resources are scarce. Thus, obtaining state-of-the-art third-party middleware and engines, commonly used in the larger studios, is not an option.

Epic Games [27] has responded to the emerging need for affordable game tools, by releasing their Unreal Engine 3 with complete toolkit, named the *Unreal Development Kit (UDK)* to the public under a special license. A licensee may use the toolkit for both educational and commercial purposes, however for any revenue above $5000 Epic claims a 25% royalty. The up-front cost for commercial projects aimed to be sold is $99, whereas a business which

internally uses the toolkit must pay a fee of $2500 per development seat per year [31]. This business scheme can be extremely benificial for indie developers since the up-front cost is a fraction of the usual six-figure price. It is important to recognize that the toolkit and engine are in no way scaled down. It is the same engine and tools used in modern AAA[1] titles such as "Unreal Tournament 3", "Gears of War 1 and 2", "Mass Effect 1 and 2" and "Batman: Arkham Asylum" [88]. The only difference between a UDK license and a full license is that the latter gets access to the source code. The first beta of the UDK was released in November 2009 and downloaded over 50,000 times the first week [29]. A new beta has been released every month since then with added features and stability improvements. The success of Epic's licensing strategy has also been noted by other engine developers. Crytek, developers of the "CryENGINE 3" recently stated that they will release a free platform based on their technology [14].

1.2.2 Personal Motivation

In the depth project preceding the thesis, the authors conducted a study of conventions and architecture of modern video game engines [7]. Unsurprisingly the research showed that modern game engines are complex behemoths, and it is unrealistic for small teams to create a competing one from scratch. As stated in the depth project's motivation we have developed an award-winning game concept [3], which we aim to implement. During the implementation part of the previous project we were not able to test our concept in a game environment at all, since the whole development effort went into creating the engine. The approach to this project will therefore be to develop a game prototype using an existing engine. The release of the UDK coincided very well with this, and is our obvious choice due to its maturity and reputation.

The game mechanics of our concept are experimental and a bit vague, thus we need to be able to rapidly prototype different ideas. This challenge is, of course, not unique in our case, thus researching the UDK as a prototyping tool is valuable for actors in the video game industry. Since the UDK is not only limited to entertainment, but can be used as a general visualization tool, research on the UDK will be of interest for other parties as well. Here lies the main value of this thesis.

Another aspect of our personal motivation is to increase our general skill set in game development. During the last project we focused on software architecture and programming, which are traditional skills in our computer science

[1] A "AAA game" is lingo for a big budget commercial game

background. Creating games is largely a multi-disciplinary effort, thus developing insight into areas beyond programming is desirable. This is particularly true for indie game development, where team members typically cover multiple roles. Larger studios also need multi-disciplinary team members to bridge the gap between art and code, e.g. "technical artists" [48].

1.3 Project Goal

The main goal of this project is to evaluate the UDK as a game prototyping tool. An important property of the evaluation is the accessibility of the toolkit to someone with no prior experience with Epic's technologies. To reach this goal a large part of the pre-study will be to familiarize ourselves with the different tools of the UDK. When a basic understanding of the toolset has been reached, the actual prototyping of the concept can commence. Since it is our concept we aim to experiment with, a set of requirements derived from the concept document will form the basis of our initial prototype. This prototype will be further refined during the course of the development, via feedback and observations from user tests. The test base will consist of a chosen few in the target demographic, including the authors.

The prototyping itself is an interesting process. A subgoal of the project will be to find a fitting prototype procedure. Here we will explore what established research on game prototyping exists, or if we must create our own scheme by lending prototyping schemes from other fields e.g. *Human Computer Interaction(HCI)* [64] and agile software development [18].

When the prototyping phase is finished, the evaluation can begin. Important criteria for the evaluation are: How much time is required to learn the basics of the toolkit until you can starting working on the actual prototype? How much time is saved versus creating something similar on your own? Here we will compare with our experiences creating the *Apocalypse Engine* in the preliminary project [7]. We will also look at the maturity and stability of the toolkit, and report potential time lost due to issues out of our control. The complete evaluation framework is described in Section 10.

Chapter 2

Research

"Foolish are those who fear nothing,
yet claim to know everything."
- Imperium Thought for the Day, Warhammer 40,000: Dawn of War (2004)

This chapter explains the research to be conducted on this master thesis and is divided into two sections. The chapter's first section will outline the research questions posed by the authors. The second describes the research methodologies applied to answer the research questions, and how we aim to use them in our project.

2.1 Research Questions

To explain where these questions are coming from, we will revisit some of what is stated in Section 1.2. First of all, we want to explore whether video game companies tend to follow any formal process for prototyping a game. This ties into our investigation on whether there exists any specific research in the area of game prototyping, or whether it exists mostly in classic areas like user interface design. If we look at game development from our own perspective, we have a small team, limited resources and a tight time schedule of 23 weeks. Teams like this are highly relevant at the time of writing, as digital distribution is getting common and the amount of independent developers is rising [51]. Such teams are in need of low-cost time-saving tools if they are to develop cutting edge 3d games, as modern 3d game engines are huge undertakings. This makes us want to evaluate the Unreal Development Kit, as it is based on a proven

engine and affordable to independent developers[1].

Thus, we pose the following questions:

RQ1: What formal game prototyping processes exist?

RQ1.1: What research exists on game prototyping?

RQ1.2: Can prototyping theory from other fields be applicable to game prototyping?

RQ2: Which factors are important in evaluating a game prototyping tool?

RQ3: How does the UDK score according to the evaluation criteria derived from RQ2?

It is important to note that there is more to RQ2 than meets the eye. The question only serves as the starting point for discovering more specific sub-questions dealing with the criteria for game prototyping tool evaluation. As these criteria are part of what is to be researched, any related questions would therefore be based on nothing. They will instead be presented as part of our own contribution.

2.2 Research Methodology

There are several established research methods applicable to software engineering. These methods are guidelines on how research may be conducted in software development projects. The ones that apply to our project will be listed in this section.

2.2.1 The Scientific Method

Basili[4] identifies and describes some of the more common research methods for software engineering. He describes the the *scientific method* as:

1. **The scientific method:** observe the world, propose a model or a theory of behavior, measure and analyze, validate hypotheses of the model or theory, and if possible repeat the procedure.

   (a) **The engineering method:** observe existing solutions, propose better solutions, build/develop, measure and analyze, and repeat the process until no more improvements appear possible.

[1] 99$ royalty-bearing license at the time of writing[31]

(b) **The empirical method:** propose a model, develop statistical/qualitative methods, apply to case studies, measure and analyze, validate the model and repeat the procedure. Statistical/qualitative methods, apply to case studies, measure and analyze, validate the model and repeat the procedure.

More thoroughly about the *engineering method* Basili says:

> "This version of the paradigm is an evolutionary improvement oriented approach which assumes one already has models of the software process, product, people and environment and modifies the model or aspects of the model in order to improve the thing being studied. An example might be the study of improvements to methods being used in the development of software or the demonstration that some tool performs better than its predecessor relative to certain characteristics. Note that a crucial part of this method is the need for careful analysis and measurement."

2.2.2 Literature Review

The *literature review* method consists of reviewing existing literature on the subject before trying to answer any research questions. This ensures that the researchers have a solid foundation on previous findings and problems, and can use this to their advantage during the research process. Of course, it is important that the literature review is done as early as possible in the project to have an effect on the rest of the project work.

2.2.3 Use of Research Methods

In this project we aim to evaluate the UDK as a game prototyping tool. To achieve this goal we must first obtain extensive knowledge about the research area. Studying the UDK will first and foremost require use of the toolkit. Learning by doing corresponds to the observation phase of the engineering method. To develop knowledge of a new toolkit, online documentation and tutorials is a convenient source. We will, however, not categorize this practice as a strict "literature review", since the knowledge obtained from the "literature" in this case does not alone cover the knowledge source. The literature is merely a supplement to learn the toolkit, it is the actual practice that will stand for most of the lessons learned.

To expand our insight into game prototyping, we need to study books, articles and web-pages. This corresponds to the observation phase of the engineering

method, though a literature review is carried out to obtain information. The result of the conducted study should be a prototyping process we find adequate. If we are unable to find such a process we will propose our own to better suit our prototyping needs, as described in the engineering method above. The process of implementing a game prototype leans toward the engineering methods in its very nature, since prototyping by definition is about iterative refinement of a product [16].

Chapter 3

Development

"May the way of the hero lead to the Triforce"
- Rescued Maiden, The Legend of Zelda: A link to the past (1991)

In this chapter we will first provide a comment on the development method to be used for the implementation part of the project. The next section will list the tools to be used in the project, both for creating the project report and for the actual implementation.

3.1 Development Method

The developing efforts in this project will lie in the implementation of a game prototype. Traditional software development make use of a development method such as the Waterfall Model [6], or some form of agile development method, e.g. Scrum [92]. Since game prototyping is not a traditional software development effort, neither will the development method be a traditional one. The development method, which in this context is the prototyping process, will be chosen or developed after the prototyping prestudy is finalized, and is described in Chapter 8. The development method will later be evaluated in Section 10.3.

3.2 Tools

This section lists the various development tools used in this project, both for creating the report and implementing the game prototype.

MiKTeX 2.7 by Christian Schenk Implementation of TEX for Windows [56]. TEX is a typesetting system used for writing books, reports and technical documents. It is especially designed for math-heavy writing.

TeXnicCenter 1.0 RC 1 by ToolsCenter.org Free open source editor for creating LATEX projects. It uses the MiKTeX or TeX Live distributions [93]. The editor gives an overview of the whole LATEX project, supplies code completion and provides means for easily building the project into an output file (like a PDF file).

Visual Studio 2008 by Microsoft An *Integrated Development Environment (IDE)*, supporting several programming languages. It is widely used for Visual C++ and C# projects, but in this project it is used in combination with *nFringe* to write UnrealScript.

nFringe by Pixel Mine Games Is a plugin for Visual Studio 2008, delivering syntax highlighting, code inspection, debug features, Intellisense and auto-completion for UnrealScript.

Tortoise SVN by Stefan Kung and Lubbe Onken A free Subversion client for Windows. It is implemented as a Windows shell extension and used for version control of code and documents. Used in this project both for this report and for the prototype implementation.

3ds Max 2010 by Autodesk An advanced 3d modeling, visualization and animation tool.

Photoshop CS4 Pro by Adobe A graphics editing program. Used for creating all sorts of graphics, like textures for 3d models, figures or manipulating photos.

Illustrator CS4 by Adobe A vector graphic editing program. Used for creating line art in the concept document.

InDesign CS4 by Adobe An application for creating publishing layouts. Used for creating the concept document.

Crazybump by Ryan Clark A tool for creating bump maps, specularity maps and height maps from regular photos or images. These can be used to simulate more detail in a 3d model than there actually is, whic is a crucial trick in real-time 3d.

Mudbox 2010 by Autodesk Is a brush-based sculpting and painting application used to create high-resolution 3d-models. Mudbox features functionality to export normal- and height-maps used with lower resolution

models.

Unreal Development Kit by Epic Games Is a complete game development kit built around the Unreal Engine 3. The UDK is subject to scrutiny throughout this project and is thoroughly described in Chapter 5.

Part II

Prestudy

Chapter 4

Game Engines

"If you want guarantees, buy a toaster..."
- Morrison, Crysis (2007)

This chapter will start with a background section with a general introduction to game engines, components commonly present in game engines and a list of relevant candidates to investigation. This section is largely based on the research conducted in the depth project preceding this thesis. When the basics of game engines are established, we move on to introduce the Unreal Development Kit in the next chapter.

4.1 Background

This section consists of three subsections giving a general introduction to game engines, their different components and a list of state-of-the-art engines used in modern, commercial games.

4.1.1 What is a Game Engine?

Jason Gregory of NaughtyDog makes the distinction between a game and a game engine in his book *Game Engine Architecture* [45] as:

> "Arguably a *data-driven architecture* is what differentiates a game engine from a piece of software that is a game but not an engine. When a game contains hard-coded logic or game rules, or employs special-case code to render specific types of game objects, it becomes difficult or impossible to reuse that software to make a

> different game. We should probably reserve the term "game engine" for software that is extensible and can be used as the foundation for many different games without major modification

A game engine is structured around different modules contributing to different aspects of a game. This can be rendering, collision detection, physics, networking, artificial intelligence and tools to name a few [83]. While early games typically where built from scratch, games nowadays often license third party tools and components to shorten development time. Such tools can come in the form of physics, sound or even an entire engine, which in turn can be modified. Examples of commercially available engines include Valve Software's *Source engine*(used in Half-Life 2 and Portal) [71], Emergent Game Technologies' *Gamebryo engine*(Used in Fallout 3 and Civilization IV) [74], and of course the *Unreal Engine 3* which the UDK is built around and has been previously used to create games such as Unreal Tournament 3, Gears of War and Batman: Arkham Asylum [27]. A list of game engines included toolset is presented in Section 4.1.3. The advantage of licensing a cutting edge engine, is that the developer can get the advanced visuals that are expected with newer games, without having world-class game engine programmers on the team. Developers can therefore put their effort into creating stunning content and entertaining game mechanics.

4.1.2 Game Engine Components

Here we will give a description of the parts that make up a modern game engine as an introduction to what might be expected from the Unreal Development Kit.

Rendering Rendering is an essential part of a game engine. It is the player's window into the game, i.e. what makes it possible for the player to get visual feedback. All the advanced game mechanics in the world is for naught if there is no way to portray it to the outside world. Performance is a crucial quality here, while the visuals at the same time should look as good as possible. These are conflicting goals, and compromises usually have to be made. Games in the FPS (First Person Shooter) genre have traditionally been the most innovative in the use of state of the art real-time rendering technologies.

Animation Few games can do without at least some sort of animation. It's essential to give the illusion of life. Be it just a few frames of simple 2d animation, or many 3d elements on the screen at the same time playing their own animation cycles. It can be anything from walking animations,

to facial expressions or explosions and other effects. 3d animation can be key-frame determined, motion-captured or procedurally created.

Artificial Intelligence *Artificial intelligence (AI)* is the intelligence of machines, and can take many forms in computer games. From a simple opponent in Pong, to a learning creature in Black & White [81]. Without artificial intelligence, many single player games would be rendered moot as there would not be an opponent. The AI in games has not had a lot in common with traditional AI research, as games have had a tendency of using *finite state machines* for the AI. Finite state machines are not considered part of the AI field, but rather as a part of computation theory. Though, as AI becomes increasingly more complex in games, more traditional AI research schemes are being put to action. *Automatic planning* and *machine learning* are two examples.

Sound Sounds play a big part in games by providing the player with audial feedback using sound effects, story through narration and mood through music. An up-to-date game engine should therefore support the playing of multiple sound effects, as well as the streaming of background music. 3d games might also have the need for *positional audio* (or "3d sound" if you will), with a movable listener and sources. There exists third-party libraries to save developer time. *FMOD* is an example, used in games like World of Warcraft and Batman: Arkham Asylum [21].

Scripted Events Scripting is important in bigger engines, as it gives artists and level designers tools to sculpt a level into their vision. This can come in the form of programming behaviors via the scripting language, to time-based or location-based events that are triggered. It is also useful for creating in-game sequences which can have the purpose of portraying the story or giving level-specific information. Scripting can make the levels or the story feel all the more interesting and dynamic, giving the player a more immersive experience than with static levels.

Physics and Collision Detection Interaction between the player and an environment where laws of physics apply are getting to be expected in modern games. It is important for players that objects respond as one would expect in the real world. This demands a powerful physic simulation engine. A lot of developers implement third party physic engines or libraries into their games. Examples include Havok Physics [47] and Open Dynamics Engine [69]. Box2D [12] is another alternative for 2D games. Collision detection is a part of physics and very important in games. It deals with the problem of determining whether two or more

objects are touching or intersecting. If a collision is detected, a proper collision response is administered. While this problem might seem simple enough, it gets extremely complex as a lot of objects are moving around simultaneously in 3D space. To add to that, you have the problem of *tunneling* [20], which can occur if a object moves sufficiently between frames to move *through* a surface without intersecting. Collision detection is therefore a complex subject, but also used all the time in games. Without it, gamers would find themselves walking through walls, falling through floors and not hitting enemies with their expensive plasma cannons.

Tools No modern game engine is complete without a suite of supporting tools. These tools all serve the purpose of easing the design and creation of a game. In short: It gives designers and artists means of focusing on creating the content that makes the game unique. Level editors are a classic example. They are usually visual and integrated with a scripting system. This way, a level designer can design a level without worrying about any internal game code and get instant feedback on how the end result will look. Modern editors also provide a host of other advanced features, like placing of AI path nodes, terrain creation and procedural creation of vegetation in selected locations.

4.1.3 Researched Game Engines

In the depth project preceding the master thesis, we researched the software architecture and design conventions in modern game engines [7]. In this section we have compiled a list of the engines subject to our investigation, outlining some of the key aspects of the Unreal Engine 3's competitors.

Gamebryo is an engine created by *Emergent Game Technologies* (EGT). It has been used in a wide array of games of different genres: *Civilization IV, The Elder Scrolls IV: Oblivion, Fallout 3, Empire Earth II* to name a few [74]. Though these have been based on older versions of the engine than is currently available.

At the time of writing, EGT's newest product is the expansion of the Gamebryo engine, called Gamebryo LightSpeed. It is a superset of the Gamebryo engine, containing additional design tools to further aid rapid development. It supports the following platforms: Windows, Wii, Xbox 360 and PlayStation 3.

While engines like Unreal Engine 3 and CryENGINE tend to focus more on

the first person shooter gaming aspect, Gamebryo tries to be a more versatile engine. This is reflected in the diversity in the games that utilizes it, as well as the range of optional third-party accessories available that integrate into the engine toolset.

Unity is a game engine created by the Danish company *Unity Technologies*. An important point to consider when comparing this engine to the others, is that it lies in a completely different price range. While the estimated cost at the time of writing for a full license for the other engines lies somewhere between 150,000$-500,000$, a Unity Pro license costs $1,500. Though, this is without the source code available.

The game creation centers around a integrated visual toolkit, the Unity Editor [75], where terrain, scripts, objects etc. is created. One has no control over the source code, unless a source code license is bought. This makes the developer reliant on Unity Technologies to provide updates to their engine to include the latest technology. Gameplay is programmed via three supported scripting languages (C#, JavaScript and Boo) [76]. Scripts in Unity are compiled to native code to ensure high performance..

With a Unity Professional license, it is possible to create games for PC, Mac, Wii and the iPhone, as well as web-based games via Adobe Flash. Unity has been used in several iPhone games, like *Star Wars: Trench Run* and *Skee-Ball*[78].

CryENGINE 3 is a game engine created by the German company Crytek GmbH [44]. Crytek is well known for their innovative solutions leading to cutting edge graphics and physics. The first CryENGINE was pushing boundaries at its arrival in 2004 with the launch title *Far Cry*. The second iteration of the engine, CryENGINE 2, was released in 2007 with the title *Crysis* and is still the defining engine for next-gen graphics. CryENGINE 2 has been successfully used in a number of titles including the mentioned *Crysis*, *Crysis Warhead* and *Far Cry 2*[1].

CryENGINE 3 steps away from being a PC-only engine and runs on both Xbox 360 and Playstation 3. The engine also utilizes a fully deferred renderer [58] as opposed to the forward renderer with early z-pass in CryENGINE 2 [57]. Another powerful feature of the CryENGINE 2 which is further enhanced in CryENGINE 3 is the so-called *What you see is what you play (WYSIWYP)*-feature of their level designer. WYSIWYP gives a level designer the opportunity to jump right into the game from the level editor.

[1]Far Cry 2 uses Ubisoft's *Dunia* engine which is a modified version of CryENGINE 2

Other engines such as the Unreal Engine 3 supports such a similar feature, but Crytek pushes it one step further by allowing WYSIWYP on consoles as well as the PC. Since CryENGINE 3 was released recently so no titles has yet been released using it.

Chapter 5

Unreal Development Kit

"Work, work, work"
- Peon, Warcraft II: Tides of Darkness (1995)

As mentioned in Section 1.2.1 the Unreal Development Kit is a free version of Epic Games' Unreal Engine 3 with a complete toolset included. When reading the chapter we use the terms "engine" and "toolkit" interchangeably, since the UDK is both an engine and a toolkit. This chapter will introduce different aspects of the Unreal Development Kit. Due to the size of the toolkit, individual sections are allocated to discuss its different aspects. Section 5.1 will introduce UnrealScript, the scripting language developed to use with Unreal Engine 3. The next section, covering the Unreal Editor, is by far the most extensive showing the scale and sophistication of the toolkit. The following section introduces the third-party tool "Speed Tree" which is included in the UDK as a separate application.

At the end of the chapter a section is devoted to summarize the chapter's content. The weighted relevance of different parts of the UDK in our project will also be discussed here.

5.1 UnrealScript

UnrealScript is the scripting language used in *Unreal Engine 3*, and therefore also in the *Unreal Development kit*. The language was created for the first Unreal Engine by Tim Sweeney, the founder of Epic Games (formerly Epic MegaGames) [72]. It has been expanded and enhanced in further versions of the engine. The language is constructed to follow the same object-oriented

principle as Java and C++, while at the same time supporting concepts not addressed by the traditional languages [40]. These concepts aim to ease game programming and include:

Time - Functions which take a some amount of time to complete can be cumbersome to write in a language like C++, often requiring a separate thread or other measures to prevent the function from choking the entire system. This is taken care of in UnrealScript in the form of *latent functions*. Latent functions run in the background without interfering with the rest of the system, simplifying chains of events that involve the passage of time.

State - Game logic and perhaps especially game AI have traditionally been highly dependent on different behaviors depending on what state an object is in [81]. This is simplified through UnrealScript's state-system, where several state-dependant versions of the same function are able to exist in the same class.

Network replication - To maintain consistency between server and client can be a nuisance for developers. Network replication in UnrealScript is done through its own replication block for variables and separate function specifiers (Server, Client, Reliable) [40].

5.1.1 Language Terminology

UnrealScript introduces some new terms that bears explaining. The words themselves might be known, but the meaning is new in this context. The most central of these is the concept of an *Actor*. Everything that exists in the game world is an actor: Players, enemies, weapons, terrain, lights etc. Most engines have a concept like this; in the "Apocalypse Engine" developed in the depth project preceding this master thesis, this was referred to as Entities [7].

The visual representation of a player in the game is called a *Pawn*. There is nothing in this class that determines the control scheme. The Pawn can be the same whether it gets controlled by the computer or by a human player. It has no "brain". The Pawn simply specifies which 3D model to use, what animations that get played, and functions for moving, attacking and whatever other actions that the pawn should be able to perform.

The "brain" in the Pawn is a binding to a *Controller*. A controller might be a PlayerController, or an AIController. The PlayerController controls the Pawn through input done through a device like mouse, keyboard, game pad, joystick or similar. The human controller scheme is set up through creating bindings

between PlayerController functions and inputs in a configuration file.

5.1.2 Language Hierarchy

To begin understanding the class hierarchy in UnrealScript, let us first look at what Tim Sweeney says in "UnrealScript Language Reference" about the object hierarchy within UnrealScript[40]:

> "Before beginning work with UnrealScript, it's important to understand the high-level relationships of objects within Unreal. The architecture of Unreal is a major departure from that of most other games: Unreal is purely object-oriented (much like COM/ActiveX), in that it has a well-defined object model with support for high-level object oriented concepts such as the object graph, serialization, object lifetime, and polymorphism. Historically, most games have been designed monolithically, with their major functionality hard coded and unexpandable at the object level, though many games, such as Doom and Quake, have proven to be very expandable at the content level. There is a major benefit to Unreal's form of object-orientation: major new functionality and object types can be added to Unreal at runtime, and this extension can take the form of subclassing, rather than (for example) by modifying a bunch of existing code. This form of extensibility is extremely powerful, as it encourages the Unreal community to create Unreal enhancements that all interoperate.
>
> Object is the parent class of all objects in Unreal. All of the functions in the Object class are accessible everywhere, because everything derives from Object. Object is an abstract base class, in that it does not do anything useful. All functionality is provided by subclasses, such as Texture (a texture map), TextBuffer (a chunk of text), and Class (which describes the class of other objects).
>
> Actor (extends Object) is the parent class of all standalone game objects in Unreal. The Actor class contains all of the functionality needed for an actor to move around, interact with other actors, affect the environment, and do other useful game-related things.
>
> Pawn (extends Actor) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.
>
> Class (extends Object) is a special kind of object which describes a class of object. This may seem confusing at first: a class is

> an object, and a class describes certain objects. But, the concept is sound, and there are many cases where you will deal with Class objects. For example, when you spawn a new actor in UnrealScript, you can specify the new actor's class with a Class object.
>
> With UnrealScript, you can write code for any Object class, but 99% of the time, you will be writing code for a class derived from Actor. Most of the useful UnrealScript functionality is game-related and deals with actors."

