Technical University of Crete

Department of Electronic and Computer Engineering

Development of a 3D Action Role Playing Game

Diploma Thesis by Grigoris Kontadakis

Greece, Crete, Chania

January 2013

Acknowledgments

Many thanks to Assoc. Prof. Katerina Mania for giving me the chance and pleasure to implement this project, fulfilling a long longing dream.

My family for their support all these years.

The Unreal Engine Community which provided useful piece of code to begin with and online support.

Antonia Papastamataki for supervising the vocabulary and grammar on a large part of this thesis.

Effie Pentaraki for creating the character textures art content.

Finally all the testers, and especially Paul Lionoudakis, Manolis Koukoumbedakis, Themis Pantelidis, Dimitris Bousias, Dimitris Kastrinakis and Dimosthenis for their helpful feedback.

Abstract

The purpose of this project is the creation of an entertainment 3D Third Person Action Role Playing Game titled Broken Dreams. A 3D game is usually developed by a team of more than one person. Nowadays, high quality games are designed by large teams that consist mainly of art designers, who are responsible for the game content creation, and programmers, who are responsible for the gameplay creation and putting all the pieces together. It is a real challenge to create a one-person project taking responsibility for both the artistic and programming parts respectively.

The Broken Dreams concept takes place in the middle-ages in a small city which is ruled by a vicious monarch. This ruler has deprived people of their dreams and forced them to sleep forever. The only person that is still unaffected by this eternal sleep is the main character of the game. His purpose is to try and wake up the residents of this town from their sleep. There is only one way to do that. By entering people's dreams and fighting with their nightmares.

Most of the gameplay involves people's dreams, where the character is confronted with nightmares that he must defeat. In order to accomplish this he must unleash Powers (magic skills) upon the nightmares of the city's residents. Currently there are ten Powers implemented in the game, which are divided in two categories: Fire and Ice Powers. As the character becomes stronger (by means of power leveling), his powers grow stronger as well.

The AI Players are also divided in two categories: Fire and Ice Pawns. For the AI movement the typical method used is an A* algorithm due to its low execution time (in comparison to other algorithms) which is crucial in a real-time application. In the current approach, the optimal paths are pre-computed and stored in a 2D array allowing the recovery of an optimal path at O(1) time. So by sacrificing a bit of memory space, optimal time performance is achieved.

The game engine used for the creation of Broken Dreams is **Unreal Engine 3** [1] via the **Unreal Development** Kit (**UDK**) [2] and **UnrealScript** [3] tools which are free for noncommercial use. UDK offers a vast amount of control over certain game elements such us meshes, particle systems, animation, sound, physics, etc. UnrealScript is a scripting language with Java-like syntax which focuses on the interactions between game objects, such as attacking an enemy, picking up an object etc. The coordination of Unreal Engine elements can produce extraordinary results which are only limited by the time available for development and one's imagination.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Purpose of thesis

The purpose of this project is the creation of an **entertainment 3D Third Person Action Role Playing Game,** titled Broken Dreams. A 3D game is usually developed from a team of more than one person. Nowadays, high quality games are designed by large teams that consist mainly of art designers, who are responsible for the game content creation, and programmers, who are responsible for the gameplay creation and for putting all the pieces together. It is a real challenge to create a one-person project, taking responsibility for both the artistic and programming parts respectively.

This project doesn't only aim in the creation of a specific game, but in the principles needed to design a game these days. Creating a game should be a dynamic and organic process. The game will be created, revised, re-envisioned, and then created again based on the feedback the developers receive. This process is known as iteration and it involves three unique stages: formulation, testing, evaluation. These core elements make up the basic progression which the development of a game will follow. These three steps are repeated continuously until a satisfying result is achieved.

The Broken Dreams concept takes place in the middle-ages in a small city which is ruled by a vicious monarch. This ruler has deprived people of their dreams and forced them to sleep forever. The only person that is still unaffected by this eternal sleep is the main character of the game. His purpose is to try to wake up the residents of this town from their sleep. There is only one way to do that. By entering inside people's dreams and fighting with their nightmares.

Most of the gameplay involves people's dreams, where the character is confronted with nightmares that he must defeat. In order to accomplish this he must unleash Powers (magic skills) upon the nightmares of the city's residents. Currently there are ten Powers implemented in the game divided in two categories: Fire and Ice Powers. As the character becomes stronger (by means of power leveling), his powers grow stronger as well.

As far as the implementation of the game is considered, besides the Power System, it was also important to have effective time performance for the Artificial Intelligence players given that we are in the process of creating an action game that needs fast reflexes. So the A.I. programming of the game shouldn't use slow algorithms in order to achieve a desired behavior. Instead the appropriate methods must be used to achieve optimal real time performance. There are two aspects into consideration when referring to enemy players: movement and attack to human players. The implementation of these aspects shouldn't be very simple in order to challenge the human player and keep him intrigued, nor very complicated because the game will then be unbeatable.

The game engine used for the creation of Broken Dreams is **Unreal Engine 3** [1] via the **Unreal Development kit (UDK)** [2] **and UnrealScript** [3] tools which are free for noncommercial use. UDK offers a vast amount of control over certain game elements such us meshes, particle systems, animations, sound, physics, etc. UnrealScript is a scripting language with Java-like syntax that focuses on the interactions between game objects, such as attacking an enemy, picking up an object etc. The coordination of Unreal Engine elements can produce extraordinary results which are only limited by development time and imagination.

1.2 Thesis Summary

Chapter 1 serves as an introduction to the thesis.

Chapter 2 describes some key points of the history of the first video games from the late 20th century, till the trends that exist at the time of writing of this thesis. It also describes the ARPG genre of the current game.

Chapter 3 introduces Unreal Engine and its components. In this section the key features of the UnrealScript language are described and a summary of all the tools used for this project is listed.

Chapter 4 presents the iterative pattern design circle used in game design. Here the design steps and main aspects for the current project are analyzed before advancing to **Chapter 5** where a simplified implementation of these steps is provided along with the AI description and implementation. This chapter concludes with the results of the testing phases.

Finally, a summary of this project and future work are presented in **Chapter 6.**

Chapter 2: History and Evolution of Video Games

2.1 Video game definition

A **game** is structured playing, usually undertaken for enjoyment and sometimes used as an educational tool [4]. Key components of games are goals, rules, challenge, and interaction. Games generally involve mental or physical stimulation, and often both. Many games help develop practical skills, serve as a form of exercise, or otherwise perform an educational, simulational, or psychological role.

An **electronic game** is a game that employs electronics to create an interactive system with which a player can play [5]. The most common form of electronic game today is the video game, and for this reason the terms are often mistakenly used synonymously. Other common forms of electronic game include such non-exclusively-visual products as handheld electronic games, standalone systems (e.g. pinball, slot machines, or electro-mechanical arcade games), and specifically non-visual products (e.g. audio games).

A **video game** is an electronic game that involves human interaction with a user interface to generate visual feedback on a video device [6]. The word *video* in *video game* traditionally referred to a cathode ray tube (CRT) display device, but it now implies any type of display device that can produce two or three dimensional images. The electronic systems used to play video games are known as platforms; examples of these are personal computers and video game consoles. These platforms range from large mainframe computers to small handheld devices. Video games have gone on to become an art form and industry. In the early days of cartridge consoles, they were sometimes called **TV games**.

2.2 First video games

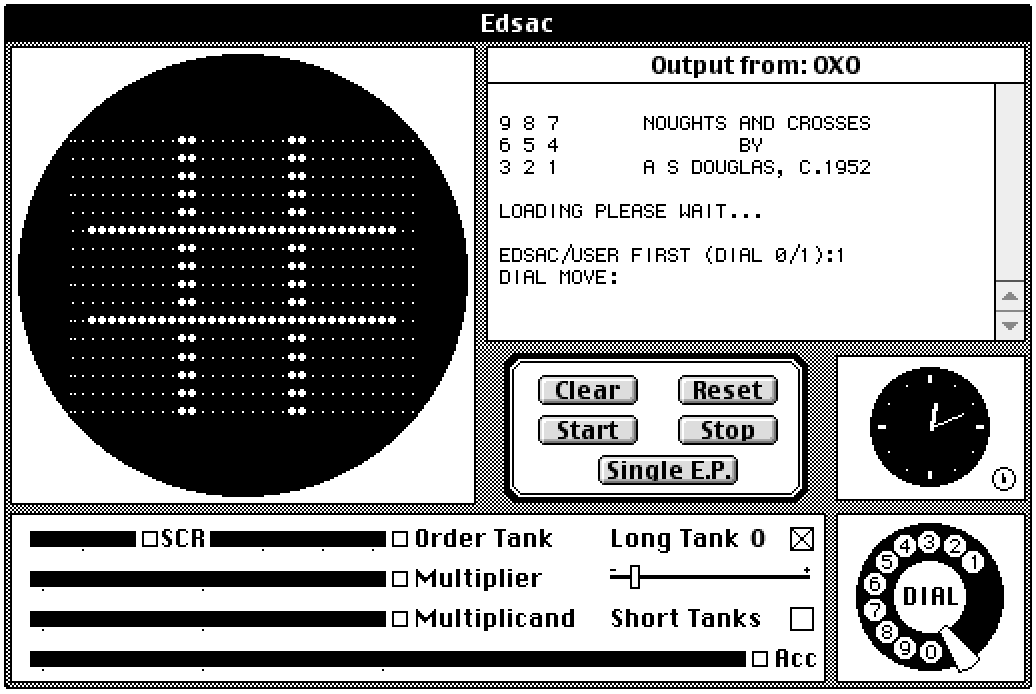
There are numerous debates over who created the **first video game**, with the answer depending largely on how video games are defined [7]. The evolution of video games represents a tangled web of several different industries, including scientific, computer, arcade, and consumer electronics.

The "video" in "video game" traditionally refers to a raster display device. With the popular catch phrase use of the term "video game", the term now implies all display types, formats, and platforms.

Historians have also sought to bypass the issue by instead using the more specific "digital games" descriptive. This term leaves out the earlier analog-based computer games.

2.2.1 List of First Video Games

In 1952, Alexander S. Douglas made the first computer game to use a digital graphical display. *OXO*, also known as *Noughts and Crosses*, is a version of tic-tac-toe for the EDSAC computer at the University of Cambridge.



```
9 8 7        NOUGHTS AND CROSSES
6 5 4                BY
3 2 1        A S DOUGLAS, C.1952

LOADING PLEASE WAIT...

EDSAC/USER FIRST (DIAL 0/1):1
DIAL MOVE:
```

Figure 2.1 – 1952 OXO game for EDSAC

In 1958, William Higinbotham made an interactive computer game named *Tennis for Two* for the Brookhaven National Laboratory's annual visitor's day. This display, funded by the U.S. Department of Energy, was meant to promote atomic power, and used a Donner Model 30 analog computer and the vector display system of an oscilloscope.

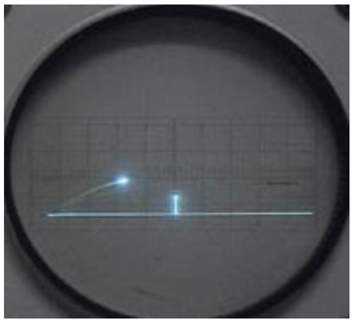


Figure 2.2 – the original Tennis for Two game.

Figure 2.3 – the original game controller.

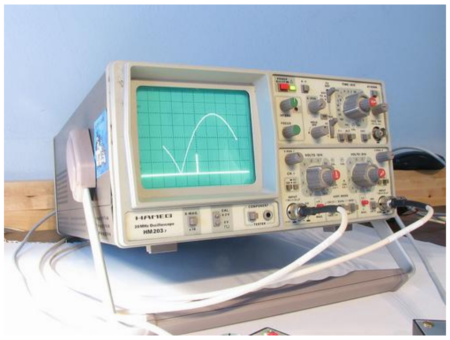


Figure 2.4 – recreation of the game in a modern oscilloscope

In 1961, MIT students Martin Graetz, Steve Russell, and Wayne Wiitanen created the game *Spacewar!* on a DEC PDP-1 mini-computer which also used a vector display system. The game, generally considered the first Shooter game, spread to several of the early mini-computer installations, and reportedly was used as a smoke test by DEC technicians on new PDP-1 systems before shipping, since it was the only available program that exercised every aspect of the hardware. Russell has been quoted as saying that the aspect of the game that he was most pleased with was the number of other programmers it inspired to write their own games.

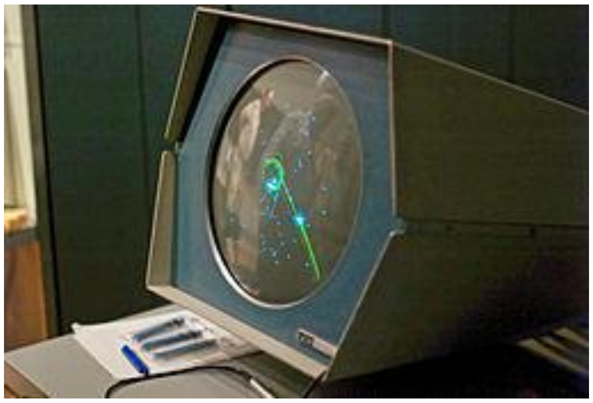


Figure 2.5 – Spacewar on PDP

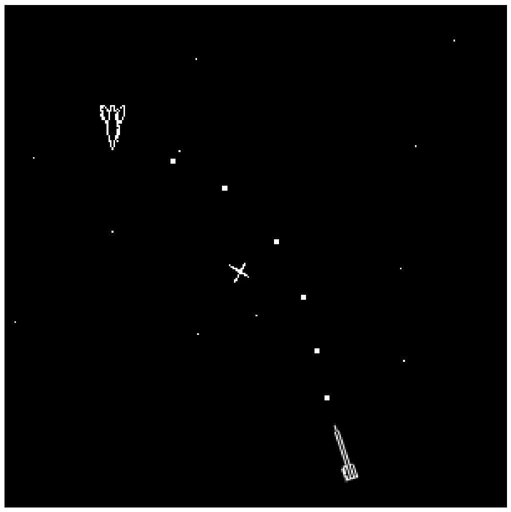


Figure 2.6 – Spacewar gameplay.

In 1966, Ralph Baer resumed work on an initial idea he had in 1951 to make an interactive game on a television set. In May 1967, Baer and an associate created the first game to use a raster-scan video display, or television set, directly displayed via modification of a video signal – i.e. a "video" game. The "Brown Box", the last prototype of seven, was released in May 1972 by Magnavox under the name Odyssey (after the Computer Space Game and before Pong). It was the first home video game console.

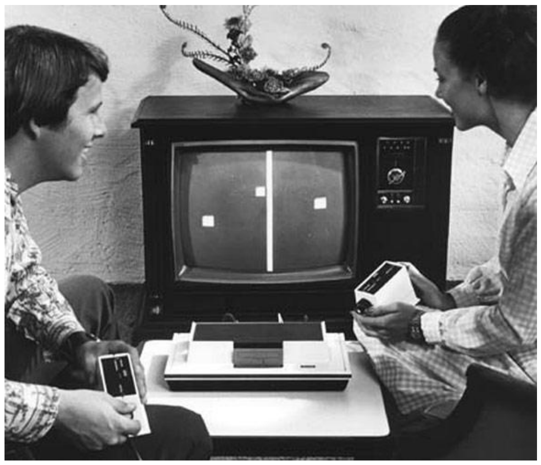


Figure 2.7 – Odyssey was the first home video game console.

In 1971, Bill Pitts and Hugh Tuck developed the first coin-operated computer game, *Galaxy Game*, at Stanford University using a DEC PDP-11/20 computer; only one unit was ever built although it was later adapted to run up to eight games at once.

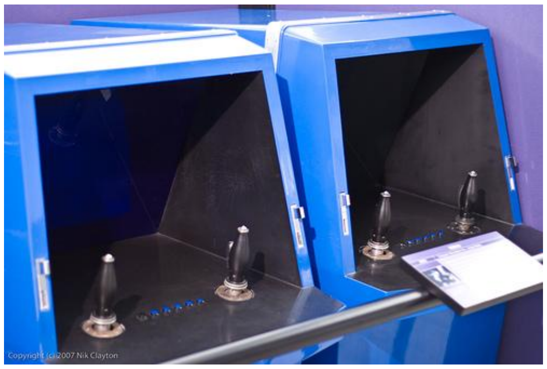


Figure 2.8 – Galaxy Game was the first coin-operated computer game.

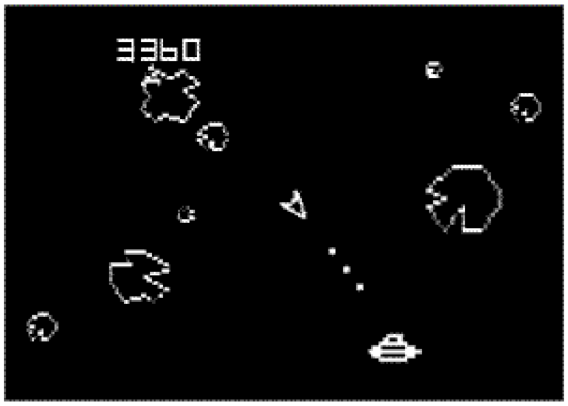


Figure 2.9 – Galaxy gameplay.

Two months after Galaxy Game's installation, *Computer Space* by Nolan Bushnell and Ted Dabney was released, which was the first coin-operated video game to be commercially sold and the first widely available video game of any kind. Both games were variations on the vector display 1961 *Spacewar!*; however, Bushnell and Dabney's used an actual video display by having an actual television set in the cabinet.

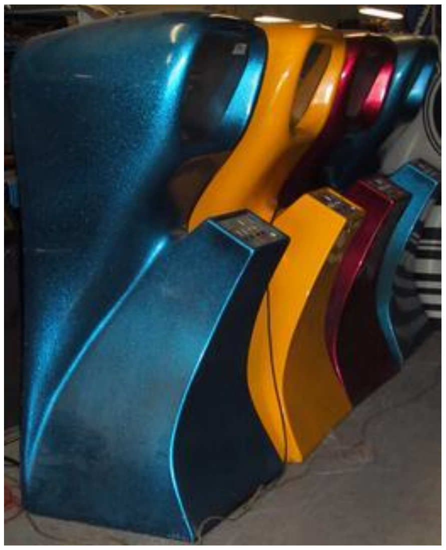


Figure 2.10 – Computer Space was the first coin-operated video game to be commercially sold

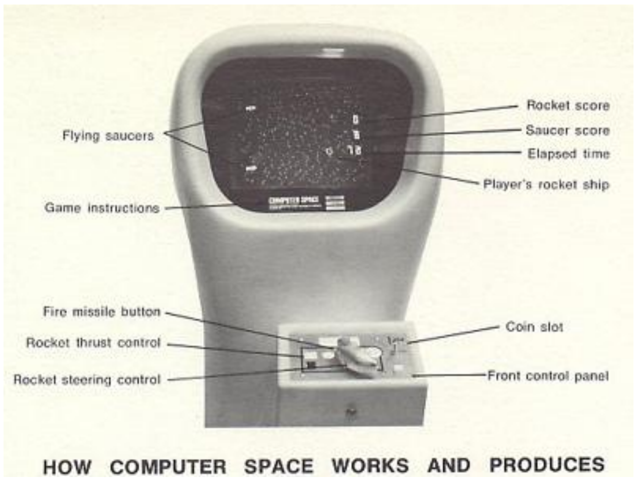


Figure 2.11(Right) – Computer Space gameplay

Pong, also by Bushnell and Dabney, used the same television set design as *Computer Space*, and was not released until 1972 – a year after *Computer Space (and six months after Odyssey)*. *Pong* was one of the first video games to reach mainstream popularity.

Figure 2.12 – Pong was one of the first video games to reach mainstream popularity

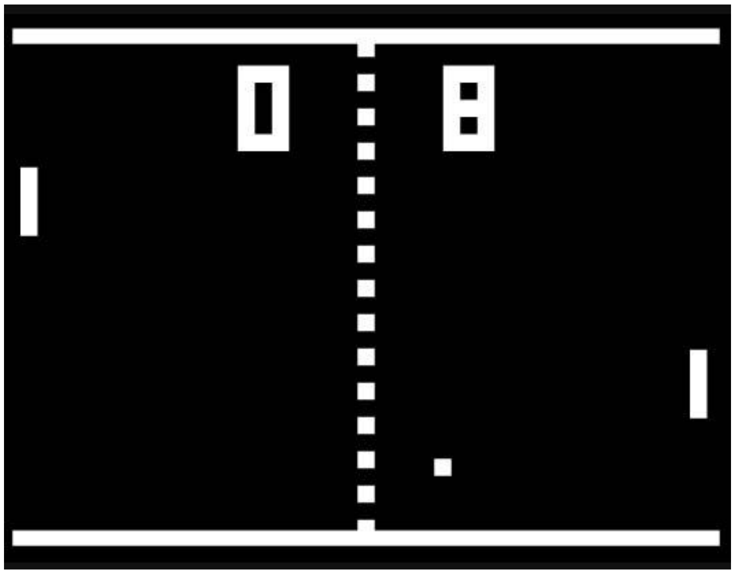


Figure 2.13 – Pong gameplay

2.2.2 Controversy and lawsuits

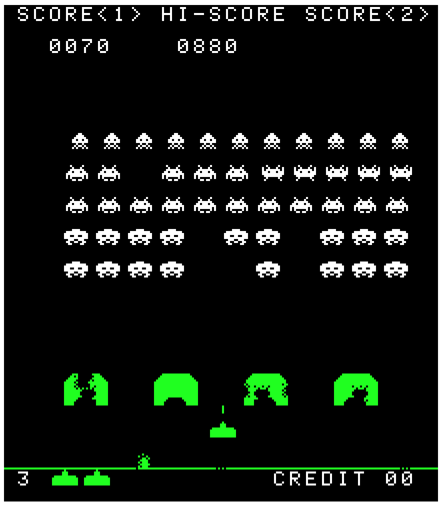
Magnavox settled a court case against Atari, Inc. for patent infringement in Atari's design of *Pong*, as it resembled the tennis game for the Odyssey. Over the next decade, Magnavox sued other big companies such as Coleco, Mattel, Seeburg, Activision and either won or settled every suit. In 1985, Nintendo sued Magnavox and tried to invalidate Baer's patents by saying that the first video game was William Higinbotham's *Tennis for Two* game built in 1958. These trials defined a video game as an apparatus that displays games by manipulating the video display signal of the raster equipment: a television set, a monitor, etc. The court ruled that this game did not use video signals and could not qualify as a video game. As a result, Nintendo lost the suit and continued paying royalties to Sanders Associates.

2.2.3 Home computer games

While the fruit of retail development in early video games appeared mainly in video arcades and home consoles, home computers began appearing in the late 1970s and were rapidly evolving in the 1980s, allowing their owners to program simple games. Hobbyist groups for the new computers soon formed and PC game software followed.

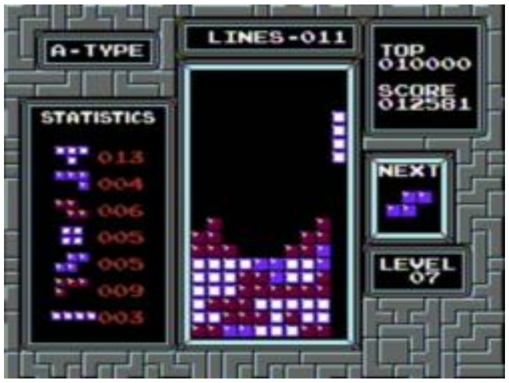
2.3 Early Trademark Games

Space Invaders is an arcade video game designed by Tomohiro Nishikado, and released in 1978. It was originally manufactured and sold by Taito in Japan, and was later licensed for production in the United States by the Midway division of Bally. *Space Invaders* is one of the earliest shooting games and the aim is to defeat waves of aliens with a laser cannon to earn as many points as possible. To complete it, Tomohiro Nishikado had to design custom hardware and development tools. It was one of the forerunners of modern video gaming and helped expand the video game industry from a novelty to a global industry. When first released, *Space Invaders* was very successful. Following its release, the game caused a temporary shortage of 100-yen coins in Japan and grossed US $2 billion worldwide by 1982.



Figures 2.14 – Space Invaders Gameplay

Tetris is a tile-matching puzzle video game originally designed and programmed by Alexey Pajitnov in the Soviet Union. It was released on June 6th 1984. He derived its name from the Greek numerical prefix *tetra-* (all of the game's pieces contain four segments) and tennis, Pajitnov's favorite sport. The *Tetris* game is a popular use of tetrominoes, the four element special case of polyominoes. Polyominoes have been used in popular puzzles since at least 1907, and the name was given by the mathematician Solomon W. Golomb in 1953. While versions of *Tetris* were sold for a range of 1980s home computer platforms as well as the arcades, it was the hugely successful handheld version for the Game Boy launched in 1989 that established the game as one of the most popular ever.



Figures 2.15 – Tetris Gameplay.

Super Mario Bros. is a 1985 platform video game developed by Nintendo, published for the Nintendo Entertainment System as a pseudo-sequel to the 1983 game *Mario Bros.* It is the first of the *Super Mario* series of games. In *Super Mario Bros.*, the player controls Mario and in a two-player game, a second player controls Mario's brother Luigi as he travels through the Mushroom Kingdom in order to rescue Princess Toadstool from the antagonist Bowser. *Super Mario Bros.* popularized the side-scrolling genre of video games and led to many sequels in the series that built upon the same basic premise. Altogether, excluding Game Boy Advance and Virtual Console sales, the game has sold forty million copies, making it the best-selling video game in the *Mario* series and the second best-selling game in the world. Almost all of the game's aspects have been praised at one time or another, from its large cast of characters to a diverse set of levels. One of the most-praised aspects of the game is its precise controls. The player is able to control how high and far Mario or Luigi jumps, and how fast he can run.

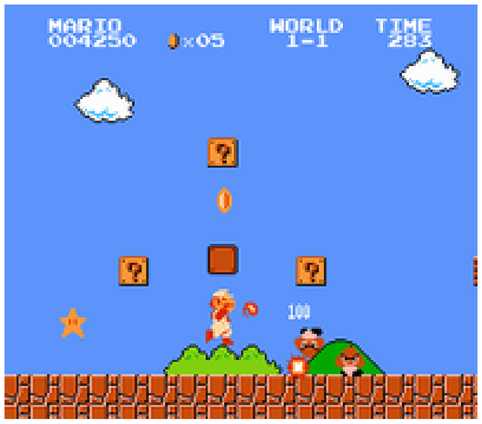
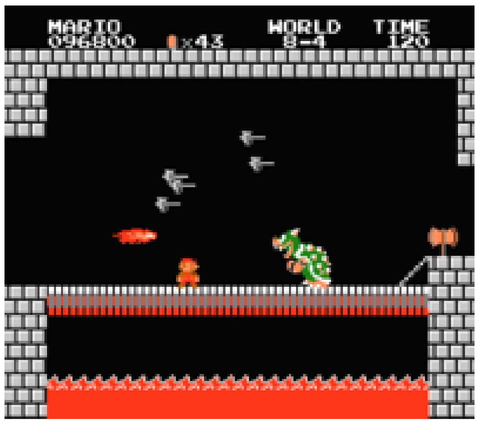


Figure 2.16 - Super Mario Bros 1 Gameplay.



Figures 2.17 - Super Mario Bros 1 Dungeon level.

