Technical University of Crete

Department of Electronic and Computer Engineering

Development of a 3D Network Game

Diploma Thesis by Grigoris Nikiforakis

Board of Inquiry

Mania Katerina, Assistant Professor (Director of Studies)

Christodoulakis Stauros , Professor

Deligiannakis Antonios, Assistant Professor

Greece, Crete, Chania

November 2010

Acknowledgements

From the beginning till the the end, there were people that assisted in making the Thauma T.v game complete. Special thanks to Nikitas Papadoulakis for the cover and the intro he made for Thauma as well as to my family for their financial and emotional support all these years.

Many thanks to Manos Karistineos for composing the piano project especially for the game which is used in the intro video, as well as Marita Karistineos who introduced us. I would also like to thank Vaggelis Gavalakis, Nikos Katzakis, Samantha Haidemenaki and Aggelos Ledakis, for their contribution in various subjects.

All the Alpha and Beta testers, and especially Vasilis Milonakis, Manos Kalaitzoglou, and Nikos Katzakis for their helpful feedback. Additionally, Epic's forum members for assisting in UDK deadends.

Last but not least, Katerina Mania for her valuable guidance and commentary during the writing of this text and for giving me the chance to create my final thesis in order to get my diploma.

Extra thanks to Nikos, Manos and WASP.

Abstract

The aim of this project was to create an entertaining **Multiplayer 3D game**. The primary objective has been to work on the most important tasks a software engineer is taught, in the context of game development. The game designed and created from this project is titled **Thauma**. Thauma is a story-based game which supports on-line multiplayer mode and includes different genres throughout the game. The game presented is Multiplayer, Story-based and falls in the category of First person shooter games. The game is about Jonathan, a man who experienced a traumatic event at a young age, but has managed to leave it behind and go on with his life. At some point in his present life, he has to confront his past painful experiences and start a battle with himself. During his journey his dual personality disorder emerged, leading to the awakening of his evil side. This evil side (Dark Jonathan in game) incarnates the negative emotions Jonathan feels during his adventure. The game starts when the subconciousncess starts revealing memories that should remain hidden. The second player is assigned to the role of Jonathan's Evil self. If he manages to win the battle against his evil self, he will have succeded in finding peace with himself, discarding his dual personality disorder once and for all. In case he loses, that will mean his dark side overcame his good side.

Game-wise the battle consists of 4 phases, depicting Jonathan's unstable mind. Each side has 4 powers (skills) that can be used spending psychic energy. This energy comes from Jonathan's mental strength and cannot overcome a certain maximum value (100). The skills Dark Jonathan can use are of evil nature, able to cause damage and control Jonathan, while Jonathan has defensive skills at his disposal. Using these skills effitiently will aid the player in killing the opponent. A point system is used to determine the winner. Points can be obtained with two different ways. The first way is by killing the opponent and the second one is by participating in the events.

The game engine used was the Unreal Development Kit (UDK). UDK is an industry-standard game engine used today for the development of many commercial high-quality games by multiple personnel teams. Unrealscript was designed to provide the developers with a powerful, built-in programming language that maps the needs of game programming.

UnrealScript is purely object-oriented and comprises of a a well-defined object model with support for high level object-oriented concepts such as serialization and polymorphism. This design differs from the monolithical one that classic games adopted having their major functionality hardcoded and unexpandable at the object level. Before working with UnrealScript, understanding the object's hierarchy within UDK is crucial in relation to the programming part.

Table of Contents

Chapter 1: Introduction

1.1 Purpose of Thesis

The aim of this project was to create a **Multiplayer 3D game**. The primary objective has been to work on the most important tasks a software engineer is taught, in the context of game development. Most projects in this field focus on the implementation part, resulting in a game which is not fun to play. Creating game parts from scratch is not an easy procedure. It is significant that the game's rules and mechanics are designed to provide fun. This project aims to design an engaging story line of a game which will result in a demanding software engineering implementation. This thesis takes an in-depth look at the narrative in games, gameplay, game prototyping as well as the implementation process. During game development attention should be devoted to all of the above.

The game designed and created from this project is titled **Thauma**. Thauma is a story based game which supports on-line multiplayer mode and includes different genres throughout the game. The Thauma TUC version (Thauma T.v) is the completed part of the game described in this thesis. The game presented is Multiplayer, Story-based and falls in the category of First person shooter games. The game is about Jonathan, a man who experienced a traumatic event at a young age, but has managed to leave it behind and go on with his life. At some point in his present life, he has to confront his past painful experiences and start a battle with himself. Memories emerge like ghosts from the past, and each has a question that needs to be answered, in order to find peace. Guiltiness, anxiety, depression and moral dilemmas comprise of the burden the hero has to carry till he reaches the end in his mind's maze. When his journey comes to an end, will he find catharsis or will the opening of his mind's hidden locks lead him to another purgatory? The second player is assigned to the role of Jonathan's Evil self. The two players have to confront each other with several skills (super-power abilities) at their disposal.

The programming environment used (game engine) was the **Unreal Development Kit (UDK). UDK is an industry-standard game engine** used today for the development of many commercial AAA (high quality) games by multiple personnel teams.

The next section presents the structure of this thesis.

1.2 Thesis Summary

Chapter 1 includes an introduction and a summary to the thesis.

Chapter 2 describes the initia form of games, their origin and their evolution

throughout the centuries. Focus is given to digital games, their past, present and a glimpse to the future, according to experts in the field. Entertainment and how it can be achieved via games is also discussed in this chapter, concluding with an analysis of the game development processes adopted by professional teams.

The tools that were needed and used, along with the tools that did not meet the requirements for this project are all described in **Chapter 3.** A thorough description of the game engine is included in this chapter as well.

Chapter 4 presents the development process adopted which is a modification of the generic game development process described in Chapter 3 because of the team's size. All the steps during game design and game development are explained in detail in Chapter 4, before proceeding to **Chapter 5** which presents the implementation of the game. This includes the 'Skills' creation, the 'Rules' settings and the effort required to implement the multi-player part of tha game.

Finally, quality assurance and future work as resulted from the players' feedback during the Alpha and the Beta testing are presented in **Chapter 6.**

1.3 Development Review

The stages of the development process for this project differ to some degree from the ones adopted in industry, mainly because there was an one-man team and secondly because the purpose of the project was not a product release, but rather a thesis for an "**Electronic and Computer Engineering**" degree. The roles needed for game development are that of Designer, Programmer and Level Designer. Each role's responsibilities are described below.

Designer:

A game designer's responsibility includes the design of the gameplay in addition to the rules and sctructure of the game. Game development teams are usually managed by a Lead Designer who is the main visionaire of the game. Conceiving the narration, writing the dialogues, commentary, journals and video game packaging are among the duties the designer has.

Programmer:

Software engineers and programmers take up this role. Software engineers primarly develop the game or the platform needed (Game Engine), while programmers mainly deal with the game's codebase. A Lead Programmer initializes the codebase programming and overviews progress. Different areas a programmer can focus are: Gameplay, Scripting, Network Play, UI and more (these four were also the author's responsibilities).

Level Designer:

The Level Designer is the person who creates the levels of the game. Level is the environment the game takes place in. A game may consist of many levels, in different places. The game's progress, missions and events are all created by the level designer often using a level editor. Level editors do not require any programming needs; high-level scripting is used in order to develop interactive environments or game AI. Level Designers also work with game prototypes before including artwork (Fig 1.1) or scripting the levels.

Fig 1.1 -Cover of the Thauma game made by Nikitas Papadoulakis.

Chapter 2: Games and How to Develop them

2.1 History of Games

2.1.1 Origin of Games

In order to succesfully grasp the concept of games, one has to look back to when and how games appeared. Games date back to ancient times, when people participated in them not only for their **entertainment**, but also for religious purposes. Every culture had its own games, which were of great importance to both the athletes and the spectators. There were some occasions where games were used as a mean to communicate with deities. Games demanded a highest level of skill that only a few could reach, since sometimes the athletes had to put their life on the line (Fig 2.1). Certain games had roots in ancient legends, such as Sumo , where each match was a representation of a ritual.

Fig 2.1 -Taurokathapsia (Bull leaping) was a ritual performed in connection with bull worship.

Different forms of ancient games shared many common characteristics and they even shared some characteristics with the form of games as we know it today. The majority of the games required courage in order to participate and the action element was apparent through the process. Apart from that, there was a **winning** condition, **obstacles**, and sometimes **enemies** and even **allies**. Hunting, for example, was done by hunters with the help of dogs, while the goal was to catch the prey and overcome the landscape's obstacles. Other common characteristics were two sides fighting each other or the protagonist fighting for a prize. The word

'Protagonist' is a greek word, where "protos" means first and "agon" means struggle and the protagonist would struggle in order to reach a goal. Moreover every game had a certain structure. Its beggining, middle and end defined the phases of the game. The rules that were set along with the landscape, limited the player's abilities to a certain degree. In the case of victory a prize was claimed, or a penalty in case of failure. Even today some types of games have maintained their original form, so for example we see a lot of "Predator-Prey" kind of games (Pacman, Fig 2.2), or "A Hero's Journey" type (Tomb Raider), or even the classic fight between two sides (Counter Strike).

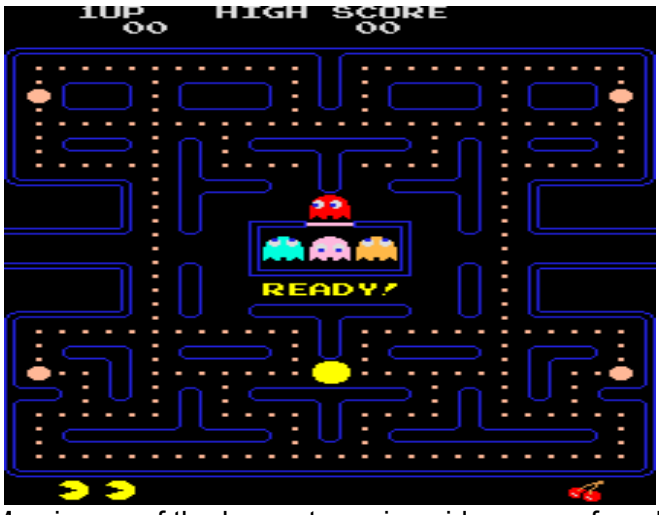

Fig 2.2 -Pac-Man is one of the longest running video game franchises from the golden age of arcade games.

2.1.2 Digital Games

When computers came along, the urge to implement a game in digital form occured. A modification of tic-tac-toe named "OXO" first appeared in **1952**, as well as a space combat simulator named Spacewar! invented by three MIT students (Martin Graetz, Alan Kotok and Steve Russell) in 1961, on a PDP-1 computer used for statistical calculations (Fig 2.3). The text game genre was the first for the Personal Computer which allowed the user to type the commands while immersed in a fantasy world described by text. Coin up machines also made their appearance in the 70's which used vector graphics. Star Wars, Death Race and Breakout (the latter was made by Steve jobs) were some of the famous coin ups at that time.

Later on that decade there was the beggining of a commercial move, when the first TV consoles became available. In 1983, the consumer's interest for personal computers rose and as a result the low-cost **Commodore 64** appeared. Large scale companies saw this as a chance to rule the market and their great support to computers followed soon. The invention of the mouse, the improvement of the sound via the new sound cards and certain succesful games helped the personal computer to become even more popular. Nintendo and Sega were dominant in the console market over the next decade when the next generation consoles appeared. A year before that, (1993) Doom appeared and nVidia was founded. Both events were of great importance for the future of the computer games.

Computers brought games to a new dimension including high quality sound effects, vivid artistic imagery and easy-to-use input, all of which led to a smooth immersion of the player to a fantasy world. Succesfull companies inactive until then in game-relevant productions (Microsoft, Sony and more) grabbed the opportunity to rule that market and started investing large amounts of money to software and hardware relevant to games. **3D graphics** also set the ground for multiple companies working on graphics cards.

Fig 2.3 -It took the three MIT students approximatelly 200 hours of work to create the first version of Spacewar.

2.1.3 Present-Future

Since the dawn of this millenium, games have been continuously growing in popularity. Online games now rule the market and the profit of the game industry surpasses this of the movie industry. The most popular Massive Multiplayer Online Role Playing Game (MMORPG) **World of Warcraft** (WoW) by Blizzard counts more than 10 million subscribers (Fig 2.4). The spread and advancement in telecommunications technology along with the mass improvements on gameplay characteristics of other popular online games, led WoW to inspiring artists to satirize or acknowledge its mark in popular culture. Another approach to gaming was also suggested by Nintendo's console Wii, where the input escapes the classical keyboard-mouse and introduces a more "active" way of input, with the help of movement sensors. A different genre then appeared, the one that demanded physical interaction. The popularity of games is increasing more and more, with worldwide events happening (BlizzCon) and millions of fans enjoying either online or single player, PC or console games.

Recently, a more accessible type of game appeared. Such games' audience were mostly non gamers, or casual gamers, so the fan base was not there to support it, but suprisingly it succeded more than expected and the profits of such games nowdays are enormous. Browser, flash and mostly facebook games give the opportunity to people with no previous game experience, to spend time with a simple game that is fun to play. **Carnegie Mellon University** Professor, Jesse Schell, dives into a world of game development which will emerge from the

popular "Facebook Games" era in his "Design Outside the Box" presentation this year[8]. According to the Professor, our future is full of gameplay characteristics, that add both to our enjoyment and to a better quality of life. So, the question arises; what is it in games, that makes them so much fun that can even affect our lifestyle?

Fig 2.4 -World of Warcraft has more than 10 million subscribers.

2.2 Entertainment in Games

Happiness and how to obtain it has always troubled mankind. In ancient Greece, philosophers reached several conclusions. According to **Aristotle**, "More than anything else, men and women seek happiness". Psychologists have revealed a strong connection between a pleasurable state called ***Flow*** and gaming experience. Mihaly Csikszentmihalyi found that "what makes experience genuinely satisfying is a state of consciousness called flow"[1]. His studies suggest eight major components that people have (one or more) when they are feeling most positive:

1. They confront tasks they have a chance of completing;
2. They must be able to concentrate on what they are doing;
3. The task has clear goals;
4. The task provides immediate feedback;
5. One acts with deep, but effortless involvement, that removes from awareness the worries and frustrations of everyday life;
6. One exercises a sense of control over their actions;
7. Concern for the self disappears, yet, paradoxically the sense of self emerges stronger after the flow experience is over; and
8. The sense of duration of time is altered

The author comments in [1] that "The combination of all these elements causes a sense of deep enjoyment that is so rewarding, people feel that expending a great deal of energy is worthwhile simply to be able to feel it."

The same 'symptoms' appear to people playing computer games. Unfortunatelly in rare occasions this flow state goes on for too long with disastrous results for the person which becomes addicted to gameplay (Fig 2.5).

The connection between Csikszentmihalyi's flaw state and the gaming experience has been studied by his colleagues, resulting in important facts. In Jenova Chen's Viewpoint "Flow in Games (And Everything Else)" he studies and analyzes that connection among others[7]. According to Mr Chen, important variables are: the player's skill, the difficulty of the game, the amount of choices the gameplay offers to the player which should be no more than the player can handle. Two different players have different skill levels and each one is challenged more or less by different situations. For these reasons the game designer should keep all these factors in mind in an effort to lead most players to the flow state. There are however many experts that do not accept the Flow theory in the gaming enviroment, however they support other theories that derive from psychology.

A step further was the work of Georgios N. Yannakakis and John Hallam titled "Capturing Player's enjoyment in computer games"[6] which presented mathematical equations that describe the amount of challenge needed, for example, in order for the game to be more enjoyable in relation to improvements of the AI methods incorporated in the game. Such work links game development with user satisfaction.

Fig 2.5 -Starcraft: "They are the types of games that completely engross the player. They are not games that you can play for 20 minutes and stop" , Psychologist Professor Mark Griffiths[10]

Games can generally bring enjoyment to people. Therefore, during the game design and development, player experiences should be taken into consideration. Quests, user abilities, game difficulty over time and varied gameplay 'tools' should be at the designers' disposal.

2.3 Game Development

In the early days of video games (1970), a programmer was responsible for creating a game from scratch and could also take on the job of the designer and artist. The limitations of computer power at that time, led to the programmer being able to handle all functions, therefore, having specialized personnel was unnecessary. Games were only meant to be played for a few minutes at a time, while art content and variations in gameplay were constrained by the limitations of computer power, making the work of the programmer easier.

Nowdays, a game development studio is composed of tens and for AAA projects even hundreds of people, seperated in teams for each task that needs to be accomplished. The most complex part of any computer game is the software behind it. The software needs to incorporate game play mechanics, art, algorithms, network protocols, real time physics, simulation, decompression and playback of music, 3d visualisation and more.

A **software engineer** holds an important role in the game development process and is sought after for his ability to assist in many areas. From early-step **designing**, to middle-step **prototyping**, to late-step **implementing**, his versatility enables him to contribute to a large part of the game creation process and most importantly, during all the different phases a game requires to be developed. Adopting software engineering processes leads to better software products and games are not an exception considering the fact that a game takes from one to four years or more to be developed.

2.3.1 Development Summary

One of the most successful methods used for game development is **agile development**[13]. It is based on iterative prototyping, a subset of software prototyping. Agile development depends on feedback and refinement of game's iterations with gradually increasing feature set. This method is effective because most projects do not start with a clear requirement outline.

The process to create a game is comprised of the stages described below. Some of the steps can be changed or left out. It is important to note that stages of development can occur in parallel. It is rare for one stage to be completed and untouched ever after, since changes and modifications occur until the last moment.

- **Stage 1: Design**

During this phase the general idea of the game is generated, along with the concept behind it. The initial **design document** is written referring to the plan the development team must follow and the basic information behind the concept of the game. At this point early game prototypes could be made; however the usual case is to have a playable prototype ready at a later stage. Concept ideas can be influenced by several sources. Books, history, mythology or even real world modern time influences can do the trick (Fig 2.6). Pokemon for instance, a very popular brand used in cartoon, video games and card games was Satoshi Tajiri's idea since younghood. He used to go to the surrounding countryside to collect bugs and insects as a hobby. In combination with his love for video games, he created Pokemon. There are many more examples where an idea like this was adjusted and perfectly fitted in the gaming world.