Like Java, all UnrealScript classes are derived from a common Object superclass.

The entrypoint for most applications will be a subclass of GameInfo [41]. The GameInfo class contains information on what PlayerController class to use, as well as information about the camera.

5.1.3 Code Examples

This section is dedicated to code examples, as this is an effective way of showing the workings of UnrealScript. It is worth noting that UnrealScript is not case-sensitive, so any use of caps are simply the result of adhering to common conventions. To start off, let us look at how states are managed in UnrealScript. Here is a small example with a imagined walk function, an idle state and an attack state:

```
class Creature

function Walk()
{
        //Walk randomly
}

auto state Idle
{
Begin:
        Sleep(3);
        if(!SpottedPlayer())
                Walk();
        else
                GotoState('Attack');
        Goto 'Begin';
}
```

```
state Attack
{
        function Walk()
        {
                //Walk Towards player
        }
}
```

The 'auto' keyword is used to determine a starting state; i.e. any new objects of this type will jump straight to the idle state. In this case, any new Creatures will start out idle. The 'Begin' label notes the start of state code. State code can contain jumps between labels and execute latent functions like 'Sleep(n)'. In fact, latent functions can only be executed in state code [40]. As we can see, the Creature class has a member function *Walk*. If different functionality for this function is desired in another state, then it is easy to override it, as in the *Attack* state.

5.2 UnrealED

This section will describe the tools of the Unreal Editor. The individual subsections will each explain a single tool or concept which together form the editor.

5.2.1 Kismet

Kismet is a visual scripting system for the UDK. It is a graphical, node-based script editor structured so that non-programmers such as artists and level designers can design game logic. Kismet script is saved into a map, thus it is not intended to drive general game rules, but per-level specific events. Figure 5.1 shows a screenshot of the Kismet editor. As seen in the figure, the user is exposed to a canvas where events and variables may be dropped onto. Logic is designed by dragging connectors between these nodes. Since kismet supports math, conditional logic, event handling and actions there is very little Kismet cannot do. An obvious advantage of this system is that a development team's programmers do not need to spend their time on writing simple boilerplate scripts.

Figure 5.1 above shows an example of setting up a simple third-person camera using Kismet. When the event "Player Spawned" is triggered, the "Attach to Actor"-action is triggered. "Attach to Actor" takes in two parameters, namely

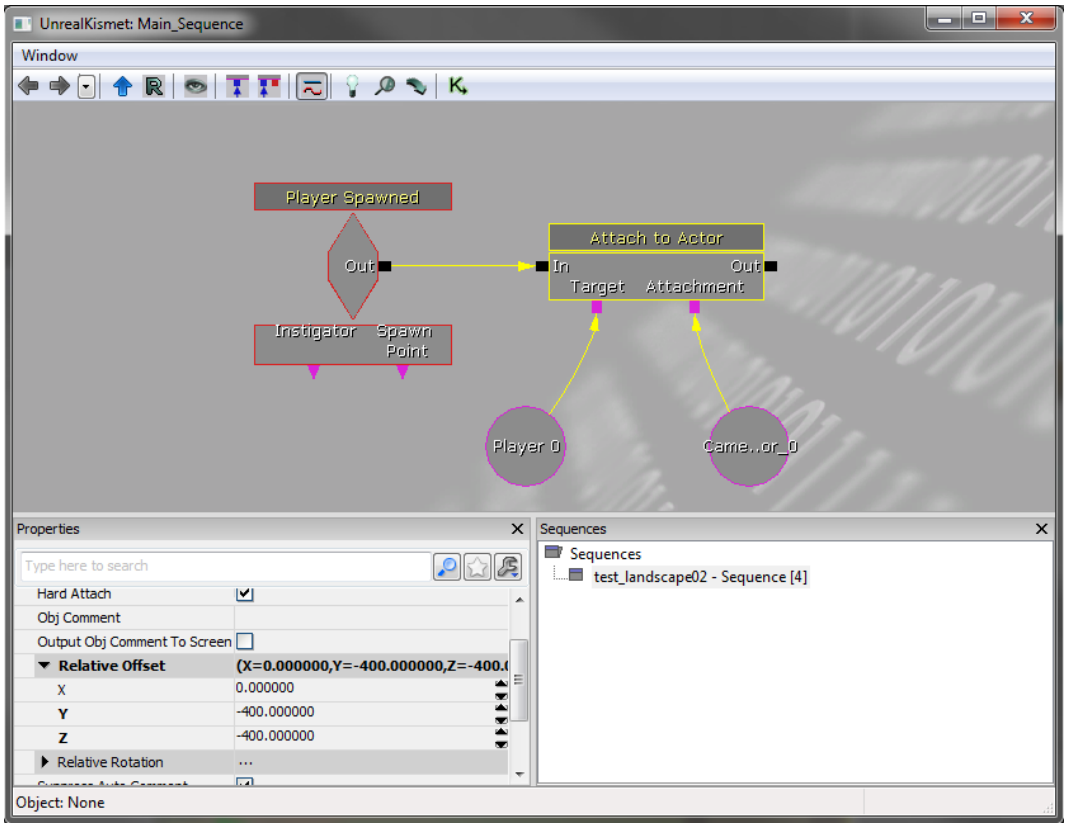


Figure 5.1: A screenshot of the Kismet editor

the target and the attachment. The parameters are respectively set to the player and a camera, and the offset from the actor to the attachment is set to move the camera up and behind the player. The game world will now be viewed through this camera at an offset relative to the player's position.

This small example shows another benefit of Kismet: It can be used to prototype functionality. Instead of using a lot of time writing a versatile third-person camera in UnrealScript, a simple routine can be created in minutes using Kismet to roughly get the same functionality.

5.2.2 Archetypes

An archetype in the UDK is a way of exposing an UnrealScript class in the editor. As an example, a simple house can be considered. A programmer creates a house class with two mesh components: A static mesh for the main building and a skeletal mesh for the animated door. In addition a particle emitter is placed in the house's chimney so that smoke can rise from it when somebody is home. The programmer, being aware of Archetypes, creates logic so that the location of the door and particle emitter is relative to the position, rotation and scale of the house. In addition a variable is created to tell whether the house is inhabited or not. If the house is inhabited, smoke is rising and the door is shut, in the opposite case no smoke is rising and the door is open. Inside the UnrealED the class is located and an archetype is created based on the class. When looking at the archetype variables the programmer chose to expose can now be altered. For instance the static mesh can be set, and the offset to the smoke emitter can be updated to fit the visual representation. Figure 5.2 shows the visual result of a similar case inside the editor, with the archetype's parameters exposed.

This example identifies two key components of what archetypes can do. First of all it is an easy way to connect logic to a game object. When the house is placed in the game world, it will operate as specified by the programmer. Secondly, it enables non-programming-savvy users to alter the properties of an object without touching the code, assuming the programmer has exposed enough variables to the archetype. In addition it is easy to create objects that behave similarly, but look different. The class in the example can create a wide range of house archetypes by simply altering the house mesh component. Another important property of archetypes is that each instance of an archetype reference the base. If a thousand house instances has been placed in a level, but the artist wants to alter the base archetype's mesh, the instances will be updated accordingly. This is a one-way relationship, thus altering the properties of an instance will not update other instances nor the base archetype.

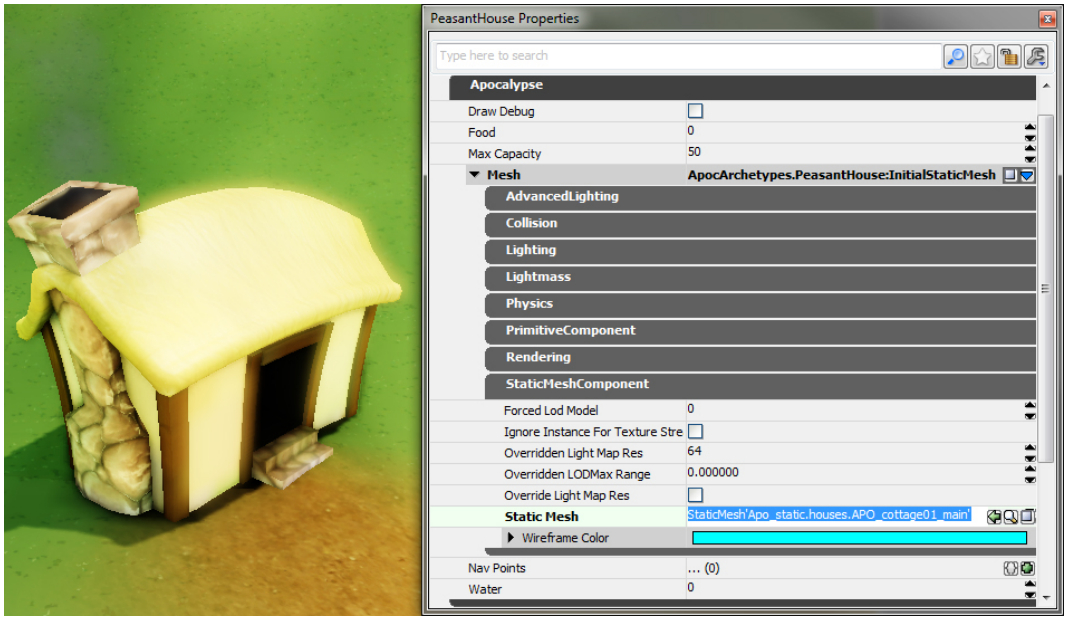


Figure 5.2: A screenshot from UnrealED showcasing a cottage visualized from an archetype. The archetype's exposed parameters are shown to the right.

The use of archetypes explained above does not cover the whole story. Any object in the UnrealED can be used to create an archetype. At any time a user may chose to save a "snapshot" of an object as an archetype. When there are as many parameters involved per object as in the UDK, this may prove to be a valuable time saving strategy.

5.2.3 Matinee

Matinee is a tool in the UDK to keyframe actors' properties over time. Its functionality ranges from creating per-level animations of movable doors and elevators, to cut-scene direction with complex animations and camera effects. Matinee is tightly coupled to Kismet, so that a Matinee sequence is initiated through Kismet. Matinee can also spawn Kismet events at arbitrary times in a sequence. A screenshot of the Matinee interface can be seen in Figure 5.3.

Matinee is structured around groups. A group contains a set of tracks controlling the properties of a single actor over time. A property is varied over time by placing keyframes on a track to control the value/time relationship. The value of a property between two keyframes is interpolated in a fashion determined by the user. The four interpolation schemes available are Curve, Curve(Break),Linear and Constant. There are many types of tracks to control

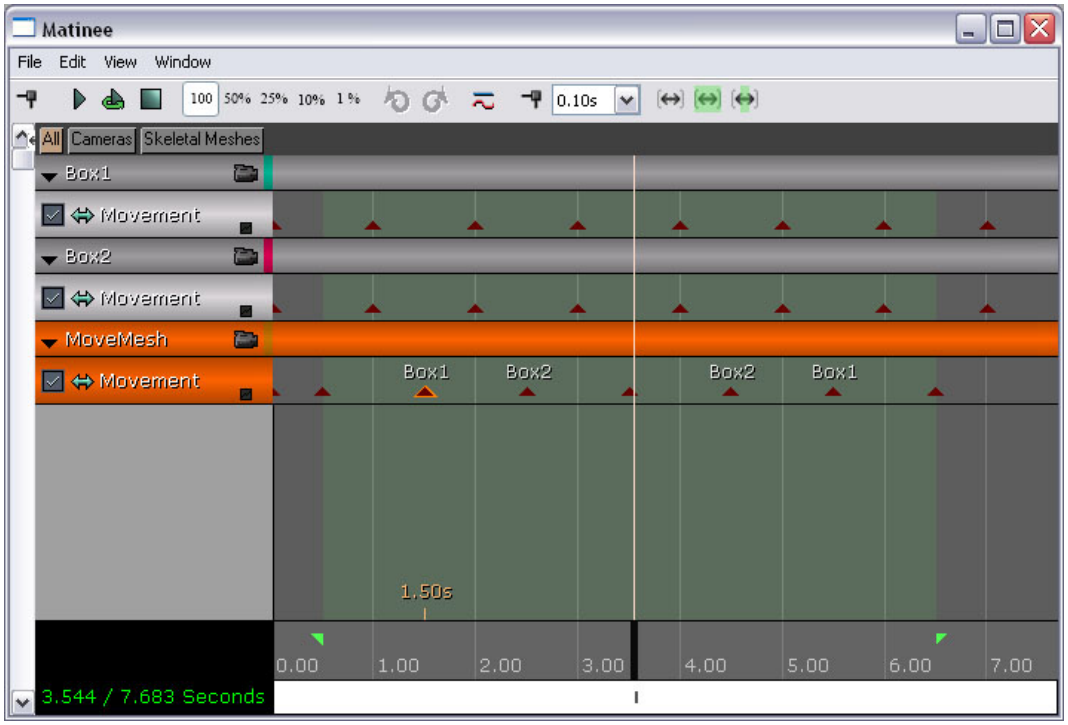


Figure 5.3: A screenshot showcasing the Matinee interface. The dark gray and orange horizontal bars are groups. All three groups have one movement track each

different properties of an actor, among these the most important track types are:

Movement Track: Is used to move an actor over time. When a movement track has been created, a curve is displayed in the editor. The curve points may be edited directly here instead of in Matinee to give the user better visual feedback. The rotation of an actor is also accessed through the movement track.

Float, Vector and Color Track: These three track types are equal, but differ in the property types they can control. The tracks let the user change an actor's property over time such as the color of a light or the DrawScale of static mesh.

Event Track: As mentioned introductory Matinee can be used to fire Kismet events. The event track contains the functionality to name an event to be fired at a specific time. When an event has been defined on an event track, it creates a new output connector on the Matinee node in Kismet. The user may then connect this connector to any Kismet event and it will be fired when the Matinee sequence reaches the specified time.

Anim Control Track: This track type is used on skeletal meshes to specify what animations are to be played at different intervals of the Matinee Sequence. In order to use this track the animation set or sets to be used must be imported into Matinee.

ParticleSystem Toggle Track: Enables the user to toggle on and off particle emitters. This track type is useful for keying particles to be spawned at specific times such as when a character lights a cigarette or a weapon fires to reveal a muzzle flash.

Director Track: Is a little peculiar compared to other track types. First off all it is not tied directly to an Actor, but is meant to control the player's view of a sequence. When doing a cut-scene from different angles the director track enables the sequence to be view through different cameras. The cameras are set up outside the director group with their own tracks for e.g movement and varying field-of-view, but the active viewpoint is determined through the director track.

Fade Track: Controls fading of the rendered Matinee sequence into a specific constant color.

Slomo Track: Controls the speed of all the game actions, physics, particles, sound etc. in the sequence by altering the playback speed. This track

type can be used to produce ”bullet-time” effects which are common in video games.

5.2.4 Material Editor

UDK's Material Editor is a visual shader and material authoring tool. Like many other tools of the UnrealED it is a node-based graph tool where arrows are dragged between nodes to control the logic. The Material Editor allows, like Kismet, non-programmers to add functionality to a game or application. The Material Editor is actually just a visual wrapper around *High-Level Shading Language(HLSL)* [87], like Kismet is a visual wrapper around UnrealScript. A screenshot of the material editor can be seen in Figure 5.4. The Material Editor can in addition to create materials for mesh-based objects also be used to create custom post-processing effects. An important notion on materials in the UDK is that for similar materials, the user can create a parameterized base material, and then multiple material instances. The parameters of an instanced material may subsequently be altered without the need of a shader recompilation.

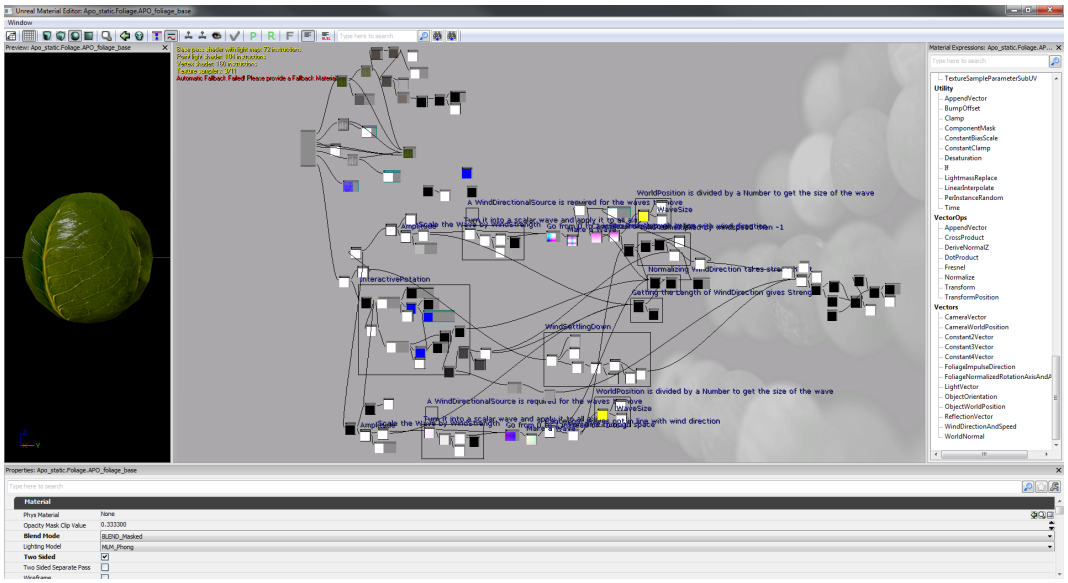


Figure 5.4: A screenshot of the Material Editor

The UDK supports the creation of a wide array of materials. There are four different available lighting models: Phong, Anisotropic, Non-Directional and Unlit. In addition, custom lighting is supported so making e.g. a cel-shader is relatively easy. A material can also be blended in six different ways: Opaque,

masked, soft-masked, translucent, additive and modulated. An opaque material means that the rendered object should be fully opaque. Masked materials use a texture to determine whether a pixel is opaque or transparent. To avoid aliasing, which is common with masked blend-mode, soft masked blending is also available. When using this blend mode, a pixel is not only either fully opaque or transparent, but can be somewhere in between. Using soft masked blending is more expensive than regular masking. The translucent blend mode is used to simulate materials such as colored glass where the material alters the colors of the underlying frame buffer. The additive and modulated blending modes respectively adds or multiplies the underlying colors with the material's output.

Advanced materials easily surpass hundreds of nodes since e.g multiplying two numbers requires three nodes. To cope with the visual complexity of a large number of nodes, a grouping tool is provided so that the user may perceive a number of nodes as a single operation.

The available material nodes are divided into 16 groups. Some of the nodes are present in multiple groups:

Color features nodes for color desaturation and color sampling behind the current pixel (for transparent materials).

Constants contains one to four dimensional floating-point vectors, and nodes sampling constant colors from either a particle emitter or the vertices of a mesh.

Coordinates contains nodes related to spatial properties of an object such as position, orientations and radius. Other coordinates-related nodes such as texture coordinates, screen position and lightmapUVs are also in this group.

Custom consists of the nodes "Custom" and "Custom Texture". Custom is a node wrapper for a custom HLSL function. The use may specify inputs, and the output format, and write a fully custom HLSL-function. "Custom Texture" is provided so that a custom-node may refer to a texture.

Depth provides nodes for sampling the depth values of a scene.

Destination contains nodes for sampling depth and color behind a pixel thus overlapping with the Color and Depth groups.

Font has nodes for using fonts with materials.

HighLevel contains the three unrelated nodes: "AntiAliasedTextureMask", "Distance" and "SphereMask". AntiAliasedTextureMask is a texture sampling node working well with soft masked alpha blending. Distance returns the euclidian distance between two 1-4 component vectors. SphereMask is a circular mask where the user can specify the opacity falloff.

Lens Flare contains nodes related to creating simulated lens flares.

Math is a large group housing common computer graphics math operations, many of which are native HLSL-functions. The group consists the nodes: Abs, Add, Ceiling, Cosine, CrossProduct, Divide, DotProduct, Floor, Floating-point modulo (FMod), Fraction, LinearInterpolate, Multiply, Normalize, OneMinus, Power, RotateAboutAxis, Sine, SquareRoot and Subtract.

Parameters houses scalar, vector and texture parameters used with material instances.

Particles contains nodes aimed at materials used with particle systems.

Texture is a large group made up of nodes related to texture sampling. The UDK handles both 2D-textures and texture cubes.

Utility contains a number of unrelated nodes. Among the more interesting are: "BumpOffset", "If" and "PerInstanceRandom". "BumpOffset" is the UDK's term for Parallax Mapping [89], a common technique for increased perceived depth on flat surfaces. "If" is a node for doing a comparison between two values A and B, where the output depends on A being greater, equal or smaller than B. "PerInstanceRandom" is self-explanatory and randomizes a number per instance.

VectorOps are vector operations, some of which are already listed in the "Math"-group. The remaining operations are: AppendVector, DeriveNormalZ, Fresnel, Transform and TransformPosition.

Vectors overlaps with the constant group, though provides additional vectors such as the camera vector, light vector and reflection vector.

5.2.5 Cascade

Cascade is the Unreal Editor's particle system authoring tool. The toolset is quite extensive and delivers functionality to create almost any kind of particle system. A screenshot of Cascade's user interface can be seen in Figure 5.5. As the figure shows, the interface is divided into four main areas. The upper

left area previews the current particle system. The upper right area visualizes emitters as described below. The lower left area exposes the parameters of a particle system to the designer. The lower right area is a curve editor. Cascade's curve editor is used to fine tune the behavior of particles over time. A Cascade particle system is constructed from one or more emitters. A particle

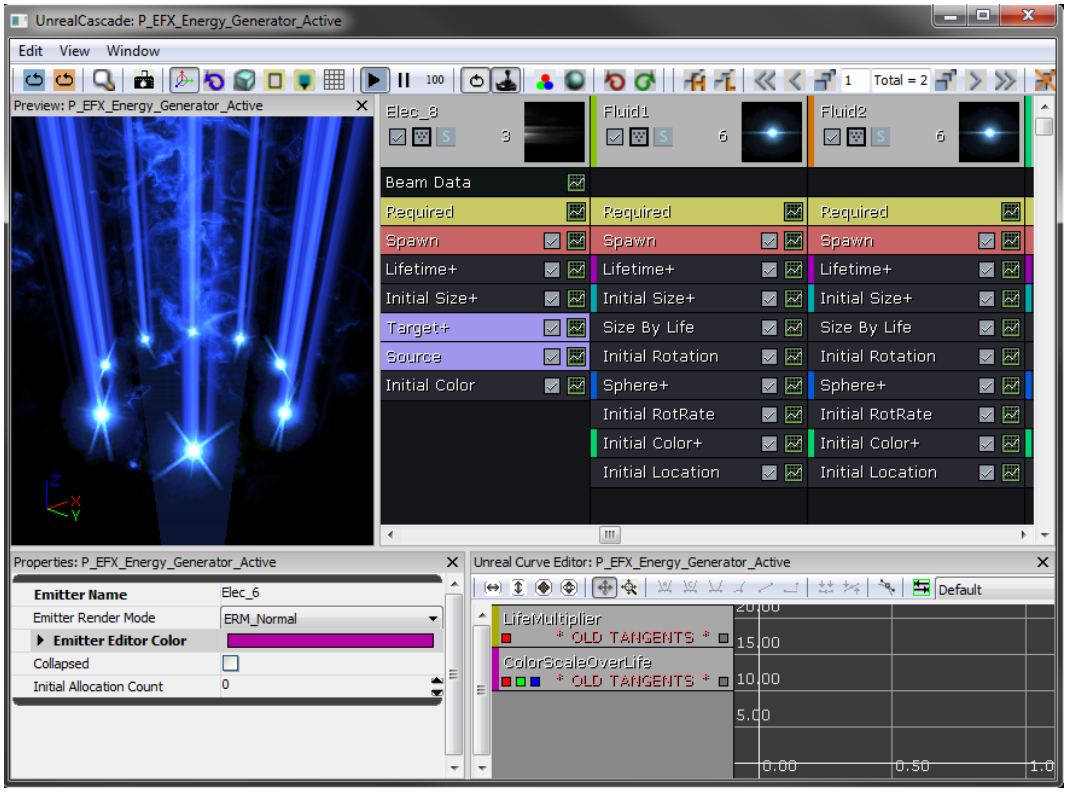


Figure 5.5: A screenshot of the Cascade, the Unreal Editor's particle system authoring tool.

emitter is the point where particles come to life, or "spawned". A particle emitter contains modules affecting how particles behave and what they look like [65]. In Figure 5.5 three emitters named "Elec_8", "Fluid1" and "Fluid2" are depicted in three columns. The boxes of each column are the modules affecting the particular emitter. All modules are optional except the *Required Module* and *Spawn Module*. The basic property groups of the Required and Spawn Module are:

Emitter: Contains properties to set the material to be used for the sprites emitted from the emitter, how the sprites should be aligned to the screen and which sorting algorithm to use for the emitter.