2.4 The Action Role Playing genre

2.4.1 Definition of ARPG

The **action game** is a video game genre that emphasizes physical challenges, including hand–eye coordination and reaction-time [11].

Role-playing video games (commonly referred to as **role-playing games** or **RPGs**, as well as **computer RPGs** or **CRPGs**) are a video game genre where the player controls the actions of a protagonist as this character lives immersed in a fictional world [12].

Action role-playing games (abbreviated **action RPG**, **action/RPG**, or **ARPG**) form a loosely defined sub-genre of role-playing video games that incorporate elements of action or action-adventure games, emphasizing real-time action where the player has direct control over characters, instead of turn-based or menu-based combat (which is mostly used in RPGs) [13].

2.4.2 ARPG examples

The Dragon Slayer video game is regarded as the one of the early precursors of the action RPG genre, developed by Nihon Falcom and designed by Yoshio Kiya. It was originally released in 1984 for the NEC PC-88 computer, and became a major success in Japan. *Dragon Slayer* was entirely real-time with action-oriented combat.

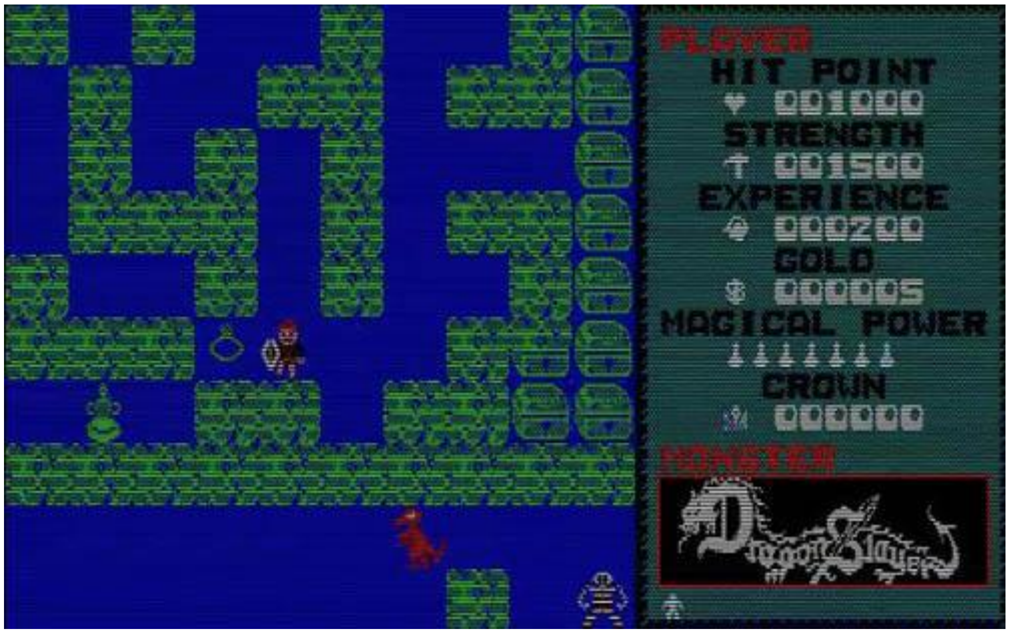


Figure 2.18 – Dragon Slayer Gameplay.

As game engines became more complex, new ARPGs were created with more freedom of movement for the player, advanced interactions, challenging AI, innovative gameplay etc. Trademark games of this genre include Gothic, Diablo, Elder Scrolls, Fable, Witcher and many more.

Figure 2.19 – Fable Gameplay

2.5 Game Engines

2.5.1 Immersion of Game Engines

With the advance of technology and increase in processing power, the development of more complex games became a reality. Developers could now focus on 3D design. At first rendering engines emerged that focused on projecting 3D models on the screen. The use of a **graphics processing unit** (**GPU**) enabled very efficient manipulation of computer graphics. Modern GPUs' highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large blocks of data is done in parallel. Besides graphic engines a game nowadays needs more functional components such as sound engine, physics engine, etc. All those subsystems of game functionality (often called middleware) are the components of a system called game engine [8]. Game engines are used primarily for developing next generation games by allowing the developers to focus on higher level aspects of a game (interaction between objects) instead of building a game from scratch as in earlier eras (the game had to be designed from the bottom up to make optimal use of the display hardware). As game engine technology matures and becomes more user-friendly, the application of game engines has broadened in scope. They are now being used for serious games: visualization, training, medical, and simulation applications. To facilitate this accessibility, new hardware platforms are now being targeted by game engines, including mobile phones and web browsers.

2.5.2 Recent trends

Game engines are broadly used for the development of AAA games. AAA games are high quality games with high budget. The cost to develop a frontline software title generally ranges from $10 million to $60 million. Although AAA games are high quality games, they don’t introduce any new innovative elements in gaming but recycle old ones and make them visually superior than any predecessors. AAA game creators cannot afford to work on new innovative ideas because of the risk of failure in the game market. Publishers are much more risk-averse to spending twenty or thirty million dollars on a title. That aversion to risk means that many of the most exciting new ideas about games may not come with new AAA title releases but from downloadable games created by small, nimble teams that can afford to take that risk [9].

2.6 Game Engine Research

Nowadays, there is a large variety of game engines that a programmer can use. In order to restrain this list, some specific requirements and constraints were specified in relation to requirements associated to the game described in this diploma thesis. These requirements were:

- The core code of the game engine should be C/C++ based, offering this way complete control along with vast amount of resouces and libraries, and fast execution speed.
- The game engine should combine the necessary photorealism with efficiently real-time execution.
- The game engine should support cross-platform distribution. The code can be executed equally from different operating systems.
- The game engine should offer a completed package containing the basic elements for creating a game (graphics engine, sound engine, physics engine, scripting language) that is distributed free of charge for non-commercial applications.

Among the engines that fulfill these requirements, the most promising and powerful ones are:

CryEngine

CryEngine is a game engine used for the first-person shooter video game Far Cry. It was originally developed by Crytek as a technology demo for Nvidia and, when the company saw its potential, it was turned into a game. CryEngine 3 has been used as a benchmark for visual graphics for some time and it continues to push the limit what games are capable of. One of CryEngine's features is its ability to produce huge beautiful, highly detailed landscapes. Free version of CryEngine 3 SDK is available with all the necessary tools to start game creation.

Figure 2.20 – CryEngine 3 graphics

Source

Source is a 3D game engine developed by Valve Corporation. It debuted in June 2004 with *Counter-Strike: Source*, followed shortly afterwards with *Half-Life 2*, and has been in active development ever since. Source was created to power first-person shooters, but has also been used professionally to create role-playing, side-scroller, puzzle, MMORPG, top-down shooter and real-time strategy games.

Figure 2.21 – Source engine graphics

Unity

Unity (also called **Unity3D**) is a cross-platform game engine with a built-in IDE developed by Unity Technologies. It is used to develop video games for web plugins, desktop platforms, consoles and mobile devices, and is utilized by over one million developers. Unity is primarily used to create mobile and web games, but can also deploy games to consoles or the PC. The game engine was developed in C/C++, and is able to support code written in C# or JavaScript. It grew from an OS X supported game development tool in 2005 to the multi-platform game engine that it is today. The game engine is downloadable from their website in two different versions: Unity and Unity Pro.

Figure 2.22 – Unity engine graphics

Unreal Engine

The **Unreal Engine** is a game engine developed by Epic Games, first illustrated in the 1998 first-person shooter game *Unreal*. Although primarily developed for first-person shooters, it has been successfully used in a variety of other genres, including stealth, MMORPGs and other RPGs. With its code written in C++, the Unreal Engine features a high degree of portability and is a tool used by many game developers today. The latest release is the UE3, designed for Microsoft's DirectX 9 (for Windows and Xbox 360), DirectX 10 (for Windows Vista) and DirectX 11 (for Windows 7 and later), OpenGL for OS X, Linux, PlayStation 3, Wii U, iOS, Android, and Stage 3D for Adobe Flash Player 11.

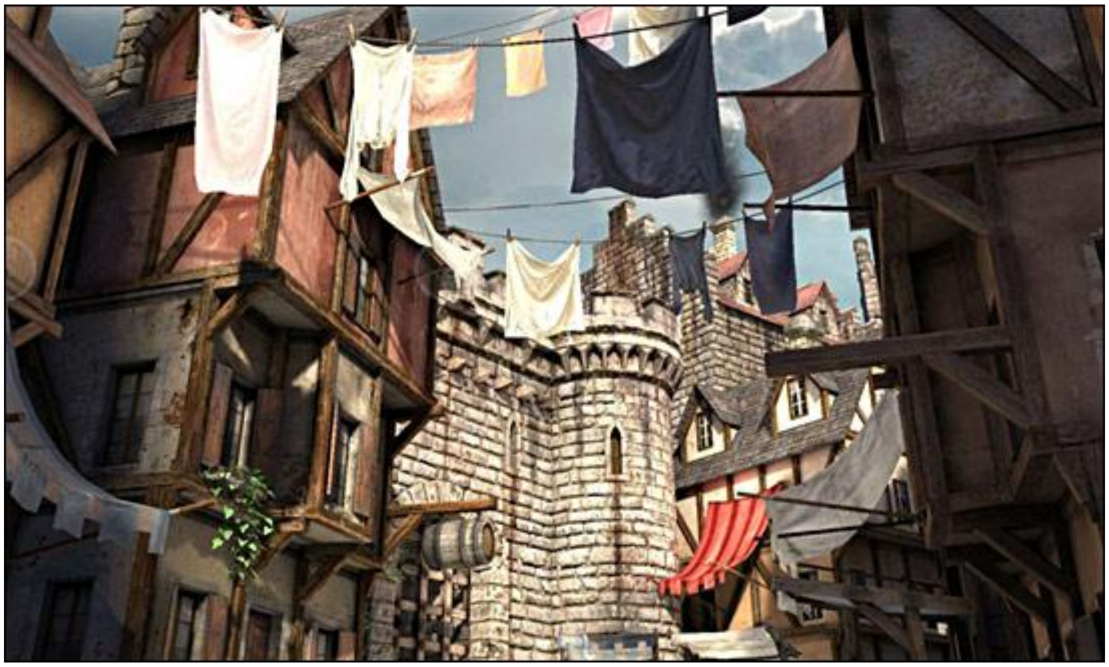


Figure 2.23 – Unreal engine graphics

The game engines listed above fulfill our requirements. For the current project the Unreal Engine 3 was chosen. The main reasons for this choice are that it is an engine which has made over one hundred AAA games, has a very large community and support through the epic games forum and is constantly updated. Having tested unreal engine’s numerous games, the game engine has immense capabilities which combined with its unique photorealism makes it an ideal choice for the current project.

Chapter 3: UDK, UnrealScript and Tools

3.1 Unreal Engine

3.1.1 Key features

In order to create a 3D computer game these days, it is necessary to have acquired the knowledge of using a game engine. A **game engine** is a system designed for the creation and development of video games [9]. For the current project Unreal Engine was chosen for the reasons described in section 2.6.

While the Unreal Engine 3 has been quite open for professional developers to work with, the ability to publish and sell games made using UE3 was restricted to licensees of the engine. However, in November 2009, Epic released a free version of UE3's SDK, called the Unreal Development Kit (UDK), which is available to the general public.

In short, Unreal Engine is a System that organizes gameplay assets (characters, artwork, music etc.) into a visually interactive environment, one that behaves the why it is programmed to (as an interactive game) [14].

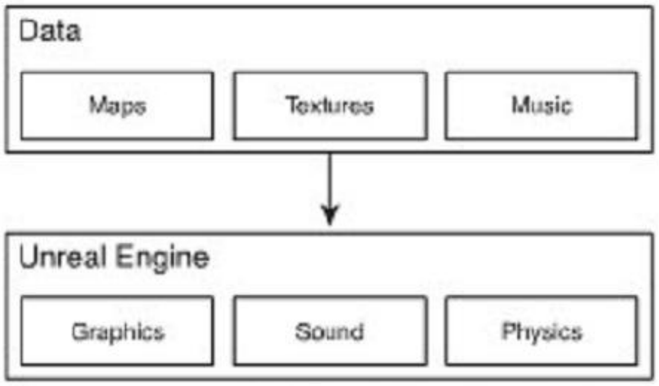


Figure 3.1 – The flow of data through the Unreal Engine

3.1.2 Unreal Engine Components

The Unreal Engine contains several components. Each can work independently, but they're kept humming along in harmony by a central "core engine." The components of Unreal Engine are the following:

The Graphics Engine

The graphics engine controls what you see while you're playing a game; it produces the jaw-dropping visuals that appear. Without it, you'd just see a tremendous list of ever-changing property values, such as the player's location, health, any weapons they are carrying, ammo, what animation is currently playing, and so on. The graphics engine translates this information into the environments you see during gameplay (figure 3.2).

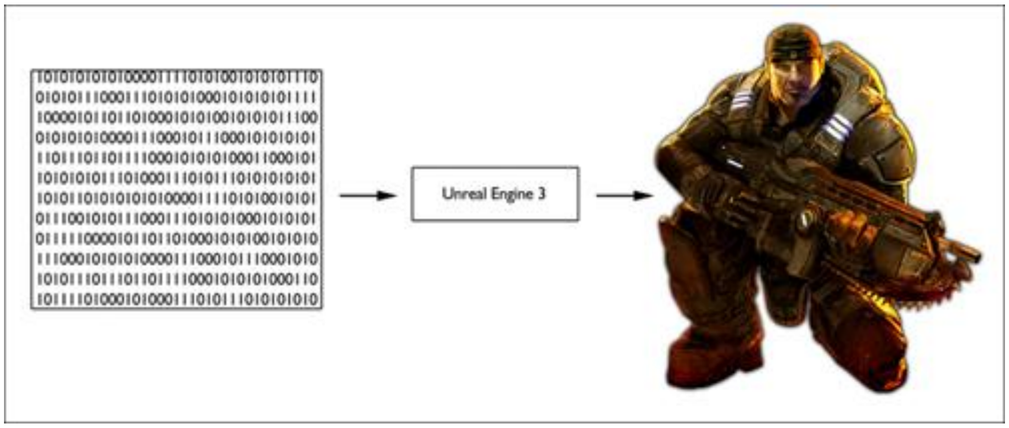


Figure 3.2 – The graphics engine is responsible for turning vast lists of properties into interactive imagery on the screen.

The graphics engine is responsible for "behind-the-scenes" calculations. For example, it determines which objects display in front of others, and which objects should "occlude" (hide) others from view.

In Unreal Engine 3, when you occlude "rear" objects from view, they don't render at all. That frees up processing cycles for something more useful.

Certain aspects of level optimization are still in place, and work with the rendering engine to help control what the player can and can't see. Unreal Engine 3 supports Level Streaming, which allows different sublevels to be loaded or unloaded from memory during gameplay. With Level Streaming, when a player enters one section of a level, the area he's exiting can be dumped from memory and no longer rendered by the graphics engine. This means you can present the player with truly vast environments without ever exposing them to a loading screen or a break from the action.

The Unreal graphics engine also renders the various shaders and materials that are applied to all your objects. (The shaders give you a *lot* more flexibility in controlling how your graphics render.)

Rendering is just the way a computer graphics program generates an image from the information it's been handed: information about things like geometry, viewpoint, texture, lighting, and shadows.

One important part of rendering is *lighting*. Unreal Engine 3 boasts an extremely robust lighting system. That allows for dynamic lighting and shadowing, soft shadowing effects, and, well, more realism.

The Sound Engine

The sound engine makes the use of sound effects possible in a game. It takes all the sound effects recorded and imported, queues them based on certain events in the game (such as playing an explosion sound when a projectile like fireball is thrown), and re-creates them as closely as possible, using the sound hardware built into the computer or gaming console.

Firstly the designer imports the sound effects or music, and then uses them to construct SoundCues. SoundCues are basically lists of instructions that control how sounds are played. For example, SoundCues might control whether a sound loops, whether its pitch or modulation should be tweaked, or whether it should be combined with other sound bites.

The Physics Engine

The in-game physics are handled through NVIDIA's state-of-the-art PhysX physics engine.

PhysX supports a whole laundry list of physical simulations, including rigid bodies, constrained bodies, dynamic skeletal meshes (ragdolls), and even cloth. Each of these can be interactive in the game and can be manipulated by the player, either directly or through scripted sequences. The physics system can also work with the sound engine to produce dynamic sound, for example whenever a physics-driven object strikes the ground (Figure 3.3).

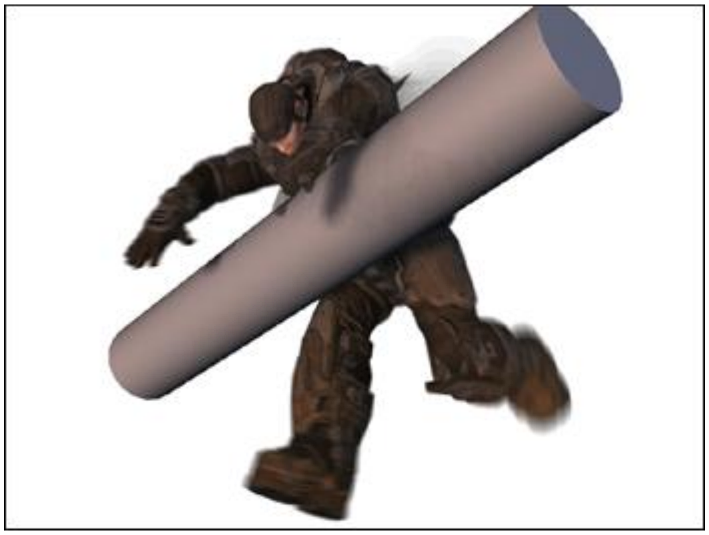


Figure 3.3 – Using the physics engine, characters can turn into ragdolls.

The Input Manager

Each time a player produces an input, the Input Manager sets a series of properties that control what should happen in the game. For example, when a player presses the “Jump” button, the Input Manager sends a signal to the core engine, which causes it to play a jump animation, send the relevant information to the graphics engine to render that action on the screen, and queue a jump sound effect from the sound engine. All that happens *very* quickly; so fast, players almost never notice the delay between their input and the game’s reaction.

One great thing about Unreal Engine 3’s Input Manager is that it’s extremely flexible. You can adjust easily when you’re moving between different gaming platforms or input devices.

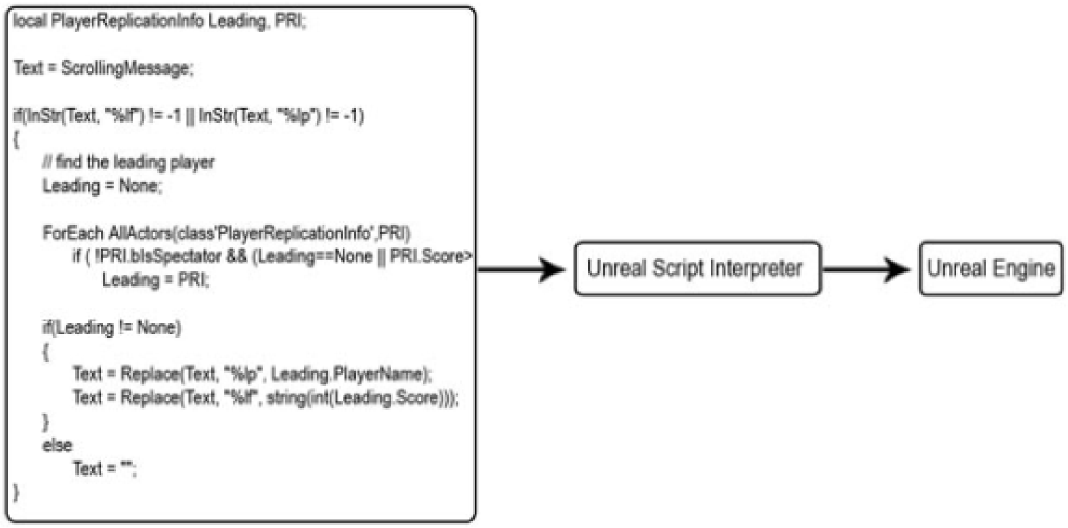
Network Infrastructure

During a multiplayer game, each player’s computer is in constant communication with another computer that acts as a server. This server computer can also run a “client” application, which means another player can actually join in and play from the server computer. When a computer is running strictly as a server (with no client), it’s called a “dedicated server.” Dedicated servers are generally preferred for online gaming, as the server can use all its resources for serving, not playing.

The key to efficient network gaming is to limit the data that’s sent across the network to only the most important aspects of gameplay, such as each player’s position and interactions with his immediate environment. By sending only the most relevant details, you can have very fast gameplay even on slower connections.

The UnrealScript Interpreter

UnrealScript is a scripting language that allows programmers or users to adjust virtually anything the Engine is doing, without touching the actual game source code. In many respects, UnrealScript is a lot like the popular languages Java and C++. But, as a scripting language, it’s intended to be relatively easy. In fact, its creator, Tim Sweeney, said that where he had to make tradeoffs, he chose simplicity over execution speed. The UnrealScript Interpreter is the component of the engine responsible for transforming the UnrealScripts you create into code that the engine can process (figure 3.4).



```
local PlayerReplicationInfo Leading, PRI;

Text = ScrollingMessage;

if(InStr(Text, "%lf") != -1 || InStr(Text, "%lp") != -1)
{
    // find the leading player
    Leading = None;

    ForEach AllActors(class'PlayerReplicationInfo',PRI)
        if ( !PRI.bIsSpectator && (Leading==None || PRI.Score>
            Leading = PRI;

    if(Leading != None)
    {
        Text = Replace(Text, "%lp", Leading.PlayerName);
        Text = Replace(Text, "%lf", string(int(Leading.Score)));
    }
    else
        Text = "";
}
```

Figure 3.4 – The UnrealScript Interpreter

3.1.3 Overview of Component Interaction

Most games are driven by a *game loop*: a part of the game's programming code that constantly repeats itself, checking for any input from the player and updating all aspects of the game accordingly (figure 3.5). Within this loop, the game engine repeatedly performs many individual checks to see what's changed in the gaming environment. For instance, *has a player moved? Fired his weapon? Taken damage? Have his enemies moved?*

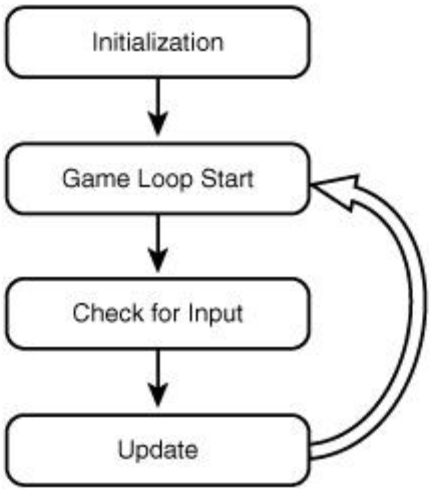


Figure 3.5 – This diagram illustrates a basic game loop.

The game loop generally also keeps track of "generic" tasks. For example, it queues the graphics engine to redraw the screen, tells the sound engine to play any sounds that have come up, and sends data across the network. Typical game loops are designed "linearly," which means that each cycle of the loop runs through its series of checks in the exact same order every time, and give each check equal priority. But, as you'll see in a moment, the Unreal Engine's game loop is a bit smarter than that.

The Unreal Engine uses an *event-driven* game loop. This means the engine contains a list of events that the various components of the engine needs to address. These events are created from many different sources, such as player inputs, data from the physics system, or communication between components. Everything is passed through the system via this event list (figure 3.6).

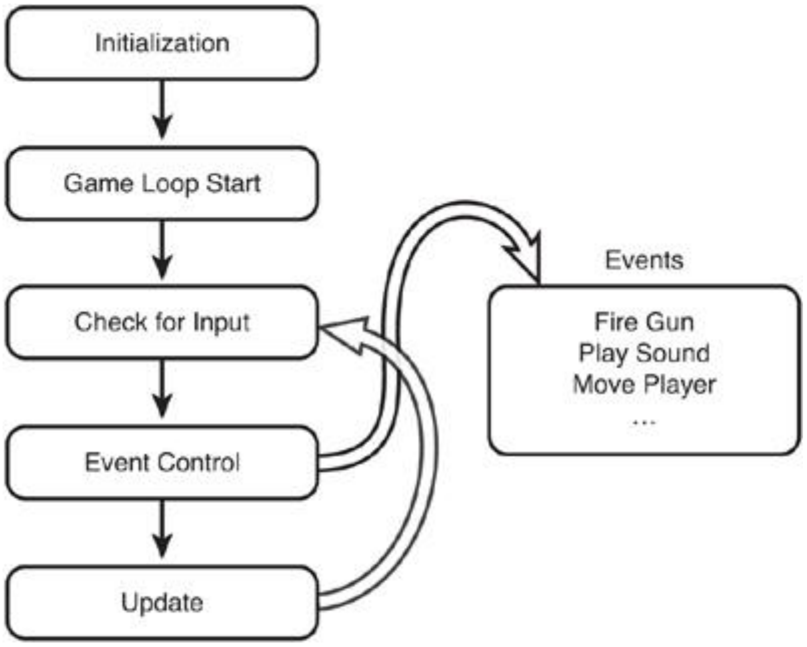


Figure 3.6 – An event-based system

In Unreal Engine 3, each event is given a specific priority. Rather than updating every single aspect of the game during every game cycle in a specific order, events are processed based on their importance. For example, a weapon is fired. The game loop recognizes this, then reacts by sending multiple new events into the queue, such as instructions to launch a projectile and play a sound. The projectile's launch is more important, as it directly affects gameplay depending on whether it strikes an enemy or some other object that must react to it. Therefore, the Unreal Engine gives the projectile launch a higher priority than the "noncritical" gunfire sound.

3.1.4 Engine Components at Work

Let's take a closer look at how these components work together during gameplay (figure 3.7). When you start an Unreal Engine-based game, each engine component is *initialized*. Initialization takes place during the loading time between launching the game and seeing the opening title screen.

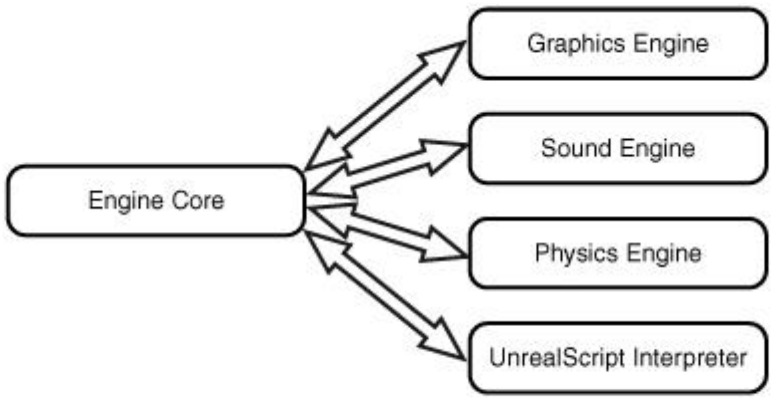


Figure 3.7 – Unreal Engine component interaction

Now the engine components start communicating with one another. At startup, the core engine initializes the graphics engine, the sound engine, the physics engine, and the UnrealScript Interpreter. The core engine also begins sending commands to each component. Some components, such as the physics engine, begin sending data back to the core engine, keeping everything in sync. Once all components are initialized, the game's user interface is displayed, and the engine waits for the player's input.