Fig 2.6 -Greek Mythology has always served as an inspiration to game designers.

- **Stage 2: Research**

During this stage the game concept is developed. Moreover, the developers choose which game engine they will utilize. Depending on the team, an already built game engine may be used (like Epic's UDK that is used for this thesis) or a new one can be built by the team's software engineers for several reasons: A commercial game engine may not offer what the game in progress demands or the team may not be able to afford buying a commercial one. In these cases, the best choice may be to build one from scratch. The genre of the game must also be considered in addition to the audience targeted. The genre includes one or more (a combination actually) of the following :

Action: Fast paced game with lots of combat and exploring. First or third person with inventory, usually single player. High reaction speed and good hand–eye coordination is needed. An example is Tomb Raider.

Adventure: In adventures the player assumes the role of a protagonist in an interactive story. Interactive media like films and novels have influenced this genre, resulting in a wide variety of literary genres. Almost all adventures are single

player, in order to focus the story around the protagonist. Exploration and puzzle-solving skills are required. An example is Monkey Island.

Puzzle: Puzzle games involves puzzle solving as the main mechanic of the game, usually involving colors, shapes and symbols in a logical and conceptual challenge. The puzzles in each game are oriented around a basic theme, like pattern recognition, logic or understanding a process. The difficulty rises and a time limit is common. An example is Lemmings.

Role Playing: Either by a party controlled by the player or by a single avatar, the player's purpose is to advance the avatar to more levels by gaining experience in order to achieve the main goal. Most often the player has to succeed in heroic deeds earning gold, items and experience during the process. Exploration and tactical combat in combination with a strong story setting are the main characteristics of RPGs which come from their ancestor, the pen and paper RPG. An example is Planescape Torment.

Shooter: The player owns one or more weapons or combat gadgets and must accompish an objective by killing his opponents directly or inderectly e.g. in Capture The Flag mode. Most of the times multiplayer gaming is supported. An example is Unreal.

Simulation: Simulation games try to simulate aspects of a real or fictional reality. Driving vehicles and playing sports are the usual themes, but nowdays more popular themes have emerged. Companies simulators give the player the chance to run a big company (Hospital, theme park and more), whereas Life simulators allow the player to run the life of the avatar. An example is Sims.

Strategy: This type requires skillful thinking and planning to achieve victory. Each move must bring the player closer to achieving victory since chance holds a smaller part. Each move may affect a number of factors such as resource management and warfare tactics. The player usually views the world from a godlike perspective to be able to manage his territory more efficiently. A combination of tactical and stragetic considerations is needed, in addition to economy management. An example is Civilization.

During the video games history different genres have been popular for a period of time, mostly depending on the available technology or one title making a great success with many similar titles following. Nowdays 30% of console games are Action themed whereas the 31% of computer games are strategy. Massive Multiplayer Online Role Playing Games (MMORPGs) also have an increasing popularity.

- **Stage 3: Prototyping**

Prototyping is the process of a fast and cheap implementation of an idea. It usually takes place before the design document is complete,however it is common to have prototypes in many stages to test various aspects of the game. For these reasons prototyping is thought to be an important process which is aided by various tools designed for this purpose. A succesfull development model is iterative prototyping, where design is refined based on current progress.

- **Stage 4: Gathering Material**

During this stage material is gathered in relation to the concept of the game in development. The material gathered will help inspire the team's work. Inspiration comes from the web as well as from other games, books, movies and real life. A sketchbook and a digital picture are usually handy when walking around in case of coming across to something interesting related to the game concept, whether it is a fancy texture or a haunted looking house (Fig 2.7).

Fig 2.7 -A digital Camera can be useful to capturing inspiring scenes from real life.

- **Stage 5: Characters Concept**

Characters are vey important in games since a well thought out character can easily be the trademark of a game, which in turn can depict the general concept of the game. Characters are also able to deliver the story such as actors in a movie do. The main difference is that designers create game characters from scratch. Characters include not only the protagonist, but also Non Player Characters (NPCs) and enemies. The main characters' back story is essential and must be in-line with the character's appearance. In a recent vote for the Greatest Game Villain of all time hosted in one of the biggest game sites, Kerrigan (Starcraft) and Darth Vader (Star Wars) made it to the last round, declaring Kerrigan the winner (Fig 2.8). The character's popularity was not only dependent on her character appearing in the video game series, but also to the StarCraft2 game being published recently, therefore Kerrigan's impact on players was fresh.

As a sidenote, while characters and a narration is not necessary depending soloely on the gameplay, investing resources on a story creation is certainly worths it, especially when a sequel game is planned. There are many examples of games based on stories prior known to the public mostly from written fiction that were very succesfull as video games. Such examples include Conan, Lord of the Rings, I Have No Mouth And I Must Scream, etc. From time to time, developers try to take advantage of popular hype and polular stories and turn them into a game. An

example of using an already existing story line and converting it to a game resulting in disatisfying success, was the adventure game Lost.

Fig 2.8 -Kerrigan, the Queen of Blades eliminated the Pac-Man Ghosts, Joker and Darth Vader, making her the greatest game villan of all time.

- **Stage 6: Environment Concept**

The environment's design can be as important as the character's design. The look and feel of the game is emitted by the environment and the challenges the landscape provides to the player. Atmospheric effects and light configurations can enhance the feeling the game is trying to convey. Fog, rain and darkness are very important for example, in a horror game. Moreover, knowledge of the psychological states of the players can affect the design of the space. Pleasant and meditative places can help the player relax, whereas tight and claustrophobic areas will impose an uneasy feeling.

- **Stage 7: Game Design Document**

This document does not add to the production process, but rather is a document that sums up the details. The **Game Design Document** can be done at any stage, as long as introductory information is available to the developers. It becomes the main point of reference of the ongoing development for every team involved. The story, genre, characters, mood of the game and core mechanics are all described in the GDD.

- **Stage 8: Level Design**

After the type, environment and core mechanics of the game have been decided, the levels can be created. When entering each level the player must fulfill a goal in

order to continue to the next level. To do so, each level's goal is divided to smaller objectives in order to provide a smoother flow to the player. This subdivision also allows the player to become more familiar with the game mechanisms. It is important that the level designers design the level to be fun, interesting and challenging, rewarding the player when needed. An interesting article of Valve's designers talking about the design decisions they had to make on their debut title is named "The Cabal: Valve's Design Process For Creating *Half-Life"* [14] (Fig 2.9). Their peculiar level design process gave many rewards and a Guinness record to the development team (Microsoft's ex-employees).

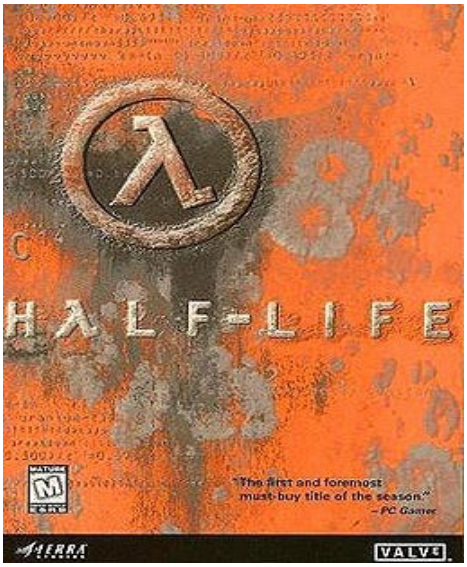

Fig 2.9 -The box art for Half-Life shows the title on a rusted orange background, below the Greek lamda letter.

- **Stage 9: Game Engine**

Game engines and game editors are at the designers' disposal in order to help materialise each level. Game Engines usually include an editor which offers an easy way to create the environement and the level events. Many times additional source code is required. Programming can start even from stage 1, after the initial desicions have been made such as how the movement will be done (WASD keys, point and click and more).

- **Stage 10:Artset**

Textures, models, animations and more visual requirements are created to fit the game world. Their creation can start as soon as the mood of the game is decided and the level design has been initiated (Fig 2.10).

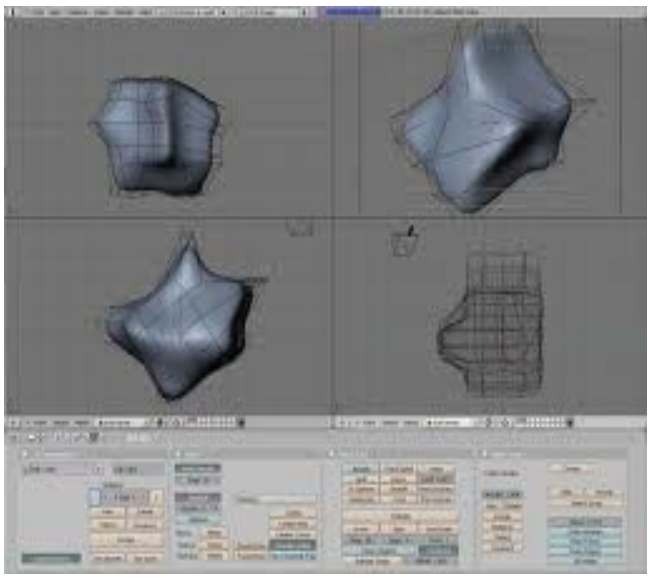

Fig 2.10 -Blender is a free 3D graphics application

- **Stage 11: Quality Assurance**

When all the stages are done, or a playable verion is avaialbe, the testing can begin, usually directed by professionals.

Chapter 3: Software and Tools

Several low-cost tools were used that assisted the implementation of the Thauma Project. These tools are the following:

3.1 Game Engine-UDK

Choosing the right game engine is crucial not only because it offers the tools and programming environment necessary for the game development but also because once a game engine is selected, considerable time should be available in order to learn its structure and tools. In the event a decision is made that a selected game engine is not suitable for the needs of a project, effort should be spent in identifying a different one which suits the projects needs. When looking for the game engine to implement the THAUMA game, various game engine options were investigated available on September 2009, based on specific requirements set:

- The engine's language should be C/C++ based. Although it is time consuming and difficult to write computer programs in C, it offers complete control along with vast amount of resouces and libraries, a fact which defines C as an idustry standard. Fast execution is another advantage to be considered. The author had previous experience with other suitable languages such as Python or Java but finding a game development environment in C for the reasons listed above was a priority;
- The game engine should be available at no cost for non commercial releases since there are effitient **free** game engines;
- There should be an active community, which also meant that the game engine's development should also be active. The official documentation usually is not enough and the experience of more experienced users is very useful to a new user;
- Support of 3D graphics was essential as well as network programming support;

The above were the main requirements. Being open source or being available for a long time (the more work done the better) was considered a plus. Having released AAA titles was a plus too. This filtering left out the following engines (some are rendering engines,however were considered to be an option as well) : GameCore, Torque, Unity, NeoAxis, Leadwerk, Irrlicht, Neoengine, Gosu, Panda3D , Plib, ZakEngine, C4, and GameMaker. The final choice was at the time, the Esenthel engine.

The Esenthel Engine is a complete game development suite allowing to create fully featured AAA titles. However, no AAA title implemented in Esenthel had been released at the time. Esenthel has been specifically designed for professional game development, which has been achieved by giving the developers full control over the game code mechanics, the next generation graphics core and a rich toolset

drastically simplifying the game development process. While mainly targeted for the professional market, the engine is easy enough to be used by independent teams or small companies with no prior game development experience.

At the time game engines were investigated as potential development environments for this thesis, Epic's UDK became free (November 2009). UDK is a complete professional development framework, with countless releases and dozens of rewards. UDK is designed mostly for professional teams. It was challenging for an one man project learn to deal with the tools that UDK offers, in order to deliver the best outcome. This practically means that the same game may be implemented by one person in another engine a lot faster than UDK, although the visual aspect may not be as good as UDK's. As far as the visual quality is concerned, nVidia has assisted Epic in writing the code for the visual aspects of the engine. As a final note, when one is aiming for the best result being able to invest time, UDK is the right choice. For the reasons analyzed above, the UDK was selected for this project.

UDK offers the following **features** [4]:

1. A Complete editing environment
2. Pure rendering power
3. State-of-the-Art animation
4. Powerful scripting
5. Real world physics
6. Eye-popping lighting and shadows
7. Gorgeous cinematics
8. Terrain
9. Built-in networking
10. Real-time shaders
11. Broad-audio support
12. Particle effects
13. Artificial intelligence
14. Distributed computing
15. Desctructible environments
16. Bink video codec
17. SpeedTree foliage editor
18. FaceFX facial animation

A new version of UDK is available every couple of months, improving several design flaws, bugs and upgrading or increasing the extra tools UDK offers (Fig 3.1). This project was finished with the March-2010 release. The subsequent versions have addresed several issues and include improvements. Some of the

flaws of the March version are:

- The editor uses a not so familiar system for the manipulation of geometry;
- There is a need for more tooltips at the editor's entities properties;
- The undo operation supports only a few undos;
- Timers do not support function calling from any class;

Fig 3.1 -Free from November '09

The following information is necessary before programming with UDK. There are also some source code examples where needed, that were written for this project. This chapter works as an intro for UDK and should in no way replace the official UDN documentation which is written by the developers.

3.1.1 Design Goals of UnrealScript

Unrealscript was designed to provide the developers with a powerful, built-in programming language that maps the needs of game programming.

The major design goals of UnrealScript are:

- Enabling time, state and network programming, which traditional programming languages do not address but are needed in game programming. C/C++ deals with AI and game logic programming with events which are dependant on aspects of the object's state. This results in long-length code that is hard to maintain and debug. UnrealScript includes native support for time, state and network programming which not only simplifies game programming, but also results in low execution time, due to the native code written in C/C++;
- Programming simplicity, object-orientation and compile-time error checking, helpful attributes met in Java are also met in UnrealScript. More specifically, deriving from Java UnrealScript offers:
    - 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;

Often design trade-offs had to be made, choosing between execution speed and development simplicity. Execution speed was then sacrificed , since all the native code in UnrealScript is written in C/C++ where performance outweighs the added complexity. UnrealScript has very slow execution speed compared to C, however since a large portion of the engine's native code is in C, only the 10%-20% of code in UnrealScript that is executed when called has low performance.

3.1.1.1 The Unreal Virtual Machine

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

The Unreal server controls all the gameplay and interaction between players and actors (placable entities). A listen server is able to host both a game and a client on the same computer, whereas the dedicated server allows a host to run on the computer with no client. All players connect to this machine and are considered clients.

The gameplay takes place inside a level, containing geometry actors and players. Many levels can be running simultaneously, each being independent and shielded from the other. This helps in cases where pre-rendered levels need to be fast-loaded one after another.

Every actor on a map can be either player control or under script control. The Script controls the actor's movement, behaviour and interaction with other actors. Actor's control can change in game from player to script and vice versa.

Time managemenet is done by dividing each time second of gameplay into Ticks. Each tick is only limited by CPU power and typically lasts $1/100^{th}$ of a second. Functions that manage time are really helpful for gameplay design. Latent functions like Sleep, MoveTo and more can not be called from within a function, only within a state however.

When latent functions are executing in an actor, the actor's state execution does not continue until the latent functions is completed. However, other actors may call functions from the actor that handles the latent function. The result is that all functions can be called, even with latent functions pending.

In UnrealScript, every actor is as if executed on its own thread. Windows threads are not efficient in handling thousands at once, so UnrealScript simulates threads instead. This means that 100 spawned actors will be executed independently of each other each Tick.

3.1.1.2 Object Hierarchy

UnrealScript is purely object-oriented and comprises of a a well-defined object

model with support for high level object-oriented concepts such as serialization and polymorphism. This design differs from the monolithical one that classic games adopted having their major functionality hardcoded and unexpandable at the object level. Before working with UnrealScript, understanding the object's hierarchy within Unreal is crucial in relation to the programming part.

The five main classes one should start with are *Object*, *Actor*, *Pawn*, *Player*, and *GameInfo*.

Object is the parent class of all objects in Unreal. All of the functions in the *Object* class are accessible everywhere, because everything derives from *Object*. *Object* is an abstract base class, in that it 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*) is the parent class of all standalone game objects in Unreal. The *Actor* class contains all of the functionality needed for an actor to move around, interact with other actors, affect the environment and complete other useful game-related actions.

Pawn (extends *Actor*) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.

Player (extends *Actor*) is the class that defines the logic of the pawn. If pawn resembles the body, *Player* is the brain commanding the body. Timers and executable functions can be called from this type of class.

GameInfo is the class that sets the rules of gameplay. Players joining will be handled in this class which and treated as programmed.

3.1.1.3 Timers

Timers are a mechanism used for scheduling an event to occur. Time management is important both for gameplay issues and for programming tricks. All Actors can have more than one timers implemented as an array of structs. The native code involving timers is written in C++, so using many timers per tick is safe, unless hundreds expire simultaneously because they execute UnrealScript when activeted. The following function starts a timer counting the cooldown until the Speed skill can be used again. MakeAvailableSpeed() function is located in the Player class.

```
function SkillCooldown(ThaumaPlayer SkillUser)
{
        SkillUser.SetTimer(Cooldown,FALSE,'MakeAvailableSpeed');
}
```

3.1.1.4 States

States are known from Hardware engineering, where it is common to see finite state machines managing the behaviour of a complex object. The same management is needed in game programming, specially when dealing with AI. The

usual case to implement states in C/C++ is to include many switch cases based on the object's state. This method, however, is not efficient, since the complexity in game AI requires many states, which results to code difficult to write and update.

UnrealScript supports states at the language level. Each actor can include many different states, and only one can be active. The state the actor is in reflects the actions it wants to perform. Attacking, Wandering, Diying are some states the pawns have.

Each state can have several functions, which can be the same as another state's functions. However only the functions in the active state can be called. For example, say you are writing a monster script, and you are contemplating how to handle the SeePlayer() function. When wandering around, one wants to attack the player one sees. When attacking the player, you want 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.