SubUV: When a material to be used with a particle system is created, it is common to sample multiple pictures from a single texture of e.g. a smoke puff so the individual particles will not look identical. SubUV properties deal with the splitting up of a texture and blending between the sub-images of the texture.

Normals: Determines whether the sampled normal of a particle sprite should be aligned planarly to the screen, or sampled from a sphere or cylinder.

Duration: The lifetime of an emitter and loop count.

Delay: This group makes it possible to create a delay between the time a particle system is activated and the spawning starts. Useful to time interaction between emitters.

Rendering: Here the designer can limit the number of particles drawn per emitter.

Spawn: Controls the distribution of particle spawning over time.

Burst: If an emitter should spawn a given number of particles at a given time, bursting controls this behavior.

Cascade: Controls how the emitter is drawn within cascade.

In addition to the "Required" and "Spawn" modules, several other exist to fine-tune an emitter. Note that some groups contain only a single module, thus the terms "group" and "module" are used interchangeably. The optional module groups are:

Type Data: Cascade supports other types of particles than screen-aligned sprites. In the type data group an emitter may be modified to spawn a mesh instance per particle instead of a sprite. Another particle type is "Beam". Beams have a source and target point and can simulate effects such as lightning between a Tesla coil and an unlucky character. A third particle type is "Animation Trails". Animation trails are set up in collaboration with the AnimSet Viewer and are used to spawn particles behind a fast-moving object to exaggerate rapid motion.

Acceleration: Contains modules to control particle acceleration and deceleration.

Attraction: Attractors are modules to draw particles towards point or lines. Examples of attractor use are simulation of effects such as worm holes or magnets.

Beam: This group of modifiers control the behavior of a beam particle emitter.

Collision: The collision allows action to be taken when a particle hits something in the game world. A common coarse of is to kill a particle upon collision.

Color: Controls the color of a particle over it's lifetime.

Event: This group handles the firing and reception of events based on particle spawning, death, collision etc.

Kill: For large systems it is beneficial to kill particles outside the view. Killing modifiers removes particles outside a bounding box volume or at a certain height.

Lifetime: Not to be confused with "Duration" under the Required group. The lifetime module contains distributions from which the lifetime of individual particles are randomly chosen.

Location: This group contains modules to position emitters and control the direction of particles.

Material: Operates on the materials used on the particle emitter. Can be used to override a material's parameters to specify it to a particular emitter.

Orbit: The orbit module is used to rotate or offset particles away from its center.

Orientation: The orientation module locks a particle sprite to a world space axis, overriding sprite particles to always face the camera.

Parameter: Interfaces with parameters from UnrealScript or Kismet to control the particle emitter through programming.

Rotation: Controls the rotation of sprite particles.

Rotation Rate: Sets the rotation behavior for mesh-based particle systems over life.

Size: Controls the scaling of particles.

Spawn Per Unit: Controls the spawning rate based on the distance an emitter has traveled. When for instance fire is emitted from a moving torch, the spawning rate can be adjusted to spawn many particle when

the torch moves fast. When the torch has little movement fewer particles are spawned, thus the fire will appear realistic in a shifting environment.

Velocity: Controls the velocity of particles.

Particle systems are largely used to simulate natural phenomenas such as fire, rain and smoke. To simulate the chaotic nature of particles, many of the modules use distributions from which random values are picked. Cascade supports a number of different distributions to accommodate different situations. The most common distribution types include constant, constant over time, vector and vector over time. The distributions varying over time are handled in the curve editor.

5.2.6 AnimSet Editor

UDK's AnimSet Editor is the base tool for setting up and previewing animations on skeletal meshes. Figure 5.6 show a screen capture from the user interface. The AnimSet Editor exposes several of the features showcasing the maturity of UDK's animation system:

Sockets are points attached to specific bone, optionally with a translation or rotation offset. When a socket has been created it will move accordingly to the bone and its offsets. The benefit of sockets is that skeletal or static meshes may be attached to them. A common area of use for sockets is weapons. Since modern combat games commonly feature an arsenal of weapons with different models, it is beneficial to separate the weapon user model from the weapon itself. Exporting a different character mesh for each weapon is a waste of memory and time, and sockets removes the need for such a process.

Animation Notifiers is meta data put into an animation sequence at certain time. Animation notifiers triggers events which are snapped up by the engine and handled. The notification types supported are: CameraEffect, FootStep, Kismet, PlayParticleEffect, Rumble, Script, PawnMaterialParam, ViewShake, Sound, Trails and PlayFaceFXAnim. The CameraEffect notify is used to notify the camera that a certain effect should take place. The FootStep is used in collaboration with physical materials so that when a character places its foot down, a sound cue is played based on the underlying material. A Kismet notify is similar to the Script notify, and spawns an event Kismet or UnrealScript can listen and react to. PlayParticleEffect is self-explanatory, and will play a particle effect. The Rumble notify sets off the vibration in connected gamepad controllers. PawnMaterialParam interfaces material parame-

ters so that e.g a magic sword can turn red when it is swung. ViewShake simply notifies the camera to shake, an effect commonly employed in first person perspective games when e.g a grenade goes off nearby or a marauding giant stomps his feet in the ground. Sound notification is probably the most commonly employed animation notifier, and lets the user time sound effects to played at a specific time in an animation sequence. The animation trails notifier works along Cascade to create particle trails, typically behind a fast moving object. PlayFaceFXAnim obviously triggers a FaceFX animation.

Mirror Tables exploits the observation that most skeletal mesh hierarchies are symmetrical. Each bone can therefore be set up to a symmetrical counterpart. This procedure allows an animation to be flipped along the symmetrical axis of a skeleton. As a result an animator may create an animation sequence for a right-handed character, and the sequence can easily be used for a left-handed character as well.

Morph Targets , or blend shapes, is a different method of doing animation than skeletal animation. A morph target is a saved pose of a mesh where some or all of the vertices differ from the base mesh. When animating with morph targets the vertex positions are interpolated between the base pose and the morph target. This scheme allows fluent transitions from one pose to another. Morph targets are commonly used in facial animation where an animator builds a library of facial poses. The poses typically represent the facial expressions of a character when pronouncing the basic sounds in a language. Such a scheme enforces reusability since all kinds of dialog can be built around a number of base poses. Another commonly employed use of morph targets is structural damage. For instance a vehicle can be modeled in a pose where a part of it is damaged. When the vehicle is damaged in-game, the mesh is morphed into the damaged pose to visually represent the damage.

Cloth Physics are parts of a skeletal mesh that should be animated by physical simulation rather than keyframed animation sequences. Cloth physics are used to give a convincing visual representation of ”flappy bits” such as capes, pony tails and flags. In the AnimSet Editor the user can assign bones to be treated as cloth. Note that these bones must be skinned to the mesh as any other bone. Finally parameters such as thickness, stiffness and damping must be set to fine tune the look of the simulation..

Soft Body Physics resembles cloth physics but are used to simulate volumes rather than thin sheets. The simulation is run on a set of tetrahedras

whose resolution is controlled independently of the graphical mesh. During simulation the graphical mesh is skinned to the simulated volumes. Two parameters controls the stiffness of the soft body preventing the change in volume and stretching. Good candidates for soft body physics simulations can be a beanbag chair, excessive body fat, the crown of a falling tree etc.

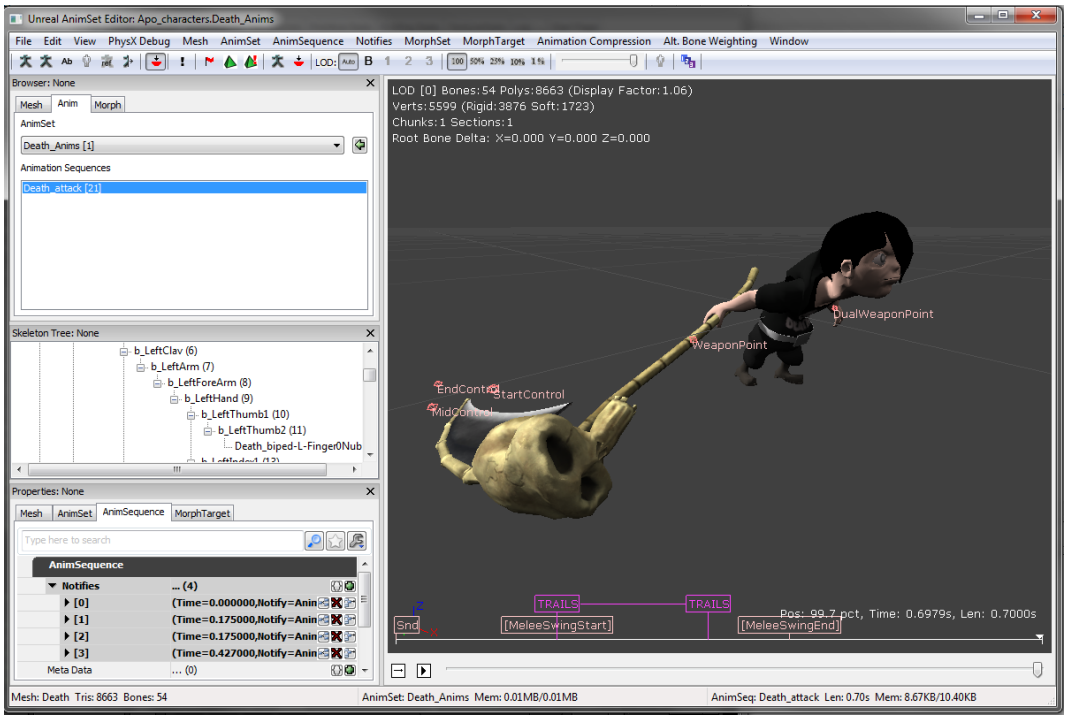


Figure 5.6: A screenshot of the AnimSet Editor

5.2.7 AnimTree Editor

This section is dedicated to describing the AnimTree Editor tool present in the UDK package. The aim will be to give a short introduction into what the tool does, and how it can be used. The concept of *animation blending* is used extensively within this section. Animation blending is the process of taking a base animation sequence and altering it by combining it with some factor. This can be another animation sequence, the speed of the game object, the physics state or anything else one can think of. The result is an interpolated animation sequence. Any deeper explanation of the subject is outside the scope of this report. Unreal Engine 3 deals with this by using something called *animation trees* or *blend trees* [24].

Overview

The AnimTree Editor is a tool for visually creating animation trees. Animation trees makes it possible for the developer to specify how and what parts of a 3d model will be animated [25]. To help visualize this concept, we will look at a concrete example. This will be illustrated by Figure 5.7.

Let us say you have a 3d model of a soldier. You might want to play an *idle* animation when he is standing still, emulating breathing, looking around and other things to create a lifelike character. You want this animation to play on the entire body. This can be done by connecting a regular Animation Node (called *FullBodyAnim* in Figure 5.7) together with a BlendBySpeed Node, telling it to only play this animation when the speed of the character is 0. For speeds above that you could specify a walking animation. You could also specify that the playback speed of the animation should match the speed in the game itself. In addition you might want to play an animation when the soldier shoots, this should ideally blend with other animations, so that only one animation sequence is needed for both shooting while running and standing still. This is done by hooking up another Animation Node (called UpperBodyAnim in Figure 5.7) and specifying that this node will only affect the upper skeleton bones of the model. That way, any animation happening in the lower part of the skeleton will be retained. Also worth noting in Figure 5.7, is an AnimNodeBlendByPhysics node. This makes for differing animations depending on what the physics state the object is in within the game.

As one can see, this makes for a fairly intuitive and very powerful system for controlling and blending together animations. A more complex tree can be seen in Figure 5.8.

In short, the main purpose of the AnimTree Editor is to provide a visual interface to programmers, animators and technical artists for specifying what parts of a skeleton should be affected by animation, blending between animations, providing *On-Demand* playback of animations and providing direct per-bone manupulation of a 3d model's skeleton [30].

Nodes

Here we will go into some further detail to explain the function of the nodes that can be used. There are essentially four different groups of nodes that can be used to create an animation tree:

Animation Node The *Animation Node* group is the largest. It contain the main animation blending nodes. What differs between them is how the

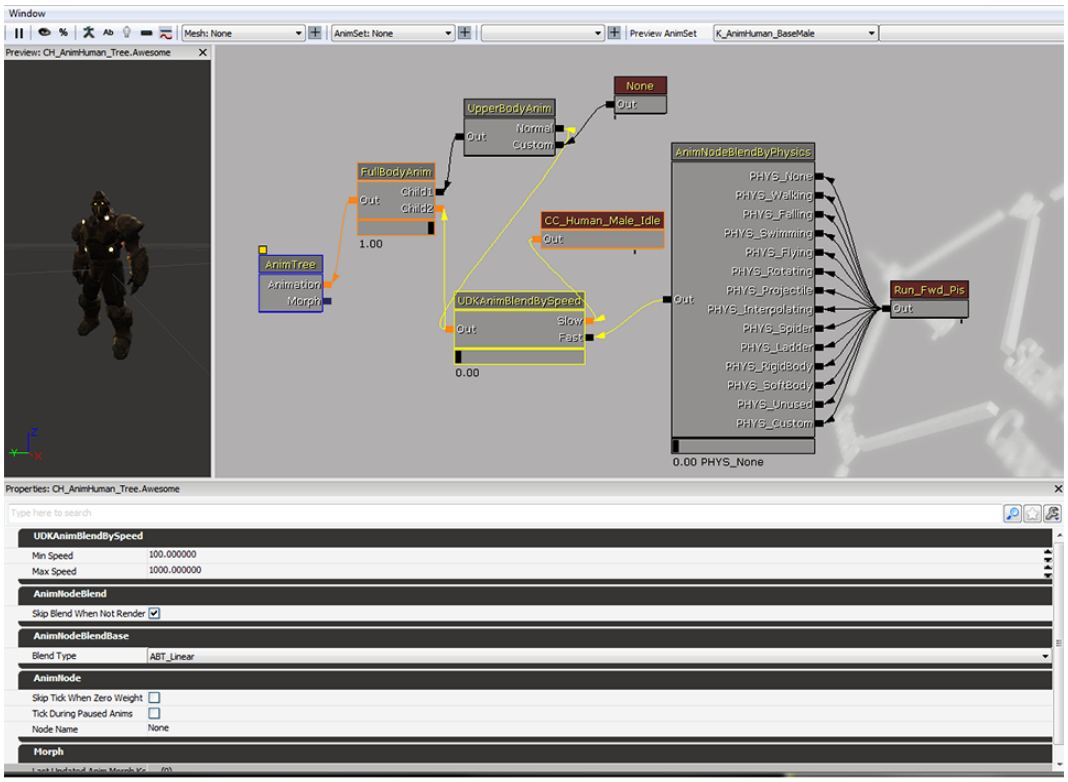


Figure 5.7: Example. Left: 3d model for running tests. Middle: The animation tree structure itself. Bottom: Properties pane for the UDKAnimBlendBySpeed node.

blending is determined. The similarities being that the nodes usually takes several inputs and output a blended animation. There are possibilities to blend by speed, posture, physics, direction. There are also some special blending cases, like a node for additive blending.

Skeletal Control The *Skeletal Control* nodes, deal with the direct manipulation of bones in the animation skeleton. As an example, this can be used to control where the player avatar is looking, by affecting the neck bone. In addition, there are controls for twisting, rotating and scaling bones, controlling entire limbs, foot placement and recoil.

Morph Node Morph Targets are a way to modify a mesh in realtime, but with more control than bone-based skeletal animation [32]. This is useful for controlling facial expressions, for example. The *Morph node* lets several Morph Targets be blended together. There's also a node for controlling the weight (strength) of Morph Targets.

Animation Sequence This is the lowest level of node and makes up the end leaves of the tree. An *Animation Sequence* is a reference to the animation that is to be played. An empty node might also be specified, to reserve the spot for later manipulation by UnrealScript (see Section 5.1) or Matinee (see Section 5.2.3).

5.2.8 Lightmass

Lightmass is the UDK's global illumination solver. It renders high-quality static light and shadow maps using techniques that are too complex for real-time integration on current hardware. The Lightmass solver must therefore be executed offline, before a level is loaded. Since complex light simulation can be extremely time-consuming, the UDK provides a tool named "Unreal Swarm" allowing distributed rendering over multiple machines. The concept of distributed rendering, exploits computational parallelism, and is ubiquitous in high-end visualization rendering for film and television where massive clusters called *Render Farms* work around the clock to crunch numbers into art [91].

The following description captures the highlights of Lightmass' features:

Area Lights and Shadows: Lightmass differs from the old Unreal Engine 3 static light system with its us of area lights. An area point light is a light source where light emits from a sphere as opposed to a traditional single point. For directional light sources the light is emitted from a disk instead of from a single direction. Area lights produce more realistic shadows, where the sharpness of the shadow edges is controlled by the

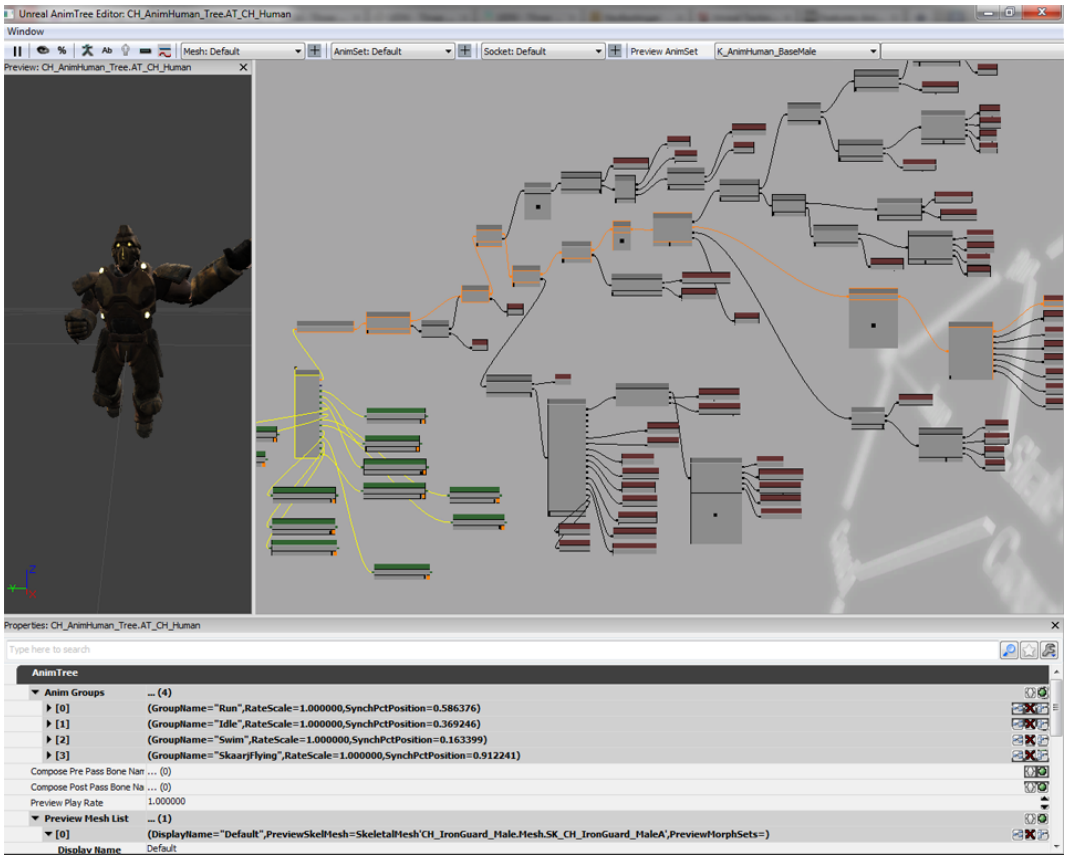


Figure 5.8: A complex animation tree in the AnimTree Editor

size of a light, and the distance to the occluder. A comparison between point and area lights can be seen in Figure 5.9.

Figure 5.9: A comparison of shadows without area lights (left) and with area lights (right). With area lights the size of the shadow penumbra [94] grows further away from the occluder.

Indirect Lighting: Lightmass has the ability to bounce light traces and create indirect lighting. True indirect lighting makes the life a lot easier for a level designer, since he or she does not need to manually place ”fill lights” to approximate lighting of surfaces that are not directly reached by a light source. Lightmass' indirect lighting system also creates shadow from indirect lighting and simulates ”color bleeding” so that the color of a surface is picked up when the light bounces [46]. A set of screenshots illustrating indirect lighting with Lightmass is shown in Figure 5.10.

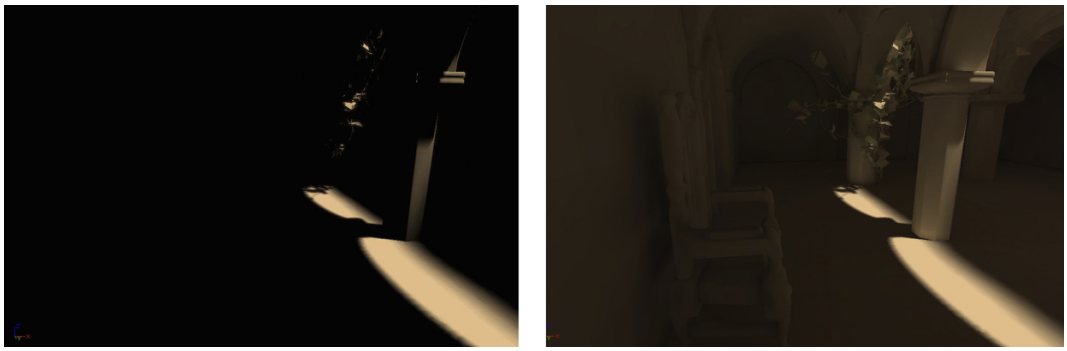


Figure 5.10: Two screenshots illustrating the visual difference between a scene lit by only direct lighting (left) and with four bounces of indirect light (right).

Translucent Shadows: Another feature of Lightmass the ability to render translucent shadows. As discussed in Subsection 5.2.4, a material must have its blend mode set to ”translucent” to enable this feature. Translucent shadows originate from light passing through a semi-transparent surface so that the colors of the surface affect the resulting shadow color. An example of translucent shadows can be seen in Figure 5.11.

Figure 5.11: Light passing through a stained glass window creates a translucent shadow when rendered with Lightmass.