To start the playable part of the game, the core engine loads a level that contains all assets needed for gameplay: textures, materials, sounds, meshes, animations, scripts, and so forth. Each asset is routed to the proper engine component. Materials are processed by the graphics engine, while rigid body objects are calculated by the physics engine. The event queue now "comes to life" and is quickly populated with events the core engine needs to manage as it delivers the gaming experience.

To summarize:

- Game initialization:
    - The core engine initializes each engine component.
    - Components begin sending data back to the core engine for synchronization.
    - Game is ready for user input.

- Game launch (choosing and starting a level):
    - A map is loaded, containing all game assets and their corresponding properties.
    - Each asset's information is routed to its respective engine component.

- Gameplay (excessive use of flak cannons, shock rifles, and weapons):
    - Each engine component sends information to the core engine, which is sorted as a series of prioritized events in the event queue.
    - The Unreal Engine's game loop runs constantly, evaluating each event by priority and performing the highest-priority tasks first.

3.2 UnrealScript

At this point, we will take a closer look at UnrealScript, since this is the medium that connects the object interactions in the game.

3.2.1 Design goals of UnrealScript

UnrealScript was created to provide the development team and the third-party Unreal developers with a powerful, built-in programming language that maps naturally onto the needs and nuances of game programming.

The major design goals of UnrealScript are [15]:

- To support the major concepts of time, state, properties, and networking which traditional programming languages don't address. This greatly simplifies UnrealScript code. The major complication in C/C++ based AI and game logic programming lies in dealing with events that take a certain amount of game time to complete, and with events which are dependent on aspects of the object's state. In C/C++, this results in spaghetti-code that is hard to write, comprehend, maintain, and debug. UnrealScript includes native support for time, state, and network replication which greatly simplify game programming.

- To provide Java-style programming simplicity, object-orientation, and compile-time error checking. Much as Java brings a clean programming platform to Web programmers, UnrealScript provides an equally clean, simple, and robust programming language to 3D gaming. The major programming concepts which UnrealScript derives from Java are:

    - a pointerless environment with automatic garbage collection;
    - a simple single-inheritance class graph;
    - strong compile-time type checking;
    - a safe client-side execution "sandbox";
    - the familiar look and feel of C/C++/Java code.

- To enable rich, high level programming in terms of game objects and interactions rather than bits and pixels. As mentioned above where design tradeoffs had to be made in UnrealScript, execution speed was sacrificed for development simplicity and power. After all, the low-level, performance-critical code in Unreal is written in C/C++ where the performance gain outweighs the added complexity. UnrealScript operates at a level above that, at the object and interaction level, rather than the bits and pixels level.

3.2.2 The Unreal Virtual Machine

The Unreal Virtual Machine consists of several components: The server, the client, the rendering engine, and the engine support code.

The Unreal server controls all gameplay and interaction between players and actors. In a single-player game, both the Unreal client and the Unreal server are run on the same machine; in an Internet game, there is a dedicated server running on one machine; all players connect to this machine and are clients.

All gameplay takes place inside a "level", a self-contained environment containing geometry and actors. Though UnrealServer may be capable of running more than one level simultaneously, each level operates independently, and are shielded from each other: actors cannot travel between levels, and actors on one level cannot communicate with actors on another level.

Each actor in a map can either be under player control (there can be many players in a network game) or under script control. When an actor is under script control, its script completely defines how the actor moves and interacts with other actors.

With all of those actors running around, scripts executing, and events occuring in the world, you're probably asking how one can understand the flow of execution in an UnrealScript. The answer is as follows:

To manage time, Unreal divides each second of gameplay into "Ticks". A tick is the smallest unit of time in which all actors in a level are updated. A tick typically takes between 1/100th to 1/10th of a second. The tick time is limited only by CPU power; the faster machine, the lower the tick duration is.

Some commands in UnrealScript take zero ticks to execute (i.e. they execute without any game-time passing), and others take many ticks. Functions that require game-time to pass are called "latent functions". Some examples of latent functions include *Sleep*, *FinishAnim*, and *MoveTo*. Latent functions in UnrealScript may only be called from code within a state (the so called "state code"), not from code within a function (that includes functions define within a state).

While an actor is executing a latent function, that actor's state execution doesn't continue until the latent function completes. However, other actors, or the VM, may call functions within the actor. The net result is that all UnrealScript functions can be called at any time, even while latent functions are pending.

In traditional programming terms, UnrealScript acts as if each actor in a level has its own "thread" of execution. Internally, Unreal does not use Windows threads, because that would be very inefficient (Windows 95 and Windows NT do not handle thousands of simultaneous threads efficiently). Instead, UnrealScript simulates threads. This fact is transparent to UnrealScript code, but becomes very apparent when you write C++ code that interacts with UnrealScript.

All UnrealScripts are executed independently of each other. If there are fifty monsters walking around in a level, all fifty of those monsters' scripts are executing simultaneously and independently each "Tick".

3.2.3 Object Hierarchy

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

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 doesn't 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)

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)

The parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.

Player (extends Actor)

The class that defines the logic of the pawn. If pawn resembles the body, Player is the brain commanding the body. Executable functions can be called from this type of class.

Class (extends Object)

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, code for any Object class can be written, but 99% of the time, the programmer code written will be for a class derived from Actor. Most of the useful UnrealScript functionality is game-related and deals with actors.

3.2.4 Anatomy of an UnrealScript

Now let's see how a class in UnrealScript looks like. Below there is the code for the class MyActor.uc that extends the basic `Actor` class. In this script, one can see the basic variable declarations of UnrealScript in a class (enum, struct, int, float, bool, Actor, etc) using the `var` keyword and some simple variable initializations in the default properties block. Furthermore a simple state is defined(`NewState`) and a simple function is declared as well(`MyFunction()`) inside the class that is overridden in the `NewState` state. The most complex of these elements are described in detail at the following section (**3.2.5**).

```
/************************ start of MyActor.uc ***********************/
/*********************
 * Class Declaration *
 *********************/


class MyActor extends Actor;

/************************************
 * Instance Variables/Structs/Enums *
 ************************************/

enum MyEnum
{
	ME_None.
	ME_Some,
	ME_All
}

struct MyStruct
{
	var int IntVal;
	var float FloatVal;
}

var int IntVar;

var float FloatVar;
var bool BoolVar;

var Actor ActorVar;

var MyEnum EnumVar;

var MyStruct StructVar;

/**********************
 * Functions & States *
 **********************/

function MyFunction()
{
      local int TempInt;

      if(ActorVar != none && BoolVar)
      {
            TempInt = IntVar;
      }
}

state NewState
{
      function MyFunction()
      {
            local float TempFloat;

            if(ActorVar != none && BoolVar)
            {
                  TempFloat = FloatVar;
            }
      }
}

/**********************
 * Default Properties *
 **********************/

defaultproperties
{
      IntVar=5
      FloatVar=10.0
      BoolVar=true
}
/************************ end of MyActor.uc ************************/
```

3.2.5 UnrealScript Elements

We will not emphasize on the simple elements of UnrealScript (variables, structs, enums, arrays, loops) as they are covered in detail at the UnrealScript Reference and are very similar to most programming languages. We will focus though, on elements that are crucial for game programming:

UnrealScript States

Historically, game programmers have been using the concept of states ever since games evolved past the "pong" phase. States (and what is known as "state machine programming") are a natural way of making complex object behaviour manageable. However, before UnrealScript, states have not been supported at the language level, requiring developers to create C/C++ "switch" statements based on the object's state. Such code was difficult to write and update. UnrealScript supports states at the language level.

In UnrealScript, each actor in the world is always in one and only one state. Its state reflects the action it wants to perform. For example, moving brushes have several states like "StandOpenTimed" and "BumpOpenTimed". Pawns have several states such as "Dying", "Attacking", and "Wandering".

In UnrealScript, you can write functions and code that exist in a particular state. These functions are only called when the actor is in that state. For example, suppose the programmer is writing a monster script, and he is contemplating how to handle the *SeePlayer* function. While wandering around, he wants to attack the player he sees. When he is already attacking the player, he wants to continue on uninterrupted. The easiest way to do this is by defining several states (Wandering and Attacking), and writing a different version of "Touch" in each state. UnrealScript supports this. Below is a simplified version of State functions for the `PlayerController` class (the brain of our Pawn). If the Player is not in state `Dead` then the `StartFire` function will call The `Pawn.StartFire` function which will do the necessary processing before in order to start shooting the enemies. But if the Player Controller is in state Dead, then the `StartFire` function will do nothing (cannot commence fire if the player is dead).

```
class BDGPlayerController extends PlayerController;
…
exec function StartFire( optional byte FireModeNum )
{
	if ( Pawn != None && !bCinematicMode && !WorldInfo.bPlayersOnly )
	{
		Pawn.StartFire( FireModeNum );
	}
}

state Dead
{
	exec function StartFire( optional byte FireModeNum ){}
}
…
```

Latent functions

A latent function is a function that executes "slowly" (i.e. non-blocking), and may return after a certain amount of "game time" has passed. This enables you to perform time-based programming -- a major benefit which neither C, C++, nor Java offer. Namely, the programmer can write code in the same way he conceptualize it; for example, he can write a script that says the equivalent of "open this door; pause two seconds; play this sound effect; open that door; release that monster and have it attack the player". He can do this with simple, linear code, and the Unreal engine takes care of the details of managing the time-based execution of the code.

There are three main latent functions available to all actors:

- `Sleep( float Seconds )` pauses the state execution for a certain amount of time, and then continues.
- `FinishAnim()` waits until the animation sequence currently playing completes, and then continues. This function makes it easy to write animation-driven scripts, scripts whose execution is governed by mesh animations. For example, most of the AI scripts are animation-driven (as opposed to time-driven), because smooth animation is a key goal of the AI system.
- `FinishInterpolation()` waits for the current InterpolationPoint movement to complete, and then continues.

Below there is a simple example of state code with the latent function sleep:

```
auto state MyState
{
Begin:
   `log( "MyState has just begun!" );
   Sleep( 2.0 );
   `log( "MyState has finished sleeping" );
   goto('Begin');
}
```

Timers

Timers are used as a mechanism for scheduling an event to occur, or reoccur, over time. In this manner, an Actor can set a timer to register itself with the game engine to have a `Timer()` function called either once, or recurring, after a set amount of time has passed. UnrealScript timers are just implemented as an array of structs inside each Actor (an Actor can have multiple timers pending). The struct contains the amount of time remaining before the timer expires, the function to call on expiry, etc.

The game loop normally ticks each Actor once per frame, and part of each Actor's `Tick()` function includes a call to `UpdateTimers()` which will check for any expired timers and call their appropriate UnrealScript function.

The granularity is limited to the frame delta time, but there are no hardware or OS resources required. All of this is implemented in C++ so the programmer could safely update hundreds of UnrealScript timers without any cause for concern. Of course he wouldn't want them all expiring simultaneously or every frame because they execute (slow) script code when they're activated.

Timer functions are only available to `Actor` subclasses. You can create multiple timers with each a different rate. Each timer has a unique target function (defaults to `Timer()`). Below there is a simplified example from the current project:

```
function StartSlowEffect(float AnimScale,float SlowDuration){

      /*
      code that slows the pawn
      …
      */
      SetTimer(SlowDuration,false,'StopSlowEffect');
}
```

The function above is located at the Pawn class. When it is called the necessary moves will be performed in order to slow the Pawn. At the end a timer will be set, with rate equal to the slow effect duration. When this duration passes the timer will be activated and will call the function StopSlowEffect, which, as the function name implies, is responsible for negating the slow effect.

Delegates

A delegate is a reference to a function bound to an object. Their main use is to provide a callback mechanism, for example to provide event notification in a user interface system.

Calling a delegate works just like calling a regular function:

```
    Class Button extends Window;

    var int MouseDownX, MouseDownY;

    delegate OnClick( Button B, int MouseX, int MouseY );

    function MouseDown( int MouseX, int MouseY )
    {
        MouseDownX = MouseX;
        MouseDownY = MouseY;
    }

    function MouseUp( int MouseX, int MouseY )
    {
        if( MouseX == MouseDownX && MouseY == MouseDownY )
            OnClick( Self, MouseX, MouseY );
    }
```

Or in case of a function argument:

```
   function DoStuff(delegate<OnClick> ClickDelegate)
   {
       ClickDelegate(Self, MouseX, MouseY);
   }
```

3.3 UDK Elements

Before going any further, a review of the most frequently used Unreal Editor assets will be performed [16]:

3.3.1 BSP Brushes

BSP Brushes: One should note that the term BSP (Binary Space Partitioning) is a data structure that is used to organize objects within a space of the level and not semantically the correct term for a type of geometry. CSG (Constructive Solid Geometry) is a more accurate term for the geometry created within the unreal engine by adding and subtracting brushes, but BSP has become the terminology to describe this geometry. So the terms BSP and CSG are often used interchangeably to refer to the geometry created within the editor using brushes.

BSP Brushes are the basic building blocks of a level. You can build a level with almost no BSP brushes, but you still have to have at least one BSP brush to "cut out" where the world is.

3.3.2 Static Meshes

Static Meshes: A Static Mesh is a piece of geometry that consists of a set of polygons which can be cached in video memory and rendered by the graphics card. This allows them to be rendered efficiently, meaning they can be much more complex than other types of geometry such as BSP brushes. Since they are cached in video memory, though, Static Meshes can be translated, rotated, and scaled, but they cannot have their vertices animated in any way.

Static Meshes are the basic unit used to create world geometry for levels created in Unreal Engine 3. These are 3D models created in external modeling applications (such as 3dsMax, Maya, Softimage, etc.) that are imported into Unreal Editor through the Content Browser, saved in packages, and then used in various ways to create renderable elements. The vast majority of any map in a game made with Unreal Engine 3 will consist of Static Meshes, generally in the form of StaticMeshActors. Other uses of Static meshes are for creating movers such as doors or lifts, rigid body physics objects, foliage and terrain decorations, procedurally created buildings, game objectives, and many more visual elements.

3.3.3 Skeletal Meshes

Skeletal Meshes: Skeletal meshes are built up of two parts, a set of polygons composed to make up the surface of the skeletal mesh and a hierarchical set of interconnected bones which can be used to animate the polygons.
Skeletal meshes are often used in Unreal Engine 3 to represent characters or other animating objects. The 3D models, rigging and animations are created in an external

modeling and animation applications (3DSMax, Maya, Softimage, etc) that are then imported into Unreal Engine 3 by using Unreal Editor's Content Browser and saved in packages.

3.3.4 Animation Aspects

Animations: Animations help to create the illusion of complex movement performed by skeletal meshes. Animations help to make skeletal meshes feel alive or have some sense of personality by defining the way characters walk, the way characters speak and act. From these behaviors, players feel that the virtual avatar is a little bit more real than just graphics on a screen.

The animation system is part of the Unreal Engine 3 pipeline. First, normal animation is processed (blending animations). Bone controllers (such as inverse kinematics) are then applied. Next morph targets are applied. Finally, the physics for the skeleton. The physics subsystem then processes the remaining physics, and the graphics subsystem then renders everything.

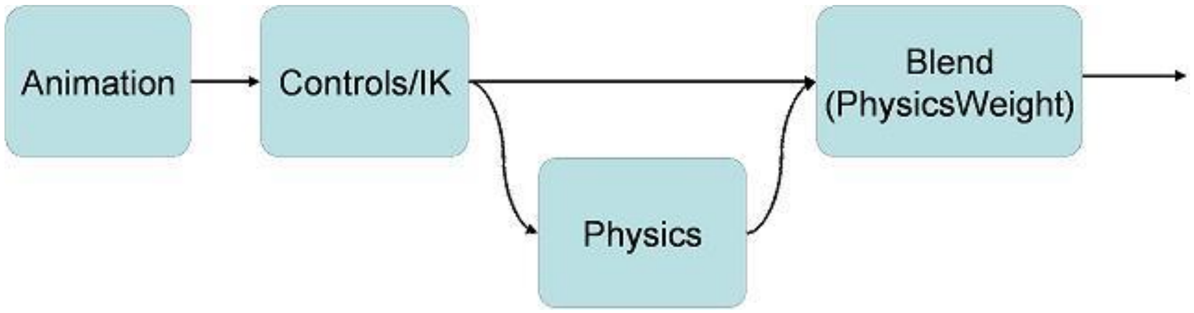


Figure 3.8 – Animation flow pattern

AnimSequence: An AnimSequence is a single animation, a collection of key frames, with its associated meta data information such as animation notifies.

AnimSet: An AnimSet is a collection of AnimSequences. They live in packages, and can be seen in the Content Browser in the same way as materials, meshes etc.

3.3.5 Materials

A material is applied in a 3d model (bsp brush, static mesh, skeletal mesh) and defines the behavior of an imported texture in lightning effects. By adjusting the proper parameters the material can simulate complex real world material behavior (specular, emissive, translucent, etc.).

3.4 Overview of Software Tools

In this section a brief list is provided with the software tools used to complete this project and the purpose of each tool.

Unreal Development Kit (UDK) [3]: Provides the Unreal Game Editor tool that helps you import, create and use the artwork you need in your game.

nFridge with Visual Studio [17]: The UDK does not contain an editor for writing UnrealScript. The most notable editor at the moment is nFringe by Pixel Mine Games. nFringe provides a total integrated development environment for UE3, and thus also for the UDK. nFringe builds upon the Visual Studio IDE provided by Microsoft. It works with both the professional version of Visual Studio as with the free Visual Studio Express.

Autodesk 3ds Max(with the appropriate fbx plugin) [18][19]: The Unreal Editor allows the developer to create basic 3D models but doesn't offer the convenience of a professional 3D Computer Graphics Software like 3ds max (or alternatively Maya). With 3ds max the designer can create models and animations and then import them to the Unreal Editor for use in your game.

Adobe Photoshop [20]: Used to create custom textures or edit existing ones before importing them to the Unreal Editor, in your level design.

Chapter 4: Game Design

4.1 Iteration

Game Development is a complex and time consuming process. What the game a developer had originally imagined will definitely differ from the final result. Most importantly it's a dynamic and organic process. The game will be created, revised, re-envisioned, and then created again based on the feedback the developers receive. This process is known as iteration and it involves three unique stages: formulation, testing, evaluation [21]. These core elements make up the basic progression which the development of a game will follow. These three steps are repeated continuously until a satisfying result is achieved.

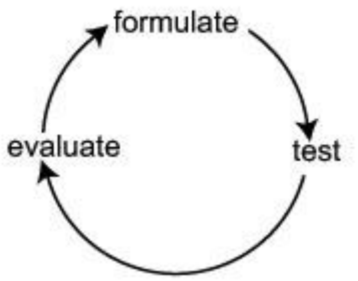


Figure 4.1 – Iterative design steps

- **Formulate**: First a game developer comes up with an idea. This idea is then refined. Next a basic implementation takes place by creating a prototype, such as a small level with simplified functionality and a small amount of art content.

- **Test**: After having implemented a basic prototype the testing phase begins. In a one man project the initial testing steps are made by the developer himself, but as the implementation progresses, testing made by the developer becomes meaningless. The game is to be played from people for a specific purpose (in this case entertainment) and must be tested by people who did not participate in its development in order to have a somewhat objective feedback. Members of the development team are often too familiar with the intricacies of the title to provide the type of criticism desired.

- **Evaluate**: After the testing phase is completed the developer gets the feedback from the testers. After gathering all the available data, concerning the current state of the game design, he analyzes them in order to determine which areas are in need of improvement. Then the cycle continues by re-formulating the project in its current state.

Although the current project was developed (and is still in development) with the use of the iterated design method described above, the description below follows a more linear approach in order to establish clearly the basic steps of the game development proceedure.

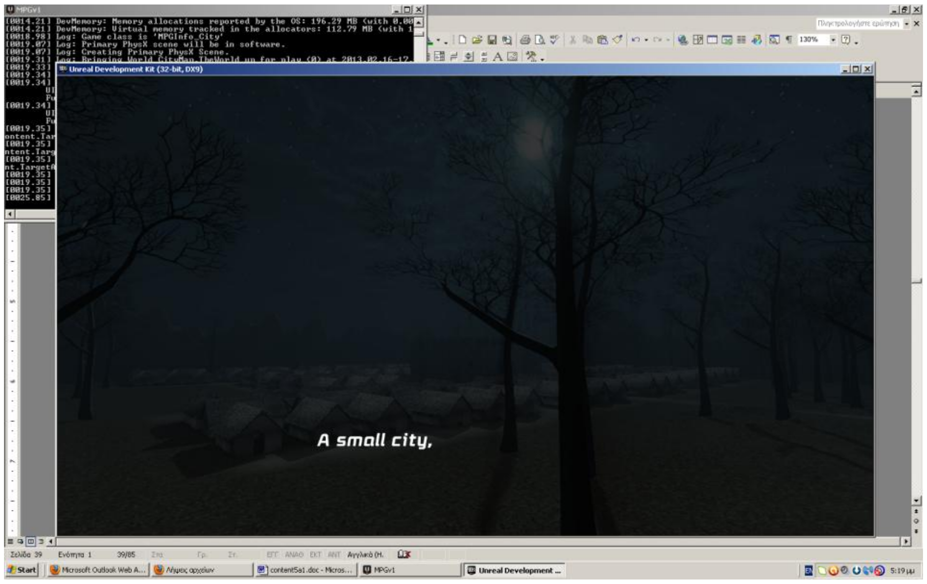
4.2 Storyline Summary

The story takes place in the middle-ages in a small city which is ruled by a vicious monarch. This ruler has deprived people of their dreams and forced them to sleep forever. The only person that is still unaffected by this eternal sleep is the main character of the game. His purpose is to try and wake up the residents of this town from their sleep. There is only one way to do that, to go into people's dreams and fight with their nightmares. As the game progresses the main character realizes that the people won't wake up from their dreams. When he has gained enough experience from fighting people's nightmares, he can go into the castle where he can face the monarch. When he faces the ruler, the game ends with two possible outcomes:

- The character loses the battle and (*spoiler alert*) is sunk at an eternal sleep.
- The character loses the battle and (*spoiler alert*) then he wake's up and realizes that all this has been nothing more than a dream of his.

Specifically, after the human player chooses to start a new game, an introduction video is displayed explaining in short the situation of the world and the human player:

'A small city, once full of life, once full of dreams, but no more. A vicious ruler, he deprived people of their dreams, they now see only nightmares, trapped at an eternal sleep. And you, the only one for some reason still unaffected, the only one who can free people from their dreams. Or not?'



```
[0014.21] DevMemory: Memory allocations reported by the OS: 196.29 MB (with 0.00
[0014.21] DevMemory: Virtual memory tracked in the allocators: 112.79 MB (with 1
[0018.98] Log: Game class is 'MPGInfo_City'
[0019.07] Log: Primary PhysX scene will be in software.
[0019.07] Log: Creating Primary PhysX Scene.
[0019.31] Log: Bringing World CityMap.TheWorld up for play (0) at 2013.02.16-17.
```

Figure 4.2 – One of the first frames of the introduction video

Then, the game begins. The player is initially located in the main city map. This consists of people's houses and the castle in which the monarch dwells. The player is originally located to his house. From there he can move to any house of his choice and enter the person's dream. Currently there are eight dream levels designed for this game, although the city has many more houses. In order for the player not to get lost among all these houses, a minimap was placed showing the current position of the player, his house location and the location of the houses that have active dreams.

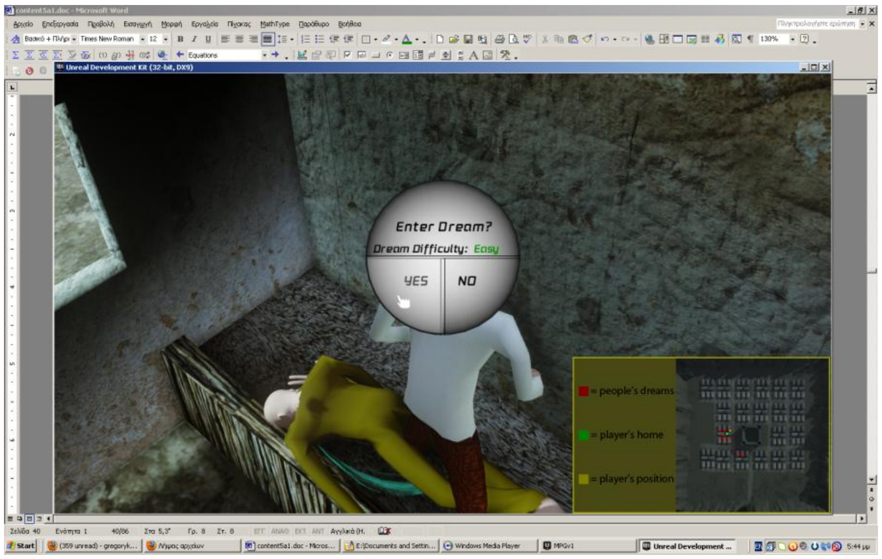


Figure 4.3 – City game play, the player is about to enter a dream level

Most of the gameplay takes place in people's dreams, where the character is confronted with nightmares that he must defeat. These dreams have also a difficulty level, depending mainly on the type of the level. E.g. a level can have AI pawns so it will be considered hard, or the dream level can be a puzzle level so that it will be considered easy.

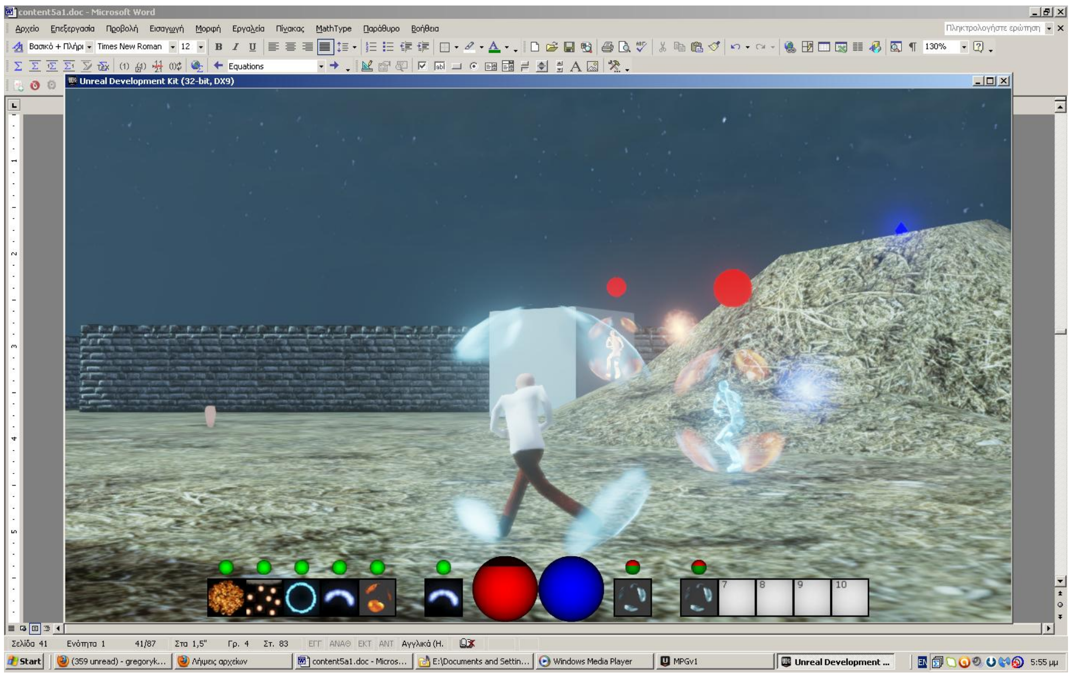

content5a1.doc - Microsoft Word
Unreal Development Kit (32-bit, DX9)
7
8
9
10

Figure 4.4 – Dream level gameplay, the player is fighting nightmares(AI Pawns)

The player will notice that people don't wake up from their dreams. So when he is strong enough he will be able to face the ruler in his dream and face the consequences described above (either loses and sleeps forever, or wakes up and realizes he is in a dream). Details about the levels of the game follow in the next chapter.