There are two major benefits to states and one complication:

- Benefit: States provide a simple way to write state-specific functions, so that one can handle the same function in different ways, depending on what the actor is doing.
- Benefit: With a state, one can write special "state code", using the entire regular UnrealScript commands plus several special functions known as "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. Time-based programming is enabled which is a major benefit that neither C, C++, nor Java offer. Namely, code can be written in the same way one conceptualizes it; for example, e a script can be written which supports the action of "open this door; pause 2 seconds; play this sound effect; open that door; release that monster and have it attack the player". This may be done with simple, linear code and the Unreal engine takes care of the details of managing the time-based execution of the code.
- Complication: Now that functions (such as Touch) override multiple states as well as in child classes, the programmer should figure out exactly which Touch() function is going to be called in a specific situation. UnrealScript provides rules which clearly delineate this process, however it is something to be aware of if creating complex hierarchies of classes and states.

The following function stops the timer used for the regenerating health skill everytime the player is in the state DEAD. If the player is in another state (attacking, running etc) he regenerates health as intended.

```
function Regeneration()
{
        if (self.IsInState('DEAD'))
                ClearTimer('Regeneration');
        else{
                if(Pawn.Health< Pawn.HealthMax && Regen.RegenCounter<10)
```

```
                {
                        Pawn.Health = Min(Pawn.HealthMax, Pawn.Health +
Regen.RegenRate);
                        Regen.RegenCounter++;
                }
        }
}
```

3.1.1.5 Interfaces

UnrealScript has support for interface classes that resembles much of the Java implementation. As with other programming languages, interfaces may only contain function declarations and no function bodies. The implementation for these declared methods must be done in the class that actually implements the interface. All function types are allowed and also events. Even delegates may be defined in interfaces.

An interface may only contain declarations which do not affect the memory layout of the class - enums, structs and consts may be declared, not variables however.

3.1.1.6 Delegates

Delegates are a reference to a function within an instance. Delegates are a combination of two programming concepts, e.g. functions and variables. In a way, delegates are like variables in that they hold a value and can be changed during runtime. In the case of delegates though, that value is another function declared within a class. Delegates also behave like functions in that they can be executed. It is this combination of variables and functions that makes delegates such a powerful tool under the right circumstances. On the following example, each skill has its own buff, buff meaning the positive power directed to the player. This means that each skill has its own declaration. These functions are declared on the *ThaumaAbilities* Class, which is the mother class of all abilities. The delegate here may call any function from any skill, dynamically.

```
delegate BuffCall( ThaumaPlayer SkillUser)
{
        //each skill has its own buff
}


function SkillCall(ThaumaAbilities Skill, ThaumaPlayer SkillUser)
{
        //check if player is dead or without a pawn
        if(SkillUser.Pawn==NONE || SkillUser.Pawn.Health <1 )
                `log(">>Abilities:No Valid Player");
        else
        {
                BuffCall = Skill.SkillBuff;        //call the skillbuff funtion of whatever ability is
needed
                TimerCall = Skill.SkillCooldown;    //call the timer of whatever ability is needed

                if (Skill.bIsAvailable)
```

```
                {
                        if(ThaumaPawn(SkillUser.Pawn).EnergyManagement(Skill.EnergyCost))
                        {
                                BuffCall(SkillUser);
                                Skill.bIsAvailable = FALSE;
                                TimerCall(SkillUser);
                        }

                }
                else
                {
                        SkillUser.ClientMessage("I can not use : "$Skill.AbilityName $", yet!");
                }
        }
}
```

3.1.1.7 UnrealScript Compiler

The UnrealScript compiler is three-pass. Unlike C++, UnrealScript is compiled in three distinct passes. In the first pass, variable, struct, enum, const, state and function declarations are parsed and remembered, e.g. the skeleton of each class is built. In the second pass, the script code is compiled to byte codes. This enables complex script hierarchies with circular dependencies to be completely compiled and linked in two passes, without a separate link phase. The third phase parses and imports default properties for the class using the values specified in the default properties block in the .uc file.

3.1.1.8 UnrealScript Programming Strategy

1. UnrealScript is a slow programming language when compared to C/C++. A program in UnrealScript runs about 20x slower than C. However, script programms written are executed only 5-10% of the time with the rest of the 90%-95% being handled in the native coden written in C/C++. This means that only the 'interesting' events will be handled in UnrealScript. For example, when writing a projectile script, you typically write a HitWall(), Bounce() and Touch() function describing what to do when key events happen. Thus 95% of the time, your projectile script isn't executing any code and is just waiting for the physics code to notify it of an event. This is inherently very efficient.
2. One should keep an eye on the Unreal log while testing scripts. The UnrealScript runtime often generates useful warnings in the log that notifies the programmer of non fatal problems that are occuring.
3. UnrealScript's object-oriented capabilities should be exploited as much as possible. Creating new functionality by overriding existing functions and states leads to clean code that is easy to modify and easy to integrate with other peoples' work. Traditional C techniques should be avoided, like writing a switch() statement based on the class of an actor or the state because code like this tends to break as you add new classes and modify states.

3.1.2 Network Overview

The UnrealScript's method for multiplayer gaming is based on shared reality. The players see the same actors and events from different viewpoints. Most of the actors will not change state in any way, so the clients only need to know their initial state. The server inform the players for the rest of the actors and events that every player needs to know.

When network capabilities (on-line gaming) are implemented, the best route is to start the implementation from the beggining working on the multiplayer along with the other features. Retrofitting a solution is hard and design decisions that make a lot of sense when networking is not considered; such as splitting functionality across many objects, may cause significant issues when multi-player gaming is implemented at a later stage.

3.1.2.1 Server's Authority

The server's game state is completely defined by the set of all variables of all actors within a level. Because the server is authoritative about the gameplay flow, the server's game state can always be regarded as the one true game state. The version of the game state on client machines should always be regarded as an approximation subject to many different kinds of deviations from the server's game state. Actors that exist on the client machine should be considered proxies because they are a temporary, approximate representation of an object rather than the object itself.

3.1.2.2 Replication

Unreal views the general problem of "coordinating a reasonable approximation of a shared reality between the server and clients" as a problem of "replication". That is, a problem of determining a set of data and commands that flow between the client and server in order to achieve that approximate shared reality.

3.1.2.3 Simulated Functions and States

Simulated functions on the client side refer to the proxy Actors, which are copies of the Actors created by the server. This is the only way for clients to control over which functions should be executed on proxy actors. Non simulated functions/states are not called that way. These proxy Actors are often moving around using client-side physics and affecting the environment, so at any time their functions can potentially be called.

3.2 IDE-Microsoft Visual Studio

Microsoft Visual Studio is an integrated development environment (**IDE**) from Microsoft. It can be used to develop console and graphical user interface

applications along with Windows forms applications, web applications and more.

Visual Studio includes a code editor supporting IntelliSense as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. It accepts plug-ins that enhance the functionality at almost every level.

Microsoft provides "Express" editions of its Visual Studio 2010 components Visual Basic, Visual C#, Visual C++, and Visual Web Developer **at no cost**. Visual Studio 2010, 2008 and 2005 Professional Editions, along with language-specific versions (Visual Basic, C++, C#, J#) of Visual Studio 2005 are available for free to students as downloads via Microsoft's DreamSpark program.

The Express 2008 Edition was used for this project inclusive of a module supporting the Unreal Scpript language. Using nFringe allows IntelliSense to automatically detects incorrect syntax[11]. However, debugging features do not work with UDK, since nFringe was not built exclusively for Unreal Script.

3.3 Materials

In addition to software tools, there were certain useful items that helped during the design and testing stages. These tools (or equivalent) while not mandatory are recommended and are always used in large scale projects.

- **WhiteBoard** is a replacement of the mood board for large projects. Gathering material relevant to the game's concept in one place helps the developers get in the game's mood and think accordingly. Ideas, plans and resources can be written on the whiteboard and by placing him in line of the working place's sight can serve as a project's cache. A 45x60 cm can be bought for 20€.
- **Board game's materials** will help build the board prototype. A board prototype is better visually and depending on the genre of the game may be overall better than other non-software prototypes. When gameplay mechanics are available, "cheap and dirty" prototypes, as they are called, should be ready to launch (Fig 3.2).
- A digital **camera** will prove handy when walking outside. A scene, a texture, a person or a situation from real life can trigger your imagination and be the source of inspiration for the work to be done.
- Finally the simplest note taking inventory, a pen and a **notebook** should be at your disposal all the time. Brainstorming is not programmable, and a good idea can come up when you least expect it. It is not coincidence that several important inventions or great ideas were first written on a papernakin.

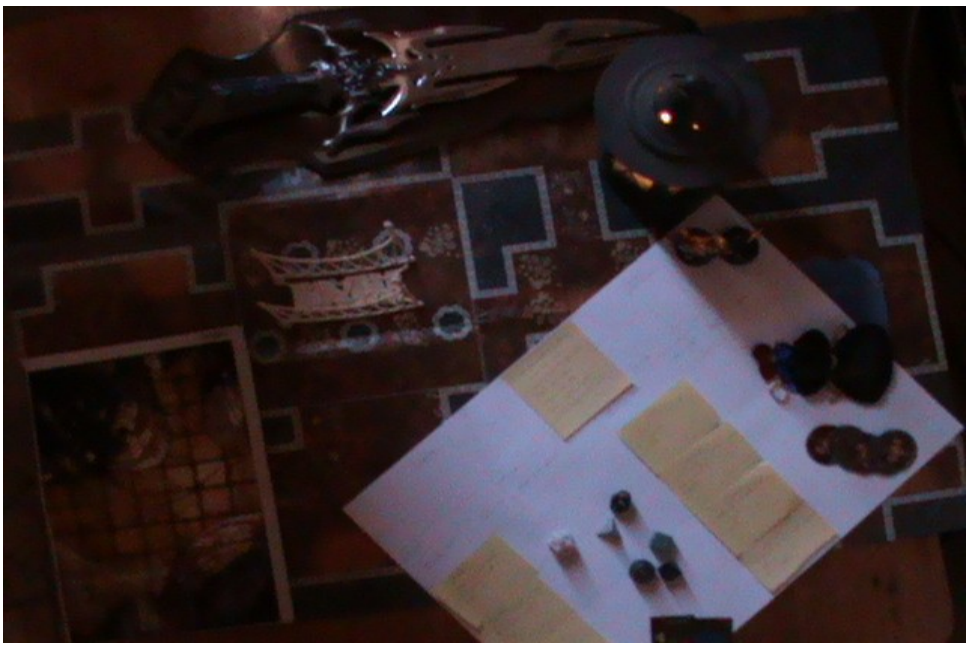

Fig 3.2 –Materials from various game types will help build a prototype.

Chapter 4: Game Design

The stages followed in this project are according to the ones professionals adopt for developing a game. Adjustments had to be made due to the developer's team size. The stages followed and subsequently detailed are the following:

Stage 1: TimeTable

The first objective was to organise the time schedule for the project in order to set milestones which lead to an easier management of the different tasks. The time available was **12 months** and the tasks were too many for this tight timeline. Careful filtering had to be made, depending on the importance of the task and the relation to the computer science field.

Stage2: The Concept

A story was obligatory to aid the player's immersion to the game world. The academic work on that subject is rich, since movies and books have examined immersion and interactive stories since long ago.

Stage3: Tools

At this point a search for the tools that would be used for the project had to be made. The search included various popular and not-so-popular game engines as detailed in Chapter 2.

Stage4: Game Design

A gameplay fitting to the story concept had to be invented in order to provide entertainement to the player via the game mechanics. Creativity, strategy and a bit of unexpectancy were the parameters that were of importance. These characteristics are expressed mainly with the skills, the combat system involving energy management and the mini events of the game. Books, guides and more importantly gaming experience were the inspirations for the gameplay.

Stage5: Prototype

The ideas derived from the game design process could not be instantly implemented and various versions were on the table not only for the skills but also for the events. A **board prototype** was created to aid choosing and modifying decisions made in the previous stages.

Stage6: Programming

Gameplay, Networking, UI and Scripting were programmed on the core of UDK. Gameplay mainly concerns the skills the players can use, the life, armor and energy management along with setting the base for future improvements, such as building

a skill tree. Network was implemeted on a higher level due to the engine's core source code with the connection being managed by UDK, using the UDP protocol. A basic UI was designed in order to provide immediate feedback to the player's actions. Finally scripting was needed first and foremost during the level design to set and manage the various events.

Stage7: Artwork

The game cover was created by an artist volunteer. Similar guidelines were given for the intro and the soundtrack developed by the same artist and a musician volunteer, in order to make the project as complete as possible. Finally, a video showing the logo of the game was developed by a 3D graphics expert [16].

Stage8: Testing

Alpha and beta versions of the project were released to gamers providing feedback not only for bugs, but also concerning the entertainment factors and possible improvemets.

As a final note, these eight Stages sum up the process of making the Thauma Project, however not in a linear way. During each stage the previous and in some occasions the next ones were also revised.

4.1 TimeTable

A timetable is necessary in software development and this project is not an exception. The need to set strict milestones emerged from the beggining when the estimated time needed to work on every aspect of the game development was calculated to be higher than expected. The method used was the **agile development**, meaning that there was an iteration during each step redifining the previous part and affecting the next ones. There were major changes happening during the process on many stages, which is natural since new ideas were on the table quite often.

Another important issue concerning time was how much of that would be spent in each subject involved. For that reason some decisions had to be made beforehand, concerning the portion of time each element would be allocated. GamePlay or Story (Fig 4.1)? Core mechanins or Interactivity (Fig 4.2)? The following figures depict the percentage of time the project's elements would take.

Finally, learning the game engine required a considerable amount of time, especially because UDK is designed for complicated projects.

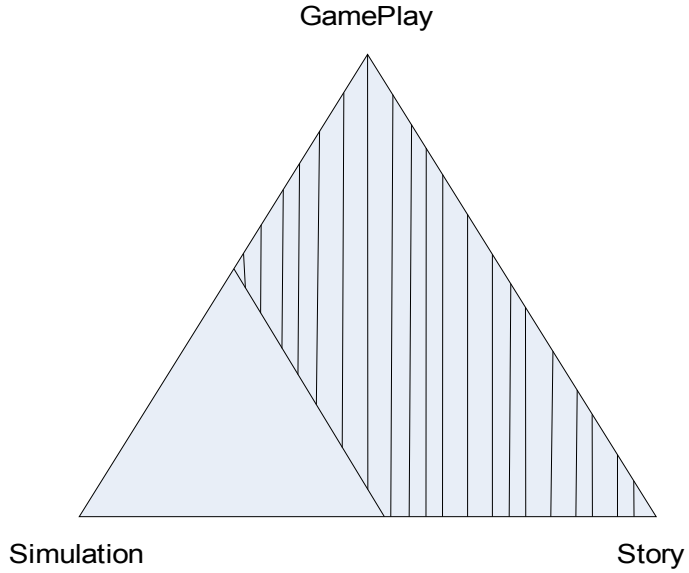

Fig 4.1 –Focus of Thauma.

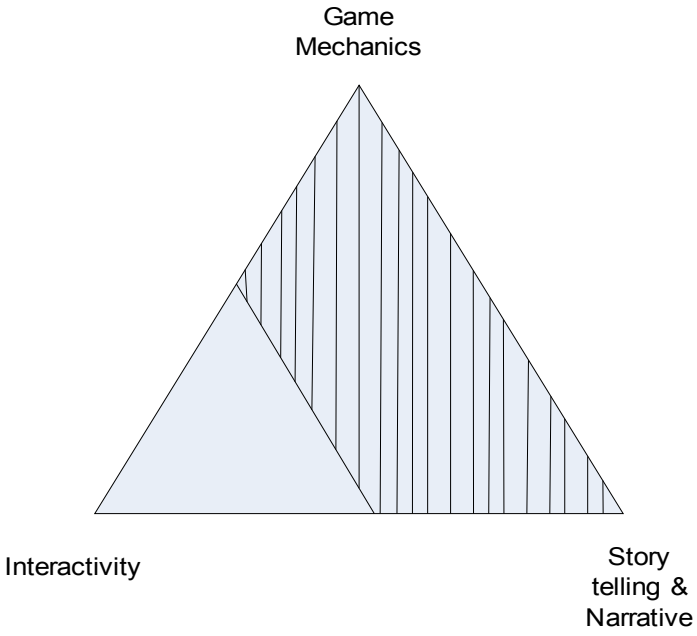

Fig 4.2 –Ingredients of Thauma.

The next step was to set the game goals in a linear way and estimate the time needed for their completion. Completing one step in order to proceed to the following was not possible as stated before. The goals set for one year were the following :

- Design the main **theme** and come up with a story.
- Gather **material** based on the concept.
- Depending on the theme and the story, **define** the game (genre, basic details).
- Create the game **mechanics**.

- Test the mechanics with the help of **prototypes**.
- Initialise the **programming** part.
- Add **artwork**.

Most of the time would be used for Programming, and less for the concept and the gameplay. The game engine learning process was also a demanding one. Designing the artwork's prerequisites would be done in any phase's spare time.

4.2 THAUMA Story

Since this is a one-man's project, it is only natural that the developer's likes and dislikes will effect everything in the game. In another case (commercial one), the trend of current games may be an important factor.

4.2.1 The Concept

Tolkien wrote "To ask what is the origin of stories, is to ask what is the origin of language and of the mind." **Fantasy** has always been in man's reality and games can immerse the player into that fantasy. Kids dream of being a knight or a ninja. Games can make that dream come true and that's how immersion begins. The experience offered by games add to this immersion, which many times surpasses that of alternative but traditional mediums, like books and movies. While in books the reader plays the part of the observer, in games the player may play the part of the observer, the protagonist or even the director. In addition to this, the story in games can be non linear, as opposed to other interactive means. The story structure of an adventure game is usually not single-threaded and the player can experience each ending, depending on the choices he made during the game, much like real life.

Despite the differences between the game narratives and traditional narratives, the similarities in creating each are many. Ella Tallyn in her PhD titled "Applying Narrative Theory to the Design of Digital Interactive Narratives with the Aim of Understanding and Enhancing Participant Engagement" compares the structures of narratives and games[3]. She concludes that there are inherent problems in emotionally engaging protagonists using the existing interactive techniques. The main problem, according to her, is the audience interaction that interrupts the author’s ability to structure a plot precisely, and as a result, many narrative techniques for promoting engagement are ineffective. The goal centred approach of computer games however, has been the most successful.