Ambient Occlusion: Ambient occlusion is a technique to enhance the sense of proximity between objects in a scene. By adding additional shadow where objects are close together a greater sense of coherence in the scene can be achieved. Ambient occlusion is not a physically correct procedure, but tends to produce result perceived as more photo-realistic since the result ”softens” the commonly ”hard” light of computer graphics Lightmass features the possibility of adding ambient occlusion in addition to indirect lighting. The ambient occlusion in Lightmass is calculated by approximating the indirect shadowing by an uniformly lit hemisphere, thus the resulting look imitates outdoor shadows on an overcast day [85]. Several parameters are available to the user to tweak Lightmass into producing the desired ”look” in a scene. A scene with ambient occlusion enabled is shown in Figure 5.12

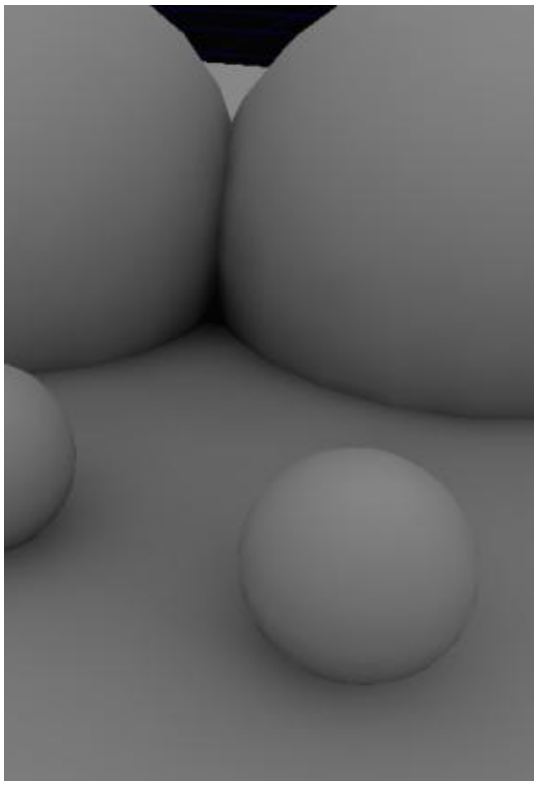


Figure 5.12: A scene shadowed with ambient occlusion. The proximity between the spheres and the floor plane is illustrated through soft shadows, simulating nearby objects occluding light from each other.

5.2.9 The Navigation Mesh

In the pre-UDK era navigation was handled by using a node graph. The node graph was created by a level designer placing out navigation nodes and defining reachability between adjacent nodes. With the first release of the UDK the old system was replaced by a navigation mesh system. The navigation mesh is connected graph of convex polygons. When an AI agent requests a path from the navigation mesh, a list of polygonal edges in the mesh the agent needs to cross to reach its destination is returned. The returned path differs from the legacy system where a list of user placed nodes the agent had to visit were returned.

Epic lists a wide number of benefits from using a navigation mesh over a node graph [79]. The most important are:

Reduction in node density: A large area can be represented by a single polygon. In the old system several nodes had to be placed over a large area for an agent to pathfind properly within it. A sparse graph reduces path finding time.

Optimization of datastructures: The legacy path nodes suffered from a big overhead from parental classes. The new system however, does not, and has consequently a smaller memory footprint.

Obviation of FindActor: The old path finding system had to locate the nearest path node through octree-lookups to locate nearby path nodes and raycasting to determine which node is the nearest. With the navigation mesh this procedure is obviated by simply finding the polygon in which the agent resides.

Better pathing behavior: The movement from a waypoint graph can be unnatural e.g when an agent first moves to the close node behind it in order to turn and move back the same direction to a node further away. Such odd behavior is avoided with the navigation mesh since the list of edges to cross will reflect the ”correct” direction of the path.

Flexibility for agents of varying size: With the navigation mesh, the length of an edge is stored in the mesh. Having this information available makes it trivial at run-time to decide whether an agent of a given size may cross the edge.

In addition to these benefits, the real ”killer feature” of the navigation mesh is that it is automatically created. Auto-generating navigation meshes is done in the UnrealED by placing specialized volumes, called ”pylons” into the game

world. A pylon is simply a cube representing the boundaries in which a navigation mesh should be created. Multiple pylons may overlap, thus non-regularly shaped levels can still be bounded properly. Figure 5.13 show the navigation mesh generated from a set of pylons.

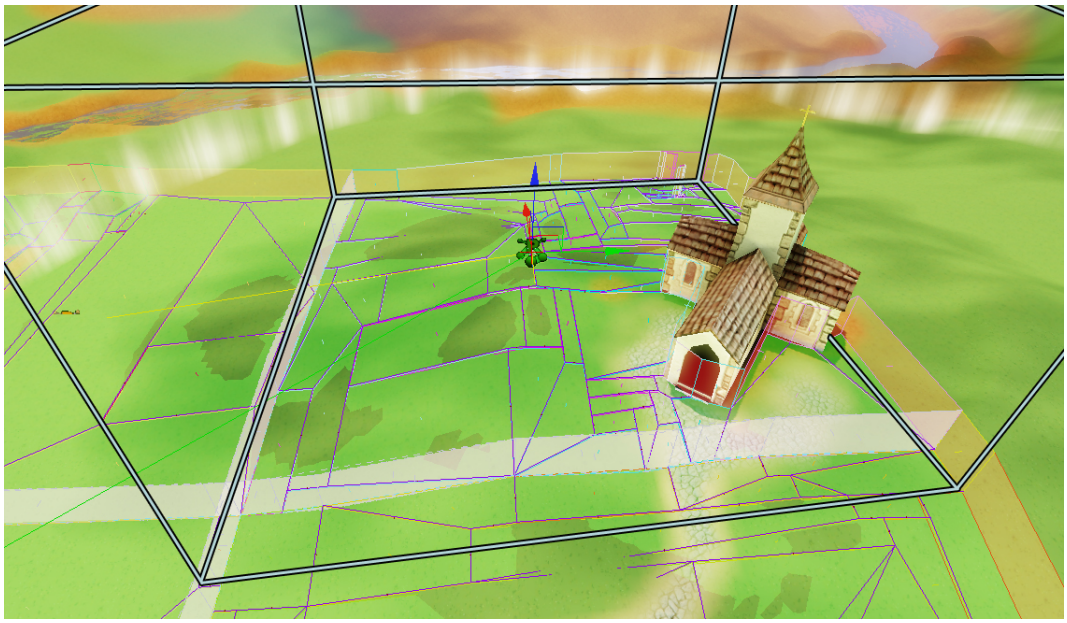


Figure 5.13: A screenshot from UnrealED showcasing a navigation mesh calculated from a set of pylons. The polygons of the navmesh are displayed with purple, cyan and dark blue lines. The outlined bounding box shows the bounds of the selected pylon. The transparent, red polygons show where the navmesh are blocking paths.

5.3 SpeedTree

This section will give a brief introduction to the SpeedTree middleware solution included with the Unreal Development Kit. The focus will be on what it actually does, rather than delving into technical details.

5.3.1 Description

SpeedTree is a middleware package created by Interactive Data Visualization, Inc. (IDV) and is used to create foliage for games and simulations [49]. The SpeedTree real-time solution, which is the one used in games, is industry-leading when it comes to foliage creation. It has been used in a number of

big titles, like *Dragon Age: Origins, Batman: Arkham Asylum, Empire: Total War, Fallout 3* and *Grand Theft Auto IV* to name a few [50]. At the time of writing, SpeedTree 5.0 is an integrated part of the Unreal Development Kit. Part of this integration are two separate tools: The SpeedTree Modeler and the SpeedTree Compiler [36].

The SpeedTree Modeler lets the user create trees or other foliage by either drawing them by hand directly in the modeler, or by creating the branches/leaves in a more traditional data-driven approach, where the user tweaks different variables to get the desired look. Figure 5.14 shows the user interface of the SpeedTree Modeler. It is possible to let various physical forces affect the look of the tree. The forces include magnet, direction, twist, curl, planar and mesh. These provide powerful tools for shaping the trees. The modeler provides the possibility to assign different materials to the branches and trunk. The transitions between materials are automatically smoothed, not requiring any tweaking from the user.

The ability to have several levels of detail (LOD) on a 3d model is important in games, as processor time is precious. There is no need to have a tree with a high level of detail if the player only sees it from a distance. LOD levels are created automatically by simply specifying how many levels you'd like. What distances they should be displayed at can be set up when they're imported into the Unreal Development Kit editor.

It is possible to import geometry into the modeler and let it interact with the foliage. The tree will automatically collide and wrap around the geometry, creating a natural look. An example of this is provided in Figure 5.15. Here we can see how the SpeedTree wraps around the well, exiting from a hole in the roof.

The Modeler also features automatic positioning of collision primitives. The user only specifies what primitives and how many should make up the collision mesh. You could for example specify two spheres and a cylinder as the collision primitives, and the software will automatically position and scale these to get the optimal coverage.

The SpeedTree Compiler compiles the tree into textures and a SpeedTree model that can be read by the editor in the Unreal Development Kit.

5.4 Summary

Throughout this chapter we have introduced the concept of game engines and the Unreal Development Kit. The chapter is included to be a suitable knowl-

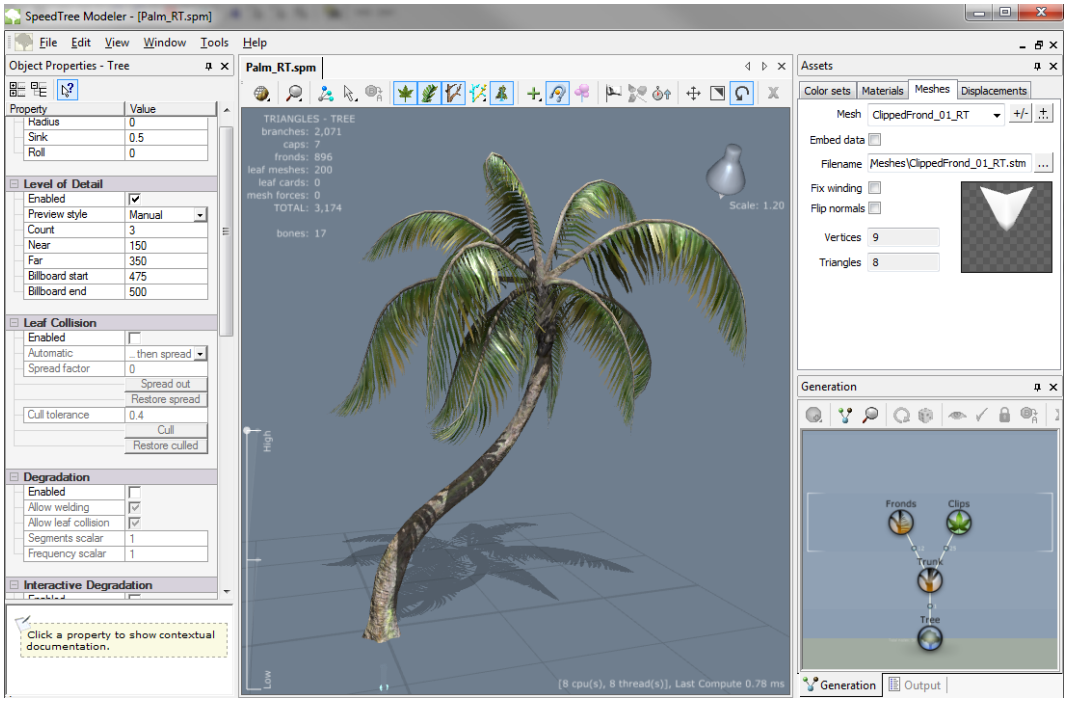


Figure 5.14: The SpeedTree Modeler user interface

edge base when we later on reference features and tools specific to the UDK. Since the project is a computer science master thesis, it is natural to assume that the only relevant parts of the chapter are the ones related to coding; e.g. UnrealScript, Kismet and Archetypes. We do, however, aim to create a game prototype, and must therefore travel beyond our native discipline. The parts of the UDK related to disciplines outside programming are therefore just as relevant. It is also worth noting that there are aspects of the UDK we have not included in this chapter e.g. fluid surfaces, fracturing of static meshes and physical materials. These are tools or concepts that were not required for our game prototype and are therefore left out.

Figure 5.15: Interaction between a mesh (well) and a SpeedTree

Chapter 6

Game Prototyping

> *"Fantastic! You remained resolute and resourceful in an atmosphere of extreme pessimism"*
> *- GlaDOS, Portal (2007)*

This chapter will describe concepts related to prototyping with focus on game prototyping. The section will start out by explaining the basic ideas and terminology related to prototyping.

6.1 Prototyping Basics

Oxford Dictionary defines a prototype as [16]:

> "A first or preliminary form from which other forms are developed or copied"

Although a little vague, this definition captures the essence of a prototype. The prototype captures the basics of a product at an early stage. Prototypes are not standardized since the prototyping concept applies to almost all constructive disciplines. There are a vast number of differences between how a new car may be prototyped compared to a software application or a vaccine. Since requirements typically form the basis of a prototype [64], prototypes from different disciplines will naturally differ. Wikipedia offers a reasonable basic categorization of prototypes [90]:

Proof-of-Principle Prototype: A prototype to test or prove functionality of parts of the functionality of a product.

Form Study Prototype: A physical prototype to establish the form of a product. The form study prototype focuses simply on the form, not texture, color or materials.

Visual Prototype: Captures the full aesthetical design of a product without showcasing its functionality.

Functional Prototype: Also called a working prototype. Covers the visual design and the functionality of a product.

As the categories show, prototypes may emphasize visual appearance and/or functionality to a varying degree. To what extent a designer wish to emphasize the form or functionality is, again, depending on the requirements for a particular product.

Another, important aspect of prototyping is the sophistication of the prototype. When for instance prototyping a car, the basic design can be drawn using pen and paper. Here the design is communicated, though not in a sophisticated manner. An improvement could be to make a 3d model of the car to better communicate surface details and to remove restrictions on the number of viewing angles for the design. The next level of sophistication could be to build a scale model of the car. In prototyping terminology a prototype's level of sophistication is called *fidelity*. A *low-fidelity* prototype should not take a long time to make, and will consequently differ a lot from the final product in both terms of design and functionality [64]. An example of a low-fidelity prototype is pen-and-paper sketching. A *high-fidelity prototype* has a higher degree of sophistication and is therefore closer to the final product. An example of a high-fidelity prototype is a scale model.

The fidelity of a prototype touches another key aspect of prototype design, namely iteration. The design cycle of a product will in most cases start out with a low-fidelity prototype such as a paper sketch. When the basic sketch has been approved, the prototype can be enhanced further using prototype techniques of higher fidelity, coinciding with Oxford's definition of the word.

6.2 Software Prototyping

This section specifies the general prototyping ideas from the section above to the field of software prototyping. The section contains three subsection. The first one discusses a common software prototyping cycle derived from *Human Computer Interaction (HCI)*. The second establishes software prototyping terminology by outlining classification schemes related to software prototyping. Last, an introduction to common software prototyping methods is given.

6.2.1 The Software Prototyping Cycle

Software prototyping is an established technique in the requirement or design phase of the software development cycle [68]. Conventional software prototyping lies within the field of HCI and involves testing user interfaces and program functionality on actual end-users at an early stage. As with general, interdisciplinary prototypes, software prototypes are iterative in their nature. Sharp [64] defines the software prototyping cycle to contain four activities:

1. Identifying Needs and Establishing Requirements
2. Developing Alternative Designs
3. Enhancing The Prototype
4. Evaluate Designs

These activities may be repeated over several iterations in an interchangeable order where the fidelity of the prototype may increase at the transition between cycles.

6.2.2 Software Prototyping Classifications

A common approach to differentiate software prototypes is to classify them in two dimensions. Prototypes can be *horizontal* or *vertical*, as outlined by usability engineering guru Jakob Nielsen in *Usability Engineering* [59]. Nielsen describes a prototype to be horizontal if it shows a broad overview of a system without focusing on the functionality of individual components. Graphical user interface prototypes are commonly horizontal. A vertical software prototype tend to focus on a single function, screen or subsystem where the functionality is close to a final product, without showcasing the unit's relation to other components of the system. The vertical prototype is used to open up and explore complex parts of an application, so that detailed requirements may be derived and clarified.

Another way to perceive different types of software prototypes is to look at their purpose. In some projects a designer or architect may wish to rapidly explore multiple designs ideas. In such a scenario multiple prototypes are designed, tested and thrown away. This prototype category is referred to as *rapid* or *throwaway* prototyping, and are typically developed using low-fidelity methods such as pen-and-paper sketching [64]. In a broad sense a rapid prototype does not form the foundation upon which an application is built, but ideas from rapid prototyping contribute to the application. A contrasting approach to rapid prototyping is *evolutionary prototyping* [13]. An evolutionary

prototype is a robust prototype which in its earliest iteration envelops the core of an application. The architecture of an evolutionary prototype should emphasize the modifiability software tactic [5] so that additional functionality can be added to the prototype at later iterations. At the final stage of the metamorphosis, the prototype has evolved into the final product. Evolutionary prototypes are helpful in uncharted territories where requirements are not fixed and evolving.

6.2.3 Software Prototyping Methods

Choosing the right prototyping method is crucial in software development. There are quite a few methods proven effective. This subsection will outline some of the most common methods.

Friedl [22] chooses to present different software prototyping methods in a diagram with increasing fidelity along the horizontal axis. Figure 6.1 shows Friedl's alignment of methods. The seven methods in the figure are:

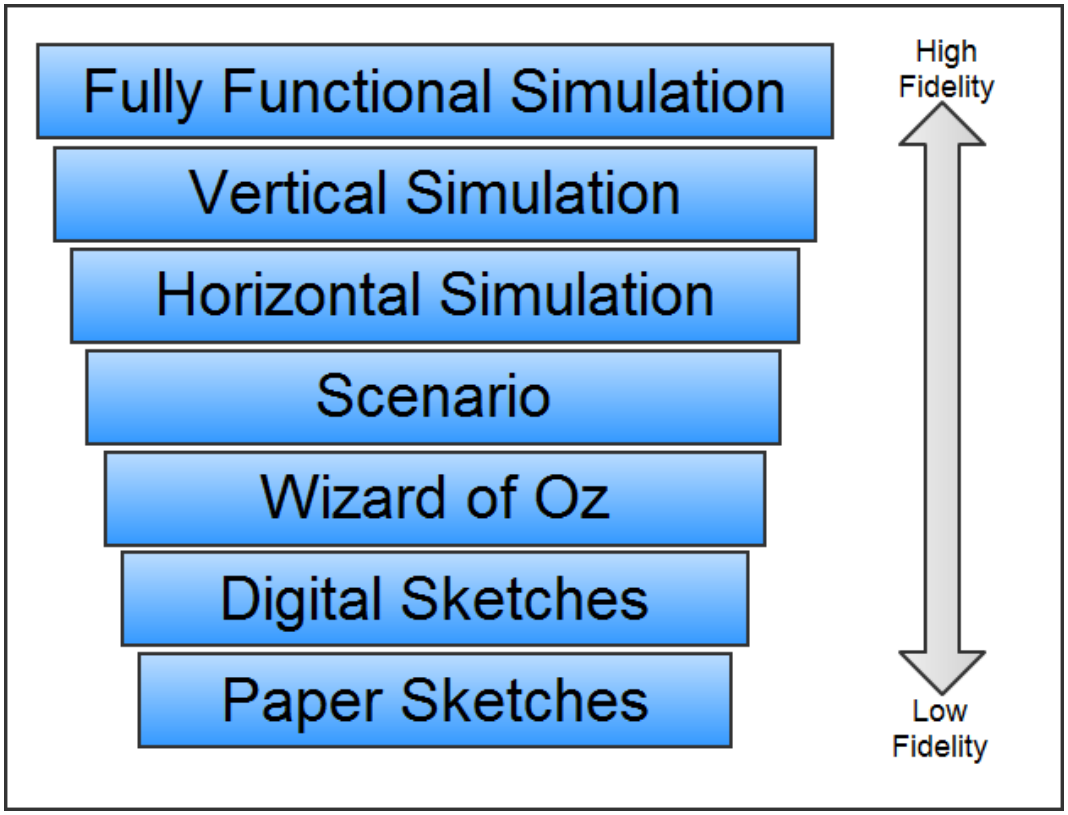


Figure 6.1: Seven different prototyping methods sorted by fidelity

Paper Sketches: Sketches are useful to design user interfaces. A paper sketch is easily created, modified and thrown away making it the method of the lowest fidelity.

Digital Sketches: As with paper sketches these are again useful to communicate a user interface. Digital sketches take longer time to create than paper sketches, but results are consequently more complete and appealing. The fidelity is therefore higher. Digital sketches will also require more sophisticated, and often expensive, tools than a sheet of paper and a pen. Examples of applications suitable for digital sketching are Adobe Photoshop, Illustrator and Flash [2].

Wizard of Oz: Wizard of Oz prototyping requires a digital prototype, and is used in user testing for graphical interfaces. A test user sits at one machine and interacts with the prototype. The feedback from the tester's interaction is not driven by software, but remotely from another machine by an operator. Wizard of Oz prototyping lets the user experience interactivity, but without the cost of actually implementing any functionality [64].

Scenario: Scenarios are written situations or cases where test users are asked to solve a problem or complete a task list using the software prototype. Scenarios typically feature a prototype of sufficient complexity for the test user to be able to complete the case. The prototype does not, however, need to be a fully functional one as with a vertical prototype. Scenarios can be combined with any of the three lower-fidelity methods e.g. trying out a scenario on a paper prototype.

Horizontal and Vertical Simulation: Horizontal and vertical prototypes have been discussed above.

Fully Functional Simulation: Combines the depth of vertical simulations with the width of horizontal ones, to create a full-scale testing environment.

In addition to the methods on Friedl's list other methods and definitions exist, and as commented under scenario, methods can be combined to suit a developers needs. An in-depth description of other prototyping methods is beyond the scope of this document.

6.3 Video Game Prototyping

This section will describe conventions around prototyping video games. Since games are large, multi-disciplinary efforts, a single prototype is rarely enough to unveil all the aspects of a game [23]. The first subsection will therefore uncover the different aspects of video game design. The section will then continue to discuss prototyping for the different aspects.

6.3.1 Aspects of Game Design

Since the term ”video game” envelops everything from a casual Flash game to a complex *Massive Multiplayer Online Roleplaying Game (MMORPG)* in a huge, persistent world, it is hard to generalize game prototyping. Differences between game genres and varying project sizes and scope call for different prototyping needs. Since video games are both highly visual, functional and interactive applications a single prototype will in most cases not suffice to capture all aspects of a game. To break down to the different aspects of game prototyping, we will look at the different aspects of game design.

Brathwaite [10] lists six types of design contributing to an overall game design:

World Design: The overall game setting and back story.

System Design: Game rules and algorithms, also referred to as game mechanics.

Content Design: The visual appearance of objects in the game world.

Game writing: Dialog and text encountered in the game.

Level Design: Envelops both the visual appearance of the surroundings of a game level, along with the events and challenges encountered in a game level.

User Interface Design: Consists of both how the user interacts with the game, and how the game gives feedback to the user. The user interface is almost always persistent between different game levels to avoid confusion around how to interact with the game.

All of these elements can be prototyped, though we will not discuss world design, content design and game writing since these aspects are purely creative efforts outside the scope of this project.

6.3.2 Game Mechanics Prototyping

The purpose of prototyping game mechanics is to get an early insight into how the game is played. Games that are aesthetically appealing with an exciting back story can still be disastrous, because the game mechanics are erroneous or simply boring. This statement, however, is not necessarily not true the other way around. Games with solid mechanics and bad graphics can be hits. The crown example of a great game with bad graphics is "Solitaire", included on all Windows operating systems since Windows 3.0. "Solitaire" is estimated to be the world's most played computer game [52]. The lesson to be learned is that game mechanics is the single most important part to "get right" when prototyping a game.

Of the software prototyping methods listed in Subsection 6.2.3, some are more useful than others to prototype game mechanics. Vertical simulation stands out as way to prototype a single game mechanic, while horizontal simulation is useful to see how the different mechanics work together. Whether to create rapid or evolutionary prototypes of a game is also a consideration. Experimental game mechanics may require many iterations to work properly and can be prototyped using throwaway prototyping. A design derived from a more established foundation of game elements such as a *First Person Shooter(FPS)* game will make sense to make use of evolutionary prototyping.