4.3 Level Functionality

The Levels of the Game are derived directly from the plot described above. So they will be described first with a short reference on their functionality.

The main map of the game will be that of the city. The city contains:

- The player's house: From there the game begins. There the player can save his progress (see Save/Load Game) so he won't have to start each time the game from scratch.

- People’s houses: The character can go there when he is ready to face other people’s nightmares. A menu appears that asks him if he is ready to enter a dream. If he agrees he goes into a new level containing the dream of the current person.

- The castle: This is the house of the ruler. As soon as the character is powerful enough he can go inside the castle and face the ruler.

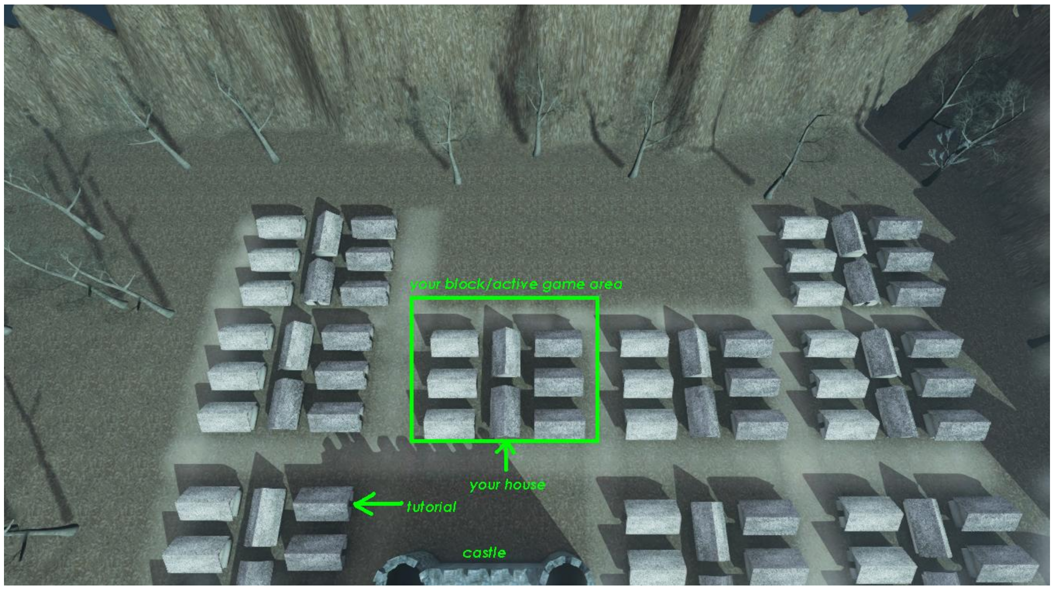


Figure 4.5 – Main City Map

The Dream levels are levels in which a character is transferred to when he decides to go in a person’s dream. When the battle is over at the dream level the character returns at the main city map.

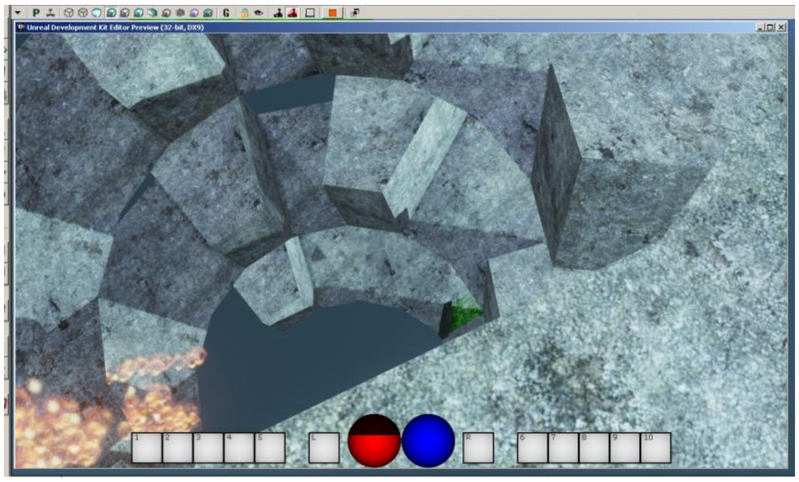


Figure 4.6 – A dream level Map (Stairs)

4.4 Level Design

These levels where created using certain aspects of the Unreal Editor. More specifically the following assets where mainly used:

- BSP brushes and Static Meshes: First the BSP Brushes are used in order to create simple geometry such as the city houses and castle. After applying the materials on the BSP Brushes, they are converted to static meshes so as to be used in the game (Static Meshes can be rendered more effectively than BSP brushes).

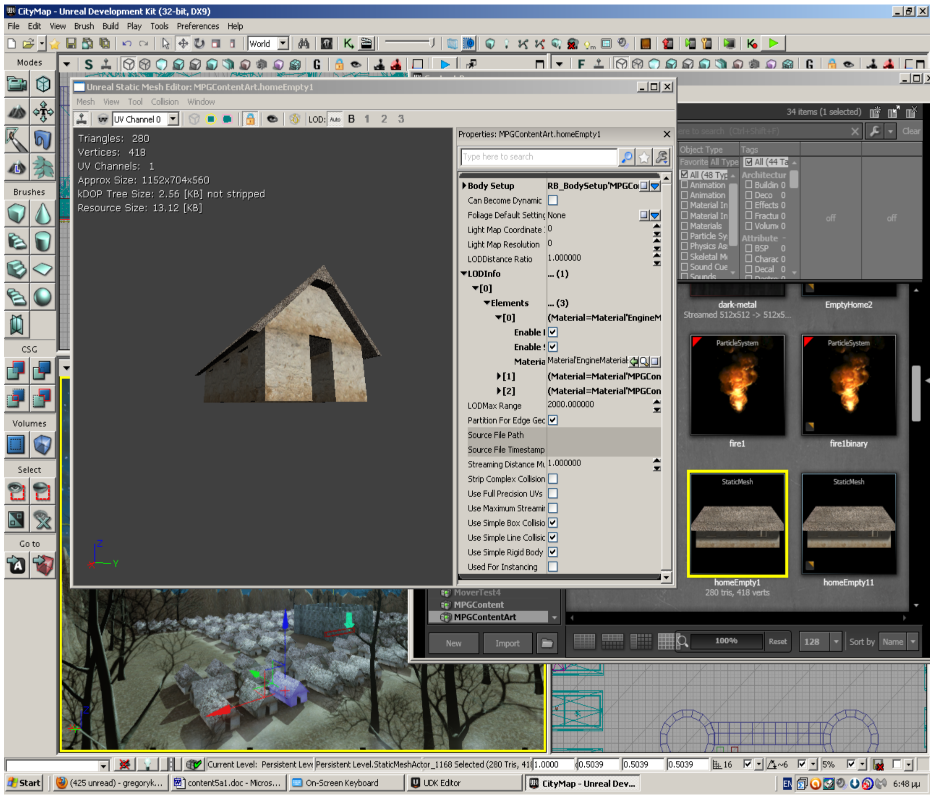

CityMap - Unreal Development Kit (32-bit, DX9)
Unreal Static Mesh Editor: MPGContentArt.homeEmpty1
Triangles: 280
Vertices: 418
UV Channels: 1
Approx Size: 1152x704x560
kDOP Tree Size: 2.56 [KB] not stripped
Resource Size: 13.12 [KB]
Properties: MPGContentArt.homeEmpty1
Body Setup
LODInfo
homeEmpty1
280 tris, 418 verts
homeEmpty11
fire1
fire1binary
dark-metal
EmptyHome2

Figure 4.7 – A House Static Mesh after its conversion from BSP and its material application as shown in the UDK browser

- Terrain: For the Terrain of a level the Terrain Editor tool was used, which can easily and without effort produce complex terrains with height differences and static mesh decorations (such as trees and other models).

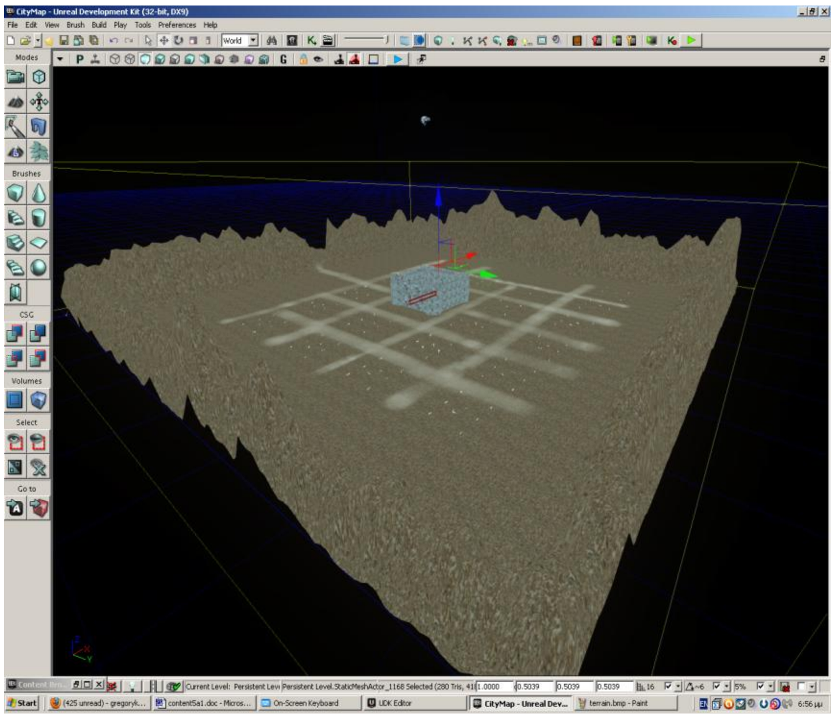


Figure 4.8 – The Main City Map Terrain without Static Mesh and without lightning applied

Using the combination of the elements described above, a level designer can create any type of geometry limited only by imagination and implementation time. Below there are some examples of the levels that where designed for these games.

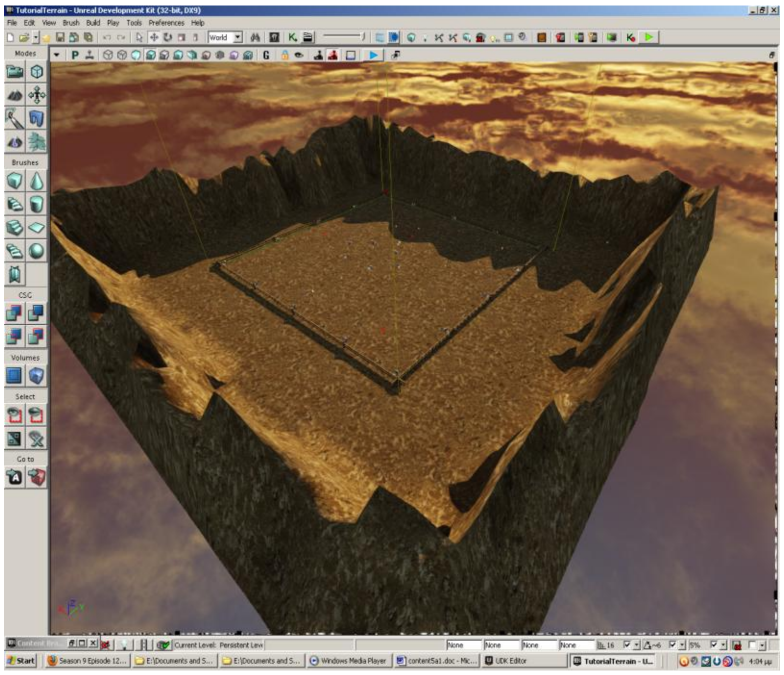


Figure 4.9 – Tutorial Map consists of mountain terrain and some fence static meshes in the middle of the map

Sometimes we want movable object on a level. A static mesh object cannot be moved. For this reason UDK provides as with the interpActor class that allows a static mesh to move inside a level.

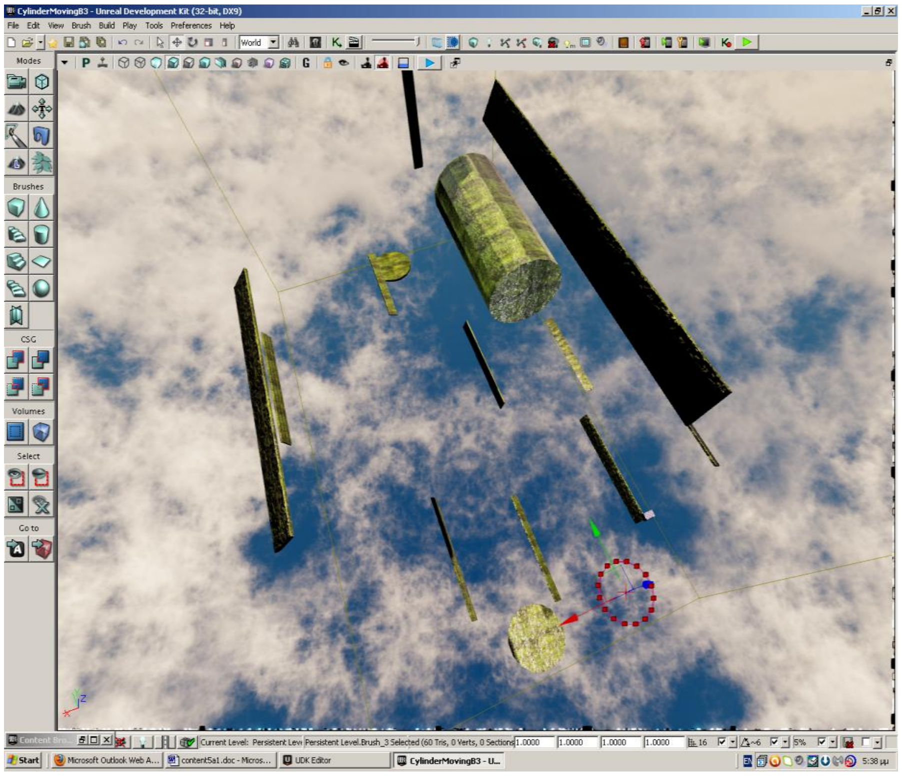

CylinderMoving83 - Unreal Development Kit (32-bit, DX9)
File Edit View Brush Build Play Tools Preferences Help
Modes
Brushes
CSG
Volumes
Select
Go to
Start

Figure 4.10 – Cylinder Dream Level consists mainly of interpolator actors.

4.5 Character Design

The creation of a complex 3D model is definitely not an easy task, especially when that model is a 3D depiction of a human character. In addition we must keep in mind that the poly count of the model should be relatively low, in order to be rendered efficiently by unreal engine 3. The process for creating a character with 3ds max and importing it to Unreal Editor as a skeletal mesh is described below:

Creation of the Bone System: the skeleton of the mesh is composed by using a hierarchy of bones. In 3ds max there are two main tools for creating the skeleton: The bone system and the biped system. The characters of unreal tournament use a combination of both techniques, though, generally, this is not necessary. One can create his own skeleton, then his own animations and import them to Unreal Editor. But in order for the skeleton to be compatible with the animations already existing in the AnimSets of UDK, the following skeleton structure is used (see figure).

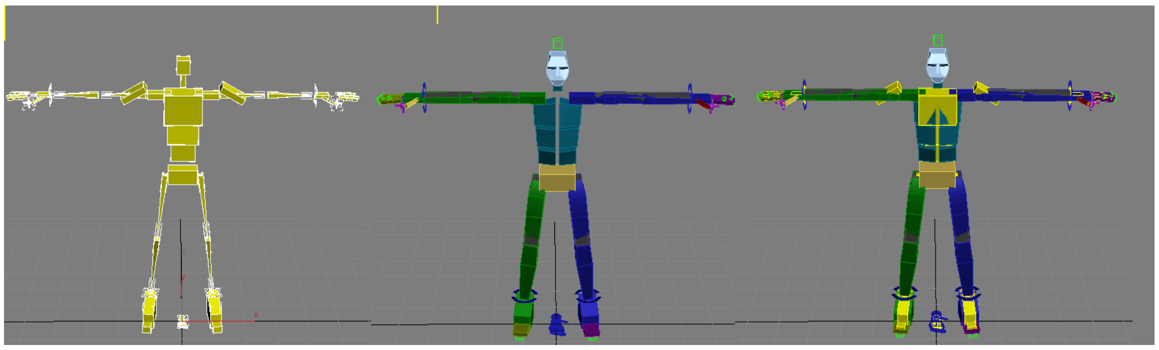


Figure 4.11 – Left: only bone system, Middle: only biped system, Right: Both Systems Combined

Creation of the actual model: This is a time consuming process mainly of an artistic nature. We tried to make the model as simple as possible, so as to minimize the time consumed in this step. The advantage of a simple model is its low poly count, which makes it easier to be rendered.

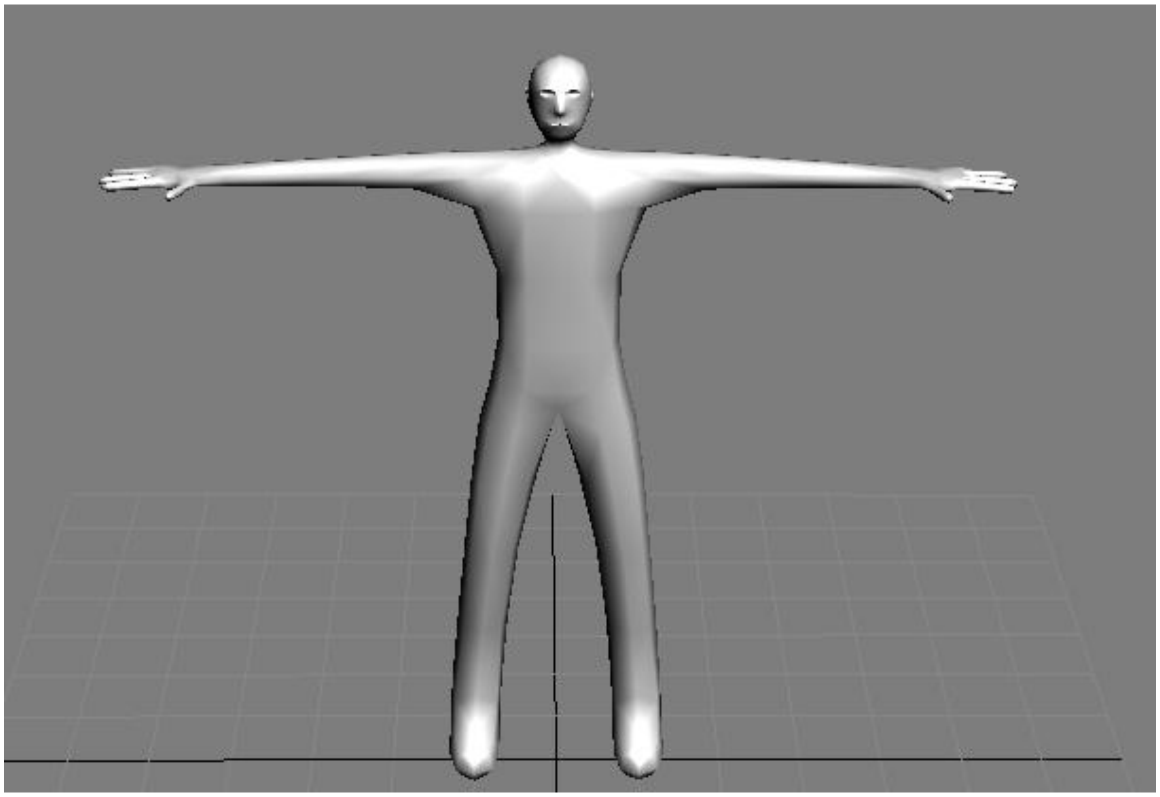


Figure 4.12 – Simple low poly model of a human character.

Apply the skin modifier: When the model is created and is fitted with its skeleton, the so called "rigging process" takes place. The rigging process starts by applying the skin modifier in the mesh. This modifier is responsible for giving weights to vertices of the model for their corresponding bones. If for example the rigging is not right when the hand moves, it might move part of the leg, which would be listed as undesirable behavior.

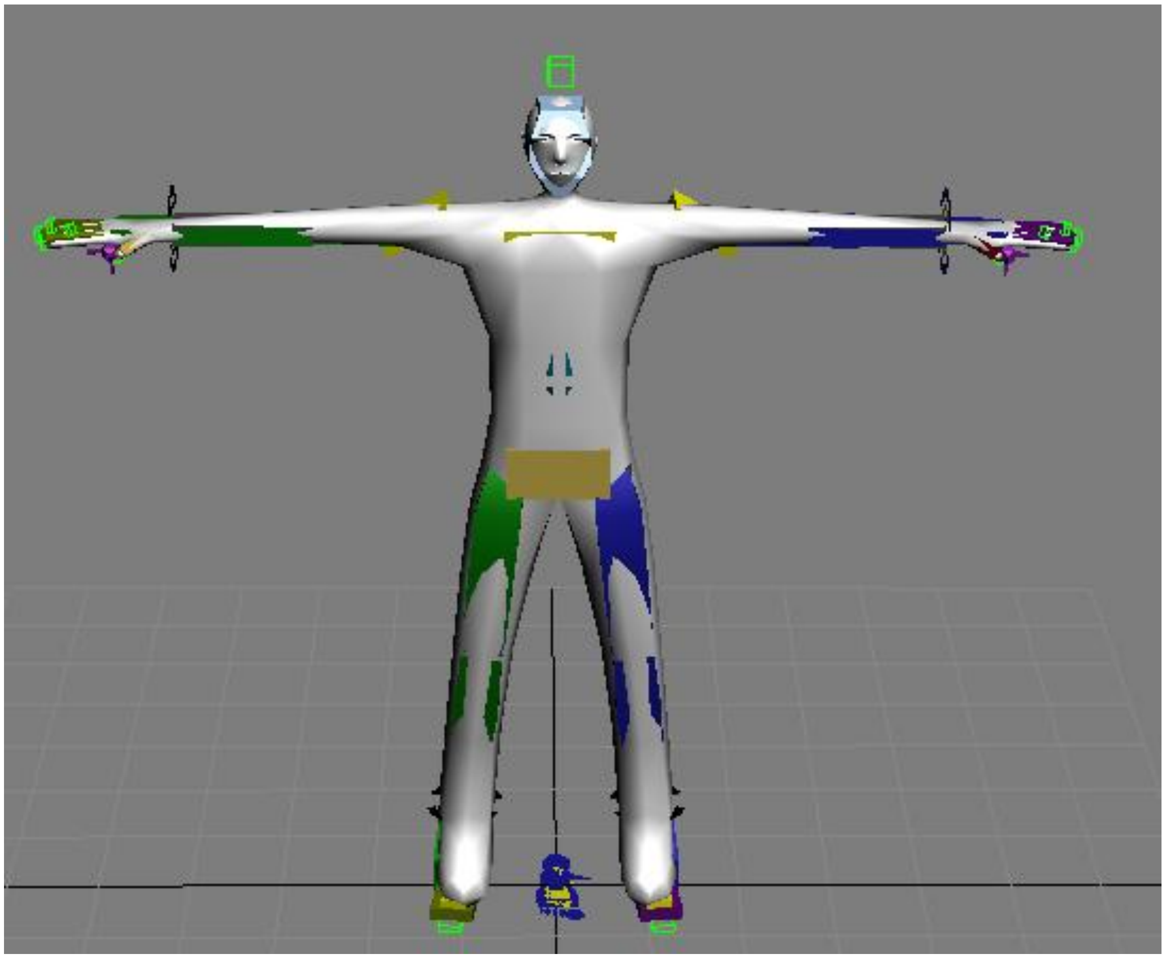
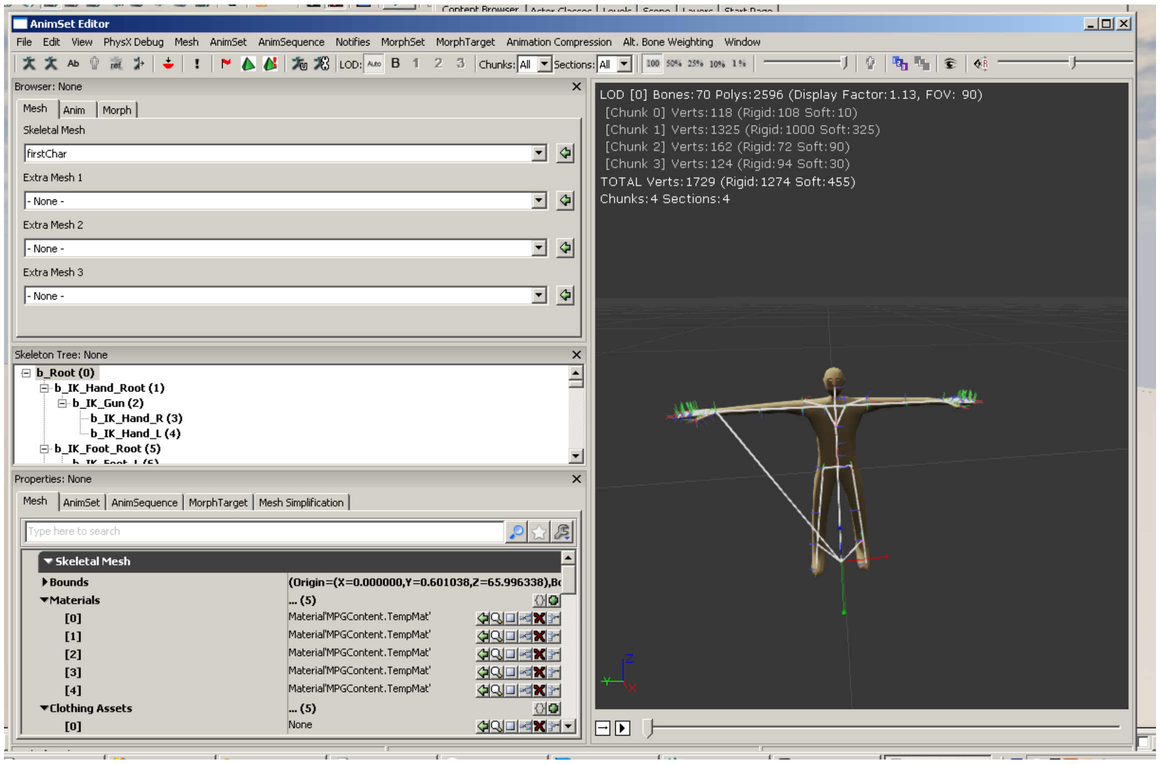


Figure 4.13 – The model attached to the skeleton with the help of the Skin Modifier

Export Skeletal Mesh: At this step the model is completed and can be exported as skeletal mesh. After selecting the model and its bones, the FBX plug-in is used to export the skeletal mesh in the appropriate file (with a '.fbx' extension).

Import Skeletal Mesh: In order to import our skeletal mesh, the AnimSet Editor of UDK is used. We simply choose the fbx file with the skeletal mesh and then import it. If the skeleton we imported is the same with the standard unreal tournament character bones, we can test the already existing animations in our skeletal mesh. In either case we can then go back to 3ds max and create our own animations.