A well established goal will not only drive the player to the end, but will also add meaning to the game. The storytelling characteristics can provide the game designer with the following:

1. **Immersion**: The player controls what happens;

2. **Plot**: Plot can provide the player with surprises;
3. **Characters**: Characters can bring the story to life. Players "become" the protagonist, rather than playing him;
4. **Setting**: The world of the game depends on the story of the protagonist;
5. **Goals and Motives**: they Define what the player tries to achieve;
6. **Obstacles and Challenges**: Either physical or puzzles, riddles;
7. **Rewards and Penalties**: They can be a score or a new skill/weapon;
8. **Emotions**: They can make a game richer, more memorable;
9. **Urgency**: The player must succed at something before time runs out.

Horror, **survival** and action were the chosen characteristics that are met frequently in games, movies and other media in order to trigger one's imagination. Games such as ASC's Sanitarium (Fig 4.3), CAPCOM's Resident Evil series, Infograme's Alone in the Dark series were engraved in the players' memory in relation to the feelings and the atmosphere they would experience playing these games. Several more horror-themed sources are H.P.Lovecraft's **Cthulhu** Mythos, Jacob's Ladder (Bruce Joel Rubin Movie,1990), Monster (manga from Naoki Urasawa) etc. The story of **Sanitarium** was considered to be a good starting point in relation to the final year project's story. The following abstract is part of the story design document, included at the appendix.

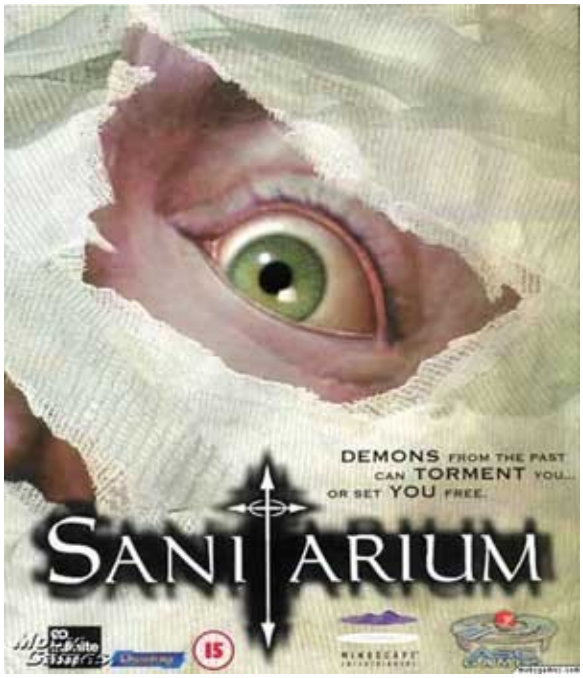

Fig 4.3 -Sanitarium is a point-and-click adventure game released in 1998 by ASC Games.

'This game (Thauma) is about a man who experienced a traumatic event at a young age, but has managed to leave it behind and go on with his life. At some point in his present life, he has to confront his past painful experiences and start a battle with

himself. Memories emerge like ghosts from the past, and each has a question that needs to be answered, in order to find peace...Guiltiness, anxiety, depression and moral dillemas are the burden the hero has to carry till he reaches the end in his mind's maze. When his journey comes to an end, will he find catharsis or will the opening of his mind's hidden locks lead him to another purgatory?'

During his journey his dual personality disorder emerged, leading to the awakening of his evil side. This evil side depicted by Dark Jonathan incarnates the negative emotions Jonathan feels during his adventure. The origin of this evil self is the guiltiness he felt after his brother died, asking for his favorite comic book. Jonathan had secretly taken the book in order to repair it and give it back to him in time. On the verge of a psychological collapse, Jonathan's parents visited a doctor and managed to hide this guiltiness inside Jonathan's subconciousness. The game starts when the subconciousncess reveals memories that should remain hidden. The battle in Thauma T.v takes place in Jonathan's subconciousness, opposing his evil self. The outcome of this battle is symbolic, however it depicts the hero's current psychosynthesis. If he manages to win the battle against his evil self, he will have succeeded in finding peace with himself, discarding his dual personality disorder once and for all. If he loses his dark side overcomes his good side. Being overun with guiltiness and depression, the protagonist loses his mind. Finally, in the case that there is no winner, then his state of mind is unstable, being sane but guilty and depressed.

Game-wise the battle consists of 4 phases, depicting Jonathan's unstable mind. Each side has 4 powers (skills) that can be used spending psychic energy. This energy comes from Jonathan's mental strength, and cannot overcome a certain maximum value (100). However it regenerates, since he is still sane during this battle. The skills Dark Jonathan can use are of evil nature, able to cause damage and control Jonathan, while Jonathan has defensive skills at his disposal. The skills' system is described in detail at the following section. Using these skills effitiently will aid the player in killing the opponent which in turn grants points. A point system is used to determine the winner. Points can be obtained in two different ways, either by killing the opponent, or by participating in the events. Due to the replayable nature of Thauma, the events may be anticipated after the first game. This is not a problem, since careful planning was meant to be required before each event. The random pick of the second and third event add a random variable that will have to be taken into account as well.

4.2.1.1 The Characters

In games there are two types of characters; the one the player controls, and the non-player characters (NPC's). Characters must be developed carefully, provided with a background story and a good visual design supporting it. Some important facts are that people have a tendency to see attractive people as more intelligent, moral and sociable than unattractive people. On the other hand, people that have more baby face features are considered more naive, helpless, innocent and honest than those with mature features. Finally, characters should have depth which is revealed throughout the story. Characters' growth is very important to the player, who will find the protagonist static and unrealistic if he does not evolve during the game.

This evolution may involve personal characteristics, such as not trusting a person that lied to him in the past, abilities, which is the more usual case, or any other form.

Certain techniques improve the player's relationship with the protagonist, such as the first person camera, used mostly in FPS games. Moreover, cinematic videos can show characteristics through dialogues or events that add to the character's prior created personality. Therefore, the advancement can occure via dialogues options the player chooses. The future became ominent for dialogue based games, like adventures, after the creation of Facade [15].

What is the player feeling as he/she play? What motivates him/her and what is the relationship to the central character? The most important feature is the characters the story provides the game with. Well-developed characters will enhance the bond between the story and the player. This means that it will be easier for the nine characteristics to be applied during his/her gaming session. On the other hand, weak characters or characters the player shows no interest at all, may easily be the reason to ignore the game. Even if the character is unimportant to the gameplay, if he/she is well developed he/she can be highlighted. Mario, for example, from the Super Mario Bros series , is a plumber with a blue outfit, a red hat, a big nose and a moustache (Fig 4.4).

Fig 4.4 -Would this Mario be as popular as the current one?

This description is almost irrelevant to the gameplay (except maybe the fact that he can go through pipes because he is a plumber) and even if he was tall, blond with a toxedo, the gameplay would be exactly the same. However, his creation as it was, helped promote the game and he is one of the most popular characters in games history. For this project, inspiration was drawn from real-life characters (please check the Story Design Document at the appendix). A short description of the protagonist of Thauma follows:

Jonathan: The tragic protagonist in this hero's journey. Jonathan was born in 1983

from middle-class parents. He and his brother, Jacob lived a happy life until Jacob got sick with a deadly disease. Jonathan's failure to fulfill Jacob's last wish caused a deep travma, which healed after a psychologist intervened. Now Jonathan is 26 years old and the demolition of his parents' mansion will make him confront these past events.

4.3 THAUMA GamePlay

The THAUMA gameplay consists of the challenges the game offers to the player in addition to the actions the player can take to overcome these challenges. Rules, balance issues, victory conditions are all part of gameplay. For this project, almost every part of the gameplay mechanics were built from scratch. This includes the skills the players can use, the resources available to player, the level design, the events and the victory condition. The challenges the players face test their physical abilities, resources management, skills and creativity.

4.3.1 THAUMA Challenges

The **physical abilities** tested in THAUMA are the typical for FPS games. Quick reflexes, physical coordination, speed timing, accuracy and precision are all needed, based on the action side of the project. More specifically, by killing the opponent the players can increase their score, and in order to do so each player must be superior from the opponents in relation to the characteristics mentioned above.

Resource management is involved mainly due to the Energy constant that is included in the THAUMA game. Energy is used on all skills and even the default attack consumes one point of energy. It represents Jonathan's mental strength. The careless use of skills will most likely result in a large consumption of energy, which in turn means that some skills will not be available when the player needs them. Skills are important because they boost the player's powers aiding the player in killing his/her opponent or surviving for that matter. An additional attribute of resource management is the cooldown (cd) of the skills. Each skill has a cooldown, which prevents it from being used again until that cooldown passses. Once again, careless use of the skills can prove fatal, since using speed in order to move faster while not in combat, will also mean that the player will not be able to use it to escape when engaged in combat. Health and Armor need management, however not immediate since the other player can affect their values too (Table 4.1).

Resource	**Min. Value**	**Max. Value**	**Special**
Health	1	150	0 = Death
Armor	0	100	Absorbs Damage
Energy	0	100	Regen 5/2sec

Table 4.1 –Player stats.

Lastly, **creativity** may be expressed mostly through the skills. Using the landscape in combination with the skills provided may provide a lot of help to the player. Skills must support that kind of creativity. The multi-purpose skills enhance the ways the player can play the game, which is desired (Table 4.2).

SkillName	Cooldown	Cost	Duration	Ability	Player
Speed	8	25	3	Haste	A,B
Invisibility	16	35	2	Ethereal	A,B
Armor	12	20	Instant	Armor	A
Regeneration	36	35	8	+40 Health over time	A
Stun	16	35	2	Stuns Enemy	B
Sacrifice	30	40	Instant	AOE damage	B

Table 4.2 –Player Skills.

Time is of great importance in games and the use of time can be a precious tool for the designer. In the THAUMA game, time equals stress and anxiety. A countdown is always visible to the players and it signifies two distinct timestamps: One is the beggining of an event. Every 90 seconds an event initializes which can provide the players with additional points, either by providing the circumstances for combat engagement or by giving points when an certain action is done by a player. Note that the second event is randomly chosen between B and C. The other purpose is to inform the player that there is time left before the end of the game. When time passes, the player with the highest score wins. Since points from killings matter a lot, even a few seoncds can be enough for the player to change the flow of the game and win. This thought adds to the anxiety of the game. Another stress factor is the choice of the right skill. A few seconds are not enough to distinguish which is the right one for the situation.

One more critical topic concerning gameplay is **balance**. Balance is a factor that when addressed successfully, a great game can be derived. Starcraft is known for its balanced gameplay among others as well as DoTA (a mod for Warcraft 3). Both games are very popular among their kind. A game can be considered balanced, when all sides available to the player are equal in power and when a high-skilled player has an advantage over a low skilled one. This can be achieved with various ways and most of them work just fine. The classic rock, paper, scissor method is a situation involving three sides, A, B, C, with A winning B, B winning C, C winning A. This of course is not a deterministic result in games, otherwise it would not be fun. Luck is affecting these relationships, however, the base is as desrcibed. Another method is to give all sides the same tools. This means that both the players and their opponents have the same skills, the same cooldowns, etc. One is a copy of the other. Only luck and skill matters in that case, however personal differences between players cannot be expressed which means that choosing a side can only be

based on the story elements of the game. Finally, another approach is to give players different powers/tools which are of equal strength. Victory then depends highly on strategy and tactics and the choices the player makes when facing an opponent.

Taken all these into consideration, the skills of Thauma were decided not to be the same for the players. The skills are described in the Story Design Document. To sum up, there are two skills of the same kind available to both players as well as two skills that differ between them. Perfect balance is a difficult achievement and this is not a game with perfect balance. The more the game is tested the more balance can be achieved. This involves professional players for commercial projects and design experience. Another option was to give all 6 skills to both players, so that the game would be balanced but that desicion created a conflict with the story elements of the game, and was set aside. The Good side of Jonathan (the protagonist) has defensive skills, whereas Dark Jonathan (Jonathan's Dual Personality Disorder) has offensive skills due to his dark nature.

4.3.2 THAUMA Level Design

Level Design is another part of gameplay. As mentioned earlier, the level is a medium to following the story and can also be used as a modifier to the player's experience. Levels may be defined as the chapters of a book. Sometimes they may occur sequentially, however this is not necessary. Depending on the genre, the level design can be more or less complicated and can serve multiple purposes. In a football simulator for instance, a plain field serves as the level. In Strategy games however, taking over a hill may mean the victory after a long-houred battle. There are certain points to consider when designing a level:

1. Its function in the game, such as providing new challenges;
2. The setting of the level, such as a deserted village in a post nuclear world;
3. The layout and the paths available to the player;
4. The main goal of this level, such as killing the dragon to save the princess;
5. The major challenges and the walkthrough;
6. The initial conditions, such as straight from a shipwreck with an empty inventory (Conan);
7. The narrative elements of this level and the flow of the story;
8. The trigger points and the actions needed to activate them;
9. The level's mood and atmosphere, such as a foggy haunted city (Silent Hill);
10. The termination conditions.

Apart from these points, there are several methods to adopt in order to succesfully design a level. A level is successful when it serves the purposes it was built for. Showing the player an NPC (Non player character) that walks into the dark water and gets massacred by a creature will give signs to the player about which way he/she can move in order to successfully complete the challenge. Moreover, the level's goals should always be clear to the player and a reward for completing is important especially when it is communicated from the start.

The first levels of Thauma are designed only roughly and are described in the SDD. This project's level was designed to be a fantastic world similar to a brain. The last fight happens in Jonathan's mind so this place would be ideal. As written before, no assets were created for this project, which means the graphics do not resemble a brain. However, apart from the visual entities, the level had to be small, circular in shape with a topology that favors the use of skills and the combat in general, therefore resembling the overall shape and idea of a brain. Speed, Invisibility (with ability to pass through the walls), Create Obstacle and Combat were the main parameters for designing the level in order to provide for creative thinking on how to use them. The Create Obstacle skill could be used best in narrow long corridors, in addition with the Invisibility skill. The aim was to be able to turn the tide of the combat by using this abilities well. Finally, wide areas in between the corridors would permit the use of events and combat, with players showing their fighting skills (Fig 4.5).

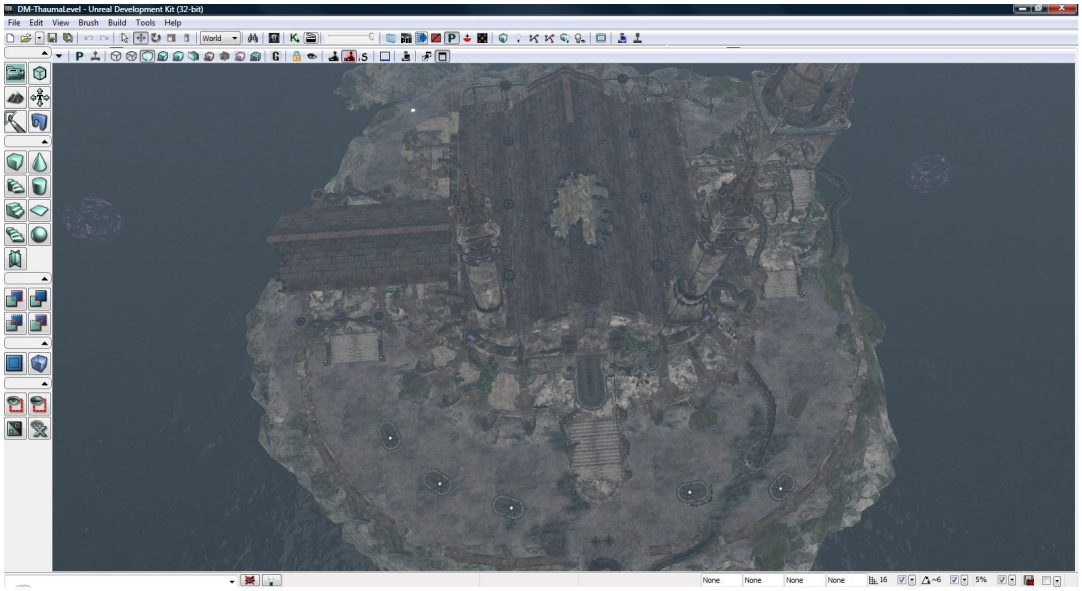

Fig 4.5 –The Level used for thauma at its final form. (DM-Sanctuary mod)

4.3.2.1 THAUMA Events

The **Events**'s purpose is to add excitement and fun to the game. The use of skills alone, while unusual and interesting as an implementation to a FPS, do not support the replayability of the game. They resemble weapons in the classic FPS games and may be boring after a couple of games. This gap can be filled with the events. The Event give the players an aim for a little while, however it is enough to keep their thought focused on achieving it. Each Thauma game lasts 6 minutes while an event is happening every 90 seconds. However, it will propably be raised to 8 minutes in subsequent versions while events will occur every 2 minutes (Table 4.3).

The first event actually marks the beggining of the game, offering extra points for each killing. It serves as a tutorial, without demanding anything specific from the player, in order to let him get accustomed to the Interface, Skills, etc.

The second event is a random choice between two, let's name them B and C. During B, the player has to step on a stone on a marked spot on the center of the map, in order to get extra points (Fig 4.6). By doing so, he also has a chance to get a buff or a debuff. Buff is the positive effect, while debuff is the negative. A battle for territory control starts at that point, with the stepping stone being the key. Will the player risk and step on it, while being in combat with the other player, or will the player let the opponent risk while trying to kill him in the meanwhile? The buffs/debuffs may turn the tide of the combat, with the random factor adding to the unpredictability which adds to excitement. One variation in mind that did not manage to be implemented due to time constraints, was to make the stepping stone random too. This action would mean that despite taking all the previous actions into consideration, the player should be aware for the randomly chosen stone.

Event C includes a teleportation of the two players to the same spot, stunning them for a few seconds and realeasing them afterwards for battle. Generally, Thauma is a fast-paced game and in combination with the kills and the players' skill, it may not be easy to kill the opponent. This Event acts like a reset, engaging the players in combat. It worths mentioning here that after a player's death, the player re-spawns at one of the two spawning points.