An alternative approach when working with experimental game mechanics is to combine the evolutionary and rapid prototyping. Developers can focus on a small set or a single game mechanic at a time using rapid prototyping. When the set of game mechanics is in a working state, additional mechanics can be added in an evolutionary manner [22].

6.3.3 Game User Interface Prototyping

Prototyping user interfaces for games is somewhat similar to any software user interface design with high focus on interactivity, and will therefore deal with concepts adapted from traditional HCI. As noted in Subsection 6.3.1 a game user interface contains both a player's interaction with the game and the feedback to the player. An alternative categorization of a game user interface prototype is to divide it into two parts: Kinesthetics and aesthetics.

Fullerton [23] describes kinesthetics as:

> "The kinesthetics are the "feel" of the game, how the controls feel, how responsive the interface is, etc."

Games from different genres for different platforms are interacted with different input controllers. For instance most games developed for the Nintendo Wii will be interacted with through a Wiimote [95]. Since a Wiimote is fundamentally different from e.g. an Xbox 360 controller, the requirements for the user interface will be different, which must be considered when designing it. Games on the same platform but with different gameplay will require different interaction mechanics. An example here could be a simple platform game created in Flash opposed to a sophisticated flight simulator. The platformer would simply require a keyboard, while the flight simulator requires support interaction through more specialized input devices such as joysticks and throttle quadrants [86]. It is therefore paramount to uncover which input devices are to be used in the game, before prototyping begins [23]. Prototyping kinesthetics with low-fidelity methods such as sketches will not be successful, since the "feel" and responsiveness of controls require interactivity between player and prototype. This is particularly true for user feedback be it, visual, audial or physical such as controller rumbling.

The aesthetical prototype of a user interface brings forward traditional HCI prototyping techniques. Low-fidelity methods are very useful at early iterations since they are quickly updated to accommodate user feedback. Building a successful user interface is about communicating information to the player in an intuitive format. Arranging early user tests is therefore a useful strategy to rapidly uncover pitfalls and weaknesses of a design [64].

6.3.4 Level Prototyping

As described in Section 6.3.1 we regard game level design to contain both the creation of visual elements in a level, as well as events challenging the player. Level prototyping is, as with other prototypes, an iterative process. Sketching a map or flowchart of a game level is a natural part of the level prototyping. This section will, however, focus on the "unique" aspects of level design, meaning the prototyping taking place when the game engine and mechanics are established.

Adams [1] identifies which features should be included in a level prototype, and how to effectively communicate a level without creating a full implementation. Features to be included are:

Basic Geometry: The basic geometry of the level should be included in a prototype. Whether the level is in 2d or 3d, including blocking geometry is still paramount to get an early impression of it.

Temporary Textures: Using a set of temporary and generic textures in

a 3d level speeds up the prototyping process. For instance it is not necessary to deploy ten slightly varying grass textures to illustrate that a part of the level is covered with grass.

Temporary Props: Using temporary placeholders to illustrate the placement of objects or *Non-Player Characters (NPCs)* speeds up level prototyping time, while still communicating the level layout. For instance a fancy 10000 polygon building with eight texture maps can be illustrated with a white box.

AI Paths: Knowing where an NPC roams is crucial to test a level. AI paths should therefore be included.

Event Triggers: Placing event triggers should be done at an early stage to create awareness of the level flow. Failing to include the horde of zombies raising from their graves when the hero enters the graveyard, does not contribute to an effective representation of how the level will play.

Placeholder Audio: If audio playback communicates important feedback to the player, it should be included. As with textures and props, it is recommended to use placeholder audio clips in a level prototype.

Lighting Design: Lighting is an important contribution to setting the "mood" of an environment. Lighting used to highlight a landmark or direct the player's attention should also be included at the prototyping stage.

When these elements are included, the level prototype is ready for reviewing. A level review is a test workshop where team members from different departments get together and deliver feedback on the level. According to Adams a number of issues should be addressed:

Scale: Determining whether the level is of right size is important. The time taken to play through the level should be balanced to meet the design aspiration.

Pace: If the events of the level arrive too close to each other, the result can be stressing for the player. Long pauses between events make the level boring, thus the event flow must be balanced to make the level exciting.

Object and Trigger Placement: The placement of objects and event triggers should be reviewed to ensure that the level produces the intended experience.

Performance Issues: Issues related to performance must be resolved to make the level playable. If the level prototype designer has cluttered

the level with expensive models and effects, these must be removed or toned down.

Other Code Issues: If the level calls for a lot of additional, expensive programming to accommodate e.g. a single event, the event must be approved by the game producer and the programming team.

Aesthetics: Reviewing the aesthetics to see if the level prototype expresses the correct atmosphere is crucial. It is less expensive to fundamental flaws in aesthetics at an early stage, than in the actual level production.

Level prototyping is an iterative process, and should be repeated until the level is approved and ready for production.

6.4 Summary

Throughout this chapter we have discussed the general concept of prototyping and increasingly directed it towards game prototyping. The chapter is included in this document as a background for later discussion of prototyping, however the research behind it had a more specific goal. As stated in Section 1.3 we ideally wanted to discover some sort of prototyping process tailored to our game prototyping needs. We have learned, however, that the due to the diversity of game design, genres, scope and platforms, there are no ”golden paths” working for all types of game prototyping. Consequently we have rather used the insight obtained from the research behind this chapter to develop our own game prototyping process we intend to follow in this project. This process is outlined in Chapter 8.

Part III

Own Contribution

Chapter 7

Game Prototype Requirements

"You require more vespene gas"
- The Overmind, Starcraft (1998)

As noted in Section 6.2.1 the first step of the prototyping cycle is to establish requirements for the prototype. The Appendix A. The requirements are divided into three categories, corresponding to three of the design domains discussed in Section 6.3.1. Consequently there are requirements for Game Mechanics, User Interface and Level Design. Each of the requirement categories are allocated a section in this chapter.

7.1 Game Mechanics Requirements

This section describes the game mechanics requirements derived from the concept document. Each table lists a set of requirements logically grouped to a game element. Table 7.1 lists requirements related to peasants and Table 7.2 lists requirements related to priests. The requirements listed in Table 7.3 describe the mechanics of the player controlled avatars, the horsepersons. Finally Table 7.4 lists requirements related to immovable objects in the game world.

7.2 User Interface Requirements

This section will list the user interface elements necessary to communicate information about the game to the player in Table 7.5. Solution specific details will be left out, since the aesthetical expression of the game is not set. The list will simply act as a reminder of what is strictly required to give the player

ID	Requirement description	Priority
GM01.1	Each peasant has two stats: Health and moral.	H
GM01.2	When a peasant's health is depleted, the peasant dies.	H
GM01.3	Peasants have souls.	L
GM01.4	Peasants need to eat and drink to stay healthy.	H
GM01.5	A peasant's health affects movement speed.	M
GM01.6	Peasants contract disease from infected water.	M
GM01.7	Diseases deteriorates health.	M
GM01.8	A healthy peasant may contract disease from a sick peasant.	L
GM01.9	A peasant's zeal is lowered by time.	M
GM01.10	Going to church to attend holy mass will restore a peasant's moral.	M
GM01.11	Low moral makes a peasant susceptible to unethical behavior including violence against fellow peasants, substance abuse and promiscuity.	L
GM01.12	Peasants can gather food from fields.	H
GM01.13	Peasants can gather water from wells.	H
GM01.14	A peasant owns a single home.	M
GM01.15	Peasants can bring food and water to their home.	H
GM01.16	Peasants have traits making them more or less vulnerable to hunger, disease and moral decay.	L
GM01.17	A peasant belongs to a single congregation.	M

Table 7.1: GM01: Game mechanics requirements regarding peasants

ID	Requirement description	Priority
GM02.1	Priests have one stat: Zeal.	H
GM02.2	Priests belong to a single church.	H
GM02.3	Zeal is inversely proportional to the moral of a priest's congregation.	H
GM02.4	High zeal makes a priest work fast.	M
GM02.5	A priest can hurt horsepersons.	H
GM02.6	A priest can disinfect wells and replenish fields.	M

Table 7.2: GM02: Game mechanics requirements regarding priests

sufficient information about the game's state. Game menus are not covered in this section, only the user interface connected to the gameplay.

ID	Requirement description	Priority
GM03.1	There are four playable horsepersons: Death, Pestilence, Famine and War.	H
GM03.2	Death can kill peasants.	H
GM03.3	Pestilence can infect water.	H
GM03.4	Famine can eat crops.	H
GM03.5	War can provoke violence.	H
GM03.6	Horsepersons collect souls of dead peasants.	M
GM03.7	Souls may be spent to increase the capabilities of horsepersons.	L
GM03.8	Horsepersons have health.	H
GM03.9	When the horsepersons' health is depleted, the player is penalized.	H

Table 7.3: GM03: Game mechanics requirements regarding horsepersons

ID	Requirement description	Priority
GM04.1	There are four immovable game elements: Churches, peasant homes, fields and wells.	H
GM04.2	Churches have holy auras, where horsepersons may not enter.	M
GM04.3	Peasant homes act as storage units for food and water.	L
GM04.4	Fields contain food.	H
GM04.5	Wells contain water.	H

Table 7.4: GM04: Game mechanics requirements regarding immovable game elements

7.3 Level Requirements

A single level is the minimum that is required to have a playable game. This section will therefore detail the requirements that are needed for a playable level in the prototype. The level requirements listed here will portray a simple level to show off the game mechanics. We start out by listing the content needed to build a prototype level in Table 7.6.

Note that there are four peasants and four peasant cottages listed in Table 7.6 requirement L01.8 and L01.9. Since the level prototype is intended to show off the game mechanics, the four peasants should be susceptible to each of the horseperson's powers. Table 7.7 shows the configuration of the individual

ID	Requirement description	Priority
UI01	The interface should display the overall corruption level of the current mission	H
UI02	The interface should display the amount of souls available and amount harvested	L
UI03	It should be possible to view current health, moral and traits of any given peasant	L
UI04	The interface should provide continuous feedback showing the current goal of all peasants	M
UI05	The interface should display the holy aura of churches and priests	M
UI06	It should be possible to view what current powers are available	L
UI07	The current horseperson in use should be marked	M
UI08	The horsepersons' health should be clearly viewable	H

Table 7.5: UI01: Requirements for the gameplay user interface

ID	Requirement description	Priority
L01.1	The level contains a church.	H
L01.2	The level contains a field.	H
L01.3	The level contains a well.	H
L01.4	The level contains a priest.	H
L01.5	The level contains trees.	L
L01.6	The level contains rocks.	L
L01.7	The level contains fences.	L
L01.8	The level contains four peasants.	H
L01.9	The level contains four peasant cottages.	H
L01.10	Death is present.	H
L01.11	Famine is present.	H
L01.12	Pestilence is present.	H
L01.13	War is present.	H

Table 7.6: L01: Required props to build a prototype level.

peasants named Peasant 1 through 4.

The last requirements listed in Table 7.8 are general requirements to the level prototype. These requirements are formed according to the features that should be included in a level prototype discussed in Section 6.3.4.

ID	Requirement description	Priority
L02.1	Peasant 1 should be immune to thirst and moral corruption, but susceptible to hunger.	H
L02.2	Peasant 1's cottage should be stocked with water.	L
L02.2	Peasant 2 should be immune to hunger and moral corruption, but susceptible to thirst.	H
L02.3	Peasant 2's cottage should be stocked with food.	L
L02.4	Peasant 3 and Peasant 4 should be immune to hunger and thirst, but susceptible to moral corruption.	H
L02.5	Peasant 3 and Peasant 4's cottages should be stocked with food and water.	L

Table 7.7: L02: Requirements for the individual peasants.

ID	Requirement description	Priority
L03.1	The level includes a terrain.	H
L03.2	The terrain is created with AI-pathfinding in mind. Peasants should be able to pathfind to and from key locations without being stuck.	H
L03.3	The level should include light sources to light the geometry in a bright and warm way.	H
L03.4	When a peasant is killed, corruption increases.	M
L03.5	When corruption increases, the lighting changes to express a more sinister atmosphere.	M
L03.5	The level is completed when all peasants are dead.	H

Table 7.8: L03: General level prototype requirements.

Chapter 8

The Prototyping Process

"Thank you Mario. But our princess is in another castle!"
- Toad, Super Mario Bros. (1985)

This chapter will outline the prototyping process we aim to work through to create a prototype from the game concept document in Appendix A. The first section of this chapter will introduce how we intend to prototype the game concept, and justify the process. We will also discuss the prerequisites needed to go through the process in terms of knowledge. The second section details the prototyping process by outlining the elements of three consecutive phases.

8.1 Prototyping Guidelines

As observed in Section 6.1, prototyping is about iteratively refining a basic sketch into a final product. Since the basic sketch of our prototype originates from a textual description, we must first concretize this document into basic requirements for the prototype, and implement these requirements. This process envelops the first phase of our prototype, and should be reiterated until we are satisfied with the basic game mechanics. The derived requirements are listed in Chapter 7. The second phase deals with blocking out a level that efficiently shows off the game mechanics. In the third phase we refine the level and finalize visual and audial content. We have chosen to divide the phase in this manner so it is easy to finalize a phase before moving on to the next. Details on the three phases are given in the next section.

The process is intended to make use of the UDK in each phase. In the first phase UDK is used to implement game mechanics through UnrealScript,

Archetypes and Kismet, discussed respectively in Section 5.1, 5.2.2 and 5.2.1. In the second phase a level communicating the game mechanics is built. Here UDK's UnrealED will play a large role importing and arranging content, blocking out the level's base geometry and make workable user interface elements. In the third phase the capabilities of the Unreal Engine 3 is put to the test. Here materials, particle effects, final animations, lighting and audio is included. The UDK support all of these features due to its sizable toolset.

Building one prototype on top of another is a principle derived from evolutionary prototyping, explained in Section 6.2.2. The process described in this chapter is an evolutionary prototype, since the underlying code and content from a previous phase forms the basis for the next. For an evolutionary prototype to be useful, it is important that the code and content from a previous phase is sufficiently robust to build upon. However, architecturing extensible code is quite time consuming and opposes the exploring mindset of quickly trying out new ideas. We will therefore use rapid iterations per phase to quickly adopt to changing terms. When the prototype has been reviewed to pass all the requirements for the phase, it may be thrown away and reimplemented or fundamentally altered to enforce robustness while still meeting all requirements. An example when using the UDK is to prototype game mechanics using Kismet, and when they work satisfyingly, reimplement the functionality in UnrealScript. Progression to the next phase will not take place before this process is completed. Evolutionary prototyping is ideally time-saving since the prototype is the foundation of the final product, and not thrown away. The discussion of whether the UDK is a well suited candidate for an evolutionary prototyping process is a key question in this thesis, and is discussed in Section 10.2.

The different phases including sub-iterations call for different prototyping methods. The intention of our prototyping process is not to set in stone what methods to use for the different phases. We have, however, found certain methods to be fitting to certain items in the process. Both vertical and horizontal simulation can be useful to prototype game mechanics. As mentioned in Section 6.3.2, vertical simulation is useful to test a single feature, while horizontal prototyping shows how the game mechanics work together. When using the UDK a lot of the prerequisites for high-fidelity game mechanics prototyping such as horizontal and vertical simulation, is already there, making the process a lot quicker. Section 6.3.2 explains two dimensions of game user interface prototyping: Aesthetics and kinestethics. Since aesthetic prototyping leans towards traditional HCI user interface prototyping, the use of paper and digital sketches are intuitively well suited methods to establish the UI elements in

the first phase and determining form and functionality in the second phase. Prototyping kinestethics can hardly be done with non-interactive prototyping methods, thus a high-fidelity vertical simulation is a logical choice. Since the UDK supports a wide amount of input devices and endless ways to deliver visual and audial feedback, the software prerequisites are again met, simplifying high-fidelity kinestethic prototyping. Finally, level prototyping, as discussed in Section 6.3.4, may require multiple prototyping methods to do right. Paper and digital sketches are useful for blocking out the level geometry and event flow, before moving onto a horizontal prototype using the UDK. The same is true for a level's visual content: In any design industry, be it web design, architecture, industry design or game development, common practice is to sketch a visual element using a simple material before moving onto refining it [62].

As mentioned above, the large feature set of the UDK supports the construction of high-level game prototypes. An issue we have not yet considered is the prerequisites of the user. A toolkit of this size will undeniably require training of the user before it can be used effectively. The learning curve and intuitivity of building prototypes with the UDK is not the subject of this chapter, and will be discussed in Section 10.2. We can, however, safely state that one of the prerequisites of users following our prototyping process, is to be proficient with the toolkit. Proficiency with the UDK is not the only prerequisite to follow our prototyping process, since high-fidelity game prototyping requires both skills in programming and game content creation. Conclusively, an individual or team aspiring to create a game prototype using the process described in this chapter must be proficient both in programming and game content creation as well as the UDK.

Throughout this section we have outlined a prototyping process tailored to use with the UDK to form a working prototype based on a non-formal, textual concept document. Even though the UDK is used throughout the described process it does not mean that another game engine toolkit, such as Unity Technologies' Unity [77] or GarageGames' Torque [42], cannot do the same job. Our evaluation of using the UDK with the described process is twofold since it deals with both the evaluation of the UDK, found in Section 10.2, and the evaluation of the prototyping process itself, found in Section 10.3. The process is also intended to be applicable to other concept documents in other genres or on a different platform, since such specifics are abstracted from the prototyping process description.

8.2 Detailed Phase Description

In Chapter 6 we investigated prototyping concepts and how they relate to the art of game creation. With this in mind, we went on a quest for a game prototyping process tailored to small team sizes. We ended up with developing an iterative approach, with three broad phases that covers the path from concept to a complete prototype which can be used directly for production. These phases will be described in detail in this section.

Each phase will follow an iterative development cycle, based on the Software Prototyping Cycle described in Section 6.2.1. Figure 8.1 illustrates this internal phase cycle.

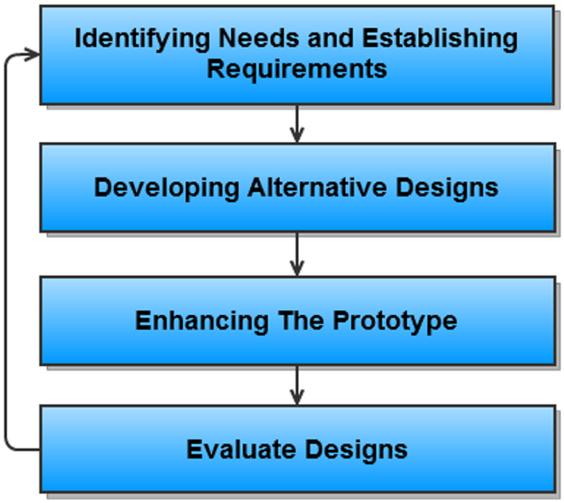


Figure 8.1: The internal development cycle of each prototyping phase.

It is important to emphasize that these steps only form a guideline. They are meant to be interchangeable and to be mixed and matched as needed. One might need to perform an evaluation of the requirements right after they are established, for example. Steps might be omitted if not needed in the current iteration, though we would recommend that the evaluation step is conducted throughout the iterations. The reason for this is to catch features that does not work at an early stage, before countless hours is spent on art and other related assets.

The thought is that this cycle runs until the prototype fulfills the goal of that phase. For the first phase, this would be to establish the core mechanics. When this is satisfactory, the prototype is ready to be taken into the next phase. A

limit to the amount of iterations per phase should be set however, or the team runs the risk of getting stuck in an infinite loop of always wanting to improve the design further. The final phase includes the touches needed to sell the idea properly. Figure 8.2 illustrates the phase process.

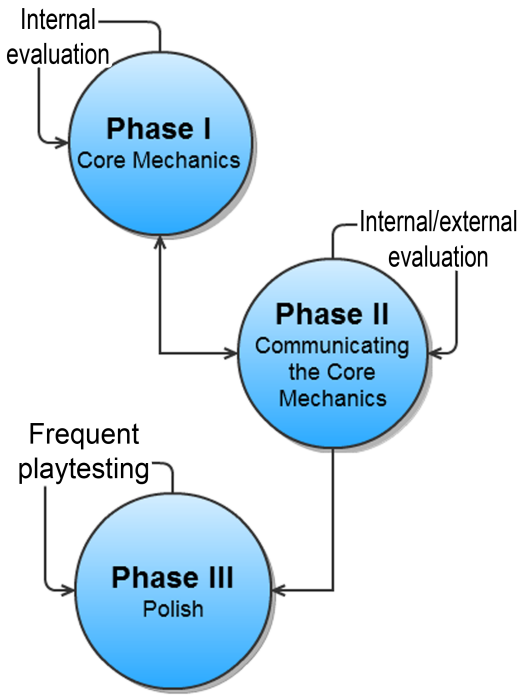


Figure 8.2: The three phases of the prototyping process.

8.2.1 Phase I: Establishing the Core Mechanics

Before this phase is started, the developers should have constructed some sort of concept document or paper outline that provides context to the game and provide goals for the various game elements. The initial requirements should be derived from this document. Focus should initially not be on developing alternative designs, but on identifying the minimum set of requirements that are needed for a playable prototype. This means finding the elements that make the game unique, and establishing the rules that will apply in the game world. The idea here is to keep it simple and find the few key concepts that will be the driving force behind the entire game. These driving forces is what is referred to as the *core mechanics*[1] and can typically be:

- The game's goal
- The player's motivation/challenges

- Vital actors in the game, both playable and non-playable
- The minimum set of available actions to the actors
- The game's economy

To paraphrase Adams, the core mechanics are the building blocks of which levels and ultimately the entire game will be created from.

A simple test level should be constructed to see game mechanics in action. This should be more like a playground than an actual level, to let the developers see the interaction between elements. This is of course in an ideal situation, as not all games might be suited for this approach. No focus should be put on creating graphics, levels, sound or other assets. Here the team need to find or create quick box placeholders. Only user interface elements that are strictly needed to perform gameplay actions should be developed. Placeholder graphics should be put in place here as well.

The focus should lie completely with the basic game mechanics. After these are identified and confirmed to be strictly necessary, they should be implemented and tested within the team. Do they work? How well do they interact? Could this be fun?

If the team believes the design/mechanics can be improved significantly, a new phase iteration should be performed. If not, the team may proceed to the next phase. Table 8.1 summarizes the highlights of Phase I, grouped by the elements of the Software Prototyping Cycle.

Cycle element	**Suggested action**
Identifying Needs and Establishing Requirements	Derived from concept document
Developing Alternative Designs	Keep in mind, but do not focus on initially. Should be considered after an unsatisfactory iteration review
Enhancing The Prototype	Simple game level to test mechanics, only placeholder assets
Evaluate Designs	Team tests and evaluates prototype

Table 8.1: The development cycle for prototyping Phase I

8.2.2 Phase II: Communicating the Core Mechanics

As core mechanics are mostly in place in this phase, the goal will be to get them to work in a more practical game setting. In Phase I the level would not be much more than a testing field. Here the team will design one or more levels that showcase the game mechanics from the previous phase in more *natural* settings. A list of the most essential assets that are required in the game should be compiled and implemented in the game. The assets do not necessarily need to be finalized in this phase, but some form of draft versions showcasing the shape and texture should be implemented. This is to provide means for more accurate placement in the showcase level, as well as providing a less abstract playing experience for testers. Level geometry, e.g. terrain, must also be designed and built in this phase with focus on providing a proper gameplay arena. While the level architecture is the key here, level geometry should be textured with generic materials to identify its different areas.

At this stage, preliminary lighting design for the level should be worked out and implemented. A basic lighting design is essential to communicate any sort of mood in a level. For instance, portraying a level in bright sunlight, as opposed to dim moonlight, is solely a matter of lighting and can create entirely different experiences for the player. The payoff in immersion versus time spent on lighting design is therefore big enough to implement in this phase. Lighting can also be an effective to highlight objectives and guide the player through a level. If the level requirements call for this type of lighting, it should be included at this stage to aid testers' understanding of the level.

The UI elements identified in the first phase should be further refined here. The developers need to decide in what form the information should be presented to the player. For example, should a player's health be presented as a number, a continuous bar or discrete set of tokens, e.g. hearts? It is not the aesthetic choices that are important, but the feedback delivered to the player when interacting with the UI.