LOD: Auto B 1 2 3 Chunks: All Sections: All

Skeletal Mesh: firstChar
Extra Mesh 1: - None -
Extra Mesh 2: - None -
Extra Mesh 3: - None -

```
b_Root (0)
  b_IK_Hand_Root (1)
    b_IK_Gun (2)
      b_IK_Hand_R (3)
      b_IK_Hand_L (4)
  b_IK_Foot_Root (5)
```

Skeletal Mesh	
Bounds	(Origin=(X=0.000000,Y=0.601038,Z=65.996338),Bo
Materials	... (5)
[0]	Material'MPGContent.TempMat'
[1]	Material'MPGContent.TempMat'
[2]	Material'MPGContent.TempMat'
[3]	Material'MPGContent.TempMat'
[4]	Material'MPGContent.TempMat'
Clothing Assets	... (5)
[0]	None

```
LOD [0] Bones:70 Polys:2596 (Display Factor:1.13, FOV: 90)
 [Chunk 0] Verts:118 (Rigid:108 Soft:10)
 [Chunk 1] Verts:1325 (Rigid:1000 Soft:325)
 [Chunk 2] Verts:162 (Rigid:72 Soft:90)
 [Chunk 3] Verts:124 (Rigid:94 Soft:30)
TOTAL Verts:1729 (Rigid:1274 Soft:455)
Chunks:4 Sections:4
```

Figure 4.14 – The model imported to UDK as a Skeletal Mesh

Create custom animations: Creating a single animation in 3ds max is an extremely time consuming procedure. To build realistic animations from scratch is a process that requires a lot of time and effort.

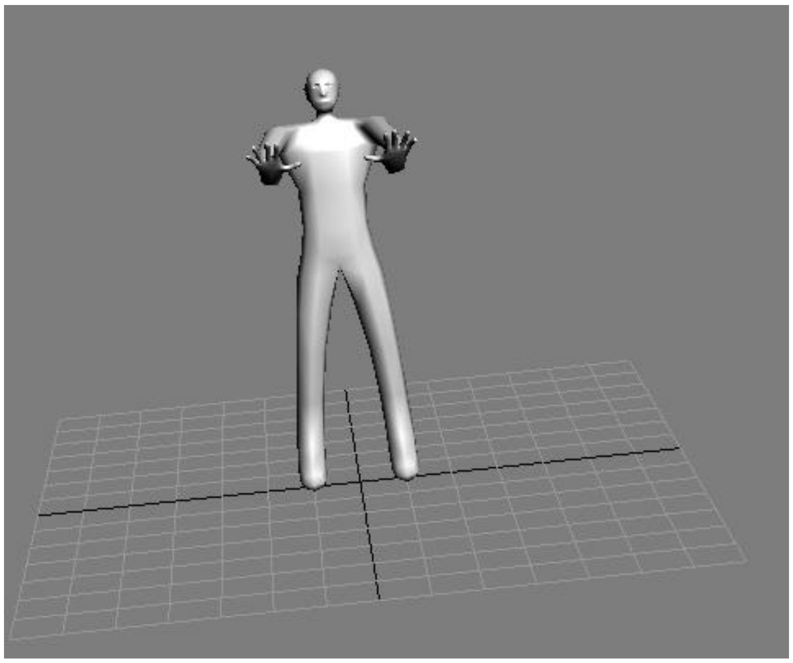
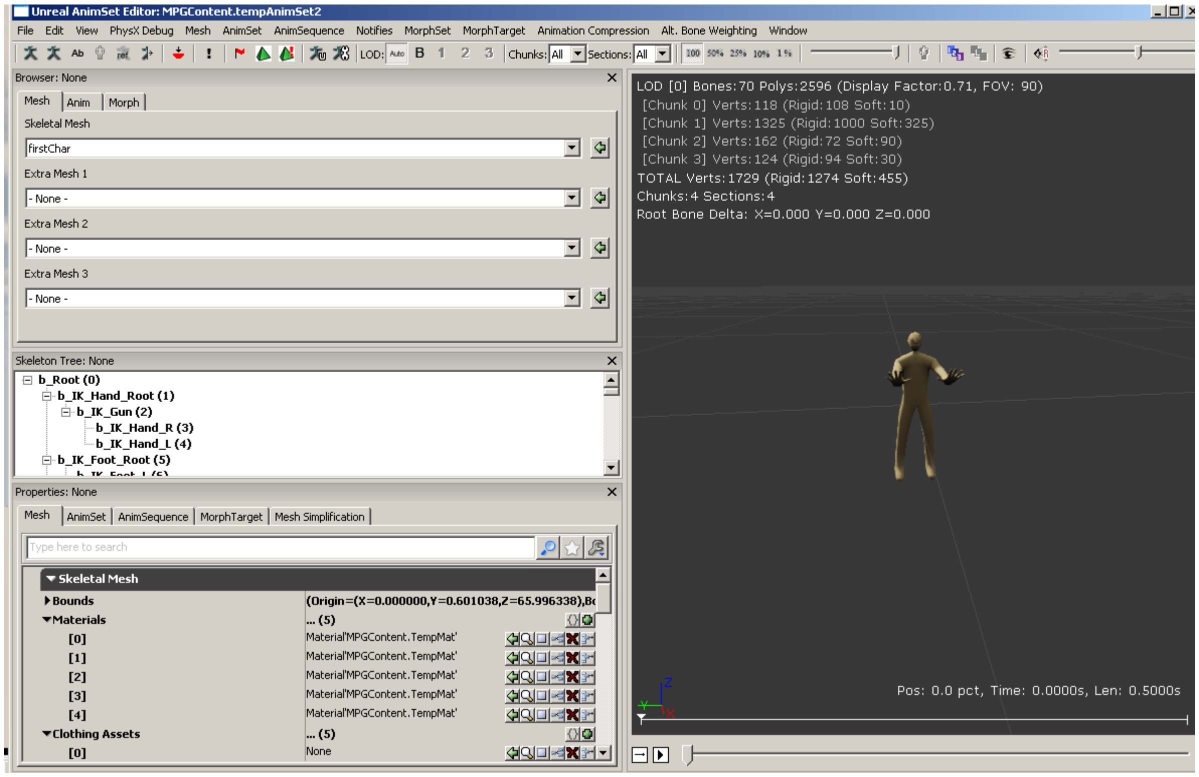


Figure 4.15 – Create Animations with 3ds max

Export custom Animations: After creating an animation and selecting the current skeleton system, the data are exported again using the fbx plugin and some additional parameters (bake animation, frames to be exported, etc.).

Import custom Animations: Again the AnimSet Editor is used and the .fbx file of the animation is imported to an AnimSet of the developer's choice. Then the AnimSet Editor can be used in order to view the imported animation on our skeletal meshes.


Unreal AnimSet Editor: MPGContent.tempAnimSet2
File Edit View PhysX Debug Mesh AnimSet AnimSequence Notifies MorphSet MorphTarget Animation Compression Alt. Bone Weighting Window
Mesh Anim Morph
Mesh AnimSet AnimSequence MorphTarget Mesh Simplification
Type here to search

Browser: None

Skeletal Mesh: firstChar
Extra Mesh 1: - None -
Extra Mesh 2: - None -
Extra Mesh 3: - None -

LOD: Auto · Chunks: All · Sections: All

Skeleton Tree: None

```
b_Root (0)
  b_IK_Hand_Root (1)
    b_IK_Gun (2)
      b_IK_Hand_R (3)
      b_IK_Hand_L (4)
  b_IK_Foot_Root (5)
```

Properties: None

```
Skeletal Mesh
  Bounds            (Origin=(X=0.000000,Y=0.601038,Z=65.996338),Bo
  Materials         ... (5)
    [0]             Material'MPGContent.TempMat'
    [1]             Material'MPGContent.TempMat'
    [2]             Material'MPGContent.TempMat'
    [3]             Material'MPGContent.TempMat'
    [4]             Material'MPGContent.TempMat'
  Clothing Assets   ... (5)
    [0]             None
```

```
LOD [0] Bones:70 Polys:2596 (Display Factor:0.71, FOV: 90)
 [Chunk 0] Verts:118 (Rigid:108 Soft:10)
 [Chunk 1] Verts:1325 (Rigid:1000 Soft:325)
 [Chunk 2] Verts:162 (Rigid:72 Soft:90)
 [Chunk 3] Verts:124 (Rigid:94 Soft:30)
TOTAL Verts:1729 (Rigid:1274 Soft:455)
Chunks:4 Sections:4
Root Bone Delta: X=0.000 Y=0.000 Z=0.000
```

```
Pos: 0.0 pct, Time: 0.0000s, Len: 0.5000s
```

Figure 4.16 – Import Animations into UDK

Creating the AnimTree: AnimTree is a way of connecting animation sequences for a skeletal mesh (or part of a skeletal mesh) with input events that are usually affected by a specific Physics asset. For example let us assume that the character presses the jump button. Then the Physics asset changes from PHYS_Walking to PHYS_Falling and the animation changes to the sequence of animations played when the character jumps (Up, Down, Pre-Land, Land). That way the basic animations are implemented for our character movement. Those include the animations for walking, jumping and being idle.

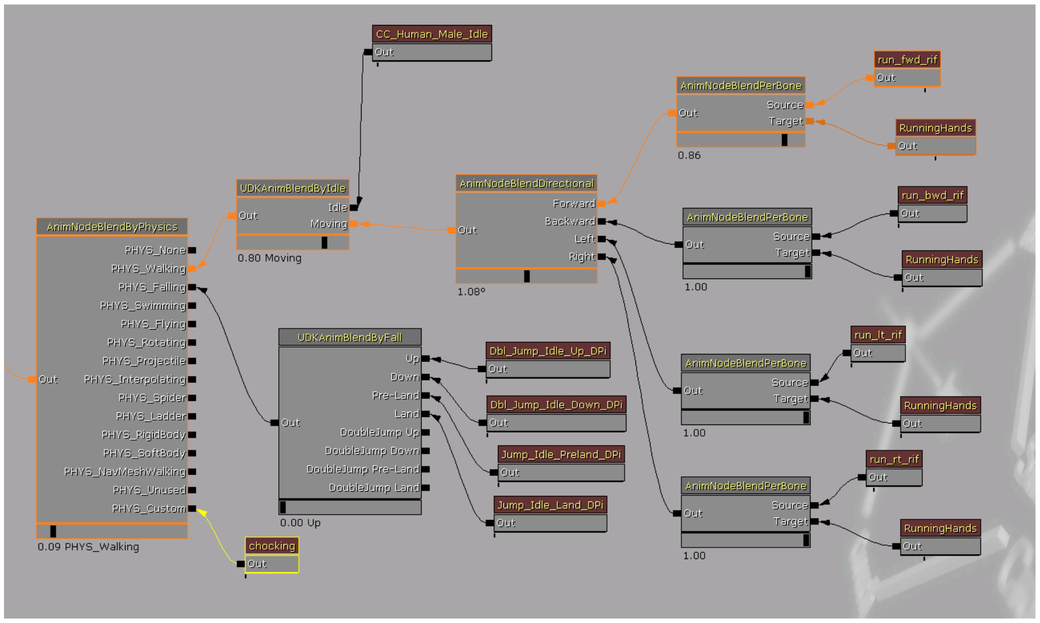


Figure 4.17 – Anim Tree

4.6 User Interface

In this section brief descriptions of the User Interface screens of the game are given, along with the purpose they serve. (For implementation of UI see next chapter)

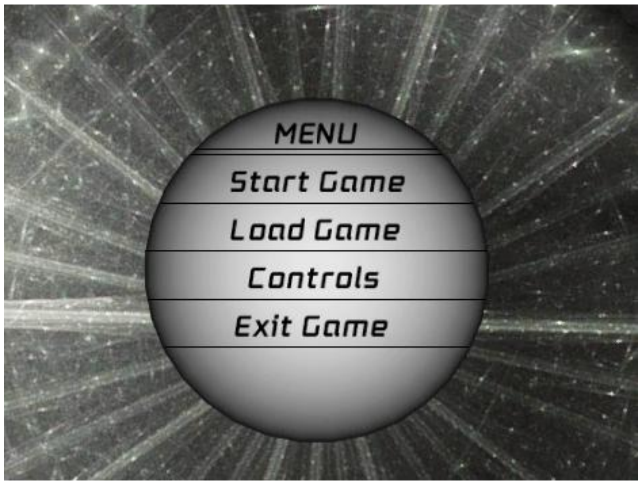


Figure 4.18 – Main Menu Screen

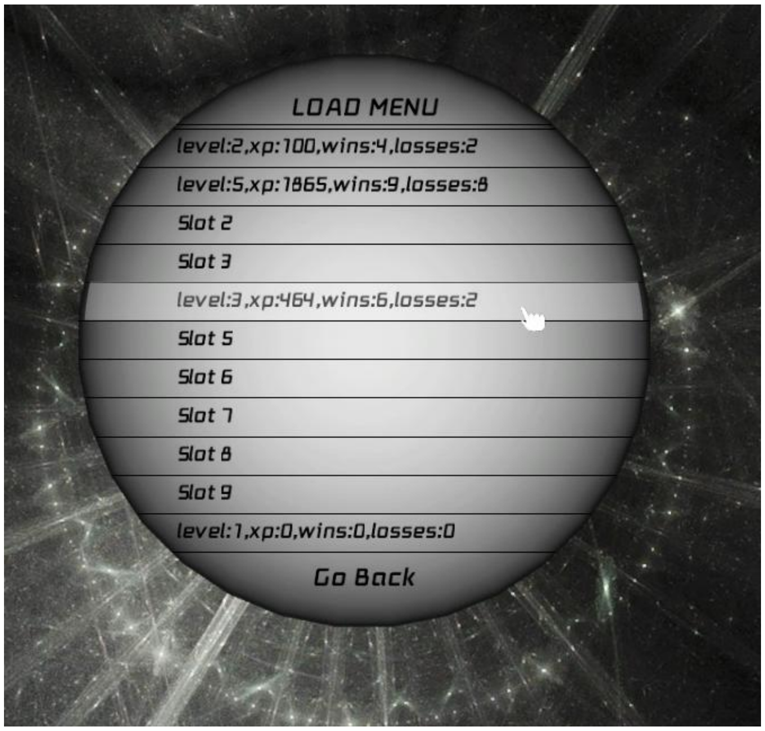
Main Menu Screen: This is the main screen the user sees as soon as he enters the game. From here the following options are available:

- Start Game: When pressed the game begins and the user is transferred to the main city map.

- Load Game: Brings up the load menu screen that contains the saved games of the user. From there a user can load a previously saved game.

- Controls: Brings up the Controls screen. This screen informs the user of the main game controls.

- Exit Game: When this option is pressed the game exits immediately to the windows screen.



LOAD MENU
level:2,xp:100,wins:4,losses:2
level:5,xp:1865,wins:9,losses:8
Slot 2
Slot 3
level:3,xp:464,wins:6,losses:2
Slot 5
Slot 6
Slot 7
Slot 8
Slot 9
level:1,xp:0,wins:0,losses:0
Go Back

Figure 4.19 – Load Menu Screen

Load Menu Screen: Here a user chooses a slot to load a game from. If this slot is empty nothing happens. If it contains a saved game, the user goes to the main city map where his game will continue with all his saved progress. The user can also press the back button at any time, in order to go to the previous menu screen.

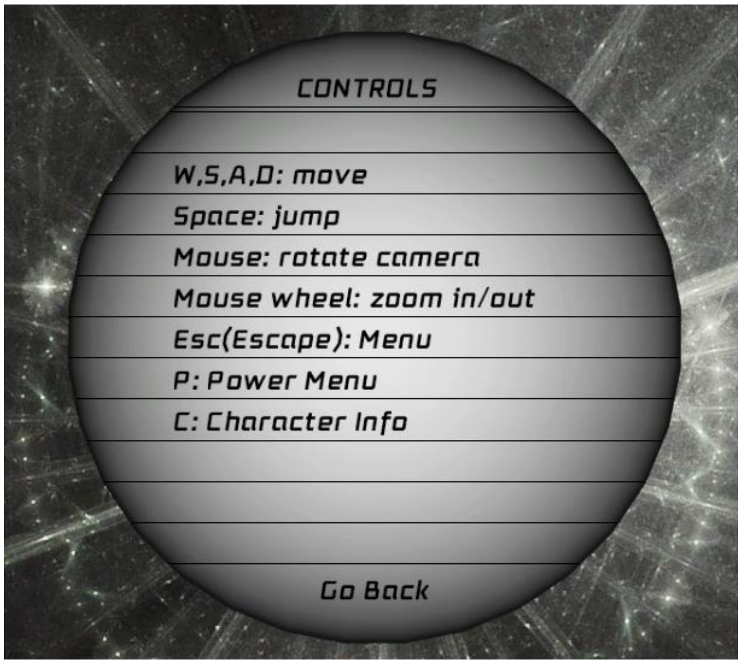


Figure 4.20 – Control Menu Screen

Controls Menu Screen: As mentioned above, this screen informs the user of the main game controls (input buttons to be pressed in order for an event to occur).

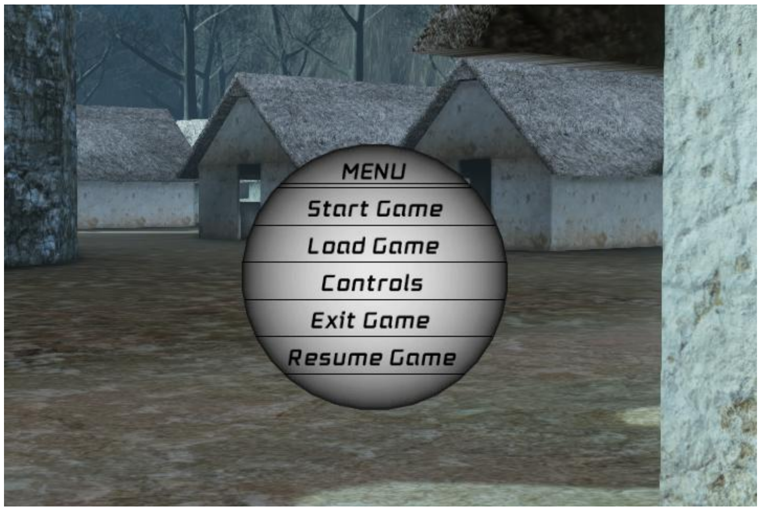


Figure 4.21 – Pause Menu Screen

Pause Menu Screen: Once in game a player can at any time press escape to view the pause menu screen. This screen has the same options as the main menu screen with one addition in the end: The resume game option. By pressing that the user can resume the game and continue playing.

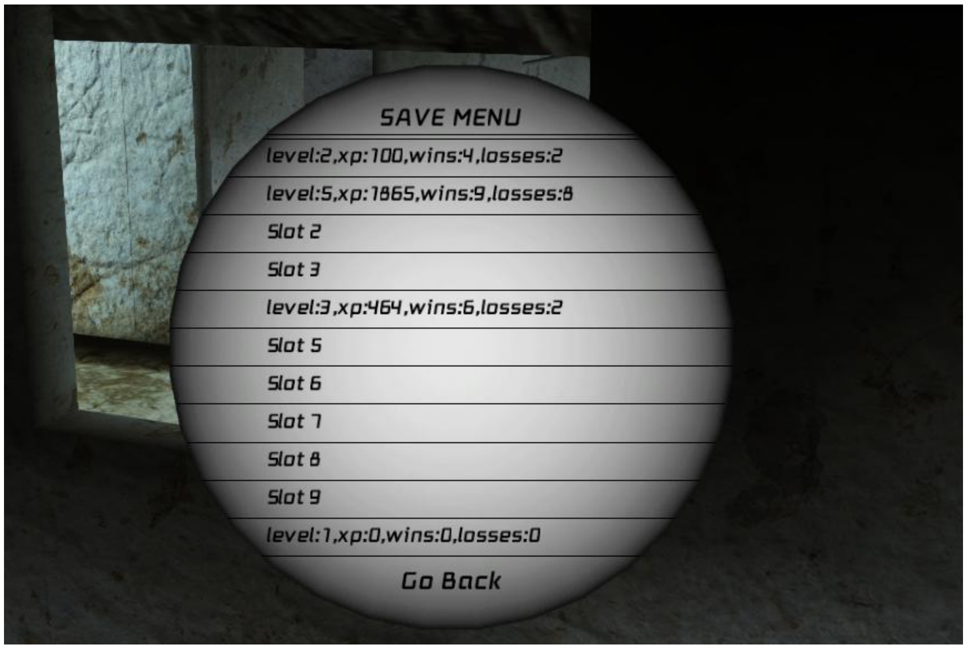
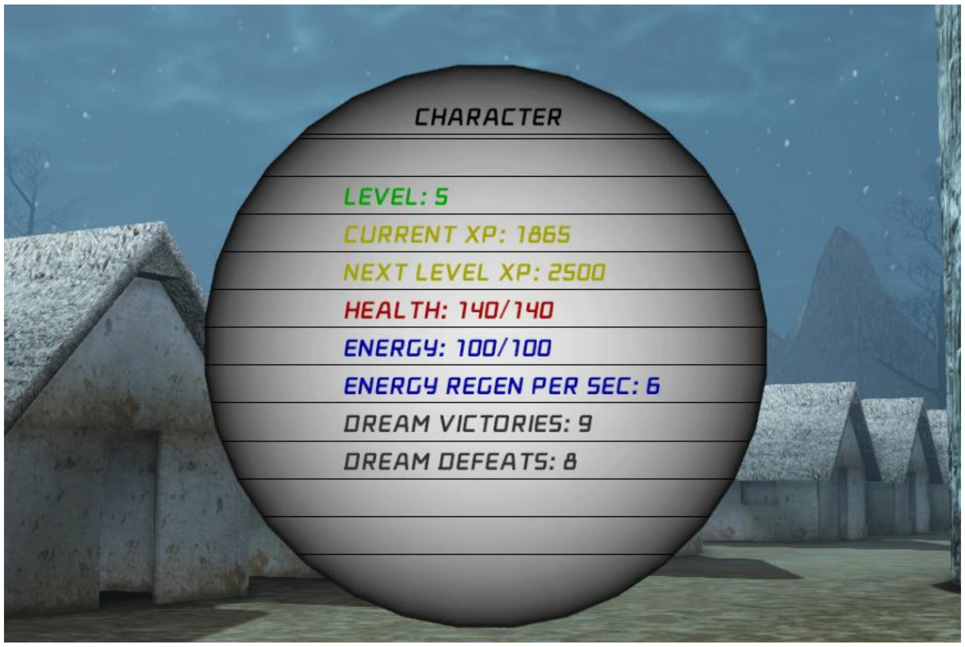

SAVE MENU
level:2,xp:100,wins:4,losses:2
level:5,xp:1865,wins:9,losses:8
Slot 2
Slot 3
level:3,xp:464,wins:6,losses:2
Slot 5
Slot 6
Slot 7
Slot 8
Slot 9
level:1,xp:0,wins:0,losses:0
Go Back

Figure 4.22 – Save Menu Screen

Save Menu Screen: This screen appears only in one spot: the player's home in the main city map. This is the only place where a user can save his progress. Its interface is identical to the Load Menu Screen, although the functionality is different. When a slot is pressed, the game progress will be saved at that chosen slot. Game progress is determined by some variables, such as Level and XP, which will be covered in the gameplay section. Only the necessary variables are saved (serialized on the hard drive).



LEVEL: 5
CURRENT XP: 1865
NEXT LEVEL XP: 2500
HEALTH: 140/140
ENERGY: 100/100
ENERGY REGEN PER SEC: 6
DREAM VICTORIES: 9
DREAM DEFEATS: 8

Figure 4.23 – Character Menu Screen

Character Info Screen: When the user presses the C button the Character Info Screen is brought up. This Screen shows vital gameplay info for the current character, such as Level, Current XP, Next Level XP, Health and Energy, which will be explained in the gameplay section. It also shows statistic information less vital for the gameplay such as Dream Victories, and Dream Defeats.

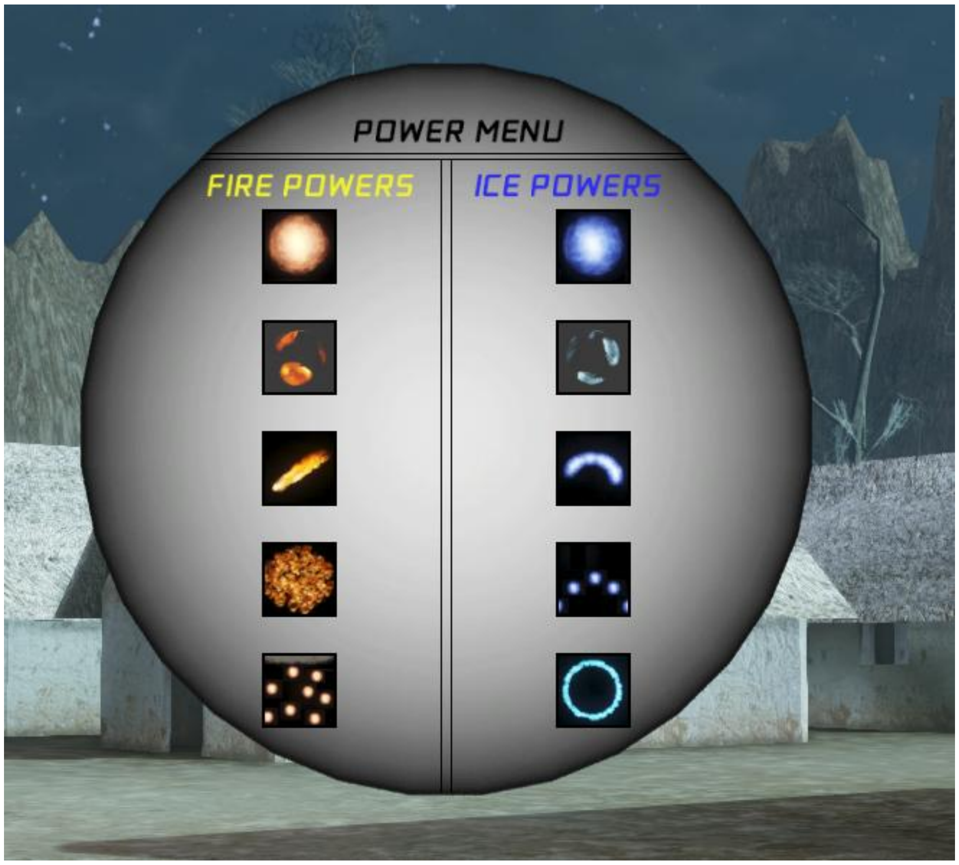


Figure 4.24 – Power Menu Screen

Power Screen: Brought up when the P button is pressed. The functionality of this screen along with the gameplay user interface are crucial elements for the Power System described in the gameplay section. This screen shows the Powers (magic skills) a player can unleash to his opponents and the effects its power has.