Finally, Event D is pre-determined and acts like a countdown for the whole game. This event changes the rules of the game a bit. The first of the three changes is related to health. Both Players' health instantly rises to 150, while a count down begins and certain health score is lost every 2 seconds. This means that by the end of the game (after 90 seconds), the players will be left with a few health points if they are not hit at all by the opponent. They are vurnerable for 90 seconds and that vurnerability can be abused by the opponent. The second change refers to the energy regeneration. The energy normally regenarates with a pace of 5 points of energy lost every 2 seconds. During this event, the regeneration drops to 2 energy points per 2 seconds. This practically means that the energy management has to be carefully designed and utilized by the players. Correct use of the skills and the default attack matter a lot at this point and the highly skilled player has better chances landing an attack or succeeding in a skill use. Finally, when a player is killed during this phase, he/she does not respawn, leaving the alive player with extra point if the alive player manages to survive the health loss every two seconds until the end of the game.

Event	Description
A	Each kill gives double points.
B	Stepping on stone gives points and a random buff/debuff.
C	Both Players are teleported to the same spot and get stunned.
D	Players' Health goes maximum. They start losing health/time. Energy regeneration goes down to 2/2sec.

Table 4.3 – Events Description

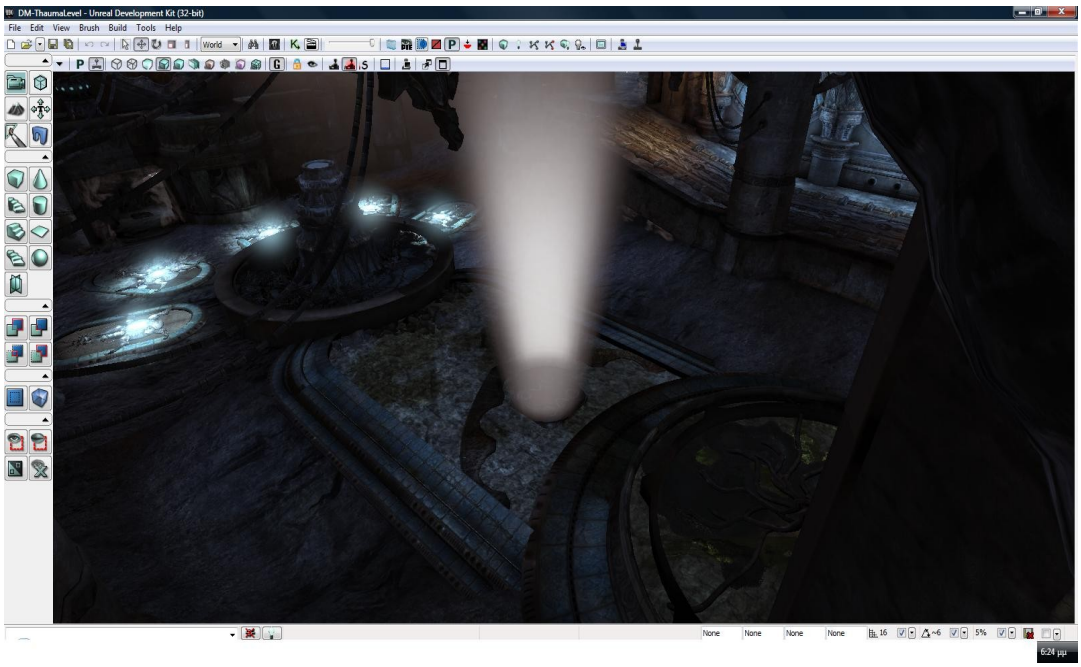

Fig 4.6 –Will you step and risk being stunned?

4.3.3 THAUMA User Interface

The User Interface (UI) is the connection between the player and the game. It gathers the information concerning player actions and sends the appropriate output based on the input. A well-designed interface will enhance the gameplay experience, whereas a poorly designed will repulse the player from spending time with the game. The ideal UI offers maximum control to the player, all the information needed to play and enjoy the game, is clear, easy to learn and very easy to use after a few seconds. The importance of the UI is highlighted by the **Human Computer Interaction** field, and one of the first principles is "The first impressions of the users on the interface are determining the attitude of the users towards the interactive system". Both appearance and efficiency are of utmost importance for the UI. A consistent Interface which provides feedback to the player for every action, while keeping things as simple as possible is the designer's goal.

A game UI may include windowed views or overlays. Windows views gather all the visual information in window-like areas at the edge of the screen (sides,up or most often at bottom). Overlays, on the other hand, intergrate the UI in the game offering an intense immersive experience because there is more screen available for the actual game. The choice is made mostly depending on the genre, along with the actions the player can perform. One may suggest that when designing a UI, innovation most of the times is not a plus. There are however, differences from game to game even of the same genre, but basic guidelines are followed. An interesting point is that UI in games does not have to follow some of the classic HCI rules in order to be successful. Games' complexity from the early years of their development have managed to get the players accustomed to a design more complicated than non-game software (Fig 4.7).

The type of UI designed for this project is a visual one, which means that the player cannot interact, but rather be informed from it. Health, Armor, Energy, skills' availability and messages are always on the player's screen. Related sounds also help to provide feedback since an exciting event may result in the player ignoring a visual indication.

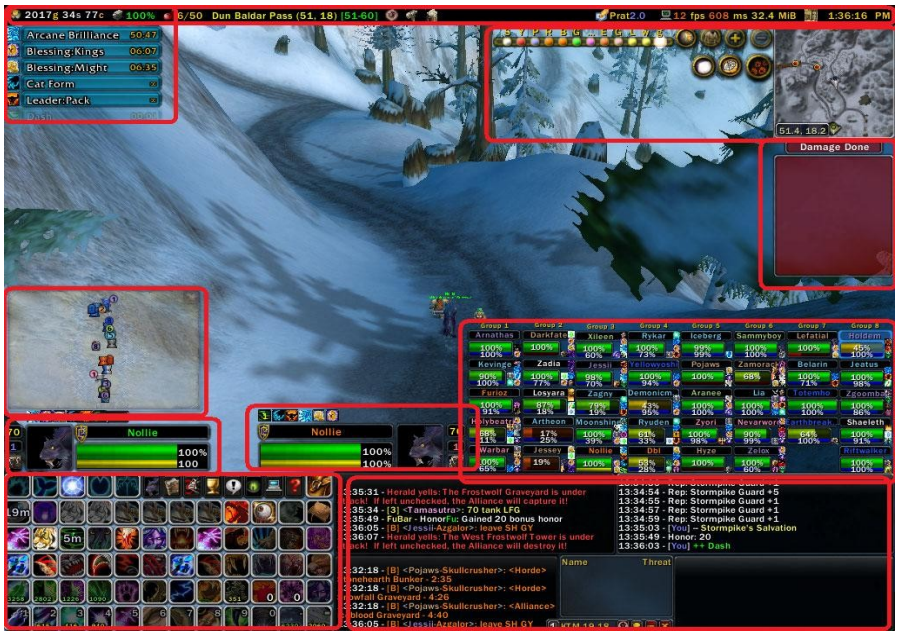

Fig 4.7 -More than half of the screen is covered from the UI. That equals to nightmare for HCI experts, but to an enjoyful experience for hardcore gamers.

Thauma UI was based on very simple graphic elements since no artist was involved and the author's skills in art are questionable (Figure 4.8). The interface is spread all around the screen, since placing all the elements in one place would cover a large portion of the screen. The important meters of Health and Energy were placed on the top of the screen which is where they are commonly placed in games of the same genre. Skills were placed on the bottom middle screen which is also common. The events messages appear on the left side, under the important Health bar, so the messages are clear to the players. The skills' icons turn to a distinguishable color when they cannot be used in order to be clearly unavailable to the players without them having to focus on them and lose attention. The colors chosen for health and energy are based on previous games as well. Red depicts blood which equals life, while blue stands for the mental strength (Fig 4.8).

Fig 4.8 -Thauma uses a visual interface.

4.3.4 THAUMA Multiplayer

Multiplayer support is very common nowdays, especially because of the fast network speeds that are available. Implementing multiplayer gaming is a desicion that needs to be taken during the first steps of the game, not only because of the different orientation the game gets concerning the gameplay, but also because of the programming part which demands networking to be taken into consideration from the beggining. The reason this project supports multiplayer is mainly because (apart from implementing it as a chosen requirement for this thesis) games are more entertaining when you play with your friends, either co-op or competitive. The fact is that 60% of games are played with friends; this is also the entertaining part of socializing. Apart from that, having experience with network programming was also a goal. Due to UDK being available for only a short period of time, there were no indie projects supporting multiplayer. Other programmer's experience and help would be too valuable for a team-focused engine like UDK.

Jonathan's story supports multiplayer, with the second player taking the part of his DPD, Jonathan's evil self. Throughout the story, DPD's role changes, along with the role of the second player, depending on Jonathan's psychosynthesis.

The taste the game leaves to the players is strange, because of the mixing of types and genres throughout the game as described in the players' feedback text having read the SDD and played with Thauma T.v. To be more specific, story-based games do not normally support multiplayer, with a few exceptions that were not successful such as the last Zork game having one player moving and the other using the mouse for the puzzles. Moreover, implementing RPG-like skills in FPS is not common either. First of all the player cannot use the skills by using the mouse (mouse is used for ratation-aim), which is the traditional way and secondly the different weapons usually cover the need for diversity.

4.4 THAUMA Prototyping

Protytype means first impression, which comes from the greek work 'protos' meaning first and 'typos' meaning impression. Creating prototypes is a critical procedure used in software development among others, which aims to serve as a typical example or basis. The main reason to create prototypes is because software development takes a great amount of time, which leaves no place for mistakes. There are various methods to prototype and each expert team can modify any method to fit their own standards. In games, this procedure is sometimes overlooked either because no personnel can handle it, since it is mainly related to software engineering and no game design, or because the complexity of the game leaves no room for extended prototyping. Prototyping is very popular however in researching new types of games. Pervasive games for example are the games that inculde real life constants in games. Time and location parameters can be used in prototypes much easier than it would be to actualy use input devices for these constants. [9]

No matter how experienced in game design a designer is, an idea can turn out different from what the designer would have thought or expected. This is the reason why prototypes are important. The main methods of game prototyping are four, however one can always experiment with a new form or a sub-form.

- **Pen and Paper:** Fastest and cheepest method to prototype and no further knowledge is needed.
- **Board Game:** No better way to show an idea for a game than implementing it in a non-software game. The entertainment factor can be measured easilly and it is a cheap method too. Some counters, miniatures and a piece of paper will do.
- **Graphics Program:** Depending on the concept, a prototype in Photoshop or elsewhere can be useful too. This is not the usual case, since it does not allow for much interactivity, but is chosen when working from distance.
- **Software Prototype:** The programmed prototype is the one with the most resemblance to the actual game. On the other hand, the time needed to create it is sometimes forbiden, and is hard to change.

Protytyping is about experimenting. Two or three or all the methods can be used for one project. The main advantages are the following:

- It can be checked if the game will be fun from early on;
- An idea can be chosen among others;
- The game rules can be balanced;
- Early feedback can be taken into account by having testers play the prototype version;

Many are the projects that managed to become popular and were based on prototypes. These games are most known for their innovations. Two examples

worth paying attention are Spore and World of Goo. Spore is a game based on the evolution of creatures the player makes. A fragile concept like that needs a lot of prototyping before certifying an idea[2]. The second game, World of Goo, came from a different procedure though. Grad enginner students of Carnegie Mellon University (CMU) spent a semester creating one-week games. The goal was to rapid prototype many new ways of gameplay. The result was a game (one of the many) that was downloaded hundreds thousands of times in a short period of time. The one-week concept fits the prototype idea fully; cheap and dirty.

For this project, a board prototype was devised and built, mainly to provide feedback for the game mechanics, the events and the balance issues (Fig 4.9). One person was needed to simulate the two avatars during the prototypes' session. The drawback of prototyping a game such as THAUMA is that while the game is played in real-time, the prototype has to be tested as turn-based. From that perspective it could not help much. However, having a visual indication of the level and the position of players in the level was enough to assist in the making of the events. Resolving balance issues was the main advantage of the THAUMA prototype, specially because the multiplayer part demanded a second person being available at working hours, which was hardly possible for this project. Board games with heroes and skills only need miniatures and cards displaying the abilities and dices. The main idea is the same, therefore it was easy to simulate an in-game combat by using two miniatures and the skills written in pieces of paper. A stopwatch (cooldowns) and tiles resembling the level would be handy as well (Fig 4.10).

Fig 4.9 -A few miniatures, a board or hexed paper, index cards, counter marks, pen and paper can assist in making a board game prototype.

Fig 4.10 -The prototype is ready for some testing. Skills and Players' stats are described in the index cards. The counters mark a skill as used.

Chapter 5: Implementation

5.1 Intro

Object-oriented programming in UnrealScript unveils no major difficulties, mainly because of the strong resemblance to the C/C++/Java syntax. The core of UnrealScript allows the programmer to accomplish any game attribute, however, the developer is centred on that core. Obtaining the core's code is highly significant to the programmer for this reason. This is proven to be a difficult task. Learning UnrealScript starts with reading the main classes' code as written by Epic, in order to become aware of what has to be written anew and what not. Learning the hierarchy of the classes is also of high importance. UDK was designed as a professional tool and professional teams consist of many people. Using UDK for a solo project meant one man had to implement the tasks that normally other experts would do. For example, UDK offers a very good base for the network programming part which usually is being handled exclusively from a network programmer, due to the complexity of that task. The same goes for the UI programmer etc. Once the programmer learns to develop game attributes using UnrealScript, the many possibilities that Unreal Script has to offer are realized.

The classes created for this project are (Fig 5.1):

- *ThaumaAbilities*
- *ThaumaAbilitiesNeutral*
- *ThaumaAbilitiesOffense*
- *ThaumaAbilitiesDefense*
- *ThaumaAbilityArmor*
- *ThaumaAbilityRegeneration*
- *ThaumaAbilitySpeed*
- *ThaumaAbilityInvisibility*
- *ThaumaAbilityStun*
- *ThaumaAbilitySacrifice*
- *ThaumaAttackSacrifice*
- *ThaumaHUD*
- *ThaumaMain*
- *ThaumaPawn*
- *ThaumaPlayer*
- *Thauma_ProjPsychic*

ThaumaAbilities includes the base attributes of the skills, such as the EnergyCost and Duration. All these attributes are inherited to the *ThaumaAbilitiesNeutral*, *ThaumaAbilitiesDefense* and *ThaumaAbilitiesOffence* classes. Each of these classes contain different skills, depending on the nature of the skill. The *ThaumaAbilityXXX* classes include the main implementation of the skills. Additional implementation had to be done in other classes, in order to connect the skill to the player. *ThaumaAttackSacrifice* is the class that creates the new projectile that is used for the Sacrifice Ability. The Heads Up Display is described and implemented in the *ThaumaHUD* class, which includes the UI. The game logic and rules are implemented in the *ThaumaMain* class, along with some connection issues between Thauma and the players. *ThaumaPawn* is the class responsible for the avatar's actions, whereas *ThaumaPlayer* contains the commands that result to the action. For instance, if the player decides to run, the code responsible for running is located in *ThaumaPawn*, but the process from pressing the button to calling the run inside Pawn is handled in *ThaumaPlayer*. Finally, *Thauma_ProjPsychic* is the class describing the projectile used for the default attack.

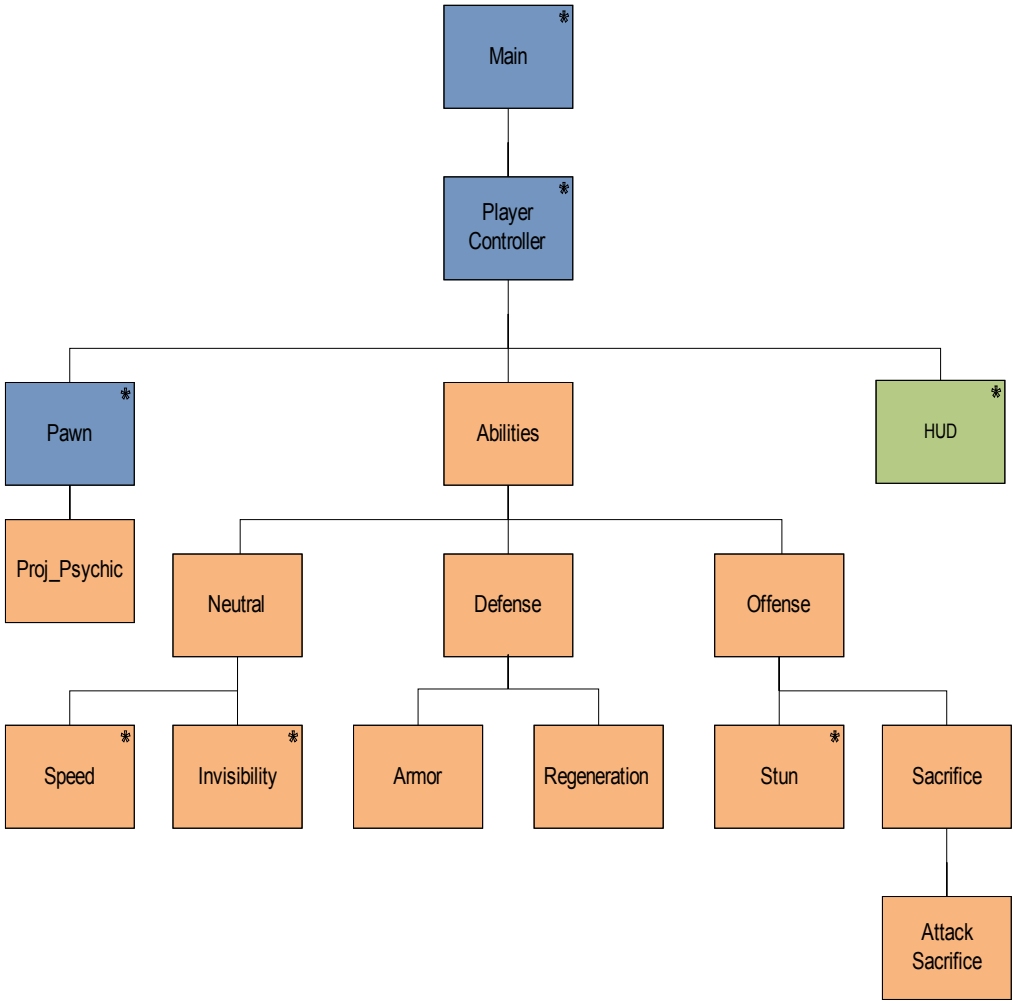

Fig 5.1 -Hierarchy of the classes written for Thauma T.v. The Blue classes derive from UTGame, the green from the engine, while the orange are made from scratch. The Classes with the asterisk include network code.