Prototyping the user interface and the level are independent processes, and can therefore be performed in parallel or individual cycles. The point of the entire phase is to end up with a user interface and a level that communicates the game mechanics in a way that is endorsed by the play testers.

Table 8.2 summarizes the highlights of Phase II, grouped by the elements of the Software Prototyping Cycle.

Cycle element	Suggested action
Identifying Needs and Establishing Requirements	Derive requirements from identified core mechanics. Compile list of essential assets. Identify form and feedback of UI elements.
Developing Alternative Designs	Work out different designs for level and UI. Choose the ones that gets the most positive user feedback. Overwhelmingly negative user feedback might suggest premature conclusion of phase I.
Enhancing The Prototype	Construct level showcasing game mechanics, basic lighting scheme and essential content. The user interface should communicate the game mechanics in an intuitive way.
Evaluate Designs	Arrange both external and internal play tests. Make adjustments according to feedback.

Table 8.2: The development cycle for prototyping Phase II

8.2.3 Phase III: Polish

The goal of this phase is to refine the prototype from the previous phase to a sufficient level of sophistication to ”sell” the game concept. In other words, this phase focuses on refining the mood, aesthetics and kinesthetics to a professional level. By the end of this phase, the prototype should effectively function as a game demo. A final game would simply include more content, and possible extensions of the game mechanics.

The work load in this phase is mainly placed on the artists, as content will need to be created and refined to fill in for placeholders and omitted elements. Typical content that needs to be created includes: Final models, textures, materials, particle systems, level geometry and lighting, audio and animation.

For the programmers, the challenge will lie in balancing and fine-tuning the game mechanics, to not only make it functional but also as entertaining as possible. This phase encourages short iteration cycles and frequent play tests, to not ruin the results achieved in the previous phases, but rather enhance it.

A pitfall at this stage, is to get hung up on details and lose perspective on the larger picture [1]. It is therefore important that the game and art director maintain a broad overview of both the gameplay and content during this phase, to ensure an overall consistent quality of the prototype.

Table 8.3 summarizes the highlights of Phase III, grouped by the elements of the Software Prototyping Cycle.

Cycle element	**Suggested action**
Identifying Needs and Establishing Requirements	Identify final models, textures, materials, particle systems, level geometry and lighting, audio and animation. Identify balancing issues.
Developing Alternative Designs	No major design alternatives should be considered at this point. Locate the various options for balancing the game mechanics and aesthetics.
Enhancing The Prototype	Refine and create content. Balance gameplay. Strive for consistency.
Evaluate Designs	Arrange frequent external and internal play tests. Make adjustments according to feedback.

Table 8.3: The development cycle for prototyping Phase III

Chapter 9

Implementation

"Great job, Gordon! Throwing that switch and all, I can see your MIT education really pays for itself."
- Barney Calhoun, Half-Life 2 (2004)

In this chapter we outline what we have implemented in regards to the game prototype. The chapter will be introduced with an overview section of the implementation in its current state. In Chapter 7 a set of requirements was formed, derived from the concept document in Appendix A. The second section, Section 9.2, gives a short summary of the level of completion of these requirements.

9.1 Overview

In the current state of the prototype, some of the game elements described in Section 7 are implemented. A lot of emphasis has been put on the peasants and their interaction with the game world. Peasants will through AI make decisions to oblige their current needs. A hungry peasant will go to the nearest field to harvest food and a thirsty peasant will go to the nearest well to fetch water. The peasants may use consume these resources to prevent being weakened and eventually dying, or store them in their homes. The decision process is implemented as a stack of states, thus a more pressing need can be pushed upon the stack at any time. For instance, a hungry peasant is on his way to a field when he is surprised by the player character "Death". Since "Death" may cause immediate death, the peasant will push a "fleeing" state upon his decision stack, and run to the sanctuary of the nearest church. When the coast is clear, the fleeing state is popped from the stack, and the peasant may return

to the previous task.

To support the process of goal-oriented AI, peasants must be able to pathfind properly from location to location as well as handle dynamic elements in their paths. Dynamic pathfinding is currently in a functional state, though a few bugs remain to be sorted out.

Of the four characters a player can control, we have only implemented one: ”Death”. The player can control ”Death”’s movement, and a third-person camera follows him around in the game world. ”Death” has also equipped a melee weapon, his infamous scythe, which he can use to kill peasants. When a peasant is hit, a particle system is spawned on the impact location simulating blood spraying from the wound.

Among the game world assets identified in Table 7.6 and 7.4 we have created two types of trees, a rock, a fence, a well, a field unit, a cottage and a church. These models are textured and have materials set up with them. In addition we have created a river model with a dynamically reflecting water surface material. The model for ”Death” is built, rigged and skinned for use in the UDK and so is his scythe. A screenshot from the game prototype, showcasing some of the models and the visual standard, can be seen in Figure 9.1.

Figure 9.1: A screenshot from the prototype implementation.

As seen in the figure, we have also developed an ”edge-darkening” post-process

shader to emphasize a stylized, cartoonish look, coinciding with the concept document in Appendix A.

It is not optimal to describe the vividness of the current implementation textually, though we have included elements such as wind affecting the trees and the wheat field, water flowing down the river, cloud shadows rolling over the landscape, and the wheat stalks bending away from a character when he moves through a field.

9.2 Completeness

Table B.1 through B.8 details which of the original requirements from Chapter 7 that have been implemented.

A total of 25 of the original 78 requirements have been implemented. This corresponds to a 32% completion rate and is lower than we had initially hoped for. There are a number of reasons why we have not reached a higher percentage of completeness. This discussion is not undertaken here, but left to the prototyping process evaluation in Section 10.3.

Chapter 10

Evaluation

"I sense a soul in search of answers."
- Adria the Witch, Diablo (1997)

In this chapter we evaluate different parts of the project. The first two sections deal with the evaluation of the Unreal Development Kit which is the core of this project. To properly evaluate the UDK, we introduce an evaluation framework in the first section. In the second, we apply the framework to the UDK resulting in an in-depth evaluation. The third section evaluates our prototyping process described in Chapter 8. Finally we concretize the answers to our research questions posed at the beginning of the project in Section 2.1.

10.1 Evaluation Framework

To properly evaluate the Unreal Development Kit, or any toolkit or framework for that sake, an evaluation framework must be formalized. This section outlines an evaluation framework we have tailored to effectively capture the different aspects of a game development toolkit. The evaluation criteria are based on the paper "An Application of a Game Development Framework in Higher Education" by Wang and Wu [82]. Due to the explorative nature of this project, the evaluation will be textual and experience-based. Identifying quantifiable measures would, from our point of view, be artificial and interfere with the quality of the evaluation. A quantitative study would require a more experimental nature, for example several groups of developers, each trying to accomplish the same set of requirements, using the UDK.

1. Learning curve. An important evaluation criterion for a game prototyp-

ing toolkit is the ease of use. Does the framework require extensive training, or is it possible to get results right away? Is it accessible to users without professional experience in game development e.g. a hobbyist? Is the programming language used with the toolkit a common language? If not, is the syntax familiar? Any unconventional GUI-elements in the tools should be identified and discussed. When evaluating the learning curve, the user background is essential. We will compile a list of prerequisite knowledge required for a user to effectively use the framework.

2. **Development speed.** This evaluation criterion is twofold. The first criterion is a measurement of the compile and deploy time when altering the prototype. Does the framework support changes without recompiles? The second criterion is the effort required to add or change a game element. Since changes or additions may be small or large, we find it impossible to generally quantify the evaluation of this criterion. A textual description of positive and negative experiences related to additions and changes will be given instead.

3. **Flexibility.** Does the framework impose any restrictions on the user? Can any type of game be created? Does the framework allow both single and multiplayer game experiences? When evaluating this criterion a generalization of what the framework does not allow the user to do will be formed.

4. **Stability.** How stable is the framework? Is work lost from crashes? Are the tools and the engine swarming with bugs? Are the developers actively maintaining the toolkit? The evaluation of this point will outline stability issues.

5. **Community Support and Documentation.** This point in the evaluation will look at the amount of documentation available, as well as the support to and from users. Is the documentation easy to find? Is it easy to understand? Are there undocumented features? Are the developers actively publishing and updating the documentation? Is there an active user community? All these questions should be evaluated at this point.

6. **Licensing.** The appeal of a development toolkit depends a lot on the licensing terms. Important factors to consider when evaluating the license are: Price, feature restrictions, ownership, closed or open source etc. The evaluation of the licensing will consist of a draft of the licensing terms, and a discussion of whether they are positive or negative.

7. **Competitiveness.** At this point we evaluate the toolkit compared to other

actors on the market. Does the feature set have an edge over the competitors? Is the toolkit sufficiently sophisticated to create state-of-the art game prototypes? Are there functionality not present in the development kit, commonly present in others? This evaluation point will result in a brief comparison between the toolkit in question and competitors.

10.2 Unreal Development Kit

This section will present the positive and negative sides of choosing the Unreal Development Kit for developing a game prototype. This evaluation will be based on the team's personal experience with the toolkit and follows the evaluation framework posed in Section 10.1. It is important to once again note that the evaluation will be from the viewpoint of someone with computer science background and hobbyist experience in 3d content creation.

10.2.1 Learning Curve

To evaluate the start learning curve, we looked at how long it took to implement a pawn the player could move around. As demo code and assets of Epic's first person shooter Unreal Tournament 3 [39] is supplied with the Unreal Development Kit, we had a custom map up and running within a day where the player could run around in first person perspective. If the goal is to prototype a first person concept, the Unreal Development Kit serves as an excellent starting point. There is a lot of reusable scripting code that can be modified, as well as existing assets. However, we wanted to see how it would be to implement something completely different than the existing demo code base, as well as implementing custom 3d content. This can be effectively called starting from "scratch", only using the engine itself as a starting point. This proved doable, but considerably harder, leading us to suspect that the UDK was initially more geared towards modifying the existing Unreal Tournament 3 code than creating entirely new games. It should be noted that Epic has taken steps to rectify this, as will be discussed in Section 10.2.5.

To someone without previous experience with Unreal Technology, there was an initial "shock" when trying to figure out where to start. This was not due to the complexity of the concept that was to be implemented, but due to all the setup required before development can begin. First of all, the project needs to tweak the engine configuration files, or the engine will not recognize the game, what level to start by default and so on. The sheer wall of information contained in some of these config files can be overwhelming for new users. It is hard to tell at first glance what settings are important, what settings can

be ignored and what needs to be changed to start making your own game. A separate tool for changing engine settings would probably be appreciated by new users, as it could control in a higher degree that only legal settings are input, as well as increase the overall readability. But what arguably is the most hampers the initial understanding of the toolkit for the user, is the tight coupling between the demo game itself and the toolkit. The distinction between the two is hard to make in places. In an ideal world, you would have the engine itself, and then you would have separate example projects showcasing features, without the examples being spread out over the rest of the development kit with settings, content and code in various places. This is of course hard to do in reality, as games are performance-heavy and naturally require a tight coupling to the engine. We missed an official tutorial on creating a simple game without relying on Unreal Tournament code. Luckily, there are lot of community-created tutorials and documentation. Learning all the workings of the toolkit is a huge undertaking in itself, as it is packed with features. In fact, a major part of this project was spent learning the intricacies of the UDK toolset. So while a huge feature set is positive in other ways, like development speed, time spent in learning/training will naturally lengthen.

Once the initial learning curve is overcome and development is actually started, there are a lot of exist of features that are intuitive to use. This holds especially true for some of the accompanying tools, like the Unreal Editor, Cascade, AnimSet Editor and SpeedTree Modeler. The use of common graphical user interface conventions helps speed the understanding along. UnrealED has a lot in common with 3d modeling packages, like 3ds Max, in that it employs four different perspectives and uses similar methods for creating geometry. Though one should note that intuitivity is not always the case. The node editors, e.g. the Material Editor, has a somewhat peculiar user interface. The Ctrl-button has to be pressed to move nodes with the mouse, not something any of the team members had encountered in any other user interface. Undo-functionality is missing as well from these tools. UnrealScript is instantly familiar to anyone who has programmed in common languages, like C++, C# or Java, as they are the inspiration source. As the team has experience with all these languages, scripting was easy to get into. The language incorporates intuitive ways to implement state and functions that deal with the passage of time, these are powerful and quick to learn.

To consider the learning curve, one should consider the required background needed to start using the development kit. It will be considerably steeper if one or more of these points are not met. The scripting language, UnrealScript, is an object-oriented language. It is therefore imperative to understand object-

oriented concepts to fully understand how UnrealScript is constructed. On the other hand, knowledge of object-oriented programming should be expected when creating games, as game creation by no means is an easy engineering task and all engines we have introduced in Chapter 5 have a C++ core [7][8]. UnrealScript is inspired by Java and C++ (as mentioned in Section 5.1), so understanding of C-style programming syntax helps as well, while not strictly required. Knowledge of a 3d modeling suite, e.g. Autodesk 3ds Studio Max, is needed to create 3d models and animations. In addition, the Flash-based UI middleware ScaleForm Gfx was added to UDK in May 2010. Knowledge of Flash and Actionscript 2.0 for user interface creation is therefore recommended as well, although there exists several community-created tutorials at the time of writing. Table 10.1 and 10.2 summarizes the required and recommended knowledge.

The main pros and cons of this evaluation criterion is summarized in Table 10.3.

Knowledge area	**Needed to know**
Programming	Object-oriented programming paradigm
3d content and animation	Autodesk 3ds Max, Autodesk Maya, or Autodesk Softimage or The Blender Foundation's Blender

Table 10.1: Prerequisite knowledge for game creation with the UDK

Knowledge area	**Recommended to know**
Programming	C-style syntax, e.g. C, C#, C++, Java
User interface creation	Adobe Flash and Actionscript 2.0

Table 10.2: Recommended knowledge for game creation with the UDK

10.2.2 Development Speed

This evaluation criterion is twofold. The first criterion is a measurement of the compile and deploy time when altering the prototype. Does the framework support changes without recompiles? The second criterion is the effort required to add or change a game element. Since changes or additions may be small or large, we find it impossible to generally quantify the evaluation of this criterion. A textual description of positive and negative experiences related to additions and changes will be given instead.

Pros	Cons
Familiar syntax in UnrealScript	No integrated IDE
Intuitive state system	Messy config file system
Helpful and descriptive functions in UnrealScript	Tight coupling between game demo and engine
Some familiar user interface choices	Some unfamiliar user interface choices
	Too many windows at times
	Huge feature set
	Some incomplete documentation

Table 10.3: A summary of pros and cons of the UDK under the "Learning curve" evaluation criterion

UDK uses compiled UnrealScript to define gameplay mechanics. As the engine itself does not need to be compiled, compile times are short. For a full compile in our project, average time was 7 seconds. Compiles are needed when something is changed in script, but recompiles can be avoided by exposing variables to archetypes which can be tweaked in the Unreal Editor. Compiles are not needed when changing archetypes, and the game can be run directly from the editor. This can be a huge time saver, especially as the engine is semi-loaded when running the editor; start-up of the game itself is therefore faster in the editor than rebooting the application after a script compile. In addition, variables can actually be changed at runtime via the Remote Control tool. It is extremely effective to get instant feedback in situations where numerous small tweaks are necessary.

On projects with a low programmers-to-artists ratio, kismet can help as well. As described in Section 5.2.1, Kismet is a node-based, graphical scripting tool, and can as such be used non-programming savvy developers. This makes it possible for artists to prototype and test features quickly in cases where programmers are swamped with higher-priority work.

When changes are made in a level, no matter how small, lighting and AI paths needs to be rebuilt to function properly. Light building in particular can waste a lot of time, as proper light building quickly can take several minutes, even when running on 8 threads. Thankfully, it is not strictly needed to run the game/level. In periods of tweaking and experimentation of mechanics and other non-graphical aspects, building of lights can therefore be omitted, until deemed necessary. Only a warning will be displayed.

Another potential time waster is to have too large packages. Each time any-

thing is changed within the package, all shaders and materials used in it will need to be recompiled. This was something the team found out the hard way while developing the prototype, as no warning of this could be found in the official documentation. Care should therefore be taken to have as small packages as possible.

There is no official IDE, which would have increased development speeds, as jumping into code definitions is essential to understand UnrealScript, as a lot of documentation exists as comments within the official core UnrealScript classes. There exists third party IDE's to remedy this, like PixelMine's nFringe plugin to Microsoft Visual Studio [43]. Though an IDE built into the Unreal Editor would have been ideal, to maintain continuity and enforce the binding between the editor and UnrealScript.

Of course, the provided tools and features in the development kit contribute to shortening development time, once the developer has learned them. The navigation mesh system simplifies path finding, and lets artificial intelligence programmers focus on the actual intelligence side. Setting up sounds and animation notifiers in the AnimSet Editor, makes animation-driven development a breeze. Latent functions and timers greatly simplifies programming functions that deal with the passage of time. Built-in functionality for e.g. interpolation, cameras, movement and pawns - these are all things that are usually needed in games, and let the programmers focus on other aspects. The downside is that it of course takes time to learn all these functions, as mentioned in the previous section. But we believe that once the developers have gotten to grips with this functionality, huge speed gains are possible. Evidence for this is Trendy Entertainment's prototype *Dungeon Defense* created with the UDK in 4 weeks with only a couple of programmers [19]. Their lead programmer, Jeremy Stieglitz, have several years of Unreal experience. The Dungeon Defense demo has become part of the official showcase for UDK [34].

As a note, in our previous project [7], we tried building an engine/framework from scratch. We can safely state that using a prebuilt framework speeds up the development by countless amounts, as this whole part of the game is taken care of for you. The developer does not have to worry about technicalities like rendering systems when using an off the shelf engine.

The main pros and cons of this evaluation criterion is summarized in Table 10.4.

Pros	Cons
Remote Control	Whole package recompile at internal
Low compile times of UnrealScript	Lighting and path build times
Kismet allows visual scripting	No IDE
Archetypes	
Huge feature set	

Table 10.4: A summary of pros and cons of the UDK under the "Development speed" evaluation criterion

10.2.3 Flexibility

For this evaluation criterion we wanted to look at the restrictions posed by the engine/framework on the developer.

The UDK gives no access to the C++ code of the internal engine, the way Unreal Engine 3 licensees get. It is therefore not possible to integrate native middleware into the engine, other than what is being supplied by Epic, or tweak the engine if new/changed core functionality is needed. The UDK loses some flexibility this way, and anyone who are thinking of implementing some radical or innovative mechanic, will have to investigate whether this can be accomplished within the confinements of the engine. That being said, UnrealScript and the features already present in the UDK provide enough flexibility to support several sorts of conventional 3d and 2.5d games. There have been created platformers, sidescrollers, top down games, third person games, in additional to the obvious choice of first person games [88]. Though, one has to consider that the engine has its origin in a first person shooter - A "crazy" concept that strays too far from the conventions established in this context might run into trouble. It is possible in a certain degree to include external code written in C++, by creating a DLL and making a binding to it. This should in theory increase the flexibility of the development kit. The team has not tested this however, and can not comment properly on its usefulness.

Perhaps unsurprisingly, with its 3d roots, the engine does not have a lot of explicit functionality dealing with 2d. There is no way to set orthographic projection style (no perspective), for example. A game concept like *Super Paper Mario* [60], a mix between 2d and 3d styles, might therefore initially seem hard to achieve in the UDK. Animated sprites in game elements, like a pawn, is not something which is directly supported, but is possible through material usage, or other creative work around. An approximation to orthographic projection can be achieved by setting the field of view of the camera

Pros	Cons
UnrealScript	Hardware demands
Engine proven in several settings	Not completely platform independent
(DLL-bind)	Limited 2d options

Table 10.5: A summary of pros and cons of the UDK under the "Flexitivity" evaluation criterion

as low as possible. In general, a lot of additional functionality which might seem "impossible" at first glance with the limitations of the engine, can be done through creative use of the already existing features. This mentality is useful when prototyping, as the player will not know the internals of the game. If something works, it does not matter if the solution is not the most elegant, or if it is "faked". The first person philosophical puzzler *Hazard: The Journey of Life* is an example of a single person being creative with the UDK [35]. In addition, the recently added Scaleform middleware allows for powerful usage of Flash and Actionscript. A 2d mini-game may be written entirely within the UI, for instance.

A proper argument against the flexibility of the UDK, is its somewhat high hardware demands. It is scalable and has support for auto-adjusting graphical settings, but with Epic's plans to remove support for Shader Model 2.0 graphics cards in the June 2010 build of UDK [33], games that hope to run on older hardware will have to look elsewhere, or use earlier builds of the development kit.

UDK is not platform independent. It runs mainly on the Windows-platform, with theoretical support for Xbox 360 and PlayStation 3. We say *theoretical* here, as Unreal Engine 3 runs on these consoles, but additional development kits and licenses will have to be acquired to deploy to these consoles. There is currently no support for Wii or mobiles, although support for Apple's 3GS iPhone is at the time of writing currently in development. A demonstration of Unreal Engine 3 running on iPhone was displayed in November 2009 [67].

The main pros and cons of this evaluation criterion is summarized in Table 10.5.

10.2.4 Stability

Before discussing the stability of the UDK, it is important to stress that the development kit is still in a beta phase. Since Epic releases a new build every

month, the toolkit is more stable at the latest release (the May 2010 Build), than it was at the first build in November 2009. The majority of stability issues are therefore likely to be addressed and eliminated by Epic in future. This section will evaluate two kinds of stability issues: Errors leading to crashes and non-fatal issues within the development kit.

We have not quantitatively measured the number of crashes we have had with the UDK, but we have experienced occasional crashes when working with the UnrealED. This is not uncommon in the world of software, and has not been very problematic. The UnrealED features customizable auto-saving functionality, where the user may specify how often an auto-save is performed and whether changes in packages or only the map should be saved. If a lot of content is lost in a crash, we will therefore put most of the blame on the user, not the UDK. We have, however, experienced a single crash that corrupted the UDK to the point that we had to re-install it.

Of non-terminal issues, we have identified three. The first issue relates to physical bodies of skeletal meshes. When turning a skeletal mesh into a "ragdoll", thus letting the physics solution calculate a character's animation rather than being driven by predefined animation, we have experienced some stability issues. If we did not remove the skeletal mesh from the map quickly, we observed a tenfold in calls to the physics unit, severely impeding engine performance. The second error is that decals [11] refuse to project on high-detailed terrain on the workstations we have used with ATI-based graphics cards. On the workstations with nVIDIA cards, the problem vanishes. The third problem is a visual one and is illustrated in Figure 10.1. When using *Cascaded Shadow Maps (CSM)* the camera frustum is split into a number of volumes, where a shadow map is calculated for each volume [17]. The idea is that the resolution of the shadow maps can vary, so that more detailed maps are used close to the camera, but not on the parts of the scene far away. As seen in the figure, visual artifacts may occur between the splits.

The main pros and cons of this evaluation criterion is summarized in Table 10.6.

10.2.5 Community Support and Documentation

As described in Section 10.1 we will at this point look at the documentation and the community support of the UDK. The questions posed in the evaluation framework are: Is the documentation easy to find? Is it easy to understand? Are there undocumented features? Are the developers actively publishing and updating the documentation? Is there an active user community?

Figure 10.1: A screenshot from the UnrealED showing a visual artifact between camera frustum splits when using Cascaded Shadow Maps. The red ring is added to highlight the problem area.

Pros	**Cons**
Great overall stability for a beta application.	Crashes occasionally.
Customizable auto-save feature reduces the severity of a crash.	Minor issues with physics, decals and cascaded shadow maps.

Table 10.6: A summary of pros and cons of the UDK under the ”Stability” evaluation criterion

Online support for the UDK lies under the *Unreal Development Network (UDN)* [38], containing support both for full licensees and UDK licensees. All of the documentation available to full licensees are available to the public, except for documentation on the inner workings of the engine. As described in 10.2.6 only full licensees have access to the engine source code, thus UDK users do not need access to this documentation. Our short answer to whether it is easy to find the documentation is: Yes. If a developer manages to find and download the UDK, he or she will also manage to find the documentation; it is right there on the website. The documentation is also linked to from the UnrealED's start page which is displayed when the UnrealED is loaded. Navigating the documentation is made easy via searching capability and an intuitive tree-structure for the documentation.