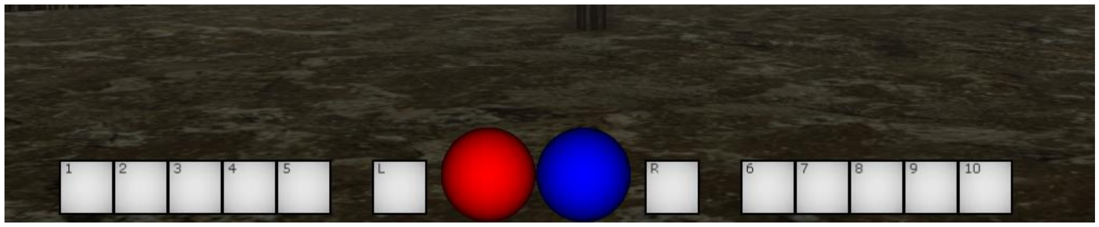


Figure 4.25 - Combat Interface

Gameplay user interface (combat interface): This interface is available on the dream levels, because only in dreams a player can engage into combat and unleash powers. In the Center there are two circle bars. The red one represents the player's health and the blue one shows the player's energy. Adjacent to the bars there are two boxes. The left box indicates which Power is equipped in the left hand and the right box indicates which Power is equipped in the right hand. Finally there are the power quickslots. On the left there are five Power quickslots for the left hand corresponding to the numbers one through five and on the right there are five Power quickslots for the right hand corresponding to the numbers six through ten. When a number from one to ten is pressed the selected Power (if any) will be equipped at the corresponding hand (for two-handed powers, the power will be equipped on both hands).

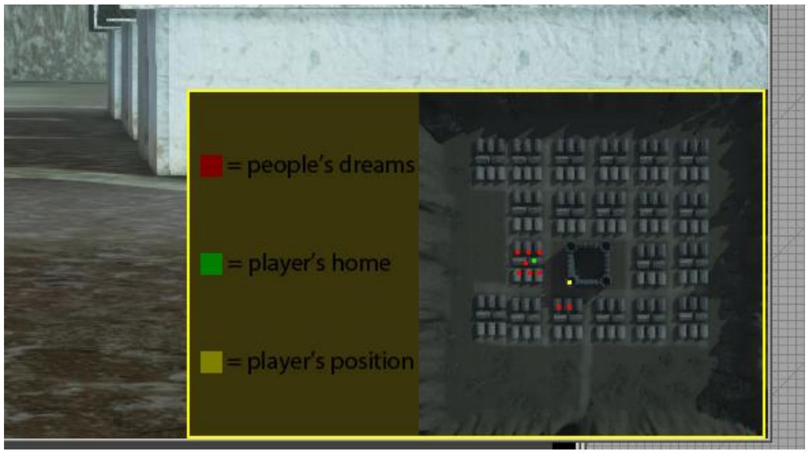


Figure 4.26 – City Minimap

City Minimap: Informs the player about his current position and the dream levels that are available in the city map.

4.7 Gameplay System

4.7.1 Basic gameplay elements

As mentioned above, the main gameplay takes place in the dreams level, where the player fights with people's nightmares. Before explaining how the combat works through the use of the Power System, let's take a look at some basic concepts that are commonly used in RPG games.

Character level: Maybe one of the most common characteristics in RPGs. When the game starts the character is assigned to level one. As the game progresses and the Character gains more experience (XP points) by successfully competing with nightmares in dreams, the character becomes more powerful. “More powerful” means that the character has more health, faster energy regeneration and stronger Powers.

XP: Experience Points are used to determine when a character advances by a level. When victorious in a dream, a character gains XP. When he gains enough XP he goes up a level. On the other hand, if a character is defeated in a dream he loses XP buthis level doesn't decrease. The level serves as a guarantee, a threshold of XP that cannot be lower from a fixed value. The relation that connects Level with XP is as follows:

Health: The Health attribute shows how many and how strong hits a character can endure. The moment that Health reaches zero the character dies.

Energy: Energy is consumed when the character unleashes a Power. If the Energy is too low, the Player cannot unleash any more Powers and must first wait for the Energy to regenerate.

4.7.2 The Power System

The Power System is essentially the heart of the gameplay.

What is a Power? A Power is essentially a magic skill that the character unleashes in order to defeat his enemies. In all respects, it acts as a Weapon but with no physical presence, meaning that in order for a Power to be unleashed it must be equipped (as described in the combat user interface) but is generated magically with no use of any kind of matter(just energy), with a simple movement(animation) of the left and/or right hand.

A player can use a magic skill only in dreams and not in the City Map. The Powers are classified in two categories: Fire Powers and Ice Powers. The Fire Powers inflict Fire Damage to an opponent and the Ice Powers inflict Ice damage. A Power can have a number of attributes such us:

DamageType: as mentioned above there are two types of Damage Fire and Ice.

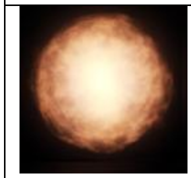
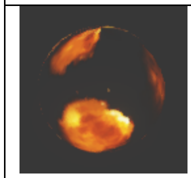
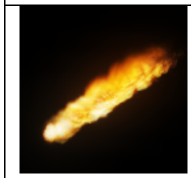
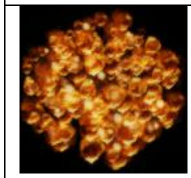
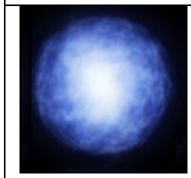
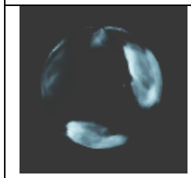
Damage: if offensive this indicates the amount of damage dealt when a Power hits the enemy. If defensive this represents the amount of damage absorbed.

Duration: Many Powers have effects which last for a period of time. Others have instantaneous effects (duration=0).

Energy Drained: Every Power in order to be unleashed consumes energy.

One/Two Handed: There are Powers that require one hand, and powers that require two hands.

Below there is a short description of each Power currently active in the game.

Icon	*Name*	*Info*	*Description*	*Cool down*	*Energy*
	Fireball	One handed	Shoots a ball of fire dealing 12 fire damage (may also inflict fire damage to nearby enemies).	0.5	3
	Fire Shield	One handed	Creates a shield of fire lasting 5 seconds that absorbs 25 points of fire damage	10	3
	Fire Ray	One handed	Shoots a short range fire ray that deals 3 fire damage every 0.2 seconds	No	3
	Fire Earth	One handed	Creates an area covered in fire for 4 seconds dealing 3 points of fire damage to all enemies inside the area every 0.25 seconds	10	12
	Armag eddon	One handed	Creates a rain of fireballs that fall for the sky for 4 seconds each dealing 12 points of fire damage to each enemy hit. The fireballs appear every 0.1 seconds	15	18
	Iceball	One handed	Shoots a ball of ice dealing 8 ice damage and slowing down the enemy to ¼ of his speed for 1 second	0.7	3
	Ice Shield	One handed	Creates a shield of ice lasting 5 seconds that absorbs 25 points of ice damage	10	3
	Ice Wave	One handed	Shoots a small ball-wave of ice dealing 12 ice damage and slowing down the enemy to the ¼	2	5

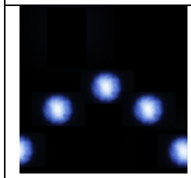
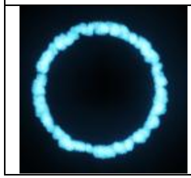
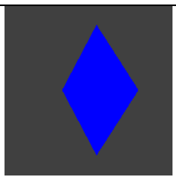
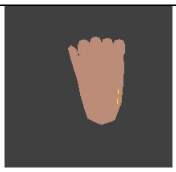
			of his speed for 1.5 seconds		
	Mass Ice Ball	One handed	Shoots 5 balls of ice each dealing 8 ice damage and slowing down the enemy to ¼ of his speed for 1 second	2.5	8
	Ice Age	One handed	Slows all the enemies in an area to the 1/16 of their speed for 1.2 seconds	15	18

Table 4.1 – Power Description.

4.6.3 The Pickup System

Pickups are items on a level that the character can gather. As soon as the player steps on a pickup item, the effect of that pickup is triggered if some condition is met (depending on the Pickup item, some don't have any conditions). These items were added into the game in order to contribute to the interactivity of the player with its surrounding world.

Below there is a short description of each Pickup item.

Icon	*Name*	*Description*
	Health Pickup	If the player has full health, the pickup is not triggered. Elsewise it cures 15 points of damage.
	Energy Pickup	If the player has full energy, the pickup is not triggered. Elsewise it restores 30 points of dream energy.
	Fist Pickup	Shoots 16 fists that knock back and damage enemies hit.
	Clock Pickup	Slows down the movement of enemy players for 5 seconds.

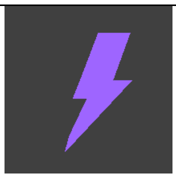
	Lightning Pickup	Spawns 8 bolts of lightning that damage nearby enemies.

Table 4.2 – Pickup Description.

Chapter 5: Implementation

5.1 Classes and functionality

The game consists of sixty-five classes. In this section the most crucial of them (from a functionality aspect) will be presented. These classes define the main functionality of the game. In many levels there are also level specific rules and events that are defined through the kismet tool of Unreal Engine. Below there is a general scheme of the implementation, and an overview of the class types that will be examined in detail in this chapter.

Class types:

- GameInfo Classes: define the rules of a game.
- User Interface Classes: responsible for in-game menus
- Controller Classes: define the mind of the players (AI and Human)
- Pawn Classes: define the physical presence (body) of the player
- Power Classes: magic Skills use by players (Human and some AI)
- Projectile Classes: objects(like fireballs) used by some Powers
- Pickup Classes: define Pickup Items that are found in game

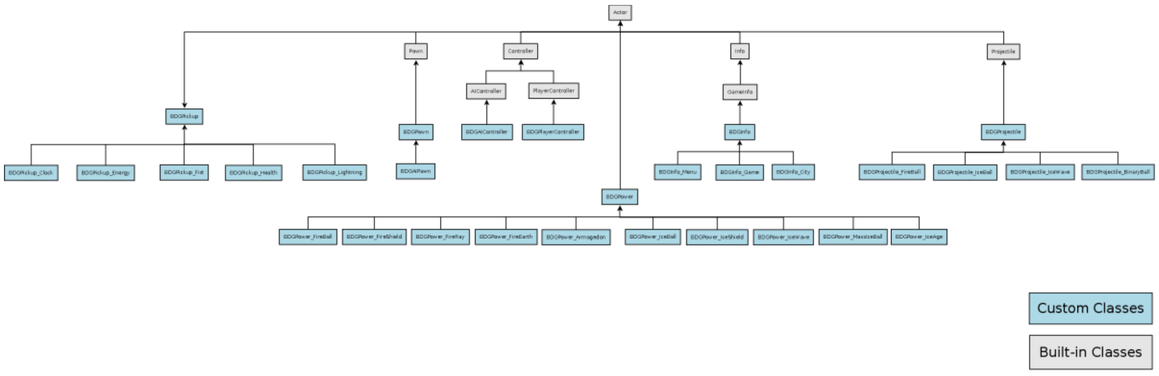


Figure 5.1 – Main game Classes Hierarchy

5.2 GameType, Game info Hierarchy

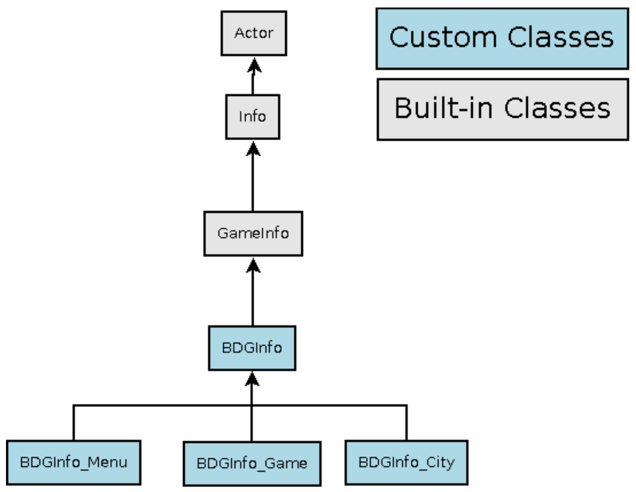


Figure 5.2 – GameInfo Classes Hierarchy

The gametype is the heart of the game. It determines the rules of the game and the conditions under which the game progresses or ends. Clearly, this is entirely game-specific. The gametype is also responsible for telling the engine which classes to use for PlayerControllers, Pawns, the HUD, etc. The example implementation of the gametype will simply specify these classes in the default properties and leave the rest of the implementation up to the programmer.

The `GameInfo` class is the base class for all gametypes in Unreal. Each time a map is loaded, a new instance of the appropriate GameInfo class is created and assigned to be gametype. So in the current game there are typically three game types.

The Game Type for the Main Menu Level: This is not actually a game level, since it has more to do with the user interface of the game, but for unreal engine in all respects it is considered to be a distinct level that ends as soon as the player starts or loads a game.

The Game Type for the City Menu Level: The City Map Level also has its own rules and restrictions (the player cannot use Powers in the city).

The Game Type for Dream Levels: The biggest part of the gameplay takes place in the dreams. This GameType determines when a player is victorious or when he is defeated in a dream.

With the above said, the GameInfo hierarchy will be now defined.

As seen on the Figure 5.2, there is a main class `BDGInfo` that extends the basic `GameInfo` class of UDK Engine. This class mostly contains declarations of variables used in its subclasses such as the current `PlayerController`. It also saves and loads variables that need to be preserved on level transition into a temporary file.

```
class BDGInfo extends GameInfo;
```

```
var BDGPlayerController SinglePlayer;

var BDGEnemyPool EnemyPool;

var int GameXP;

var PathNode PathNodes[256];
var byte NumOfNodes;
var byte NextNode;
var BDG2DimByteArray Paths;
var BDG2DimFloatArray Weights;

//called when player controller is confirmed
event PostLogin( PlayerController NewPlayer ){
      super.PostLogin(NewPlayer);
      SinglePlayer=BDGPlayerController(NewPlayer);
      BDGPawn(SinglePlayer.Pawn).InitPlayerPawn();

      //load//////////////////////////////
      BDGPawn(SinglePlayer.Pawn).Load("temp");
}

//called upon level transitions
//(e.g. a player must be send in another level)
function SendPlayer( PlayerController aPlayer, string URL )
{
      //save//////////////////////////////
      if(BDGPawn(aPlayer.Pawn).Save("temp"))
            super.SendPlayer(aPlayer,URL);
}

defaultproperties
{
   PlayerControllerClass=class'BDGv1.BDGPlayerController'

   DefaultPawnClass=class'BDGv1.BDGPawn'

   HUDType=class'BDGv1.BDGHUD'

   bDelayedStart=false
}
```

The `BDGInfo_Menu` is the GameInfo class used in the main menu screen. After the user logs into the game, it is responsible for disabling the current player Pawn and enabling the mouse input through the player's HUD object. It is also responsible for playing the background music of the intro level with the use of sound cues.

```
class BDGInfo_Menu extends BDGInfo;

//disables player movement, gives command to HUD to load Menu screen
//and starts playing the main menu sound
event PostLogin( PlayerController NewPlayer ){
      super.PostLogin(NewPlayer);
```

```
	SinglePlayer.GotoState('PlayerInactive');
	BDGGHUD(SinglePlayer.myHUD).UIType=102;
	PlaySound(SoundCue'BDGGContent.BDGMenuSoundCue',true,true,true,,)
;

}

function SendPlayer( PlayerController aPlayer, string URL )
{
	//change location before saving!!!!!!!
	BDGGPawn(aPlayer.Pawn).StartLocation=class'BDGPawn'.default.Start
Location;
	super.SendPlayer(aPlayer,URL);
}
```

The `BDGInfo City` class handles the behavior of the Pawn in the main city map. It is responsible for deactivating the user while in loading city screen and re-enabling him when loading is completed. It also searches for an appropriate starting location for the current player by examining nearby nodes (navigation points).

```
class BDGInfo_City extends BDGInfo;

event PostLogin( PlayerController NewPlayer ){
	local NavigationPoint N;

	super.PostLogin(NewPlayer);

	//change starting location
foreach Worldinfo.RadiusNavigationPoints(
class'NavigationPoint',N,BDGPawn(NewPlayer.Pawn).StartLocation,
256){
		break;
	}
	NewPlayer.Pawn.SetLocation(N.Location);

	//make player inactive while loading
	BDGHUD(SinglePlayer.myHUD).UIType=120;
	SinglePlayer.GotoState('PlayerInactive');
	SinglePlayer.UnPauseGame();
	SetTimer(7,false,'FinishedCityLoading');
}

function FinishedCityLoading(){
	SinglePlayer.GotoState('PlayerWalking');
	BDGGHUD(SinglePlayer.myHUD).UIType=0;
}
```

The `BDGInfo_Game` is the class that is used in all dream levels. The EndGame defines when a player has lost (`health <= 0`) or won (all enemies defeated). The `BDGInfo_Game` also computes the optimal paths (Paths) of the Node Network, a crucial functionality for the A.I., that will be described in the A.I. Section. Below the non-AI relative functionality of the class is displayed.

```
function EndGame(PlayerReplicationInfo Winner,string Reason){
        local BDGHUD tempHUD;

        if(SinglePlayer.Pawn.Health<=0){
                SetTimer( 4,false,nameof(Defeat) );
                GameXP=BDGPawn(SinglePlayer.Pawn).Defeat(GameXP/2);
        }else if(EnemyPool.IsEmpty()){
                SetTimer( 4,false,nameof(Victory) );
                BDGPawn(SinglePlayer.Pawn).Victory(GameXP);
        }
}

//display UI message dialog when defeated
function Defeat(){
        SinglePlayer.GotoState('LevelEnded');
        BDGHUD(SinglePlayer.myHUD).UIType=13;
}

//display UI message dialog when vitorious
function Victory(){
        SinglePlayer.GotoState('LevelEnded');
        BDGHUD(SinglePlayer.myHUD).UIType=12;
}
```

5.3 User Interface

The functionality of the game's user interface is defined in the `BDGHUD` class that extends the Engine `HUD` class. This class defines the event `PostRenderer()` which is called every Tick and draws the appropriate menus, screens, gameplay UIs each time. What will be displayed during each tick in the screen is dependent from the UIType variable, which is essentially an integer id that defines different type of menu screens and UIs. It's main event is the `PostRender()` event which is called every Tick. The programmer simply overrides this function and draws the appropriate UI each time (defined by the `UIType` variable).

```
event PostRender()
{
        Super.PostRender();

        if(BDGGPlayerOwner.bPlayCinematic)
                return;

        if (BDGPlayerOwner != None && BDGPlayerOwner.IsPaused())
        {
MouseInput = BDGMouseInterfacePlayerInput(BDGPlayerOwner.PlayerInput);
        }

        bOverButton=false;

        if(UIType<=101 && BDGInfo_Game(WorldInfo.Game)!=none)//more maybe
                DrawGameplayHUD();

        if(UIType==103 || UIType==104)
```

```
			DrawLoadMenuHUD();
		else if(UIType==6)
			DrawCharacterInfoHUD();
		else if(UIType==105)
			DrawControlsHUD();
		else if(UIType==102)
			DrawMainMenuHUD();
		else if(UIType==101)
			DrawPauseMenuHUD();
		else if(UIType==1)
			DrawPowerMenuHUD();
		else if(UIType==10)
			DrawConfirmationMessage("Save Completed!");
		else if(UIType==11)
			DrawConfirmationMessage("Save Error!");
		else if(UIType==12)
DrawEndLevelMessage("Victory!","XP
Gained:"$BDGInfo(WorldInfo.Game).GameXP);
		else if(UIType==13)
DrawEndLevelMessage("Defeat!","XP
Lost:"$BDGInfo(WorldInfo.Game).GameXP);
		else if(UIType==14)
			DrawConfirmationDialog("Enter Dream?");
		else if(UIType==20)
			DrawInfoMessage();
		else if(UIType==21){
			DrawPowerMenuHUD();
			DrawInfoMessage();
		}
		else if(UIType==120)
			DrawCityLoadingScreen();
		else if(UIType==130)
			DrawBlackScreen();

		if(UIType>0)
			DrawMouseHUD();



		if(!bOverButton)
			CurrentButton=EmptyButton;
		else{
			if (CurrentButton.isPowerButton())
				DrawPowerInfo();
		}

		//Player Message For Kismet
		if(IsTimerActive('PlayerMessageFunction')){
			DrawPlayerMessageForPostRenderer();
		}
}
```

A Helper class, called `BDGButton`, is also used along with the `BDGHUD`. This class defines the action to be taken when a button is pressed. For this purpose a delegate is defined (pointer to function) that states which function will be called for a specific button click.

For example for the ExitGame Button the action taken, when pressed, will be to Exit the Game.

```
class BDGButton extends Object;
var String ButtonName;
var float StartPointX,StartPointY,EndPointX,EndPointY;
var const float Width,Height;

delegate OnClickAction();
…

//code on the BDGHUD class
ExitGame.OnClickAction=ExitGameAction;

function ExitGameAction(){
      ConsoleCommand("Quit");
}

//called when left or right mouse button is unpressed
function ButtonClicked(){
      if(!bDragging){
            if(CurrentButton!=None && bCurrentButton){
//the function adressed by the OnClickAction of the
//CurrentButton delegate is called
                  CurrentButton.OnClickAction();
            }
      }else{
            bOverButton=false;
            bDragging=false;
            bFinishDragging=true;
      }
      bCurrentButton=false;
}
```

Let’s focus on a simple function that draws a simple message and an ok button, to understand better how this works. The main component for drawing something in the screen is the canvas. It has functions that can Draw Text and Textures in the screen. A combination of these two was used in order to create menus and buttons. In the end a check is performed to see if the mouse is over the OK button. If yes that button is saved on the `CurrentButton` variable and when a mouse click event is performed that button’s action will be called by the use of the `CurrentButton` delegate (see the section describing unrealscript delegates).

```
//draws a message on the screen. Here a lot of built-in unrealscript
//functions are used for drawing on the screen
//e.g. SetDrawColor(), DrawTexture(), DrawText() etc.
function DrawConfirmationMessage(String Message){
      local float XL,YL;
      //pause if not paused mainly for kismet
      if(BDGPlayerOwner.bCanUnpause)
            BDGPlayerOwner.PauseGame();

      Canvas.SetPos(SizeX*0.4, SizeY*0.24);
      Canvas.SetDrawColor(255, 255, 255,255);
      Canvas.DrawTexture(MenuTexture,(0.2*SizeX)/(MenuTexture.SizeX));

      Canvas.Font = class'Engine'.static.GetLargeFont();
      Canvas.SetDrawColor(0, 0, 0);
      Canvas.StrLen(Message,XL,YL);
      Canvas.SetPos(SizeX*0.5-XL/2, SizeY*0.38);
      Canvas.DrawText(Message);

Canvas.Draw2DLine(SizeX*0.422, SizeY*0.235+13*0.2*SizeX/16,SizeX*0.578,
SizeY*0.235+13*0.2*SizeX/16,LineColor);
Canvas.Draw2DLine(SizeX*0.424, SizeY*0.24+13*0.2*SizeX/16,SizeX*0.576,
SizeY*0.24+13*0.2*SizeX/16,LineColor);

      Canvas.Font = class'Engine'.static.GetLargeFont();
      Canvas.SetDrawColor(0, 0, 0);
      Canvas.StrLen("OK",XL,YL);
      Canvas.SetPos(SizeX*0.5-XL/2,SizeY*0.24+13.5*0.2*SizeX/16);
      Canvas.DrawText("OK");
bOverButton=OK.isPressedCircled(Canvas,MouseInput.MousePosition,
SizeX*0.4+0.1*SizeX, SizeY*0.24+0.1*SizeX,0.1*SizeX, 0, SizeX,
SizeY*0.24+13*0.2*SizeX/16, SizeY);
      if(bOverButton){
            CurrentButton=OK;
            return;
      }
}
```

5.4 Controller Hierarchy

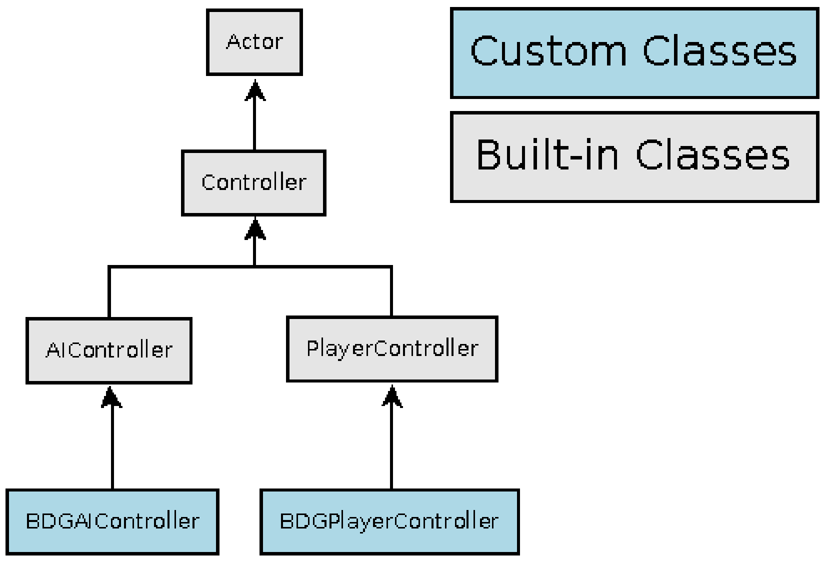


Figure 5.3 – Controller Classes Hierarchy

As it was mentioned above, the Controller is the "mind" of a Pawn. There are two subclasses of the Engine `Controller` Class. The `PlayerController` which is the mind of the Player and the `AIController` which is the mind of the A.I. Player.

In the current implementation the structure includes two Controller classes: the `BDGPlayerController`, that extends the Engine `PlayerController` Class and the `BDGAIController`, that extends the Engine `AIController` Class.

`BDGPlayerController`: Here is the implementation of how a Player Pawn interacts with the game world by moving, jumping, starting to fire etc. The `BDGPlayerController` inherits its basic functionality from the Engine `PlayerController` class. This class declares the `PlayerWalking` state which defines the basic moves of the Pawn (left, right, up, down, jump). For additional functionality exec functions should be declared. These functions are connected in the configuration file with an input button. So when that button is pressed the corresponding function is called. Supposing that the player wants to enter the power menu, he presses the P button, which is linked in the *input.ini* file with the exec `PowerMenu()` function. The function is then called and instructs the HUD to draw the PowerMenu screen if needed (if for example the player is at the main menu, he doesn't have access to the Power Menu Screen).

```
//called when P(shortcut for power menu) is pressed.
exec function PowerMenu(){
      if(bPlayCinematic) return;
      if(BDGHUD(myHud).UIType<100)
//if game not paused, then pause and make UIType=1 (power menu)
            if(!IsPaused()){
                  SetPause(true,CheckCanUnpause);
                  bCanUnpause=false;
                  HandlePause();
                  BDGHUD(myHud).UIType=1;
            }else{
//if already at Power screen, then unpause and se UIType=0 (gameplay)
                  if(BDGHUD(myHud).UIType==1){
                        bCanUnpause=true;
                        SetPause(false,CheckCanUnpause);
                        HandleUnpause();
                        BDGHUD(myHud).UIType=0;
                  }else{
                        BDGHUD(myHud).UIType=1;
                  }
            }
}
```

Other Executable function headers and their functionalities are described below:

```
//performs Zoom out with mouse wheel if no obstacle is in the way of
the camera
exec function ZoomOut();

//performs Zoom in with mouse wheel
```

```
exec function ZoomIn();

//displays the Main Menu when Esc is pressed
exec function ShowMenu();

//displays the Character Menu when C is pressed
exec function CharacterMenu();

//starts firing an equipped Power
//The argument shows which mouse button was pressed(0->left,1->right)
exec function StartFire( optional byte FireModeNum );

//stops firing an equipped Power
//The argument shows which mouse button was unpressed(0->left,1->right)
exec function StopFire( optional byte FireModeNum );

//The power mounted in the quickslot indicated by the argument num, is
prepared to be equipped in the left/right hand (1-5 -> left, 6-10 ->
right, if power is two handed -> both)
exec function switchPower(int num);
```

`AIController`: This class defines the behavior of an A.I. Pawn in the game. By behavior we mean the moving pattern which the AIPawn uses and the decision concerning whether the Enemy should start or stop firing. More Details are given on the AI implementation Section.