5.2 Players

ThaumaPlayerController is the class ordering the Pawn what to do. It derives from *UTPlayerController* class inheriting all the base functionality from it, enabling the programmer to deal with the important parts and the parts that need to be changed. For example, the movement functions, the physics and more basic features are all implemented in *UTPlayerController*, saving precious time which can be spent in different aspects, such as skills. One may decide to escape the movement type of Unreal however. These critical desicions are left for the designer to take. The complete source code for this class is available at the appendix.

```
...
var ThaumaAbilities Abilities;
var ThaumaAbilitySpeed Speed;
var ThaumaAbilityInvisibility Invis;
var ThaumaAbilityArmor Armor;
var ThaumaAbilityRegeneration Regen;
var ThaumaAbilityStun Stun;
var ThaumaAbilitySacrifice Sacri;

var int PsychicAttackCost;
var UIScene PauseMenu;
var Console PlayerConsole;
var bool bPhaseD;

replication
{
        if ( bNetDirty )    //when replicated variable changes server side, send its value to all
clients
                bPhaseD;
}
...
```

The declaration of variables always comes first, followed by the replication statement. For this class the bPhaseD boolean must be known to both to the client and server so this statement has the server send bPhaseD's value everytime it changes.

```
...
simulated function InitializeAbilities()
{
        Abilities = new class'Thauma2.ThaumaAbilities';
        Abilities.AbilitiesOffenseConstructor();
        Abilities.AbilitiesNeutralConstructor();
        Abilities.AbilitiesDefenseConstructor();
        Speed = Abilities.GoodAbiNeu.NeutralTier.AbiSpe;
        Invis = Abilities.GoodAbiNeu.NeutralTier.AbiInv;
        Armor = Abilities.GoodAbiDef.DefenseTier.AbiArm;
        Regen = Abilities.GoodAbiDef.DefenseTier.AbiReg;
        Stun = Abilities.GoodAbiOff.OffenseTier.AbiStu;
        Sacri = Abilities.GoodAbiOff.OffenseTier.AbiSac;
}
...
```

InitializeAbilities() works as a constructor for all the skills. While not all skills are used by both players, the client server will propably use the server's abilities during

the event that gives buffs. Some of the buffs serve as the skills themselves. The server however, will not use the stun or the sacrifice skill, so this initialization could have been altered depending on whether the machine is a client or not. This is bound to change in the next patch.

Whenever there is a a function call with the keyword 'Energy', this function is located in the *ThaumaPawn* class, where the Energy variable is declared and managed..

```
...
/due to timer's restriction (functions calls with no parametres) Regen() is in here
function Regeneration()
{
	if (self.IsInState('DEAD'))
		ClearTimer('Regeneration');
	else{
		if(Pawn.Health< Pawn.HealthMax && Regen.RegenCounter<10)
		{
			Pawn.Health = Min(Pawn.HealthMax, Pawn.Health +
Regen.RegenRate);
			Regen.RegenCounter++;
		}
	}
}
...
```

The Regeneration() function should best be declared in the *ThaumaPawn* class, however that was not possible due to the constraints the Timers have in UDK. To be more specific, the Timer's function call ought to be in the *PlayerController*'s class. SetTimer() has several parameters including a function declaring the function that will be called when the timer expires. This function must be in the *PlayerController* class.

The functions involving the skills follow, requiring additional management depending on the side from which the skill is called (either server, client) and the complexity of the skill.

The Kismet is a tool UDK offers that will be described below. Kismet actions, are also in this class because the functions have to be executable and only the *PlayerController* class can have executable actions. Executable is the action that can be called from the console.

5.3 Skills

A large portion of the programming part of this project, refers to the skills the players can use. Skills are the main characteristic of the Role Playing Games, a genre with great potential due to MMO games' popularity nowdays. Skills can be

compared to super powers or special moves. Tenths of different skills are at the player's disposal while playing these games, allowing for differencies between player's characters' builts. Therefore, implementing a skill is a significant task for a game programmer. The full description of two of the skills utilized follows. The network part of the skill is also described, since Thauma is multiplayer game.

The base on which the skills were set, was designed like a common skill tree similar to the one many games include (Fig 5.2). The origin of this tree may well be the PnP RPG skills, which most often demanded a low level skill before allowing the player to learn a higher one. This progression allows the player to use a new power at each level, the next being greater than the current. In order to resemble this tree, a hierarchy had to be developed. The class *ThaumaAbilities* is the super class of every tier and the three classes that derive from it are *ThaumaAbilitiesNeutral*, *ThaumaAbilitiesOffense* and *ThaumaAbilitiesDefense*. All the skills extend from these three classes, which in turn extend from the basic one. This means that a large amount of the source code would be declared in the basic one, since all the subclasses inherit its functions. The functions code would differ, however, calling a function would need a call from the basic one, due to the inheritance. All the common skills attributes would also be declared in the super class.

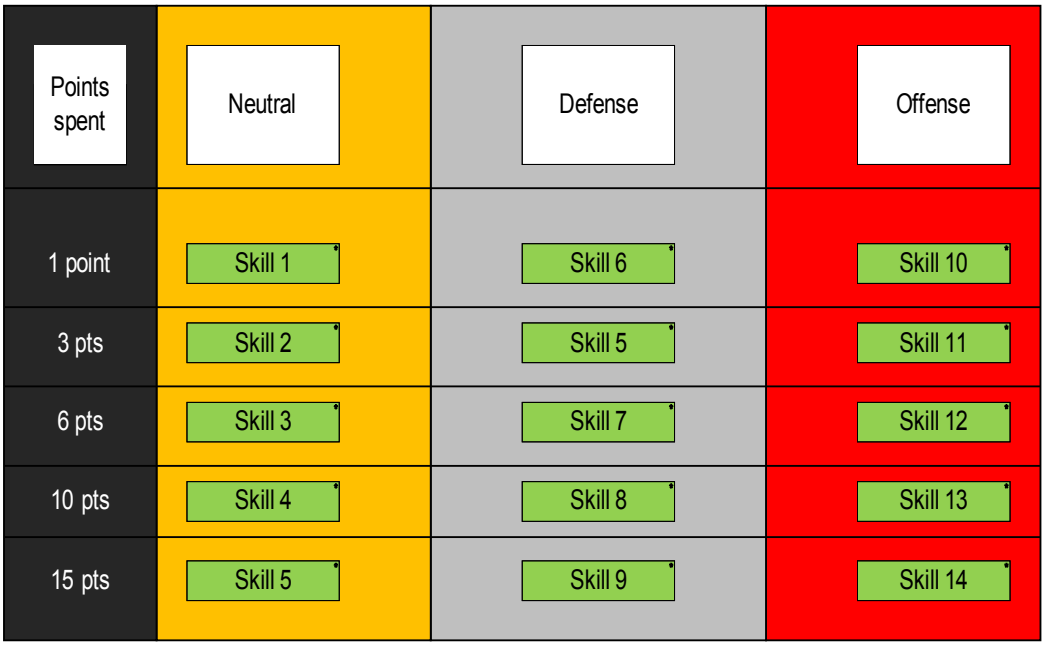

Fig 5.2 -Adding more skills can easily result to a classic skill tree met in RPGs, with points spent to learn each skill.

The complete code for the *ThaumaAbilities* class can be found in the Appendix.

Now that the base is set, the subclasses can be declared. Only one is presented, with the others being similar. The implementation could be the same without this hierarchy, but as stated before this convenience was applied to allow for future modifications. Adding more skills and including a skill tree can be easily developed that way.

The first skill analyzed is the Speed skill. Using it offers the player a movement speed bonus for a small amount of time. The main implementation is in the class presented below. The ThaumaPlayerController class and the ThaumaPawn class include part of the skill's functionality in order to connect the skill with the player succesfullyin the client or server.

```
...
simulated function SkillBuff( ThaumaPlayer SkillUser)
{
	if(SkillUser.Pawn != NONE )
	{
		SkillUser.SetTimer(Duration, FALSE, 'SpeedEnded');
		if( SkillUser.Pawn.Role < Role_Authority )
			SkillUser.ServerSpeedBuff();
		else
			SkillUser.Pawn.GroundSpeed =
ThaumaPawn(SkillUser.Pawn).MaxSpeed;
		if( SpeedUseSound != None && SkillUser.Pawn != None )
			SkillUser.Pawn.PlaySound(SpeedUseSound, false, false);
		SkillUser.ClientMessage("My feet feel too light!");

	}
	else
		`log(">>AbilitySpeed:this shouldn't happen");
}

function SkillCooldown(ThaumaPlayer SkillUser)
{
	SkillUser.SetTimer(Cooldown,FALSE,'MakeAvailableSpeed');
}

function SkillEnded(ThaumaPlayer SkillUser)
{
	if( SkillUser.Pawn.Role < Role_Authority )
		SkillUser.ServerSpeedEnded();
	else
		SkillUser.Pawn.GroundSpeed = ThaumaPawn(SkillUser.Pawn).BaseSpeed;
	SkillUser.ClientMessage("My feet feel heavy again!");
}
...
```

The full course of action in order to use any skill, in this case speed is the following (Fig 5.3):

Each player has his HUD drawn either on the side of the server or the client. The first skill is represented by the number one key. The player presses the "1" button which is assigned in the configuration file of the game to the function PlayerAbilitySpeed() in the *ThaumaPlayerController*. The PlayerAbilitySpeed() function calls the generic function used for all skills located in the *ThaumaAbilities* class. This function then checks whether the player that uses the skill is valid, meaning not dead or not spawned in game, whether the ability can be used (cooldown wise) and then it calls for the skill's ability and starts the cooldown if the player has enough energy to spend in order to use it. The ability Speed checks the side of the client that called for it mainly because it involves movement which is related to the gameplay rules the server needs to know and authorise. One more timer is then needed to count the skill's duration while the previous timer utilized counts the cooldown. The speed of that player is set to the maximum for the

duration of the skill. When the skill ends, the movement is set to the standard one, which is applied on the client or server sider accordingly. A boolean is set to mark the skill as used, so when the cooldown ends the boolean will be set to true again.

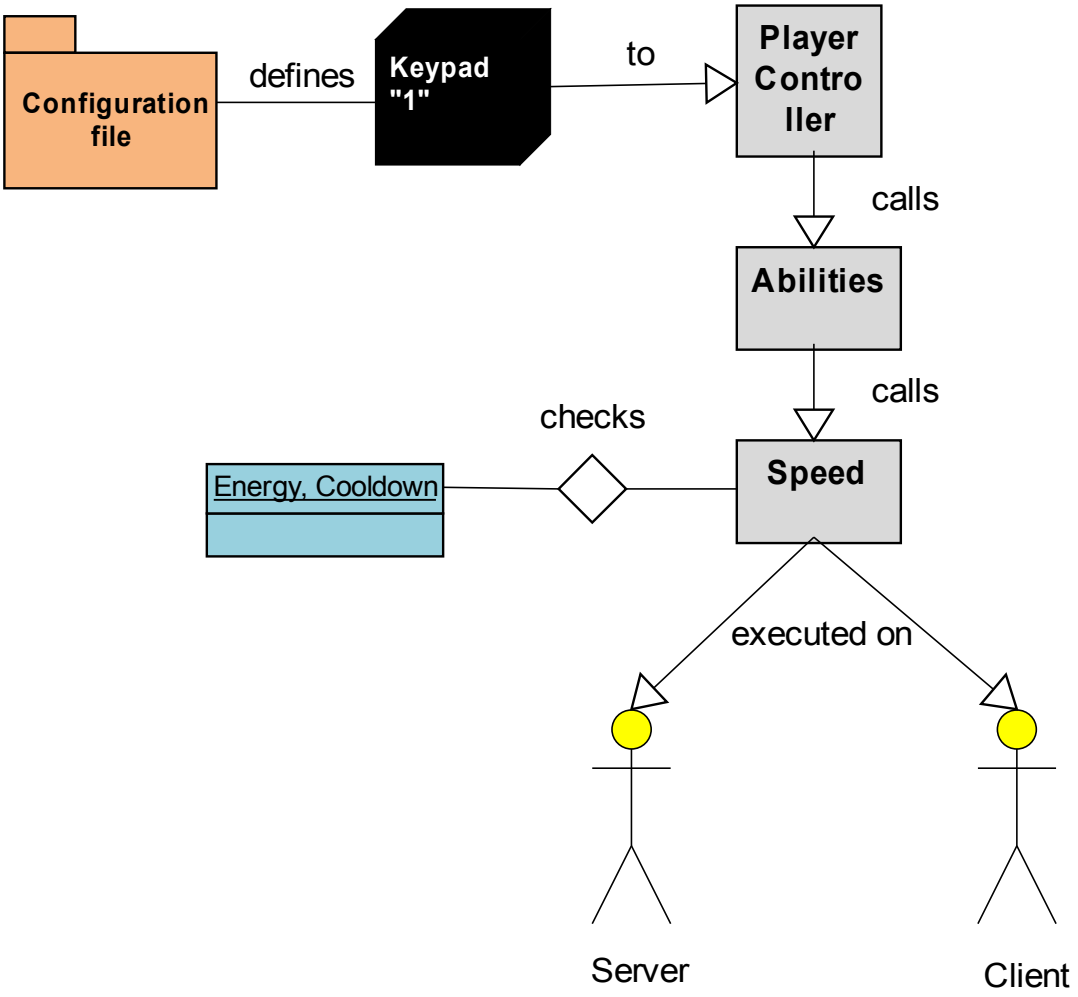

Fig 5.3 – Using a skill process.

The code for Stun skill follows, used only by the client side and player Dark Jonathan.

```
...
simulated function SkillBuff( ThaumaPlayer SkillUser)
{
	if(SkillUser.Pawn != NONE )
	{
		//Only client uses Stun. If server can use it too, some changes are needed in
Pawn's CheckForStun function.
		if( SkillUser.Pawn.Role < Role_Authority )
			SkillUser.ServerStunBuff();
		SkillUser.ClientMessage("I am aiming for his kidney.");
	}
	else
```

```
                `log(">>AbilityStun:this shouldn't happen");
}

function SkillCooldown(ThaumaPlayer SkillUser)
{
        SkillUser.SetTimer(Cooldown,FALSE,'MakeAvailableStun');
}

//stun ended for the stunned player!
function SkillEnded(ThaumaPlayer SkillUser)
{
        SkillUser.SpeedEnded();
        ThaumaPawn(SkillUser.Pawn).bEnergyUsable = TRUE;
}
...
```

Stun is a skill more complicated than the others due to the fact that using this skill can affect another player too. When this skill is used, the next attack (Right click this time) will shoot a special projectile that can stun the player it hits. Stun disables both the movement and the energy use of the enemy, making him very vurnerable for a short amount of time. Compared with the sacrifice attack it maybe deadly. However Jonathan will use 50 points or Armor if available, in order to avoid the stun effect. The damage of the Stun attack will be applied which is more than double of the default psychic attack. The procedure for this skill's implementation uses the same base as the one described for the speed skill above.

The player presses the "3" button as shown in his HUD, which shows a message about the next attack being in a critical spot. In the configuration file the button 3 is assigned to the function PlayerAbilityThree() in the *ThaumaPlayerController*. This function calls the correct skill 'number three' depending on the player. Attacking with the right click button now shoots the projectile which does damage and causes the stun effect. This attack is written in the *ThaumaPawn* class with this function.

```
reliable server function StunAttack()
{
        local Projectile StuProj;
        local vector newLoc;
        local rotator Aim;

        Aim = GetViewRotation();
        newLoc = Location;
        newLoc.Z += 10;
        StuProj = Spawn(class'UTProj_ShockBall',self,,newLoc);
        StuProj.Damage = 20;
        StuProj.Init(Vector(Aim));
        if( StunFireSound != None || Instigator != None )
                PlaySound(StunFireSound, false, false);
}
```

Note that since this is an action that affects both players, it is very important for the server to know, since the gameplay state lies on the server and the client is a proxy one. This is the reason why there is a boolean informing the server about the

stun attack in a replication statement in the *ThaumaPawn* class. If the attack hits the opponent, there is a series of checks in the *ThaumaPawn* class (this is the class managing the damage done to the player) that check each time the player is damaged, whether this damage is damage from the stun attack (each attack can have its own damage type), in which case if the player has 50 or more armor points the stunn effect is avoided, otherwise the stun effect is applied. The checks concerning the energy the cooldown etc. are done as described before.

5.4 Events

A part of the implementation, the events were written in UnrealScript. A significant amount of events' gameplay was also implemented using the Kismet tool. UnrealKismet is a flexible and powerful tool that allows non-programmers to script complex gameplay flow in level. It works by allowing the developed to connect simple functional Sequence Objects to form complex sequences. Every sequence is translated into UnrealScript, which means that a programmer can skip the Kismet and write the code for the gameplay events. However using Kismet saves a great amount of time and is fun to use (Fig 5.4, Fig 5.5).