In our experience, the documentation is well written and easily understandable. Since it is not always easy to describe interactive user interfaces textually, Epic Games has hired "3D Buzz", a company specialized in creating educational video tutorials for 3d content creation applications, programming and game design since 2001 [73], to create a series of video tutorials covering different aspects of the UDK. We have found these video tutorials to be an excellent supplement to the textual documentation of the UDK.

We have yet to discover any major features to be undocumented. However, some of the documentation is a little sparse on details, and suffers from inconsistency. As an example, there are in the particle system editor "Cascade" inside the UnrealED, seven types of data type modules: Animation Trails, Beam, Mesh, PhysX Sprite, PhysX Mesh, Trails and Ribbon. The documentation for Cascade does, however, only list three module types: Beam, Mesh and Trails [65]. Browsing around in the documentation will reveal that the data types related to PhysX particles are discussed in a document of its own, and so is Animation Trails. The example illustrates another point. As described is Section 1.2 the UDK is still a beta, and Epic releases a new build every month with added features and fixes. It is apparent that the documentation is not always in up to date with the current build, however, we deem it likely that Epic will remedy this when the development kit is finalized. Note that Epic does release and update documentation with every new build, but as the example illustrates, some inconsistencies are missed. Even though there are no major features uncovered in the documentation, a nuisance when starting to learn the UDK was, as mentioned in Section 10.2.1, that there was no tutorials on "cleaning" the development kit of *Unreal Tournament (UT)* specific elements. The subsequent builds since November 2009 shows that Epic are actively isolating more and more of the UT-specific elements not only from

Pros	Cons
Extensive documentation with no major undocumented features.	Occasionaly documentation is not up to date.
Free, professional video tutorials	No official tutorials on starting from scratch.
Large, active user community.	

Table 10.7: A summary of pros and cons of the UDK under the "Community Support and Documentation" evaluation criterion

native engine code, but also from heavily referenced UnrealScript classes such as *Actor* and *Pawn* described in Section 5.1.1. It is therefore safe to assume that the process of starting from scratch will not remain as complex in the future.

The fact that Epic releases a new build every month, with added features and bugs fixed, is in itself a testament to the commitment Epic show to increase the awareness and use of the UDK. As noted in Section 1.2 there were over 50,000 downloads of the November 2009 build, when the UDK was released. The user community is therefore a large one, and growing. There are at the time of writing over 160,000 registered users on Epic Games' forums [28]. Note that this number is merely an indication of the user mass, since Epic Games' forums envelop all Epic Games' products and it is not necessary to be a registered member to read forum posts. The benefit of having a large user community is twofold: When posting specific questions on the user forums, it is likely to be answered quickly. Second, a lot of user-created tutorials are available. In the case of the problem with "cleaning" the UDK for UT-specific features mentioned above, we were able to solve it through tutorials created by individuals outside of Epic.

The main pros and cons of this evaluation criterion is summarized in Table 10.8.

10.2.6 Licensing

Under this evaluation criterion, we will, as described in Section 10.1, look at the licensing terms of the UDK. Factors to consider here are: Price, feature restrictions, ownership, closed or open source etc.

The UDK pricing model is perhaps the biggest selling point for indie developers. The pricing was to us an important motivational contributor to undertake

this project, as described in Section 1.2. As stated previously: A licensee may use the toolkit for both educational and commercial purposes, however for any revenue above $5000 Epic claims a 25% royalty. The up-front cost for commercial projects aimed to be sold is $99, whereas a business which internally uses the toolkit must pay a fee of $2500 per development seat per year [31]. When using the toolkit to develop a game prototype, the cost is therefore null and void. For indie developers and small studios, which are financially vunerable, this pricing model evens the competition with more established studios since they get access to similar tools. The risk involved in paying a huge up-front cost is also eliminated: In a failed project, developers and investors may walk away with significantly lower financial losses. If the project is a success, it will also benefit Epic financially, thus enabling continued support for the UDK and its community. Everybody wins.

On the negative side, a 25% royalty to Epic may be pricy. It is important to recognize that the 25% cut is not the only cost involved in an indie game development project when using the UDK. If a licensee wishes to release a title on the Xbox 360 and/or the Playstation 3, they must contact Epic Games to discuss "additional terms" [31]. Since Epic refuses to publish these terms, we will not hold it against them though there might be a hidden cost here. In addition, all developers for both the Xbox 360 and the Playstation 3 must purchase a development kit for the console. While the price of the Xbox 360 Development Kit is unknown for the public, the price of a Playstation 3 Development Kit is currently $2000 [70]. As a general note, the use of digital distribution channels such as Steam for PC, Xbox Live Arcade for Xbox 360 and Playstation Store for Playstation 3, are not free, though this cost is not exclusively tied to the UDK.

Mark Rein, vice president of Epic Games stated in an interview with Ars Technica [84] that:

> "It (the UDK) isn't watered down in any way, so we could expect to see anyone from beginners to professional developers using it."

He also explains that the only differences between a full licensee and a UDK licensee, is that UDK licensees does not have access to the underlying C++ source code of the Unreal Engine 3 and its tools, nor access to direct support from Epic Games' engineers. As described in Section 10.2.5 Epic has moved towards a cleaner engine by moving native Unreal Tournament 3 code to script and exposing more of the engine's core features to scripting. When taking .dll-binding into account as well, there really are not that many cases where a developer needs to alter the underlying source code. Direct support from Epic's

engineers would have been a nice gesture. However, the strong user community and wealth of documentation, commented in Section 10.2.5, largely makes up for the lack of direct support. A quick glance at some of the questions posed on the community forums makes it understandable that Epic Games chooses to only give direct support to full licensees.

Regarding ownership, the UDK *End-User License Agreement (EULA)* included in the UDK installer says [26]:

> "As between the parties, Epic or its suppliers
>
> ...
>
> own the title, copyright, and other intellectual property rights in the UDK, including all derivative works of the UDK. You own the title, copyright, and other intellectual property rights in the applications you develop using the UDK and any derivative works thereof, but ownership of the UDK and derivative works of the UDK, and any portion(s) of the UDK and derivative works of the UDK remains with Epic."

Effectively this means that Epic owns the UDK, and developers own the games and content they have created using the UDK. Regarding closed or open source, Epic says [37]:

> "You can't release your UDK project under terms other than the UDK EULA (like GPL, LGPL, open source, etc.). You don't have the right to encumber the UDK with terms that we have not already granted to you."

When evaluation these terms, we find them to be more than reasonable. For Epic to maintain control of the intellectual property of their own product, it is understandable that UDK licensees may not change the licensing terms to open source, nor obtaining the intellectual property of the toolkit itself.

The main pros and cons of this evaluation criterion is summarized in Table 10.8:

10.2.7 Competitiveness

As described in Section 10.1 we will at this point look at the competitive strengths of the UDK compared to market rivals. Since we have only tested the UDK and not the competitors, the arguments related to the competitors will not be based on experience with the engines. We have, however, in the

Pros	Cons
No up-front cost. No feature restrictions beyond closed source code. Low risk.	A 25% cut to Epic may be pricy. Console development requires additional licensing (development kit).

Table 10.8: A summary of pros and cons of the UDK under the "Licensing" evaluation criterion

depth project preceding the master thesis conducted a study of these engines and acquired substantial theoretical knowledge about them. Another factor to consider is the track record of published games for the rival engines, since strengths and weaknesses of an engine are ultimately reflected in the games using it. The rivals we have considered are the ones listed in Section 4.1.3: Gamebryo, Unity and CryENGINE 3.

The first question posed in the evaluation framework is to determine whether the feature set of the UDK have an edge over the competitors. The general answer is: No. The feature set of the UDK is huge, versatile and very good, which is evident by a glance at Chapter 5. However, none of these features are strictly unique to the UDK when comparing it to e.g. CryENGINE 3 [15]. The reversed form of the question; whether there are features lacking from the UDK present in other engines, is neither positive for the UDK. CryENGINE 3 appears to have an edge over UDK, due to its *What You See Is What You Play (WYSIWYP)* functionality which allows real-time editing on a PC to be propagated to other platforms (Xbox 360 and Playstation 3). In UDK changes must first be made on a PC and then "cooked" to update on other platforms. The lighting solution of the CryENGINE 3 does also have an edge over UDK since it is fully dynamic and does not need offline builds, which Lightmass does. The CryENGINE 3 lighting solution does still allow indirect lighting and ambient occlusion, but here it is computed at real-time. The benefit of the UDK's solution, however, is that the visual quality will be the same, or even better, after the offline lighting build is completed, at lower computational cost. Another annoyance we have identified with the UDK compared to the CryENGINE 3 is the absence of a dedicated road and river creation tool. Road and river networks are common in game maps, and the CryENGINE 3 makes it easy for the user to effortlessly update the terrain height map to integrate such networks. It is possible to achieve the same results in the UDK, but it requires a substantial amount of time. To do it in UDK the road or river must first be created in a 3d content creation application, and then imported into

UDK as a static mesh where the terrain height map must be manually tweaked to integrate the network into the level. Compared to Unity and Gamebryo, UDK have an edge with the inclusion of AI functionality, though CryENGINE 3 offers such functionality as well.

Answering the second question, of whether the sophistication of UDK allows creation of state-of-the-art game prototypes, we look to games created using Unreal Engine 3. In Section 1.2.1 we mention "Unreal Tournament 3", "Gears of War 1 and 2", "Mass Effect 1 and 2" and "Batman: Arkham Asylum" as examples of titles using the engine. Even though tools additional to the ones included in the UDK are developed for these titles, such as the facial animation system for Mass Effect 2 [61], the track-record of titles using the Unreal Engine 3 is unmatched. Among these example titles, "Batman Arkham Asylum" was rated the second best PC game of 2009 with a score of 91/100 [54] by "metacritic.com", a web service providing an average of all the scores assigned by other online reviewing institutions [53]. The Xbox 360 version of "Mass Effect 2" has a score of 96/100 [55] which is the fourth best of all times for an Xbox 360 title, and is a likely candidate to win "Best Xbox 360 Game" of 2010. When comparing the track-record of Unreal Engine 3 titles to CryENGINE 3 it is an easy match for Unreal, since no titles have yet been released on the latter platform. The short version of answering whether UDK allows creation of state-of-the art games and game prototypes is therefore a resounding: Yes.

A discussion of the platform, and hardware, flexibility of UDK compared to the rivals is also fitting under the "Competitiveness"-criterion. UDK allows multi-platform development for Windows-based PC, Xbox 360 and Playstation 3, but so does CryENGINE 3. Unity, in the other hand supports "normal" windowed or fullscreen games and web-browser games on the PC (thus platform-independent), as well as deployment to Apple's iPhone and Nintendo Wii [77]. The engine is therefore demonstrably more scalable to different hardware than the UDK since it works both on lower-fidelity platforms such as the iPhone, while delivering competitive results on PC. This statement may not stand the test of time, as Epic Games has announced that they are working on porting the Unreal Engine 3 to the iPhone [66].

The big selling point of the UDK compared to its rivals is, however, the licensing. With no up-front costs, no watering down of the toolkit, and the potential proven by its track record, the UDK stands out as an extremely attractive choice for indie developers. Unity has a free indie license, but this license provides a watered down version of the toolkit, neither does Unity have the track-record of Unreal. Gamebryo has, like Unreal Engine 3, an impressive

Pros	Cons
No up-front cost.	Lighting solution not completely dynamic.
Richness of tools, including AI.	No road and river tool.
Engine proven successful through track record.	Not demonstrably scalable to low-fidelity platforms.

Table 10.9: A summary of pros and cons of the UDK under the "Competitiveness" evaluation criterion

track record, but comes with a huge up-front cost. No titles are yet released using CryENGINE 3, thus even if the feature set looks promising, it is hard to say if the games will deliver. Licensing the CryENGINE 3 comes with a huge up-front cost at the moment, however, Crytek recently stated that they will release a free platform based on their technology [14]. What this upcoming license, and platform, will look like remains to be seen.

The main pros and cons of this evaluation criterion is summarized in Table 10.9.

10.3 Prototyping Process

Before discussing the prototyping process, we should consider the results from Section 9.2: Throughout the project, we have implemented 32% of the original requirements from Chapter 7. Before we conclude our prototyping process from Section 8.2 to be useless, we should consider our background.

Revisiting the contents of Table 10.3, we have identified both proficiency in programming and 3d content creation as prerequisites for efficient use of the UDK. This project is a computer science master thesis, thus we meet the programming requisite through our education. Our 3d content creation background is somewhat more sketchy. To prosper in this field, we signed up for web-lessons at Digital-Tutors.com, a professional training site for digital art used to train artist in companies (and government agencies) such as Pixar, Disney, Sony, Microsoft, MTV and CIA among others [80]. Even though the tutoring was excellent, not to mention expensive, a lot of time went into progressing to a level where we could create something useful.

As stated in Section 8.1, a prerequisite for using our prototype process was to have a working knowledge of the UDK. Due to its large feature set, obtaining a "working knowledge" of the toolkit consumed a very large portion of the 23

weeks we had available for the thesis as a whole. The goal of evaluating the toolkit, thus gaining as much insight as possible in it, conflicts somewhat with the rigidness of the prototyping process. For instance, the first phase of the prototyping process stresses that all game mechanics should be implemented before moving forth. However, for us to get an overview of the UDK feature set, we had to work with elements from other phases e.g. creating animated characters. To be as time-efficient as possible, we tried to create material relevant to the concept, when in the phase of exploring the UDK. Thus, as the implementation clearly shows, there is content in the prototype that should have been created at later prototype phases, even though not all of the prototype's game mechanics have been implemented. In essence, this means that when we designed the prototyping process, we did not foresee that were unable to meet the prerequisite of having a "workable knowledge" of the toolkit and the proposed process has not been tested, nor verified. We do not, however, find our inability to follow the the prototyping process to hurt the project as a whole, since it is the evaluation of the UDK in Section 10.2 that is the main product, not the prototype implementation.

Even though we were unable to get through the prototyping process as intended, we still believe in it, seeing that it is founded on deep, theoretical insight in game prototyping, as well as previous experience in game development. As commented in Section 11.2, we intend to stick to it in the future. In the depth project preceding the master thesis, we implemented a game engine, the *Apocalypse Engine* [7], from scratch. When comparing what we have accomplished in this prototype to the *Apocalypse Engine*, we can safely say that a lot more has been achieved now even though we had to tame the behemoth known as the UDK.

10.4 Research

In this section we revisit the research questions posed at the beginning of the project in Section 2.1, and summarize the answers based on the insight we have gained throughout the project. An individual subsection has been allocated to each of the research questions.

10.4.1 RQ1: Game Prototyping Process

The first research question RQ1, posed in Section 2.1, asked: "What formal game prototyping processes exist?". To investigate this research question we looked into existing literature on the subject. Disappointingly, we did not manage to come up with a lot of useful literature on the subject, though

”Fundamentals of Game Design” [1] by Ernest Adams and ”Game Design Workshop” [23] by Tracy Fullerton provided some insight. The conclusion we reached, as previously noted in Section 6.4, is that due to the multi-disciplinary nature of game design, the differences between genres and the varying scopes of games, there is no ”golden path” when it comes to game prototyping. Based on this conclusion we went on to outline our own process, described in Section 8.

To detail RQ1.2, ”Can prototyping theory from other fields be applicable to game prototyping?”, we saw in Chapter 6 that understanding prototyping as a general concept is valuable for game prototyping. More specific knowledge of software prototyping can also help when working with games, since the prototyping methods and categorization are all applicable to prototyping games. Insight in software prototyping concepts and terminology is also an asset to game developers, since it forms a common communication platform. As a specific prototyping field applicable to game prototyping, user interface prototyping from HCI is highly relevant, since user interfaces is an important aspect of games.

10.4.2 RQ2: Game Prototyping Tool Evaluation

Research question 2 asked: ”Which factors are important in evaluating a game prototyping tool?”. To investigate this research question, we experimented a lot with the UDK to derive key elements we found necessary to point out in an evaluation. A broad categorization of these key elements was the first step towards the evaluation. Secondly, we tried to identify common methods to evaluate game development frameworks in general. Even though we were not able to find an evaluation framework exactly fitted to our needs, we discovered that the evaluation criteria in ”An Application of a Game Development Framework in Higher Education” by Wang and Wu [82], created a solid basis for what we were looking for. The combination of the criteria in Wang and Wu’s paper and the elements identified through exploration of the UDK are therefore the foundation we build our evaluation framework upon.

The factors we identified to answer RQ2 are, as listed in Section 10.1: Learning Curve, Flexibility, Stability, Community Support and Documentation, Licensing and Competitiveness.

10.4.3 RQ3: Evaluation of the UDK

In research question RQ3 we asked: ”How does the UDK score according to the evaluation criteria derived from RQ2?”. The term ”score” here came out

to not be a quantitaive measure, but a textual evaluation. The key pros and cons of the UDK according to the evaluation framework is listed in Table 10.3 to 10.9.

Chapter 11

Conclusion and Further Work

"You must gather your party before venturing forth"
- Narrator, Baldur's Gate (1998)

In this final chapter, we summarize our findings in the section aptly named "Conclusion". The final section will discuss the additional work to be done for the project to reach its optimal state.

11.1 Conclusion

The goal of this project was to evaluate the *Unreal Development Kit (UDK)* as an evolutionary game prototyping tool. This would be done by implementing a prototype of the award-winning[1] game concept *Apocalypse: The four Unmounted Horsepersons* using the UDK. Through our research, we wanted to investigate any existing theory on game prototyping, as well as determine how traditional prototyping techniques can be utilized in a game prototyping environment.

We were unable to locate an existing prototyping process tailored to evolutionary game prototyping. To fill this void, we created our own based on the deep theoretical insight gained through our research on general prototyping processes. Due to time constraints, we were unable to test this process extensively in a realistic environment. This is work that remains before the process can be deemed a success.

Heavy experimentation with the UDK, as well as the framework evaluation criteria posed by *Wang* and *Wu* in their paper "An Application of a Game

[1]Committee's Choice, Norwegian Game Awards 2009 [3]

Development Framework in Higher Education" [82] formed the basis for the criteria used when evaluating the UDK. The key points identified in the evaluation was that the UDK offers low-risk licensing terms for a game engine suite with an outstanding track-record of successful game titles. To properly utilize the speed gains that can be achieved through the UDK, a deep understanding is needed of its feature set. We learned this the hard way, by underestimating the time and effort needed to reach the required level of insight. In the depth project preceding this thesis, we implemented a game engine from scratch [7]. When comparing to this project, it is apparent that the time invested in learning the UDK clearly outweighs the effort of trying to implement something remotely similar from the ground up.

We deem the main goal of this thesis to be fulfilled. The evaluation framework is an asset for anyone aiming to evaluate similar tools. The evaluation itself is a valuable resource for anyone considering using the UDK, or professionals interested in a novice's perspective on the toolkit.

11.2 Further Work

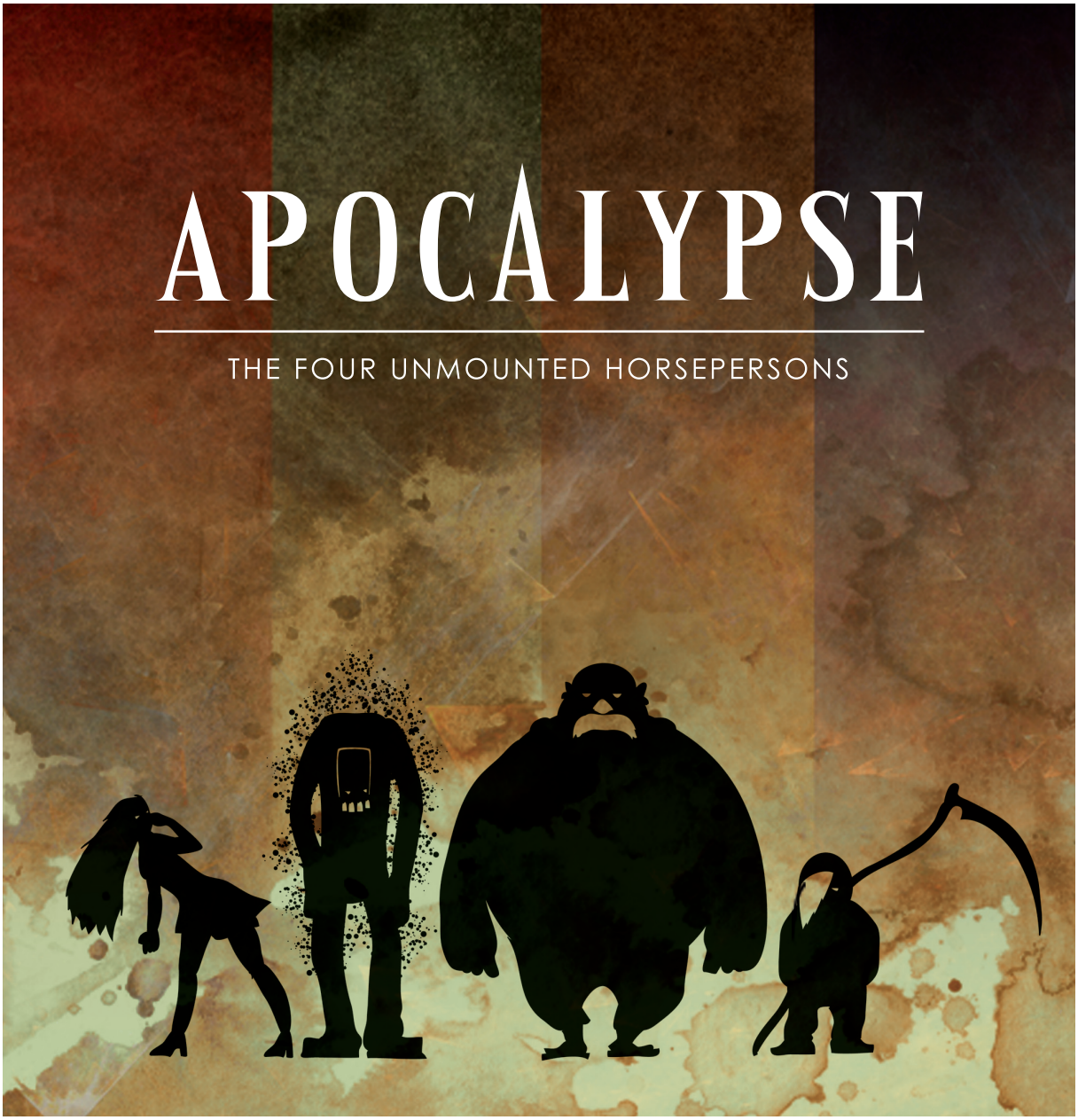
Looking ahead, an obvious task for us is to finalize the game prototype, using the process from Chapter 7. When we actually meet the prerequisite of having working knowledge of the UDK, prototyping other game concepts could also be useful to conform more strictly to the process. The usefulness of our prototyping process may then be re-evaluated.

It would be interesting to evaluate other game development toolkits, both for the game development community, or purely as an academic pursuit. By applying the same evaluation framework as we have to the UDK to e.g. Unity or CryENGINE 3, the result will be a solid foundation for a comparison between the engines. Such a comparison would be a useful asset for anyone looking into licensing an engine for game development or academic purposes.

Appendix A

"Apocalypse" Concept Document

1. Game name: Apocalypse: *The Four Unmounted Horsepersons*
2. Team name: HAX.EXE
3. Game genre: Strategy
4. Multiplayer support: Yes
5. Platform: Xbox 360



APOCALYPSE: THE FOUR UNMOUNTED HORSEPERSONS

Apocalypse is an action-oriented, real-time tactics game, where your ultimate goal is to corrupt mankind, thus bringing the End of Days. By taking command of the Four Horsemen, which ironically neither have horses nor are all men, you will plunge into the land of happiness and bliss, leaving only death and darkness in your wake.