5.5 Pawn Hierarchy

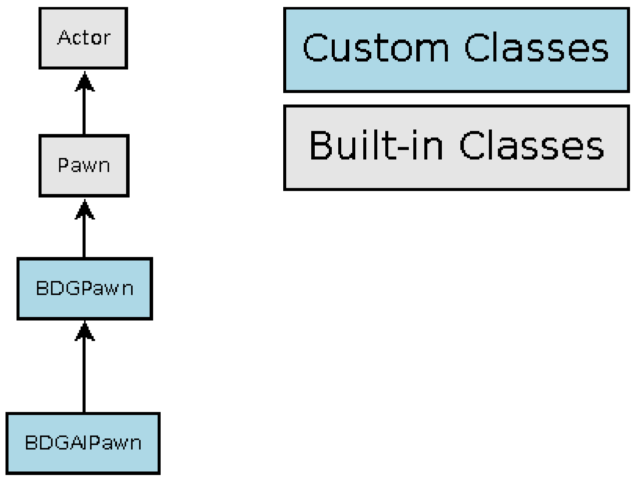


Figure 5.4 – Pawn Classes Hierarchy

A Pawn is the physical presence of a character. In this class the following aspects of a character are defined: The Skeletal Mesh with any Animation Trees, Animation Sets, Physics Asset, Collision component.

Implementation of what happens when a Pawn interacts with other objects of the world. For example when a Pawn is hit, the `takedamage` event is called and distributes the damage to the Pawn, if certain conditions apply (if fireshield is active and the damage is of fire type, then the damage is subtracted from the shield and not the pawn).

```
//this function checks if the current Pawn will takes Damage of type DamageType caused by
InstigatedBy
event TakeDamage(int Damage, Controller InstigatedBy, vector HitLocation, vector Momentum,
class<DamageType> DamageType, optional TraceHitInfo HitInfo, optional Actor DamageCauser)
{

        if(InstigatedBy.Class!=Controller.Class){
//if the type is fire damage then check if fire shield is on, and apply damage reduction
                if(DamageType==class'BDGFireDamage' &&
AllBDGPowers[1]!=none && AllBDGPowers[1].bActivated &&
BDGPower_FireShield(AllBDGPowers[1]).DamageAbsorbed>0){
                        if(BDGPower_FireShield(AllBDGPowers[1]).DamageAbsorbed>=Damage){
                                BDGPower_FireShield(AllBDGPowers[1]).DamageAbsorbed-=Damage;
                                Damage=0;
                        }else{
                                Damage-=BDGPower_FireShield(AllBDGPowers[1]).DamageAbsorbed;
                                BDGPower_FireShield(AllBDGPowers[1]).DamageAbsorbed=0;
                                BDGPower_FireShield(AllBDGGPowers[1]).DeactivateEffects();
                        }
                }
//if the type is ice damage then check if ice shield is on, and apply damage reduction
                else if(DamageType==class'BDGIceDamage' &&
        AllBDGPowers[6]!=none && AllBDGPowers[6].bActivated &&
        BDGPower_IceShield(AllBDGPowers[6]).DamageAbsorbed>0){
                        if(BDGPower_IceShield(AllBDGPowers[6]).DamageAbsorbed>=Damage){
                                BDGPower_IceShield(AllBDGPowers[6]).DamageAbsorbed-=Damage;
                                Damage=0;
                        }else{
                                Damage-=BDGPower_IceShield(AllBDGPowers[6]).DamageAbsorbed;
                                BDGPower_IceShield(AllBDGPowers[6]).DamageAbsorbed=0;
                                BDGPower_IceShield(AllBDGPowers[6]).DeactivateEffects();
                        }
                }


        super.TakeDamage(Damage,InstigatedBy,HitLocation,Momentum,DamageType,HitInfo,DamageCauser);
        }
}
```

The base class for Pawns in the game is `BDGPawn` that extends `Pawn`. This could be used for Player and AI Pawns since they have the same functionality. But the `BDGAIPawn` uses additional functionality at the game initialization, so it is a subclass of the `BDGPawn`, which is the basic Pawn for this game.

```
//used at the AI Pawn initialization
function InitAIPawn(){
        local int i;
        TargetSelectionHUD=Spawn(class'BDGTargetSelectionHUD',self,,);
        HealthBar=Spawn(class'BDGHealthBarHUD',self,,);
        Shield=Spawn(class'BDGShield',self,,);
        for(i=0;i<4;i++){
                MeshMat[i]=Mesh.CreateAndSetMaterialInstanceConstant(i);
                MeshMat[i].SetScalarParameterValue('AIPawn',type);
        }

        initPowers();

        if(type==0){

        TargetArea_Armageddon=Spawn(class'BDGTargetArea_Armageddon',self,,);
```

```
            TargetArea_FireEarth=Spawn(class'BDGTargetArea_FireEarth',self,,);
        }else if(type==1){
            TargetArea_IceAge=Spawn(class'BDGTargetArea_IceAge',self,,);
        }
        ApplyLevelsDelayed();

}
```

5.6 Power Hierarchy

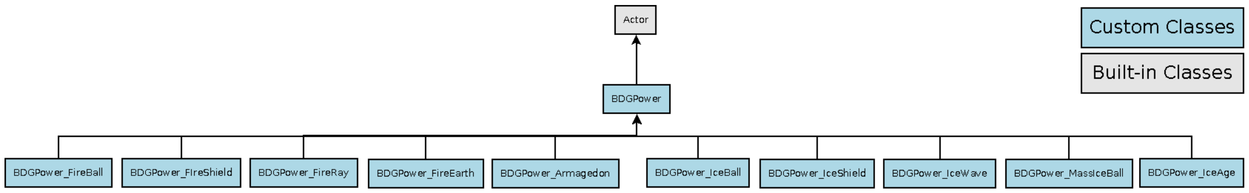


Figure 5.5 – Power Classes Hierarchy

The Power System for this game was designed from scratch. The main class of this system is the `BDGPower` abstract Class that extends the basic Actor Class. This class contains all the logic required to unleash a Power and all the functions' declarations required (with no body). The programmer's only task is to extend this class and design his own Powers. So after this class is completed, it is divided in sub-classes in order to create the Powers required. Let's see how the `BDGPower` class logic is implemented. This class uses mostly timers. Firstly the `equip` function must be called in order to equip the Power to the left and/or right hand. This function is called by an input event, i.e. either by pressing a button from one to ten or by using the mouse cursor.

```
function Equip(byte myhand) //myhand arg (0->left,1->right,2->both)
```

After a Power is equipped it is ready to be unleashed as soon as the corresponding input event occurs. In this case the input event is a mouse click. If a Power is one-handed it can be a left or right mouse click for the left or right hand respectively. If it is two-handed, both mouse buttons are needed, first the right mouse button is needed in order to aim by holding it down, and then a click of the left mouse button.

In this section the procedure of a one-handed Power will be described (for two-handed Powers the procedure is very similar). After the appropriate mouse button is clicked (let's suppose it's the left button) the `StartFire()` Function is called. This function is responsible for starting the animation sequence for the left hand. As soon as the left hand is in place (after an amount of time has passed) the `PrefireAnimEnd()` is called. This function is the one that actually activates the effects of a Power. If the Power has instant effects, the `InstantEffect()` function should be subclassed. If the effect has some duration then the `ActivateEffects()` and `DeactivateEffects()` functions should be subclassed and the `duration` float variable should have a value (representing seconds) greater than zero. If, additionally, the Power has periodic effects the `PeriodicEffects()` should be subclassed and the `Period` float variable should be set in a value smaller than `duration`. After a Power is unleashed it cannot be used immediately. Once the time indicated by the `CoolDownTime` variable has passed (which is greater than `duration`), then the `StartFire()` function can be called again. To sum up, the order of the functions called is the following:

```
Equip(byte myhand)

StartFire()
PrefireAnimEnd()

InstantEffect()
ActivateEffects()
PeriodicEffects()
DeactivateEffects()
```

Let us now see the basic implementation of a single one-handed Power.

```
class BDGPower_FireBall extends BDGPower;

//Spawns the fire projectile
function InstantEffect(){
        local Vector Loc;
        Local BDGProjectile SpawnedProjectile;

        BDGPawn(Owner).Mesh.GetSocketWorldLocationAndRotation(SocketName,Loc);
        SpawnedProjectile=Spawn(ProjectileClass,self,,Loc,,,true);
        SpawnedProjectile.SetLevelChanges(Level);
        if( SpawnedProjectile != None && !SpawnedProjectile.bDeleteMe ){
                SpawnedProjectile.Init( Vector(Owner.Rotation) );
        }
}

DefaultProperties
{
        ProjectileClass=class'BDGProjectile_FireBall'
        duration=0
        CoolDownTime=0.5
}
```

The Fireball class implements only the `InstantEffect()` function and essentially it is responsible for spawning and initializing an Actor of the `BDGProjectile_FireBall` class.

Let us now examine an implementation of a Power with duration effects

```
class BDGPower_IceShield extends BDGPower;

var int DamageAbsorbed;

function ActivateEffects(){
        DamageAbsorbed=25+5*(Level-1);
        SetLevelChanges();
        if(BDGPawn(Owner)!=None){
                BDGPawn(Owner).Shield.EnableIceShieldEffect();
        }
}

function DeactivateEffects(){
	if(BDGPawn(Owner)!=None){
		BDGPawn(Owner).Shield.DisableIceShieldEffect();
	}
}


DefaultProperties
{
	duration=5
	CoolDownTime=10
	Energy=5
}
```

The IceShield Power absorbs ice damage done to the Pawn while it is in effect, so an extra variable is declared and then the `ActivateEffects()` and `DeactivateEffects()` functions are implemented. Upon activation the shield effect is enabled and the `DamageAbsorbed` variable is reset. On deactivation the shield effects are disabled.

Let us now see a Power which additionally has Periodic Effects.

```
class BDGPower_Armageddon extends BDGPower;

var Vector ArmageddonLocation;

function ActivateEffects(){
	ArmageddonLocation=BDGPawn(Owner).TargetArea_Armageddon.Location;
	BDGPawn(Owner).TargetArea_Armageddon.Activate();
}

function DeactivateEffects(){
	BDGPawn(Owner).TargetArea_Armageddon.Deactivate();
}

//the fire projectiles are spawned every 0.1s in a random location
within a circular plane of specific radius(256)
function PeriodicEffects(){
	local Vector Loc;
	Local Projectile SpawnedProjectile;
	Loc=ArmageddonLocation;
	Loc.X=Loc.X+RandRange(-511,511);
	Loc.Y=Loc.Y+RandRange(-511,511);
	SpawnedProjectile=Spawn(ProjectileClass,self,,Loc);
	if( SpawnedProjectile != None && !SpawnedProjectile.bDeleteMe ){
		SpawnedProjectile.Speed=500;
		SpawnedProjectile.Init( Vect(0,0,-1) );
	}
}

DefaultProperties
{
	hand=2
	CoolDownTime=20
```

```
        duration=5
        Period=0.1
        ProjectileClass=class'BDGProjectile_FireBall'
        Energy=18
}
```

This Power shoots fireballs from the Sky every 0.1 seconds and lasts for 5 seconds. The `ActivateEffects()` function initializes the target area and location upon which the Power takes place. The `DeactivateEffects()` function deactivates the target area. Finally the `PeriodicEffects()` function handles the periodic effects. It is called every 0.1 seconds (Period time) and spawns a projectile at the location specified.

5.7 Projectile Hierarchy

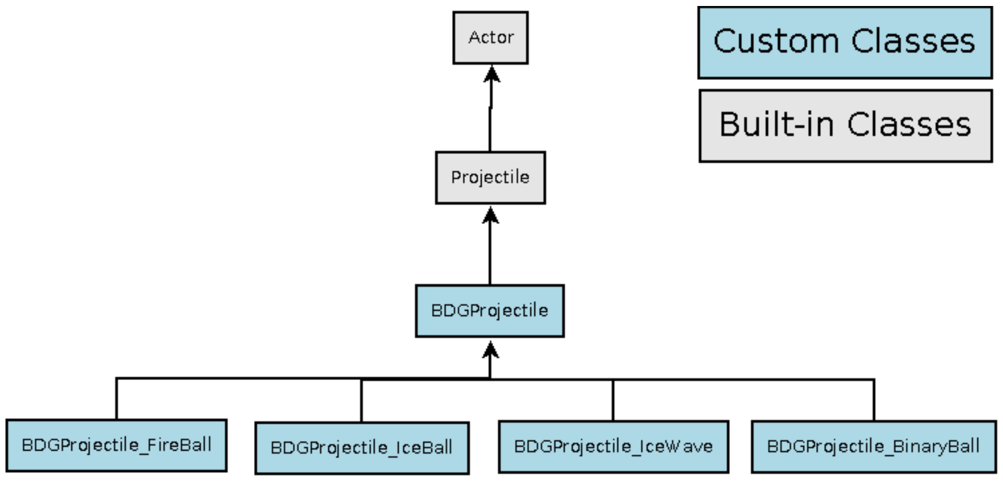


Figure 5.6 – Projectile Classes Hierarchy

Most of the Powers' functionality depends upon the `Projectile` Class. A Projectile is an Actor that moves around for some time after it is created. As soon as it interacts with another actor or its span time expires, the Projectile Actor is destroyed. The basic class of the current implementation is the `BDGProjectile` class that extends the Engine `Projectile` class. This class essentially defines a collision component, the projectile’s speed and an empty Particle System component that will be subclassed.

```
class BDGProjectile extends Projectile;

function SetLevelChanges(int lvl){}

DefaultProperties
{
        Begin Object Name=CollisionCylinder
                CollisionRadius=25
```

```
            CollisionHeight=16
      End Object

      MaxSpeed=+900.000000
      Speed=+900.000000
    MomentumTransfer=0

      begin object class=ParticleSystemComponent Name=MyParticles
      end object
      Components.Add(MyParticles)
}
```

Below you can see an example of a subclass for the `BDGProjectile` Class. This class defines an iceball behavior. The behavior consists of the following aspects: The particle System used and the effects that occur when a touch event happens (damage, sound, slow effect).

```
class BDGProjectile_IceBall extends BDGProjectile;

var float SlowDuration;

function SetLevelChanges(int lvl){
      Damage=8+3*(lvl-1);
      SlowDuration=1+0.25*(lvl-1);
}

//upon touch event start slow effect
simulated function ProcessTouch(Actor Other, Vector HitLocation, Vector 
HitNormal){
      super.ProcessTouch(Other,HitLocation,HitNormal);
      if(BDGPawn(Other)!=none){
            BDGPawn(Other).StartSlowEffect(0.25,SlowDuration);
      }
}

//called after Touch and adds effects e.g. explosion particles, sound
simulated function Explode(vector HitLocation, vector HitNormal)
{
      super.Explode(HitLocation,HitNormal);
      WorldInfo.MyEmitterPool.SpawnEmitter(
      ParticleSystem'BDGContent.IceBall_Impact', Location);
      PlaySound(ImpactSound);
}

DefaultProperties
{
      DamageRadius=0
      MomentumTransfer=0
      ImpactSound=SoundCue'A_Character_Footsteps.FootSteps.A_Character_
Footstep_MetalJumpCue'
      MyDamageType=class'BDGIceDamage'
      begin object Name=MyParticles
            Template=ParticleSystem'BDGContent.IceballParticles'
      end object
}
```

5.8 Pickup Hierarchy

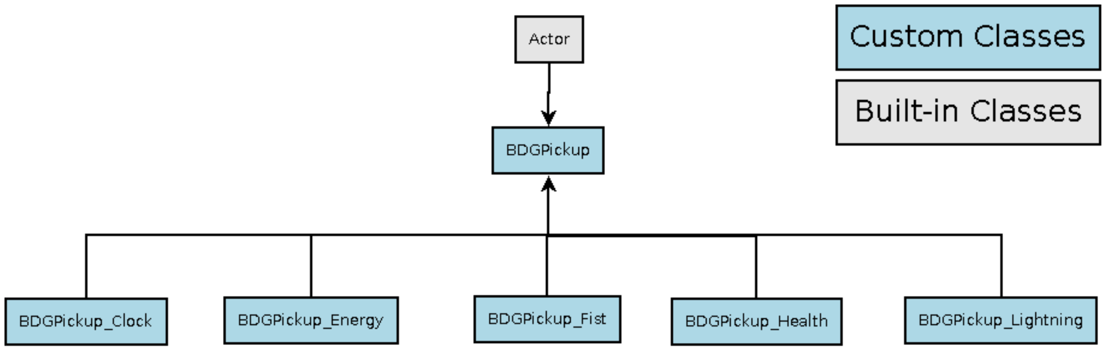


Figure 5.7 – Pickup Classes Hierarchy

A pickup is triggered when a player collides with it and if a specific condition is met, the effects of that pickup item take place. The main class of a pickup item is the `BDGPickup` class that extends the `Actor` class and overrides the `Touch` function which is called when the player steps on the mesh component of that item.

```
event Touch( Actor Other, PrimitiveComponent OtherComp, vector
HitLocation, vector HitNormal ){
	super.Touch( Other, OtherComp, HitLocation, HitNormal);
	if(MPGPawn(Other)!=none  &&
MPGPlayerController(MPGPawn(Other).Controller)!=none){
		P=MPGPawn(Other);
		PC=MPGPlayerController(P.Controller);
		if(Condition())
			Destroy();
	}
}
```

The function states that if the `Condition` function returns true the pickup is destroyed. The subclasses of the `BDGPickup` class, only need to specify the Condition function in which the effects of the pickup take place. Below is the implementation of `BDGPickup_Health` class.

```
class BDGPickup_Health extends BDGPickup;

function bool Condition(){
	//if the pawn has full health do nothing else heal 15 points
	if(P.Health<P.HealthMax){

	P.PlaySound(SoundCue'MPGContent.PickupCue',false,true,,Location,true);
		P.Health=P.Health+15;
		if(P.Health>P.HealthMax)
			P.Health=P.HealthMax;
		return true;
	}
	return false;
}
```

```
DefaultProperties
{
        begin object Name=MyMeshComp
                StaticMesh=StaticMesh'MPGContent.heart'
     end object
}
```

The `Condition` function states that if the Health of the player is at maximum, then the pickup will not be triggered. In other case the pickup will be triggered and the player will restore fifteen points of damage.

5.9 Artificial Intelligence

The AI of an enemy Player depends on two things that happen simultaneously:

- The movement of the enemy Pawn
- The choice of which Power to use

5.9.1 AI movement

In the game the attacks that can be initiated based on the appropriate Powers are mostly ranged attacks at a distance from the human player. So the desired movement of the enemy Pawn will be very specific and will, ideally, follow the pattern shown below:

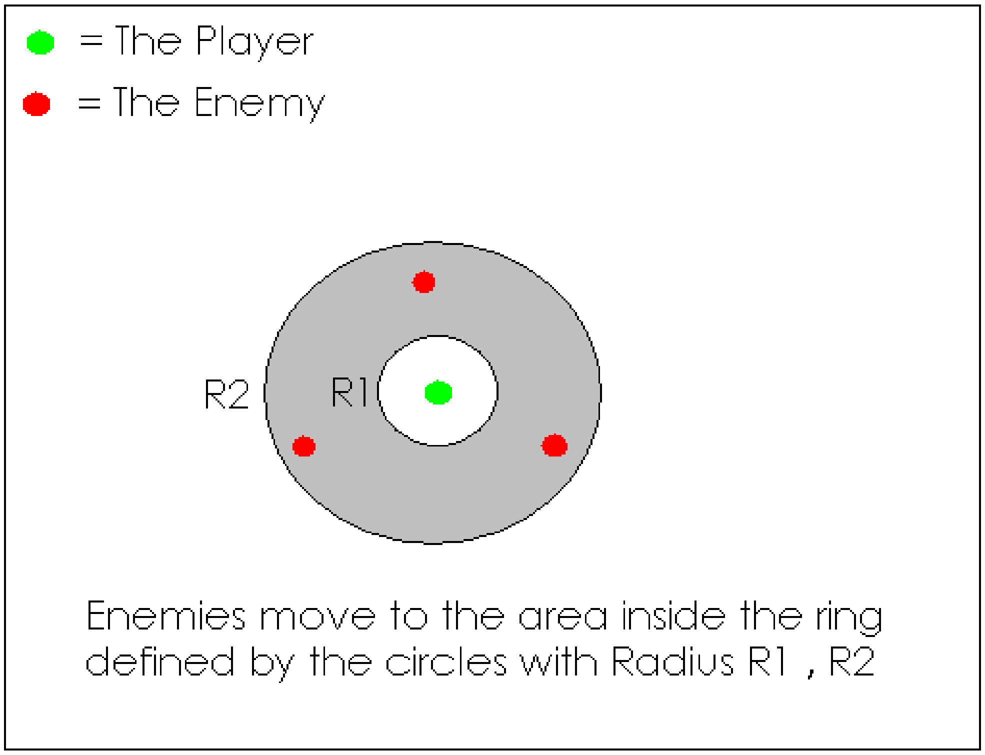

= The Player
= The Enemy
R2
R1
Enemies move to the area inside the ring defined by the circles with Radius R1 , R2

Figure 5.8 – AI desired movement behavior

This can be expanded into three dimensions (where we have spheres instead of circles) but the logic is the same. If the altitude differences of our map are small then our players essentially move to two dimensions and the above can be applied.

What we want to achieve is the enemy to follow the Player Character but never come too close to him. If the enemy finds himself inside the circle with Radius R1 then he will try to walk away from the player and enter the gray area. If the enemy is outside the circle with radius R2, he will try to move closer to the player and enter the gray area.

How this is achieved:

The game map consists of Path nodes that form a graph used to locate a specific actor or point (here this actor is the human player).

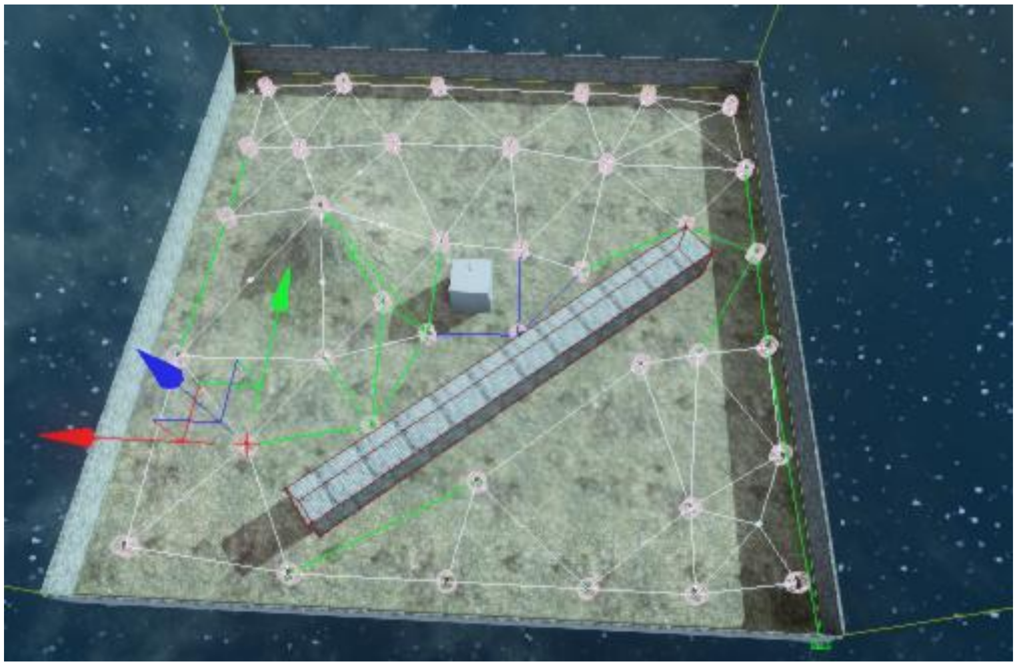


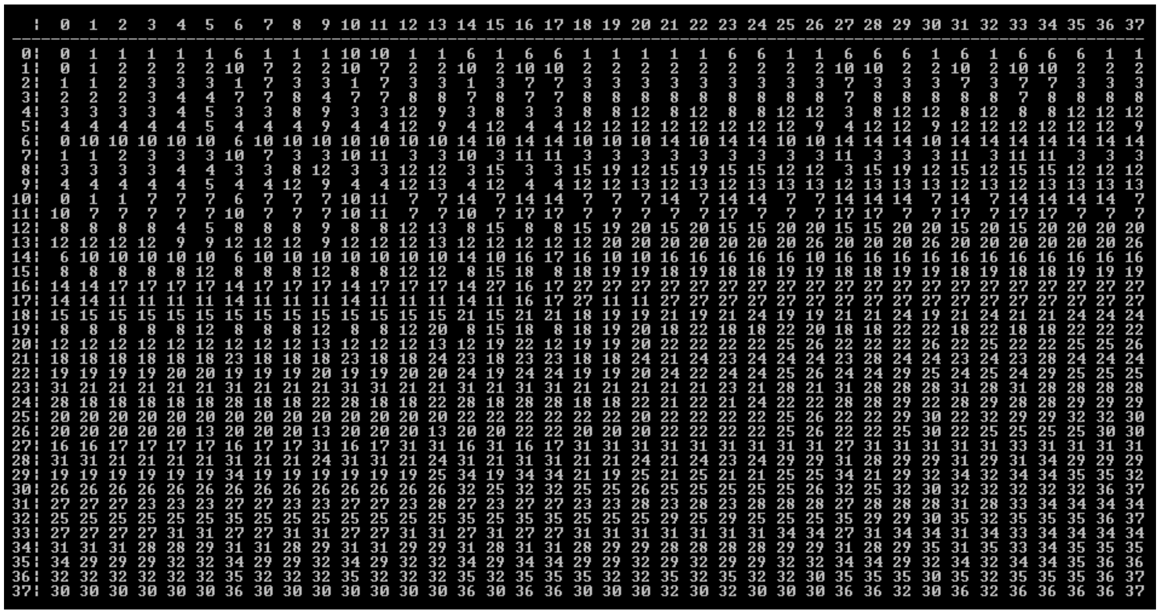
Figure 5.9 – Test map pathnode graph for AI movement

Note: the above map is just a test map, and may not be used in the final game else wise it will be changed appropriately and will consequently have a slightly different graph.

When the enemy sees the target he performs pseudo random moves, in order to confuse the human player and become a more challenging target. If at any time the enemy leaves the gray zone (desirable area), it performs the necessary move in order to stay at a given distance from the player. If at any time the enemy loses sight of the human player (fastTrace function returns false), then the path node graph is used in order to find the Next Move Location of our Pawn.