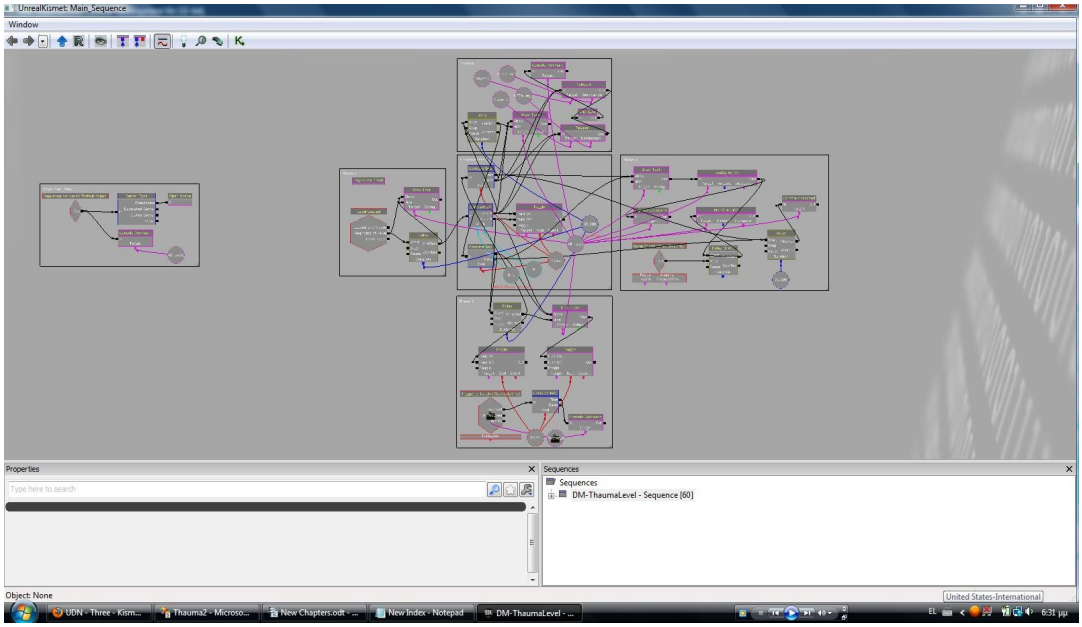

Fig 5.4 - Overview of the Kismet sequences made for the Thauma T.v Game.

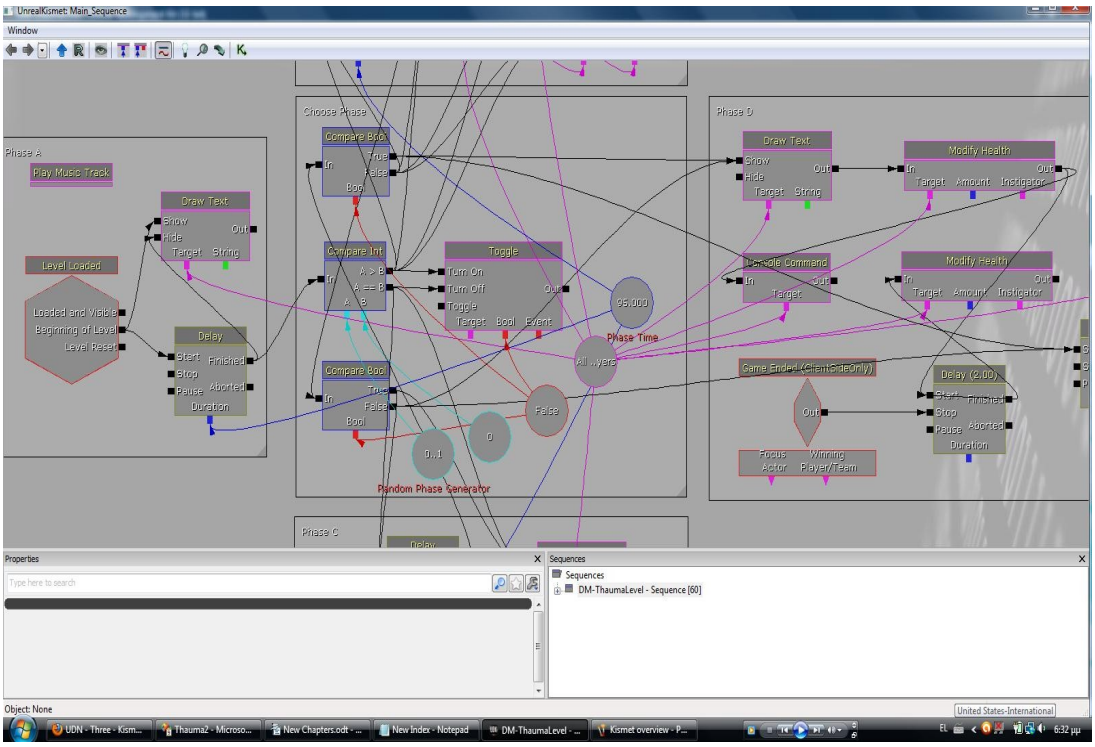

Fig 5.5 - Clear view of Kismet sequences.

The sequences look like a Finite State Machine (FSM). There is visible progression from one state to another when and if some conditions allow it. In this example, when the level is visible (condition checked by UDK) a text is drawn on the screen and a countdown begins. When the countdown ends, the integer comparison is applied and according to the result the correct sequence is called. This game-wise checks the result of a random integer generator and calls an event based on the integer generated. This sequence of Kismet actions affect the gameplay but not immediately. There are more immediate means such as making damage to a player, or teleporting him. The actions the Kismet may use to make sequences can be either premade or created by the level designer. The basic actions are pre-made and available from a menu in Kismet. In order to implement new actions, an amount of programming may be involved.

5.5 Multiplayer

What is important when implementing multiplayer support is to know exactly what needs to be sent from the server to the clients and vice versa and what not. The second most important thing is to know how to do this data transfer. A general rule is that every action that involves gameplay rules must be known to the server. As mentioned before the true game state is that of the server. The clients only see a proxy of that gameplay. If too much info is sent from the server to the clients the game will have latency and lag is hardly tolerated in action games where every second counts. The setup for a network connection is written in the native code of UDK, which means that the programmer does not need to spend time opening sockets and using protocols (UDK uses the UDP protocol).

The keyword simulated before the function's name means that this funcion will be executed in the server side as well as the client side. The functions that do not include this keyword are executed in the computer the game is played at, without sychronization or information transfer. Most of the functions are declared that way, in order to not burden the bandwidth. This also means that the gameplay rules that will be executed in this machine, (like the cooldown timer) are prone to hacking since the server cannot check the functions that are executed in that machine. However security was not taken into consideration in this project. There are certain methods that protect the code's execution in each machine.

In order to check the side the code is being executed in, there is a variable named 'Role' at each machine, getting values depending on whether the machine is a listen server, a dedicated server or a client. This check is useful when server data are involved. In the following example, the stun attack is used from the client and the server has to know that the client used the stun attack. Movement, attacks, sounds and more actions the server must know are being treated likewise.

```
simulated function SkillBuff( ThaumaPlayer SkillUser)
{
	if(SkillUser.Pawn != NONE )
	{
		//Only client uses Stun. If server can use it too, some changes are needed in Pawn's
CheckForStun function.
		if( SkillUser.Pawn.Role < Role_Authority )
			SkillUser.ServerStunBuff();
		SkillUser.ClientMessage("I am aiming for his kidney.");
	}
	else
		`log(">>AbilityStun:this shouldn't happen");
}

reliable server function ServerStunBuff()
{
	ThaumaPawn(Pawn).bNextAttackStun = TRUE;
}
```

The reliable keyword tells the side (server in this case) to definitelly execute it, in the same order it was sent to the client. When the stun skill is used from the client, the server will set this boolean to true.

```
replication
{
	if ( bNetDirty )    //when replicated variable changes server side, send its value to all clients
		bNextAttackStun;
}
```

The boolean's new value will be sent to all the clients, including the one that used the stun skill. The next successful stun attack will check for this boolean in order to know whether the skill was used or not and the server will know that the player in that machine will have to get stunned whereas the player in the client's machine will know that the stun attack was used. The next step is to set the boolean to false,

meaning the next attack will not be a stun attack since it has already been used.

Data involving players' and game's information are replicated in the native code, therefore all the sides can access them. The programmer has to know these functions since it is most likely that they are going to be needed at some point. One example is the *GameReplicationInfo (*GRI). This structure keeps the score for each player and can be accesed from any side. This is the code accesing the opponent's score.

```
...
for (i=0; i < WorldInfo.GRI.PRIArray.Length; i++)
        {
                if ( WorldInfo.GRI.PRIArray[i]!= none )
                {
                        TotalScore += WorldInfo.GRI.PRIArray[i].Score;
                }
        }

        OpponentScore = TotalScore - PlayerScore;
...
```

As a final note, it is important to grasp the whole idea behind the server's authority in the game. Certain classes do not need to be replicated to the clients such as the *PlayerController* or the *GameType*. The client uses the *PlayerController* to command the pawn what to do, then the server sends that action to all clients. Finally, the server reports the data back to the pawns or their controllers.

5.6 Quality Assurance

Quality assurance is described here as the last step of developing a game but should begin as soon as a playable version of the game is available, depending on the resources (testers). There are many techniques a game can be tested with, some including psychologists or mathematicians apart from engineers. These techniques inquire into the entertaining part of the game as well as checking for bugs. One such technique is the TRUE System (Tracking Real-time User Experience System), implemented by Microsoft Game Studios, more specifically the Games User Research team. This system records the player's experience and the data such as place and number of deaths mined from all sessions are analysed by experts to improve the game[12].

While techniques such as the TRUE System are used to improve the gameplay experience, there are more traditional methods to testing a game (Table 5.1). Focus Groups, Usability Testings and Surveys are all methods that provide feedback to the game developers[5]. Getting good feedback may be difficult especially for a solo project such as the one presented in this thesis and is often misjudged. Good feedback is not the feedback that re-confirms the developer's pre-existing opinion. The purpose of the quality assurance stage is to receive specific comments which are not obvious to the developer. Evaluation test questions such as "how great the game was" will most likely bear no useful results.

Goals	**Focus Group**	**Usability Test**	**Survey**
Objective	NO	Good	NO
Subjective	OK	Good	Good
Qualitative	OK	Good	Good
Quantitative	NO	OK	Good
Evaluation	NO	Good	Good
Generation	Good	OK	OK

Table 5.1 – Methods of research in games.

For this project, a Usability Testing method was used, focusing on Think Aloud sessions while testers played the game. Thauma is Player versus Player and the volunteer testers four of which provided useful feedback were playing from distance. This only allowed for Think Aloud sessions of the developer and one player at a time. Skype was used to communicate with the other player. During the Alpha testing, when the game could be tested in single-player mode the testers could report their comments either verbally in small talk sessions if possible or e-mailed.

5.6.1 Testing

There were two testing phases for this project, the **Alpha** testing and the **Beta** testing. The purpose of each testing session was different.

The Alpha testing was run when the Alpha version of the game was available. This version included only the implementation of the skills, did not support multiplayer and the level used was a pre-made Unreal map which did not include the requirements of the level design phase. The purpose of this version was to check whether the skills were fun to use, whether any changes on the skills' design were needed and whether the skills should be all available to both players or not. There was enough feedback since it was easier to test the game in single-player, mode which led to some skills' changes.

The Beta testing (Beta is also the current version of the game, not the initial though) provided the players with 95% of the functionality of the game. The level was the one designed, the skills had been tested before and their paramaters were set accordingly. Most importantly, the multiplayer part was ready to be tested. The original idea was to make Thauma T.v a FPS with skills. The need to include events emerged at this point, since the gameplay was rightfully considered to be poor. The events were designed and tested, with some changes happening before reaching their current state. A few recommendations and bugs have not been fixed yet, but most of them will be implemented as a patch in the future. Creating a patch for a game requires a different procedure than the standard programming one mostly required for a PC game. Feedback that is taken into consideration suggests the Event B to change from a specific stone to a random one around that area. This

will have the player paying attention to his surroundings as well and not only the opponent and the specific stone. The increase of the game's duration from 6 minutes to 8 is another change pending. The gameplay relies on replayability, allowing the players to play many fast and different from one another sessions. Eight minutes of total game with a different event every two minutes is better than the current approach. Finally, several User Interface improvements were noted such as the need for a different color to signify the used abilities, adding text messages in certain situations and making the text under the Join button in the menu more apparent.

Another purpose the testing had to fulfill was the balance between the two players' abilities. The importance of a balanced gameplay was described in a previous chapter and testing it included two seperate phases. The first phase was done via the board prototype that was built for this game and offered a fast and easy way to test the balance components of the game. The balance components consist of the various parameters that define the gameplay. Usually in FPS games, special powers and weapons' damage are the gameplay variables that need to be balanced, apart from map events. In accordance with Thauma T.v, the parameters that were subjected to balance issues were: Health, Energy, Armor and the Energy cost, duration, cooldown and effect of all the skills. The prototype assisted in defining the damage and the energy cost. Duration and cooldown could not be tested in a turn-based enviroment, therefore these attribute were tested at the Beta test phase two, involving real players.

Chapter 6 : Results

6.1 Summary

The purpose of this thesis was to design and develop a networked computer game. This thesis focused on the development part as well as the design part of gameplay resulting in a game with entertaining elements and interesting gameplay mechanics. The Thauma Story was based on the creation of the characters' profiles and their background stories and the design proccess includes the levels, the game mechanics and prototyping. Finally, implementation based on the design results along with quality assurance conducted by testers in two phases using Usability testing are all parts of this project adopting a user-centered design using the agile development method.

Thauma T.v is a multiplayer game. Two players confront each other which is challenging since the opponent is not an A.I agent. This is entertaining since one is able to spend time playing the game with a friend. The game's storyline can be extended to include additional parts and chapters. Jonathan, the protagonist, is on a quest to find peace with himself after the death of his brother at a young age. The antagonist (the second player) is Jonathan's evil self, summoned by a series of events at the beggining of the game.

The implementation part included the creation of skills, the definition of the gameplay mechanics and the network part. UDK was chosen among others for this project. The inherent team-oriented design of this game engine as opposed to the limited resources available for the implementation of THAUMA led to a difficult start with excellent results nonetheless.

6.2 Future Work

Creating a game requires a team consisting of many skills inclusive of scientific and artistic. Working alone requires careful time planning. Software engineers can spend time on the design and development stages such as coming up with the game mechanics, prototyping and programming (gameplay, AI, network etc).

Future work could include the provision of the 'save' option which saves the player's current state and offers the capability of starting a game at a later stage from that point. 'Save' is used in many game genres, mainly because completing a game nowdays requires tenths of hours. Careful thinking is required when designing a save routine in order to pause and continue the game in a smooth transition. This option was neither discussed nor implemented in this project. The FPS nature of the game makes does not need, however, a 'Save' routine would be handy in the context of the Thauma game when implemented in full.

One significant programming aspect that was not addressed in this project was the process of creating and implementing a patch. Using pre-made code as a basis and implementing new pieces of code requires a separate process. Based on testers' feedback, patches may be implemented.

As far as the design part is concerned, a thorough investigation of the relevant psychology bibliography would come up with interesting results.

Adding new skills and creating a skill tree that allows the players to choose their favourite among many, depending on the experience level, is a welcome change in the future. The basis is set on the programming part, needing only the new skills to be designed and created, as well as a leveling system, which is very common nowdays, even in FPS games.

Lastly, completing the rest of the Thauma Game is a challenge by itself.

Appendix

A.1 Story Design Document

THAUMA

Made by: Gregory Nikiforakis
Mail: gnikiforakis@isc.tuc.gr

document version 0.8

Overview:

(some of the following are subject to change)
This game is about a man who experienced a traumatic event at a young age, however, he has managed to leave it behind and go on with his life. At some point in his present life, he has to confront his past painful experiences and start a battle with himself. Memories emerge like ghosts from the past and each has a question that needs to be answered in order to find peace... Guiltiness, anxiety, depression and moral dillemas are the burden the hero has to carry till he reaches the end in his mind's maze. When his journey comes to an end, will he find catharsis or will the opening of his mind's hidden locks lead him to another purgatory?

The game's purpose is only to entertain by presenting a story in which you participate and decide the finale.

It will be available for Windows OS with the possibility of additional platforms compatability in the future.

THAUMA is an atmospheric horror adventure-action game(1st person POV) which targets the T(13+) group. Prior experience in games may help through the game.
The game engine used was Epic's UDK free release version.

Player mode: Single/Multi Player. The multiplayer part will offer a new experience to the players who will have to confront each other.

The Goal of the game is to finish all the chapters. In terms of multiplaying there will be alternate endings depending on the multiplayer sections in each chapter. The players can lose, (e.g by having another player reach his goal first) however he/she will not have to restart the chapter. It will afffect, however the ending of the game.

The major challenge for the hero is to find the truth in relation to one incident in the past. This is done piece by piece as the hero recovers memories. Minor challenges include riddles, dialogue riddles, skill puzzles, time limits and player vs player combat through the chapters.

The action takes place at the mansion where the hero had the trauma in the past, a distorted version of a fairy tale book's world and the hero's mind's imaginary labyrinth.

Structure:The game is seperated in levels/chapters (one chapter for the purpose of this proejct but more after that). The player has to beat each chapter in a linear manner. Each chapter has different goal/setting.

Gameplay: Players can walk/run, jump, can't crawl etc. They have an inventory for items that will use through their way at each chapter. Time is important in some chapters, as there is one chapter with time limit and another with a “faster wins”

condition of winning. The control is done using the keyboard for moving and the mouse for interaction. There are also menus for options like exit, save/load etc.
There are audio tracks appropriate to the mood of the game. Gameplay will be further discussed at each level.

Prologue:

(shown in handsketch pictures as an intro)
This is an image showing Jonathan eating alone on a rainy day. He receives a call from a demolition company, announcing to him that the day the demolition will take place is the day after tommorrow (through dialog text or voise acting). The house in question is his parents' house. Jonathan remembers a scene shown in the image below. He finds himself in a psychotherapist's office during a council at young age as a patient talking about a house.

The next frame shows Jonathan driving (to the Mansion). He arrives only to see a previously majestic building, now being an abandoned wreck waiting its destruction. As Jonathan walks through the yard, he remembers playing at the same yard with his little brother, Jacob, in a happy and bright environment.