The game is set in the Dark Ages, where lawful and God-fearing peasants happily harvest the fruits of the land. Peasants are strong and sound, attending every mass in the local church, gratefully praising the Lord for their good health, their wealthy crops and their pure and fresh water and air. This happy ecosystem is for you to destroy.

Your various unmounted horsepersons, have demonic powers related to their apocalyptic domain. "Pestilence" is the manifestation of sickness and plague, infecting men and livestock alike, leaving the lucky dead and the unlucky suffering horrendously on their deathbeds. The she-devil "War" possesses demonic beauty and immense wealth, corrupting the weak minds of men, turning brother on brother leaving villages in shambles and ruins. The mustachioed and fatally obese "Famine" has the appetite of a thousand hogs, consuming crops and livestock as he plods through the land, leaving the populace starved and weakened in body and mind alike. "Death" is the ultimate release for the plagued and starved peasants, harvesting souls to be sacrificed upon unholy altars in the eternal fires of Hell. These sacrifices enable the unmounted horsepersons to call upon the Seven Deadly Sins to aid their sinister work.

Unfortunately the Holy Church of Mankind stands in your way thwarting your every move, undermining your powers by the hands of their zealous priests. The priests can cure the sick, and remove the taint of corruption from the land. Also the priests seek to repel the unmounted horsepersons with their blessed auras. Their one weakness lies in their dependency on the peasants attending their holy mass, fueling their powers. The peasants also benefit from attending the mass as their curses are lifted and they recover their strength from the Holy Commotion. When peasants weaken in body and spirit, their faith and ability to attend church diminishes and they stop attending mass, weakening the powers of the priesthood and decreasing the

In-game screenshot showcasing the character Pestilence

holy aura surrounding the church. As the situation in the church deteriorates, the priests are blessed with increased speed and zeal, counteracting the onslaught of chaos. At the time of ultimate corruption, when no peasant is left to serve the holy cause of the church, the priests in their despair find the temptation of sacramental wine to be overwhelming, and the corruption is total.

The peasants lead a simple life, having only three major concerns: Gather precious food to store in their houses, get pure, pristine water from the surrounding wells, and attend holy mass. The unmounted horsepersons collectively possess the necessary skills to disrupt these activities. Each peasant has a combination of strengths and weaknesses making them more resilient or more vulnerable to different types of corruption. It is paramount that the unmounted horsepersons combine their strengths and abilities to destroy each soul.

Gameplaywise Apocalypse is a multiplayer game, intended for 1-4 players. where one player may control one, two or all four “hero“-characters. The game

APOCALYPSE: THE FOUR UNMOUNTED HORSEPERSONS

Concept art for the Famine character

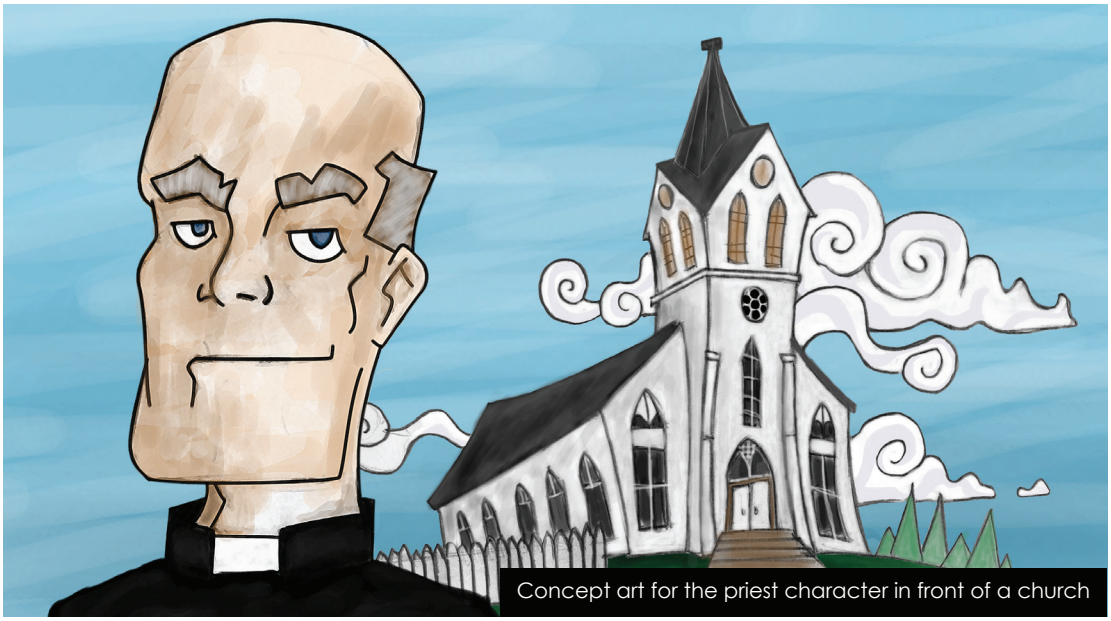
is not intended as a network-multiplayer game, but as a simultaneous hotseat game, where players may dynamically join and quit as the game passes. Depending on the number of players, different players will be assigned different unmonted horsepeople. Since all characters share the same screen, one is assigned leader of the group, and is targeted by the camera.

Leadership is easily transferred between the different characters, and the presence and direction of off-screen char-acters is visualised. The game is presented through a top-down camera as seen in many RTS-games, enabeling broad overview of all players and the surrounding environment.

The game features a campaign with several levels. Each level introduces the players to a peasant community under one or more church jurisdictions. A level is completed when all churches are completely corrupted. Levels start out in an overly happy, bright and idyllic environmentwith strong, bright colors, dan cing peasants, rainbows and singing birds. As the taint spreads the environment turns gloomy , the colors dims, birds die, the sky turns red and pigs start flying. The music changes from happy, medieval tunes, to heavy metal.

Despite of the dark theme of the game, all characters and the environment are presented in a highly cartoonish style, contributing to a comical and unrealstic look and feel.

Peasants have two major statistics: Health and zeal. Their health is depen-



Concept art for the priest character in front of a church

dent on three factors: Food, pure water and combat, influenced by the abilities of respectively Famine, Pestilence and War. A peasant with low health moves slower and is thus more vulnerable to the scythe of the relatively slow moving Death.

The zeal of the peasants is dependent on how often the peasant has attended mass. When zeal is low the peasant is more receptive to moral corruption such as bribes and mindless violence against their fellow peasants.

Priests have one major stat, fanatism, which increases as peasants in their jurisdiction start dying and stop attending mass A priest with high fanatism runs faster and performs prayers and blessing faster.

Appendix B

Requirements Fulfilled

This chapter contains a set of tables, Table B.1 to B.8, which are copies of the tables in Chapter 7. Here we show exactly which requirements that have been implemented, and the ones remaining.

ID	Requirement description	Rank	Fulfilled?
GM01.1	Each peasant has two stats: Health and moral.	H	No
GM01.2	When a peasant's health is depleted, the peasant dies.	H	Yes
GM01.3	Peasants have souls.	L	No
GM01.4	Peasants need to eat and drink to stay healthy.	H	Yes
GM01.5	A peasant's health affects movement speed.	M	No
GM01.6	Peasans contract disease from infected water.	M	No
GM01.7	Diseases deteriorates health.	M	No
GM01.8	A healthy peasant may contract disease from a sick peasant.	L	No
GM01.9	A peasant's zeal is lowered by time.	M	No
GM01.10	Going to church to attend holy mass will restore a peasant's moral.	M	No
GM01.11	Low moral makes a peasant susceptible to unethical behaviour including violence against fellow peasants, substance abuse and promiscuity.	L	No
GM01.12	Peasants can gather food from fields.	H	Yes
GM01.13	Peasants can gather water from wells.	H	Yes
GM01.14	A peasant owns a single home.	M	Yes
GM01.15	Peasants can bring food and water to their home.	H	Yes
GM01.16	Peasants have traits making them more or less vulnerable to hunger, disease and moral decay.	L	Yes
GM01.17	A peasant belongs to a single congregation.	M	No

Table B.1: The implemented requirements of Table 7.1

ID	Requirement description	Rank	Fulfilled?
GM02.1	Priests have one stat: Zeal.	H	No
GM02.2	Priests belong to a single church.	H	No
GM02.3	Zeal is inversely proportional to the moral of a priest's congragation.	H	No
GM02.4	High zeal makes a priest work fast.	M	No
GM02.5	A priest can hurt horsepersons.	H	No
GM02.6	A priest can disinfect wells and replenish fields.	M	No

Table B.2: The implemented requirements of Table 7.2

ID	Requirement description	Rank	Fulfilled?
GM03.1	There are four playable horsepersons: Death, Pestilence, Famine and War.	H	No
GM03.2	Death can kill peasants.	H	Yes
GM03.3	Pestilence can infect water.	H	No
GM03.4	Famine can eat crops.	H	No
GM03.5	War can provoke violence.	H	No
GM03.6	Horsepersons collect souls of dead peasants.	M	No
GM03.7	Souls may be spent to increase the capabilities of horsepersons.	L	No
GM03.8	Horsepersons have health.	H	Yes
GM03.9	When the horsepersons' health is depleted, the player is penalized.	H	No

Table B.3: The implemented requirements of Table 7.3

ID	Requirement description	Rank	Fulfilled?
GM04.1	There are four immovable game elements: Churches, peasant homes, fields and wells.	H	Yes
GM04.2	Churches have holy auras, where horsepersons may not enter.	M	No
GM04.3	Peasant homes act as storage units for food and water.	L	Yes
GM04.4	Fields contain food.	H	Yes
GM04.5	Wells contain water.	H	Yes

Table B.4: The implemented requirements of Table 7.4

ID	Requirement description	Rank	Fulfilled?
UI01	The interface should display the overall corruption level of the current mission	H	No
UI02	The interface should display the amount of souls available and amount harvested	L	No
UI03	It should be possible to view current health, moral and traits of any given peasant	L	No
UI04	The interface should provide continuous feeback showing the current goal of all peasants	M	No
UI05	The interface should display the holy aura of churches and priests	M	No
UI06	It should be possible to view what current powers are available	L	No
UI07	The current horseperson in use should be marked	M	No
UI08	The horsepersons' health should be clearly viewable	H	No

Table B.5: The implemented requirements of Table 7.5

ID	Requirement description	Rank	Fulfilled?
L01.1	The level contains a church.	H	Yes
L01.2	The level contains a field.	H	Yes
L01.3	The level contains a well.	H	Yes
L01.4	The level contains a priest.	H	No
L01.5	The level contains trees.	L	Yes
L01.6	The level contains rocks.	L	Yes
L01.7	The level contains fences.	L	Yes
L01.8	The level contains four peasants.	H	Yes
L01.9	The level contains four peasant cottages.	H	Yes
L01.10	Death is present.	H	Yes
L01.11	Famine is present.	H	No
L01.12	Pestilence is present.	H	No
L01.13	War is present.	H	No

Table B.6: The implemented requirements of Table 7.6

ID	Requirement description	Rank	Fulfilled?
L02.1	Peasant 1 should be immune to thirst and moral corruption, but susceptible to hunger.	H	No
L02.2	Peasant 1's cottage should be stocked with water.	L	No
L02.2	Peasant 2 should be immune to hunger and moral corruption, but susceptible to thirst.	H	No
L02.3	Peasant 2's cottage should be stocked with food.	L	No
L02.4	Peasant 3 and Peasant 4 should be immune to hunger and thirst, but susceptible to moral corruption.	H	No
L02.5	Peasant 3 and Peasant 4's cottages should be stocked with food and water.	L	No

Table B.7: The implemented requirements of Table 7.7

ID	Requirement description	Rank	Fulfilled?
L03.1	The level includes a terrain.	H	Yes
L03.2	The terrain is created with AI-pathfinding in mind. Peasants should be able to pathfind to and from key locations without being stuck.	H	Yes
L03.3	The level should include light sources to light the geometry in a bright and warm way.	H	Yes
L03.4	When a peasant is killed, corruption increases.	M	No
L03.5	When corruption increases, the lighting changes to express a more sinister atmosphere.	M	No
L03.5	The level is completed when all peasants are dead.	H	No

Table B.8: The implemented requirements of Table 7.8

Bibliography

[1] Ernest Adams. *Fundamentals of Game Design*. New Riders, 2^{nd} edition, 2009.

[2] Adobe. Adobe creative suite 5 design premium [online]. Available from: `http://www.adobe.com/products/creativesuite/design/whatisdesignpremium`.

[3] Norwegian Game Awards. Winners of nga09 [online]. Available from: `http://gameawards.no/news/winners09/`.

[4] Victor R. Basili. The experimental paradigm in software engineering. *IEEE Transactions on Software Engineering*, 12:733–743, 1986.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2^{nd} edition, 2003.

[6] Eric J. Baude. *Software Engineering: An Object-Oriented Perspective*. Wiley, 1^{st} edition, 2000.

[7] Kjell Iver Bekkerhus and Storstein Kjetil Guldbrandsen. Apocalypse engine - a study of software architecture and conventions in modern game engines [online]. Available from: `http://dl.dropbox.com/u/2086993/ApocalypseEngine.pdf`.

[8] Jonathan Blow. Game development: Harder than you think. *Queue 1*, 10:28–37, February 2004.

[9] Gillian Bowditch. Grand theft auto producer is godfather of gaming [online]. Available from: `http://www.timesonline.co.uk/tol/news/uk/scotland/article3821838.ece`.

[10] Brenda Brathwaite and Ian Schreiber. *Challenges for Game Designers*. Charles River Media, 1^{st} edition, 2009.

[11] David Burke. Using decals [online]. Available from: `http://udn.epicgames.com/Three/UsingDecals.html`.

[12] Eric Catto. Box2d [online]. Available from: `http://www.box2d.org/`.

[13] John Crinnion. *Evolutionary Systems Development: A Practical Guide to the Use of Prototyping Within a Structured Systems Methodology*. Pitman Publishing, 1^{st} edition, 1990.

[14] Rob Crossley. Free-to-use cryengine plans emerge [online]. Available from: `http://www.develop-online.net/news/34466/Free-to-use-CryEngine-plans-emerge`.

[15] Crytek. Cryengine 3 - specifications [online]. Available from: `http://mycryengine.com/index.php?conid=2`.

[16] Oxford Dictionaries. *Concise Oxford Dictionary: Luxury Edition*. OUP Oxford, 11^{th} edition, 2009.

[17] Rouslan Dimitrov. Cascaded shadow maps [online]. Available from: `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`.

[18] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.*, 50(9-10):833–859, 2008. `doi:http://dx.doi.org/10.1016/j.infsof.2008.01.006`.

[19] Trendy Entertainment. Dungeon defense blog - day 26 [online]. Available from: `http://utforums.epicgames.com/showthread.php?t=714489`.

[20] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 1^{st} edition, 2005.

[21] Ltd. Firelight Technologies Pty. Fmod music & sound effects system [online]. Available from: `http://www.fmod.org/`.

[22] Markus Friedl. *Online Game Interactivity Theory*. Charles River Media, 1^{st} edition, 2002.

[23] Tracy Fullerton. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, 2^{nd} edition, 2008.

[24] Epic Games. Animation system overview [online]. Available from: `http://udn.epicgames.com/Three/AnimationOverview.html`.

[25] Epic Games. Animtree editor user guide [online]. Available from: `http://udn.epicgames.com/Three/AnimTreeEditorUserGuide.html`.

[26] Epic Games. Download the unreal development kit [online]. Available from: http://download.udk.com/UDKInstall-2010-05-BETA.exe.

[27] Epic Games. Epic games [online]. Available from: http://epicgames.com/.

[28] Epic Games. Epic games forums [online]. Available from: http://forums.epicgames.com/index.php.

[29] Epic Games. Epic games' unreal development kit eclipses 50,000 users in one week [online]. Available from: http://www.udk.com/udk50k.

[30] Epic Games. Features: Animation - epic udk [online]. Available from: http://www.udk.com/features-animation.

[31] Epic Games. Licensing - epic udk [online]. Available from: http://udk.com/licensing.

[32] Epic Games. Morph targets [online]. Available from: http://udn.epicgames.com/Three/MorphTargets.html.

[33] Epic Games. News - epic udk [online]. Available from: http://www.udk.com/news-beta-may2010.html.

[34] Epic Games. Showcase: Dungeon defense - epic udk [online]. Available from: http://udk.com/showcase-dungeon-defense.

[35] Epic Games. Showcase: Hazard - epic udk [online]. Available from: http://www.udk.com/showcase-hazard.

[36] Epic Games. Speed tree [online]. Available from: http://udn.epicgames.com/Three/SpeedTree.html.

[37] Epic Games. Udk frequently asked questions [online]. Available from: http://udn.epicgames.com/Three/DevelopmentKitFAQ.html.

[38] Epic Games. Udn-three-developmentkithome [online]. Available from: http://udn.epicgames.com/Three/DevelopmentKitHome.html.

[39] Epic Games. Unreal tournament 3 [online]. Available from: http://www.unrealtournament.com/.

[40] Epic Games. Unrealscript language reference [online]. Available from: http://udn.epicgames.com/Three/UnrealScriptReference.html.

[41] Epic Games. Your first unrealscript project [online]. Available from: http://udn.epicgames.com/Three/DevelopmentKitFirstScriptProject.html.

[42] Garage Games. Game development software and tools torquepowered.com [online]. Available from: `http://www.torquepowered.com/`.

[43] Pixel Mine Games. Tools:nfringe - pixel mine games wiki [online]. Available from: `http://wiki.pixelminegames.com/index.php?title=Tools:nFringe`.

[44] Crytek GmbH. Crytek gmbh: Home [online]. Available from: `http://www.crytek.com/`.

[45] Jason Gregory. *Game Engine Architecture*. A K Peters, Ltd., 1^{st} edition, 2009.

[46] Steven Haines, Daniel Wright, and Derek Cornish. Unreal lightmass - static global illumination for unreal engine 3 [online]. Available from: `http://udn.epicgames.com/Three/Lightmass.html`.

[47] Havok. Havok physics [online]. Available from: `http://www.havok.com/index.php?page=havok-physics`.

[48] Jason Hayes. The code/art divide - how technical artists bridge the gap. *Game Developer Magazine*, 8, August 2007.

[49] IDV inc. Speedtree [online]. Available from: `http://www.speedtree.com/`.

[50] IDV inc. Sppedtree image gallery [online]. Available from: `http://www.speedtree.com/gallery/`.

[51] Mary Jane Irwin. Indie game developers rise up [online]. Available from: `http://www.forbes.com/2008/11/20/games-indie-developers-tech-ebiz-cx_mji_1120indiegames.html`.

[52] Josh Levin. Solitaire-y confinement [online]. Available from: `http://www.slate.com/id/2191295/`.

[53] Metacritic.com. About metascores [online]. Available from: `http://www.metacritic.com/about/scoring.shtml`.

[54] Metacritic.com. Best of 2009 [online]. Available from: `http://www.metacritic.com/games/bests/2009.shtml`.

[55] Metacritic.com. Mass effect 2 [online]. Available from: `http://www.metacritic.com/games/platforms/xbox360/masseffect2?q=masseffect2`.

[56] MiKTeX. About miktex [online]. Available from: http://www.miktex.org/about.

[57] Martin Mittring. Finding next gen: Cryengine 2. *Siggraph 2007*, pages 97–121, 2007. doi:http://doi.acm.org/10.1145/1281500.1281671.

[58] Marting Mittring. A bit more deferred - cryengine 3 [online]. Available from: http://www.crytek.com/fileadmin/user_upload/inside/presentations/2009/A_bit_more_deferred_-_CryEngine3.ppt.

[59] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 2^{nd} edition, 1994.

[60] Nintendo. Super paper mario at nintendo :: Games [online]. Available from: http://www.nintendo.com/games/detail/_bua93nkRXBBWiJ8ulRPXASuK0xbcL8l.

[61] Andy Price. Mass effect 2 revealed. *3d World*, February 2010.

[62] Pamela Schenk. The role of drawing in the graphic design process. *Design Studies*, 12(3):168–181, 1991.

[63] Ben Schlichter. Xbla sales charts [online]. Available from: http://news.vgchartz.com/news.php?id=2957.

[64] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-computer Interaction*. John Wiley & Sons, 2^{nd} edition, 2007.

[65] Scott Sherman and Wyeth Johnson. Particle system reference [online]. Available from: http://udn.epicgames.com/Three/ParticleSystemReference.html.

[66] Anand Lai Shimpi. Epic demonstrates unreal engine 3 for the ipod touch/iphone 3gs [online]. Available from: http://www.anandtech.com/show/2892.

[67] Anand Lal Shimpi. Epic demonstrates unreal engine 3 for the ipod touch/iphone 3gs [online]. Available from: http://www.anandtech.com/show/2892.

[68] Michael F. Smith. *Software Prototyping: Adoption, Practice and Management*. McGraw-Hill Publishing, 1^{st} edition, 1990.

[69] Russell Smith. Open dynamics engine [online]. Available from: http://www.ode.org.

[70] Blake Snow. Sony tries to boost ps3 development with dev kit price cut [online]. Available from: http://arstechnica.com/gaming/news/2009/03/sony-announces-lower-cost-ps3-dev-tools.ars.

[71] Valve Software. Source engine [online]. Available from: http://source.valvesoftware.com/.

[72] Tim Sweeney. Unrealscript language reference (1998) [online]. Available from: http://unreal.epicgames.com/UnrealScript.htm.

[73] The 3D Buzz Team. About 3d buzz [online]. Available from: http://www.3dbuzz.com/vbforum/content.php?153.

[74] Emergent Game Technologies. Gamebryo [online]. Available from: http://www.emergent.net/en/Products/Gamebryo/.

[75] Unity Technologies. Unity: Features [online]. Available from: http://unity3d.com/unity/features/.

[76] Unity Technologies. Unity: Features - scripting [online]. Available from: http://unity3d.com/unity/features/scripting.

[77] Unity Technologies. Unity: Game development tool [online]. Available from: http://unity3d.com/unity/.

[78] Unity Technologies. Unity's iphone momentum [online]. Available from: http://unity3d.com/company/news/unity-iphone-momentum-press.html.

[79] Matt Tonks. Navigation mesh reference [online]. Available from: http://udn.epicgames.com/Three/NavigationMeshReference.html.

[80] Digital Tutors. Digital tutors customers [online]. Available from: http://www.digitaltutors.com/09/customers.php.

[81] Michael van Lent. Game smarts. *Computer*, 40(4):99–101, April 2007.

[82] Alf Inge Wang and Bian Wu. An application of game development framework in higher education. *International Journal of Computer Games Technology*, 2009.

[83] Jeff Ward. What is a game engine? [online]. Available from: http://www.gamecareerguide.com/features/529/what_is_a_game_.php.

[84] Andrew Webster. Unreal deal: Ars talks unreal dev kit with epic's mark rein [online]. Available from: http://arstechnica.com/gaming/news/2009/11/early-this-month-epic-games.ars.

[85] Wikipedia. Ambient occlusion [online]. Available from: http://en.wikipedia.org/wiki/Ambient_occlusion.

[86] Wikipedia. Game controller [online]. Available from: http://en.wikipedia.org/wiki/Game_controller#Throttle_quadrant.

[87] Wikipedia. High level shader language [online]. Available from: http://en.wikipedia.org/wiki/Hlsl.

[88] Wikipedia. List of unreal engine games [online]. Available from: http://en.wikipedia.org/wiki/List_of_Unreal_Engine_games#Unreal_Engine_3.

[89] Wikipedia. Parallax mapping [online]. Available from: http://en.wikipedia.org/wiki/Parallax_mapping.

[90] Wikipedia. Prototype [online]. Available from: http://en.wikipedia.org/wiki/Prototype.

[91] Wikipedia. Render farm [online]. Available from: http://en.wikipedia.org/wiki/Render_farm.

[92] Wikipedia. Scrum (development) [online]. Available from: http://en.wikipedia.org/wiki/Scrum_(development).

[93] Wikipedia. Texniccenter [online]. Available from: http://en.wikipedia.org/wiki/TeXnicCenter.

[94] Wikipedia. Umbra [online]. Available from: http://en.wikipedia.org/wiki/Penumbra.

[95] Wikipedia. Wii remote [online]. Available from: http://en.wikipedia.org/wiki/Wii_Remote.