How is the graph used? Unrealscript has a FindPathToward function that finds the closest path within a certain graph given a source node and a destination node. This implementation is using an algorithm with relatively low complexity in order to compute the closest path in real-time. From a time (not space) perspective the optimal solution, given a connected static graph where no new nodes are inserted, would be to have pre-computed the optimal paths in a 2D array at the beginning of the game or compute them for a map and save them to the hard disk and load them into main memory. Then we can simply get the next best node from any given node we currently are, at O(1) time.

Example:
At the above map, we have 38 nodes. Their indexes range from 0 to 37. If we assume that the maximum number of nodes in a graph does not exceed 256 (which would be a very big map representation), every pointer has a length of one byte. So we only need a 2D array of bytes of the following form:



```
  ¦  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
-----------------------------------------------------------------------------------------------------------------------
 0¦  0  1  1  1  1  1  6  1  1  1 10 10  1  1  6  1  6  6  1  1  1  1  1  6  6  1  1  6  6  6  1  6  1  6  6  6  1  1
 1¦  0  1  2  2  2  2 10  7  2  2 10  7  2  2 10  2 10 10  2  2  2  2  2  2  2  2  2 10 10  2  2 10  2 10 10  2  2  2
 2¦  1  1  2  3  3  3  1  7  3  3  1  7  3  3  1  3  7  7  3  3  3  3  3  3  3  3  3  7  3  3  3  7  3  7  7  3  3  3
 3¦  2  2  2  3  4  4  7  7  8  4  7  7  8  8  7  8  7  7  8  8  8  8  8  8  8  8  8  7  8  8  8  8  8  7  8  8  8  8
 4¦  3  3  3  3  4  5  3  3  8  9  3  3 12  9  3  8  3  3  8  8 12  8 12  8  8 12 12  3  8 12 12  8 12  8  8 12 12 12
 5¦  4  4  4  4  4  5  4  4  4  9  4  4 12  9  4 12  4  4 12 12 12 12 12 12 12 12  9  4 12 12  9 12 12 12 12 12 12  9
 6¦  0 10 10 10 10 10  6 10 10 10 10 10 10 10 14 10 14 14 10 10 10 14 10 14 14 10 10 14 14 14 10 14 14 14 14 14 14 14
 7¦  1  1  2  3  3  3 10  7  3  3 10 11  3  3 10  3 11 11  3  3  3  3  3  3  3  3  3 11  3  3  3 11  3 11 11  3  3  3
 8¦  3  3  3  3  4  4  3  3  8 12  3  3 12 12  3 15  3  3 15 19 12 15 19 15 15 12 12  3 15 19 12 15 12 15 15 12 12 12
 9¦  4  4  4  4  4  5  4  4 12  9  4  4 12 13  4 12  4  4 12 12 13 12 13 12 13 13 13 12 13 13 13 12 13 12 13 13 13 13
10¦  0  1  1  7  7  7  6  7  7  7 10 11  7  7 14  7 14 14  7  7  7 14  7 14 14  7  7 14 14 14  7 14  7 14 14 14 14  7
11¦ 10  7  7  7  7  7 10  7  7  7 10 11  7  7 10  7 17 17  7  7  7  7  7 17  7  7  7 17 17  7  7 17  7 17 17  7  7  7
12¦  8  8  8  8  4  5  8  8  8  9  8  8 12 13  8 15  8  8 15 19 20 15 20 15 15 20 20 15 15 20 20 15 20 15 20 20 20 20
13¦ 12 12 12 12  9  9 12 12 12  9 12 12 12 13 12 12 12 12 12 20 20 20 20 20 20 20 26 20 20 20 26 20 20 20 20 20 20 26
14¦  6 10 10 10 10 10  6 10 10 10 10 10 10 10 14 10 16 17 16 10 10 16 16 16 16 16 10 16 16 16 16 16 16 16 16 16 16 16
15¦  8  8  8  8  8 12  8  8  8 12  8  8 12 12  8 15 18  8 18 19 19 18 19 18 18 19 19 18 18 19 19 18 19 18 18 19 19 19
16¦ 14 14 17 17 17 17 14 17 17 17 14 17 17 17 14 27 16 17 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27
17¦ 14 14 11 11 11 11 14 11 11 11 14 11 11 11 14 11 16 17 27 11 11 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27 27
18¦ 15 15 15 15 15 15 15 15 15 15 15 15 15 15 21 15 21 21 18 19 19 21 19 21 24 19 19 21 21 24 19 21 24 21 21 24 24 24
19¦  8  8  8  8  8 12  8  8  8 12  8  8 12 20  8 15 18  8 18 19 20 18 22 18 18 22 20 18 18 22 22 18 22 18 18 22 22 22
20¦ 12 12 12 12 12 12 12 12 12 13 12 12 12 13 12 19 22 12 19 19 20 22 22 22 22 25 26 22 22 22 26 22 25 22 22 25 25 26
21¦ 18 18 18 18 18 18 23 18 18 18 23 18 18 24 23 18 23 23 18 18 24 21 24 23 24 24 24 23 28 24 24 23 24 23 28 24 24 24
22¦ 19 19 19 19 20 20 19 19 19 20 19 19 20 20 24 19 24 24 19 19 20 24 22 24 24 25 26 24 24 29 25 24 25 24 29 25 25 25
23¦ 31 21 21 21 21 21 31 21 21 21 31 31 21 21 31 21 31 31 21 21 21 21 21 23 21 28 21 31 28 28 28 31 28 31 28 28 28 28
24¦ 28 18 18 18 18 18 28 18 18 22 28 18 18 22 28 18 28 28 18 18 22 21 22 21 24 22 22 28 28 29 22 28 29 28 28 29 29 29
25¦ 20 20 20 20 20 20 20 20 20 20 20 20 20 20 22 22 22 22 22 22 20 22 22 22 22 25 26 22 22 29 30 22 32 29 29 32 32 30
26¦ 20 20 20 20 20 13 20 20 20 13 20 20 20 13 20 20 22 22 20 20 20 22 22 22 22 25 26 22 22 25 30 22 25 25 25 25 30 30
27¦ 16 16 17 17 17 17 16 17 17 31 16 17 31 31 16 31 16 17 31 31 31 31 31 31 31 31 31 27 31 31 31 31 31 33 31 31 31 31
28¦ 31 31 21 21 21 21 31 21 21 24 31 31 21 24 31 21 31 31 21 21 24 21 24 23 24 29 29 31 28 29 29 31 29 31 34 29 29 29
29¦ 19 19 19 19 19 19 34 19 19 19 19 19 19 25 34 19 34 34 21 19 25 21 25 21 21 25 25 34 21 29 32 34 32 34 34 35 35 32
30¦ 26 26 26 26 26 26 26 26 26 26 26 26 26 26 32 25 32 32 25 25 26 25 25 25 25 25 26 32 25 32 30 32 32 32 32 32 36 37
31¦ 27 27 27 23 23 23 27 27 23 23 27 27 23 28 27 23 27 27 23 23 28 23 28 23 28 28 28 27 28 28 28 31 28 33 34 34 34 34
32¦ 25 25 25 25 25 25 35 25 25 25 25 25 25 25 35 25 35 35 25 25 25 29 25 29 25 25 25 35 29 29 30 35 32 35 35 35 36 37
33¦ 27 27 27 27 31 31 27 27 31 31 27 27 31 31 27 31 27 27 31 31 31 31 31 31 31 34 34 27 31 34 34 31 34 33 34 34 34 34
34¦ 31 31 31 28 28 29 31 31 28 29 31 31 29 29 31 28 31 31 28 29 29 28 28 28 28 29 29 31 28 29 35 31 35 33 34 35 35 35
35¦ 34 29 29 29 32 32 34 29 29 32 34 29 32 32 34 29 34 34 29 29 32 29 32 29 29 32 32 34 34 29 32 34 32 34 34 35 36 36
36¦ 32 32 32 32 32 32 35 32 32 32 35 32 32 32 35 32 35 35 35 32 32 35 32 35 32 32 30 35 35 35 30 35 32 35 35 35 36 37
37¦ 30 30 30 30 30 30 36 30 30 30 30 30 30 30 36 30 36 36 30 30 30 32 30 32 30 30 30 36 36 32 30 36 32 36 36 36 36 37
```

Figure 5.10 – Array with optimal paths for all nodes

Let's read an entry of the above array. If we are at the node (source node) with index 12 (row 13) and wish to go to the node (destination node) with index 18(column 19), then the next node we must go to (which is the optimal node according to the pre-computed closest path) is the node with index 15. So in pseudo-code we have:
array[12][18]=15

The space overhead for the 2D array of bytes (assuming a maximum of 256 nodes) is 256*256 cells * 1 byte per cell = 65536 bytes =65 Kbytes, which is minor compared to the 200 Mbytes of main memory that the UDK uses.

So by sacrificing a bit of space, we achieve optimal time performance.

5.9.2 AI Power Chooser

In the game the AI players have the same Powers as the main character, but are divided in two categories: The fire Pawns and the ice Pawns. The fire Pawns mainly unleash fire Powers while the ice Pawns unleash ice Powers. The method for a Pawn to choose a Power is quite straightforward. The Powers with the longest cooldownTime and the most effective will be the first to be unleashed. Then more common Powers with smaller cooldownTime are unleashed. Also a Power is not unleashed if the enemy of the AI Pawn representing the main character is not in Range or if his Energy is too low and he cannot unleash the Power. The implementation of the PowerChooser function is in the `BDGAIPawn` class (see below).

5.9.3 AI Implementation:

The classes with AI functionality are the following.

```
BDGInfo_Game
BDGAIController
BDGAIPawn
```

`BDGInfo_Game`: This class is initialized at the beginning of every dream level. Here the 2D array described above is computed only once for the current map. Once computed it can be accessed by the AI Players at any time to get the shortest path for a given location. The steps used are the following:

```
InitPathNodeList();
InitPaths();
ComputeWeights();
DijkstraAlgorithmForAllNodes();
Paths.print();
```

The `InitPathNodeList()` function puts all the path nodes in a list for easier access during the shortest algorithm phase.

The `InitPaths()` function simply initializes the `Paths` 2D array in order to store the result of the algorithm.

The `ComputeWeights` function computes the weights of all nodes and stores the results in a 2D float array. The weight is simply the Euclidean distance of two path node Location in the 3D space. They are computed with the use of the VSize function:

```
VSize(PathNodes[i].PathList[j].End.Actor.Location-PathNodes[i].Location)
```

The `DijkstraAlgorithmForAllNodes` function is where all the computations take place. Here the Paths 2D array is filled by finding the closest paths for all the nodes using the dijkstra algorithm. The algorithm was chosen mostly because of its simplicity and not for his speed and alternatives such as the A* algorithm could be used. The speed is not crucial because as described above, the computation of the array can be made only once for every map and then access the array at O(1) when it is needed.

BDGAIPawn: Here, as described above, the PowerChooser function implementation takes place.

```
function PowerChooser(){
        if(type==0){
                if(!AllBDGPowers[1].bActivated &&
        AllBDGPowers[1].bCooldownFinished &&
        ((Energy-AllBDGPowers[1].Energy)>=0)){
                        EquipPower(1,1);
                        EquipPower(6,0);
                        return;
                }
                if(!AllBDGPowers[4].bActivated &&
        AllBDGPowers[4].bCooldownFinished &&
        IsInRange(BDGAIController(Controller).Target) &&
        ((Energy-AllBDGPowers[4].Energy)>=0)){
                        EquipPower(4,0);
                        return;
                }
                if(!AllBDGPowers[3].bActivated &&
AllBDGPowers[3].bCooldownFinished && IsInRange(BDGAIController(Controller).Target) &&
        ((Energy-AllBDGPowers[3].Energy)>=0)){
                        EquipPower(3,0);
                        return;
                }
                EquipPower(0,0);
        }else if(type==1){
                if(!AllBDGPowers[6].bActivated &&
        AllBDGPowers[6].bCooldownFinished &&
        ((Energy-AllBDGPowers[6].Energy)>=0)){
                        EquipPower(6,1);
                        EquipPower(1,0);
                        return;
                }
                if(!AllBDGPowers[9].bActivated &&
        AllBDGPowers[9].bCooldownFinished &&
        IsInRange(BDGAIController(Controller).Target) &&
        ((Energy-AllBDGPowers[9].Energy)>=0)){
                        EquipPower(9,0);
                        return;
                }
                if(AllBDGPowers[8].bCooldownFinished &&
        ((Energy-AllBDGPowers[7].Energy-AllBDGPowers[8].Energy)>=0)){
                        EquipPower(7,0);
                        EquipPower(8,1);
                        return;
                }
                EquipPower(7,0);
                EquipPower(5,1);
        }
}
```

When the type variable equals to zero we have a fire Pawn and the appropriate choices for that kind of Pawn. If the type equals to one then we have an ice Pawn and a different variety of Powers.

BDGAIController: This class implements the way the Pawn moves and, if needed, uses the Paths array computed in the BDGInfo_Game class. The BDGAIController has two States: Idle and Hunt. While in the Idle state the Pawn does nothing. The main functionality is implemented at the Hunt state, where the AI Player behaves as described above (figure). In Pseudocode the state does the following:

```
state Hunt{
Begin:
      Find the next location to move to that meet certain
specifications.
      goto('Begin');
}
```

Now let's examine how exactly the Pawn moves. If we have visual confirmation of the Target then the code is simple:

```
if(FastTrace(Target.Location,Pawn.Location)){
            CurrentNodeIndex=-1;
            NextNodeIndex=-1;
            TargetNodeIndex=-1;

            MoveGuidelines(MoveLocation,OffsetLocation);
            MoveTo(MoveLocation,Target,OffsetLocation);
}
```

When the fastTrace function returns true this means that the AI Pawn has visual contact with the Player and doesn't need to use the path node graph. Then the Pawn moves according to some Guidelines in order to stay in the ring described above. The Guidelines are defined below:

```
function MoveGuidelines(out Vector ReturnLoc,optional out float
offset){
      if(VSize2D(Target.Location-Pawn.Location)>1400){
            ReturnLoc=findNextLocationForFar();
            offset=1000;
      }else if(VSize2D(Target.Location-Pawn.Location)<750){
            ReturnLoc=findNextLocationForClose();
            offset=0;
      }else{
            ReturnLoc=findNextLocationForInRange();
            offset=0;
      }
}
```

The code above translates to this:

If the distance of the target player is long (>1400), move towards him (not next to him, but with a specific offset from him).
If the distance is too short (<750) then try to get away from the player.
If the Pawn is in range then don't stay completely still but make small random moves in order to be a more difficult target to hit and a challenge for the human player.

In the case that the human Player is not visible by the human player (the fastTrace function returns false) then the following code takes place:

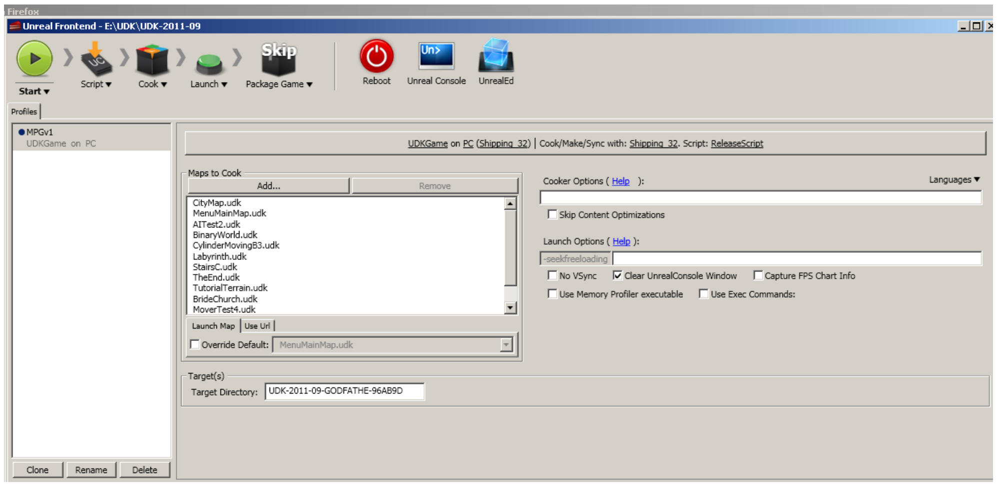
```
if(CurrentNodeIndex==-1){
        Temp=TraceNodeFrom(Pawn.Location);
        if(Temp!=none){
                CurrentNodeIndex=byte(Right(Temp.Tag,3));
                MoveTo(Temp.Location,Target);
        }else{
                MoveGuidelines2(MoveLocation,OffsetLocation);
                MoveTo(MoveLocation,Target,OffsetLocation);
        }
}else{
        TargetNodeIndex=byte(Right(Temp.Tag,3));
        NextNodeIndex=BDInfo(WorldInfo.Game).Paths.get(CurrentNodeIndex,TargetNodeIndex);
        CurrentNodeIndex=NextNodeIndex;
        MoveTo(BDInfo(WorldInfo.Game).PathNodes[NextNodeIndex].Location,Target);
}
```

In short, if the AI Pawn isn't currently on a node (`CurrentNodeIndex==-1`) then the script will attempt to find a node close to the AI Pawn. The unreal engine uses an octree implementation to store objects and this is relatively fast code. If a node is found and the player is still not visible, then the Paths structure is used to get the next node in the shortest path for the current target location. This happens at O(1) time. So the complexity depends on the octree use made by unreal engine to access nearby nodes. This way there is no need for a shortest path algorithm to run each time the Pawn moves, thus saving execution speed.

5.10 Producing the Executable

After the code is completed, the last stage is to produce the executable. Unreal Engine provides a tool called Unreal Frontend that takes the game content (levels, models, code) and produces an '.exe' file with the final game. This procedure consists of three steps:

- Compilation: the code is compiled ensure there are no errors.
- Cooking: cooking content is how content is converted and massaged into the format that supports consoles. PCs, as well, can now use cooked data, which will result in much faster loading speed.
- Packaging: After the cooking is complete the package is created and the final executable is ready.



MPGv1
UDKGame on PC

UDKGame on PC (Shipping_32) | Cook/Make/Sync with: Shipping_32. Script: ReleaseScript

Maps to Cook
CityMap.udk
MenuMainMap.udk
AITest2.udk
BinaryWorld.udk
CylinderMovingB3.udk
Labyrinth.udk
StairsC.udk
TheEnd.udk
TutorialTerrain.udk
BrideChurch.udk
MoverTest4.udk

☐ Override Default: MenuMainMap.udk

Cooker Options (Help):
☐ Skip Content Optimizations

Launch Options (Help): -seekfreeloading
☐ No VSync ☑ Clear UnrealConsole Window ☐ Capture FPS Chart Info
☐ Use Memory Profiler executable ☐ Use Exec Commands:

Target(s)
Target Directory: UDK-2011-09-GODFATHE-96AB9D

Figure 5.11 – The Unreal Frontend tool

5.11 Technical Implementation Summary

We will now summarize the process of the implementation for this project. We assume that we have configured properly the tools described in section 3.4 and that we have made an initial design of the game with paper. So, in this point we know the resources that will be needed for this game as well as the class diagram that will be implemented. From this point we proceed as follows:

The resources needed for the game are gathered. This mainly involves the artistic part of the game. In particular, the textures and 3D models needed are created in Photoshop and 3DS Max respectively, or are collected through the internet. This content is imported in UDK along with any additional resources needed, such as character animations which are also created in 3DS Max. In UDK these resources are combined, e.g. textures and animations and are applied to the 3D Models. Then the content created is ready to be used for the implementation phase of the game.

During the implementation phase, the core classes needed for the gameplay functionality are implemented. The 3D models created now become gameplay objects which through Unrealscript interact with each other and form the main elements of our game. These elements are Player Pawns, AI Pawns, Powers, Pickup objects, Map terrain, Environment Meshes such as houses, Menus etc.

After the basic interactions are completed and the game has reached an early level of functionality, the game is exported into a windows executable file and is delivered to the testers. The testing results will contribute to the reevaluation of some aspects of the game as described in the next chapter. This procedure continues until the final result is considered satisfactory by the programmer/designer and the testers.

Chapter 6: Evaluation

6.1 Testing

There were two testing phases for this project, the **Alpha** testing and the **Beta** testing. In both phases the testing aimed to evaluate both artistic and programming aspects of the game, however, we were mainly interested on the programming parts and specifically on the following aspects:

- The Power System
- The AI difficulty

The testing phases differed according to the group of people that performed the testing of the game. During both phases, the subjects were given the same testing form which included certain additional evaluation elements in relation to the beta testing phase as described below:

- The first part included general questions about the subjects and the broken dreams game. The subject reported how often he/she plays video games. Concerning the actual game we asked the subject to rate several aspects of the game on a scale of one to ten. For instance, in relation to the perceived difficulty of the game, one was equivalent to a 'very easy' response while ten communicated that the game was perceived to be very hard. These aspects included the plot of the game, the design of the level environment, character, user interfaces (and their ease of use), the music of the game, the AI challenges and the overall difficulty of the broken dreams game. At the end of this testing phase a mean was calculated representing the average ratings of all subjects for each question. Although these questions gave an overall experience about what the subjects thought of the game, they didn't give any indications about what would have been beneficial to change in order to improve the gaming experience. The next parts of the testing form were more crucial in improving the game.

- The second part was added in the beta testing phase and involved specific questions about the AI difficulty and the overall game difficulty. This part was added because in the alpha testing phase one game difficulty level was utilized making the game hard to play, whereas in the beta testing phase there were three level of game difficulty on offer (easy, normal, hard). Subjects were asked which level of difficulty they had selected to play the game and whether they had any observations about each difficulty.

- The third part constituted of questions about the gameplay of the broken dreams game, and especially about the use of the Power System. Subjects where asked which of the powers they used more often, if they used powers with both hands and if the Power System was generally convenient.

- The fourth and final part included questions about the experience of the subjects as a whole, e.g. aspects of the game they liked a lot or not at all, additional observations and criticism, and report of any bugs.

After the subjects tested the game they provided us with their feedback using either the testing form detailed above or just stating their observations and criticism verbally. Either way their opinion was noted in order to reevaluate the game and apply the necessary changes to the next version.

6.1.1 Alpha Testing

During the Alpha Testing Phase the people that played the game were experienced casual gamers chosen by the programmer and could more easily adapt to the gameplay of the current project. The results of their testing of the aspects into question were the following:

- Power System: They found the Powers very intriguing to use, but very difficult due to limitations of the configuration of the Powers. In other words, the testers were disappointed that there wasn’t a way to load and save certain powers when in a dream. This way, every time a player enters a new dream, players must reconfigure their powers, a process extremely time consuming when repeated every time.

- AI difficulty: It is important that the AI is challenging but not unbeatable. The experienced gamers found the AI quite challenging scoring 7.5 points out of 10 using a simple mean metric when rating the difficulty of the game. This means that for a non-casual gamer the AI would be unbeatable, e.g. 10 points out of 10 in the difficulty scale.

6.1.2 Beta Testing

Using the results of the Alpha testing, the game was modified accordingly:

- In relation to the Power System aspect, one significant change was applied in order to make the configuration of the powers easier: the player is now able to save and load his Power Configuration. This way, the players won’t reconfigure their powers each time they go inside a new dream level. They simply press the load power button (CTRL+F1/F2/F3) and their predetermined configuration appears.

- In relation to the AI difficulty aspect, three levels of AI difficulty were implemented: Easy, Normal and Hard with Hard being the original game difficulty used during the previous testing phase. The differences between the AI difficulty levels are explained below:

- Easy: Enemy players use only certain powers that cause only one point of damage (minimum). In addition they have slower movement speed compared to the human player.

- Normal: Enemy players use only certain Powers that cause normal damage. Their speed is normal.

- Hard: Enemy players use more effective powers that cause larger amount of damage damage. Their speed is normal.

During this phase the testing group was larger and included casual and non-casual gamers. The testing was open so anyone could play the game and share his experience, in contrast to the Alpha phase where only selected individuals were chosen. The casual gamers where satisfied with the changes that were included in the next release of the game. The non-casual gamers, however, found it difficult to adapt to the Power System, mainly due to the lack of fast reflexes needed in the game (although on Easy mode the opponents are slower), and the lack of an appropriate tutorial concerning the proper usage of the Powers although a tutorial was offered which the testers didn't find it thorough enough. The next version of the game must attend to their needs too.

6.2 Future Work

Currently the game, despite having most of its functionality come to life, can still be improved in several ways. For starters, a thorough tutorial must be implemented that will explain the Power System to non-casual players. Moreover, although the main plot is complete, the game currently has only a few dream levels designed. Based on new innovative ideas, more levels can be created and easily embodied in the main game without altering the plot. Furthermore, an interesting piece of functionality would be a cooperative multiplayer section for this game, in which two or more players can work together in defeating nightmares. As described above, game design is a dynamic progress, as it is time consuming. A game can constantly be updated with new ideas, but in an one man project, a great amount of time is required in order to reach the ideal result.

6.3 Summary

The purpose of this project was the introduction in the principles and tools needed for the creation of a 3D computer game. At first the concept of the game was conceived. Then a draft design on paper was made. Later on came the familiarization with the Unreal Engine, a very efficient game engine which provided the means of transferring the original idea from paper, into a fully operational 3D computer game. During the creation of this game one thing became apparent: Creating a game is no longer an one person task as was in the early eras of video games. A whole team is needed for that purpose, which should be consisted mainly by 3D art designers and 3D programmers. The art designers provide the models for the 3D world that is to be created and the programmers take these models and give them life with their gameplay rules and interactions with its various objects. Besides the human resources, the process of creating a game these days is only limited by development time and imagination.

Bibliography & References

[1] Unreal Engine, http://www.unrealengine.com/

[2] Unreal Development Kit (UDK), http://www.unrealengine.com/udk/

[3] UnrealScript, http://udn.epicgames.com/Three/UnrealScriptHome.html

[4] Game, http://en.wikipedia.org/wiki/Game

[5] Electronic Game, http://en.wikipedia.org/wiki/Electronic_game

[6] Video Game, http://en.wikipedia.org/wiki/Video_game

[7] First Video Games, http://en.wikipedia.org/wiki/First_video_game

[8] Game Engine, http://en.wikipedia.org/wiki/Game_engine

[9] AAA games article, http://uk.ign.com/articles/2012/07/30/are-aaa-hardcore-games-doomed

[10] Busby J., Parrish Z., Wilson J. (2009) Mastering Unreal Technology, Volume I Introduction to Level Design with Unreal Engine 3

[11] Action Game Genre, http://en.wikipedia.org/wiki/Action_game

[12] Role Playing Game Genre, http://en.wikipedia.org/wiki/Role-playing_video_game

[13] Action RPG Genre, http://en.wikipedia.org/wiki/Action_role-playing_game

[14] Busby J., Parrish Z., Wilson J. (2009) Mastering Unreal Technology, Volume I Introduction to Level Design with Unreal Engine 3

[15] UnrealScript Reference http://udn.epicgames.com/Two/UnrealScriptReference.html

[16] UDK Content Creation http://udn.epicgames.com/Three/ContentHome.html

[17] nFringe, add-on for UDK, http://wiki.pixelminegames.com/index.php?title=Tools:nFringe

[18] Autodesk 3ds Max, http://usa.autodesk.com/3ds-max/

[19] FBX plugin for 3ds Max, http://usa.autodesk.com/adsk/servlet/pc/item?id=10775855&siteID=123112

[20] Adobe Photoshop, http://www.adobe.com/products/photoshop.html

[21] Iterative Design Article,
http://www.gamecareerguide.com/features/577/iterative_.php