In the next frame, Jonathan is in the mansion gathering some stuff that has already been packed. At the top of an opened box lies an old mistreated book. He glances upon it and freezes for a second. His heartrate rises high and cold sweat runs from his forehead. That book brings up memories he tried hard to hide in his subconsciousness... He can see little Jacob lying sick in a bed, asking for his book. Jonathan's heartrate raises. His hands can't hold the box and he lets it fall. Whispers

sound in the room. He looks up, to the direction of Jacob's room. Now he can see him passing away in bed and his younger self stands right next to him.

Chapter 1:

~~this chapter will not be available in the current version of THAUMA~~
Room door knocks are heard, faintly at the beggining, but louder as seconds pass by. Jonathan opens the door and hesitates. A man stands behind the door who looks like a twisted elder version of his brother. When asked he implies he is Jacob. In reality he is Jonathan's DPD (Dual Personality Disorder) which has just surfaced due to his guiltiness being reminded. The spirit urges Jonathan to find a video that will help him remember. This is another long forgotten memory. On the one hand he wants to remember what happened, however he is being protected by himself not to remember because these memories will bring all the pain back.

GAMEPLAY:(adventure-like, game mechanics, gameplay etc still in development)

Single: Player 1, as Jonathan in young age having to find the projector's whereabouts through dialogs with his parents and riddles.

Multi: Player 2 has to prevent player 1 to find the projector's pieces in time.

Place: Old version of mansion.

End: Full video shows Jonathan giving Jacob the book on his last birthday.

The Strange figure, however destroys the mood by accusing/questioning Jonathan that afterwards he stole the book. That's why Jacob was asking for it before he passed away. All hell breaks loose for Jonathan...
Deep in his mind, Jonathan remembers that he took the book from Jonathan after he had given it to him. That caused him this self-questioning.

Chapter 2:

~~this chapter will not be available in the current version of THAUMA~~
Jonathan enters the imaginary world of the book and has to fight his way to the end in order to realize that the last page is missing.

GAMEPLAY:(game mechanics, gameplay etc still in development)

Single: Player 1 vs AI.

Multi: Player 2 vs Player 1.

Place: Twisted world of the book.

End: Once again the Strange figure appears, in a more complete and divine form, almost like God. Could he be God all this time? He tries to calm Jonathan down and prompts him to find the missing page.

The missing last page symbolizes two things. One thing is that the last page was actually missing and that was the reason why Jonathan took the book. This incident creates this nightmare in Jonathan's mind. He wanted to repair it, as the last page was torn apart. Another thing could be that Jonathan is always missing one piece of each memory he gains, however, that piece is the most important.

Chapter 3:

In yet another gothic fantasy world the players are called to find the way to Jacob through a maze and give him his precious book. Reflexes, wise use of your abilities but also prediction of your opponent's moves are all part of your ticket out of this maze.

GAMEPLAY:(action based)

Single: Player 1 vs AI .
Multi: P2 vs P1. P1 has to find Jacob, and P2 has to prevent it.
Place: Gothic labyrinth.

MECHANICS:

Both Player1 and P2 have the following abilities, each with a seperate cooldown.

TIER0:

RUSH: movement speed increased.
INVISIBLE: Stealth and the ability to fly through obstacles.

Player1 has the following abilities along with TIER0:
TIER1:

ATTACK: A psychic attack.
ARMOR: gain an amount of armor.
REGENERATION: you regenerate health per time.

Player2 has the following abilities along with TIER0:
TIER2:

ATTACK: A psychic attack.
SACRIFICE: deals area of effect damage.
STUN: If enemy is hit, he is stunned for a period of time.

End: In order to find Jacob, they will have to confront each other.

This chapter is Jonathan asking Jacob for forgiveness. He has to find peace with himself first though and his DPD makes that hard. This chapter will highly affect the end too.

Characters' profile:

Jonathan: The tragic protagonist in this hero's journey. Jonathan was born in 1983 from middle-class parents. He and his brother, Jacob lived a happy life until Jacob got sick with a deadly disease. Jonathan's failure to fulfill Jacob's last wish caused a deep travma which healed after a psychologist intervened. Now Jonathan is 26 y.o and the demolition of his parents mansion will make him confront these past events. Jonathan is controlled by Player1.

Photo 1 - Jonathan

Jacob: Jonathan's young sibling. His precious book meant a lot to him. While in a critcal situation he was asking for it. Jonathan couldn't find the book and that created Jonathan's Erinyes.

Parents: Both loving and caring for their children, their profile is vague through the game. They only apper in the old mansion's chapter. Their role is to assist Jonathan.

Strange figure: This character serves multiple purposes through the story. At the beggining he assists the player in finding the answer behind his unexplained turmoil. Later he appears as a villain accusing him for stealing the book from his sick brother. Then he depicts God offering salvation. He is Jonathan's image of his **D**ouble **P**ersonality **D**isorder that he suffered at a young age due to his self-loathing (faillure to fulfill Jacob's dying wish). It surfaced when his memories from that incident started coming back and he had to experience the same situation again.

The multiple personalities depict the different levels that the DPD goes through and the resistance Jonathan shows. At first he is trying to find the reason why he is upset after viewing the book. Later he finds out he took the book and he is afraid he

had stolen it from his dying brother. At the end he turns to God to help him and his DPD takes that role. Finally, he will have to confront his DPD and the conclusion is up to the player's actions. Will his DPD take control which will lead him to madness or will he manage to overcome his fears and by finding the truth, he will reach catharsis? In multiplayer the 2nd player takes control of the strange figure, most of the times with a specific objective he has to complete.

A few changes have been applied for the purposes of this thesis, however, the story design document's purpose to inform the team (if there was one) would not be altered. In this case, the Story Design Document were given to various people to get feedback. In addition to this, the gameplay mechanics and the level walkthrough are included in this document which results in a game design document rather than a story design document.

B.1 Source Code Samples

The classes ThaumaPawn, ThaumaAbilities, ThaumaAbilitiesDefense with the skills ThaumaAbilitySpeed and ThaumaAbilityStun are presented here. The programming of new skills can be done in a similar way, resulting in a richer gameplay.

B.1.1 ThaumaPawn

```
/**
 * DESCRIPTION :   Player's Pawns. Handles the health, armor, energy and damage system.
 *
 * Written by : Grigoris Nikiforakis for "Thauma" project, 2010 */


class ThaumaPawn extends UTPawn;


//Energy Management
var    int     EnergyMax;         //maximum Energy a player can have
var    int     EnergyMin;         //minimum Energy a player can have
var    int     EnergyCur;         //current Energy a player has
var    int     EnergyRegen;           //energy regeneration per sec
var    bool    bEnergyUsable;         //certain Player states like Death prevent the Energy usage

var int BaseSpeed;          //base ground speed
var int MaxSpeed;           //maximum speed obtained through the ability

var() int ArmorMax;         //maximum Armor a player can have
var int ArmorMin;           //minimum Armor a player can have
var() float AbsorptionRate;         //rate at which the armor absorbs damage
var bool bNextAttackStun;           //shows whether next attack is a stun attack or not

var SoundCue PsychicFireSound;
var SoundCue SacrificeFireSound;
var SoundCue StunFireSound;
var SoundCue StunnedSound;

//Only essential variable for server to know is bNextAttackStun. More values could be replicated in
//order to prevent cheats, but cheat protection is not a concern at this stage.
replication
{
        if ( bNetDirty )    //when replicated variable changes server side, send its value to all clients
                bNextAttackStun;
}

//partially tries to solve the invisibility collision bug
simulated function UnStuck()
{
        local vector NewDest,TraceEnd,HitLocation,HitNormal;

        TraceEnd = Location + vect(0,0,1) * GetCollisionHeight();
        if (Trace(HitLocation, HitNormal, TraceEnd, Location, true, GetCollisionExtent()) ==
None )
        {
                HitLocation = TraceEnd;
        }
```

```
		newdest = HitLocation + GetCollisionRadius() * vect(1,1,0);

		if ( SetLocation(newdest) && CheckValidLocation(Location) )
			SetLocation(NewDest);
		else
		{
			newdest = HitLocation + GetCollisionRadius() * vect(1,-1,0);
			if ( SetLocation(newdest) && CheckValidLocation(Location) )
				SetLocation(NewDest);
			else{
				newdest = HitLocation + GetCollisionRadius() * vect(-1,1,0);
				if ( SetLocation(newdest) && CheckValidLocation(Location) )
					SetLocation(NewDest);
				else
				{
					newdest = HitLocation + GetCollisionRadius() * vect(-1,-1,0);
					if ( SetLocation(newdest) && CheckValidLocation(Location) )
					SetLocation(NewDest);
				}
			}
		}
}

simulated function PostBeginPlay()
{
		Super.PostBeginPlay();
		if(Role==ROLE_Authority)
			SetTimer(2.0,true,'ClientEnergyReport');		//timer for energy regen
}

reliable client function ClientEnergyReport()
{
		if(self!=NONE)
		{
			if(ThaumaPlayer(Controller).bPhaseD)
			{
				if (EnergyCur < EnergyMax)
					EnergyCur = Min(EnergyCur + 1, EnergyMax);
			}
			else
			{
				if (EnergyCur < EnergyMax)
					EnergyCur = Min(EnergyCur + EnergyRegen, EnergyMax);
			}
		}
}

//Returns true when player has enough energy AND can use energy.False otherwise.It checks
whether
// an ability can be used, and returns appropriate message, but can be helpful to return bool.
function bool EnergyManagement(int EneCost)
{
		if(bEnergyUsable)
		{
			if (EnergyCur - EneCost >= 0)
			{
				EnergyCur = EnergyCur - EneCost;
				return TRUE;
			}
```

```
                else
                {
                        ThaumaPlayer(Controller).ClientMessage("I do not have enough
Energy!");
                        return FALSE;
                }

        }
        else        //e.g when player is stunned he cant use abilities
        {
                ThaumaPlayer(Controller).ClientMessage("I...cannot move... at all!");
                return FALSE;
        }
}

//Checks if energy is enough for hud information
function bool EnergyCheck(int EneCost)
{
        if (EnergyCur - EneCost >= 0)
                return TRUE;
        else
                return FALSE;
}

//does not work for UTBot yet!Only for ThaumaPlayer
simulated function CheckForStun(Controller instigatedBy)
{
        local UTPlayerController WillBeStunned;

        if(instigatedBy.Pawn != NONE)
        {
                WillBeStunned = UTPlayerController(Instigator.Controller);
                //player will lose 50 armor to avoid being stunned
                if(ThaumaPawn(WillBeStunned.Pawn).GetShieldStrength() >50)
                {
                        ThaumaPawn(WillBeStunned.Pawn).VestArmor -=50;
                        WillBeStunned.ClientMessage("I avoided a stunning attack, but not
without cost...");
                }
                else
                {
                        ThaumaPawn(WillBeStunned.Pawn).PlayerStunned(true);
                        WillBeStunned.SetTimer(ThaumaPlayer(WillBestunned).Speed.Duration,
FALSE, 'StunnedEnded');
                }
                ThaumaPlayer(Instigator.Controller).ServerStunFalse();
        }
}

simulated function PlayerStunned(bool bStunned)
{
        GroundSpeed = 00.00;
        bEnergyUsable = FALSE;
        if( StunnedSound != None )
                PlaySound(StunnedSound, false, true);
}

/**
 * calculates the damage done by the weapon, after the armor absorbs the correct amount
 **/
```

```
simulated function AdjustDamage( out int inDamage, out Vector momentum, Controller
instigatedBy, Vector hitlocation, class<DamageType> damageType, optional TraceHitInfo HitInfo )
{
	local int PreDamage;

	if(damageType==class'UTDmgType_ShockBall')
		CheckForStun(instigatedBy);
	if ( bIsInvulnerable )
		inDamage = 0;
	if ( UTWeapon(Weapon) != None )
		UTWeapon(Weapon).AdjustPlayerDamage( inDamage, InstigatedBy, HitLocation,
Momentum, DamageType );
	if( DamageType.default.bArmorStops && (inDamage > 0) )
	{
		PreDamage = inDamage;

		//if Health of attacker is below 30, he does 50% extra damage
		if(Instigator.Health < 30)
			inDamage = 1.5 * inDamage;

		inDamage = ArmorAbsorb(inDamage);
		// still show damage effect on HUD if damage completely absorbed
		if ( (inDamage == 0) && (Controller != None) )
		{
			Controller.NotifyTakeHit(InstigatedBy, HitLocation,
Min(PreDamage,10), DamageType, Momentum);
		}
	}
}

/**
 *  ArmorAbsorb()returns the resultant amount of damage after shields have absorbed what they can.
*/
simulated function int ArmorAbsorb( int Damage )
{
	if ( Health <= 0 )
		return damage;

	//armor prevents 30% of the damage, but is damaged by 60% of the dmg. Also blocks the
stun ability
	if(GetShieldStrength() > 0)
	{
		bShieldAbsorb = true;
		VestArmor = Max( ArmorMin, AbsorbPsychicDamage(Damage));
		if (VestArmor == 0)
			SetOverlayMaterial(None);
		if ( Damage == 0 )
		{
			SetBodyMatColor(SpawnProtectionColor, 1.0);
			PlaySound(ArmorHitSound);
			return 0;
		}
	}

	return Damage;
}

simulated function int AbsorbPsychicDamage(out int Damage)
{
	local int MaxAbsorbedDamage;
```

```
        MaxAbsorbedDamage = Min(Damage * AbsorptionRate, VestArmor);
        Damage -= MaxAbsorbedDamage;
        return VestArmor - MaxAbsorbedDamage/AbsorptionRate;
}


simulated function bool TakeHeadShot(const out ImpactInfo Impact, class<UTDamageType>
HeadShotDamageType, int HeadDamage, float AdditionalScale, controller InstigatingController)
{
        if(IsLocationOnHead(Impact, AdditionalScale))
        {
                if ( VestArmor > 0 )
                {
                        VestArmor = 0;
                        bShieldAbsorb = true;
                        Spawn(class'UTEmit_HeadShotHelmet',,,Impact.HitLocation);
                }
                else
                {
                        TakeDamage(HeadDamage, InstigatingController, Impact.HitLocation,
Impact.RayDir, HeadShotDamageType, Impact.HitInfo);
                }
                return true;
        }
        return false;
}

reliable server function SacrificeAttack()
{
        local Projectile SacProj;
        local vector newLoc;
        local rotator Aim;

        Aim = GetViewRotation();
        newLoc = Location;
        newLoc.Z += 10;
        SacProj = Spawn(class'UTProj_ScorpionGlob',self,,newLoc);
        SacProj.Init(Vector(Aim));
        if( SacrificeFireSound != None || Instigator != None )
                PlaySound(SacrificeFireSound, false, false);
}

reliable server function PsychicAttack()
{
        local Projectile PsyProj;
        local vector newLoc;
        local rotator Aim;

        Aim = GetViewRotation();
        newLoc = Location;
        newLoc.Z += 10;
        PsyProj = Spawn(class'ThaumaProj_Psychic',self,,newLoc);
        PsyProj.Init(Vector(Aim));
        if( PsychicFireSound != None || Instigator != None )
                PlaySound(PsychicFireSound, false, false);
}

reliable server function StunAttack()
{
```

```
        local Projectile StuProj;
        local vector newLoc;
        local rotator Aim;

        Aim = GetViewRotation();
        newLoc = Location;
        newLoc.Z += 10;
        StuProj = Spawn(class'UTProj_ShockBall',self,,newLoc);
        StuProj.Damage = 20;
        StuProj.Init(Vector(Aim));
        if( StunFireSound != None || Instigator != None )
                PlaySound(StunFireSound, false, false);
}


DefaultProperties
{
        Health = 100
        HealthMax = 150
        ArmorMax = 100
        ArmorMin = 0
        AbsorptionRate = 0.3
        EnergyMax = 100
        EnergyMin = 0
        EnergyCur = 100
        EnergyRegen = 5
        BaseSpeed = 280
        MaxSpeed = 560
        bEnergyUsable = TRUE

        PsychicFireSound = SoundCue'A_Weapon_ShockRifle.Cue.A_Weapon_SR_FireCue'
        SacrificeFireSound =
SoundCue'A_Weapon_RocketLauncher.Cue.A_Weapon_RL_GrenadeFire_Cue'
        StunFireSound = SoundCue'A_Weapon_Link.Cue.A_Weapon_Link_FireCue'
        StunnedSound =
        SoundCue'A_Gameplay.CTF.Cue.A_Gameplay_CTF_EnemyFlagReturn01Cue'
}
```

B.1.2 ThaumaAbilities

```
/**
 * DESCRIPTION :    Abilities tiers. Consisting of Neutral, Defense and Offense tiers
 *      In addition here is the definition of the main functions used by all skills.
 *      Good Side:  Neutral + Defense
 *      Evil Side:  Neutral + Offense
 *
 * Written by : Grigoris Nikiforakis for "Thauma" project, 2010 */


class ThaumaAbilities extends Object;

// Variables needed for all abilities
var int EnergyCost;         //energy Cost for the ability
var float Cooldown;         //time needed to use ability again
var float Duration;         //duration time of the ability
var bool bIsAvailable;      //check if ability is ready to use (energy/cooldown)
var string AbilityName;     //name needed in hud
```

```
// Three different skill tiers. Can be used as such in a skill tree in the future
var ThaumaAbilitiesNeutral GoodAbiNeu;          //neutral
var ThaumaAbilitiesDefense GoodAbiDef;          //defensive
var ThaumaAbilitiesOffense GoodAbiOff;          //offensive


//=============================================================
================
// Tiers Constructors.
function AbilitiesNeutralConstructor()
{
        GoodAbiNeu = new class'Thauma2.ThaumaAbilitiesNeutral';
        GoodAbiNeu.AbilitiesConstructor();
}

function AbilitiesDefenseConstructor()
{
        GoodAbiDef = new class'Thauma2.ThaumaAbilitiesDefense';
        GoodAbiDef.AbilitiesConstructor();
}

function AbilitiesOffenseConstructor()
{
        GoodAbiOff = new class'Thauma2.ThaumaAbilitiesOffense';
        GoodAbioff.AbilitiesConstructor();
}

//=============================================================
================
// Delegates for the the common functions that need to know the ability used.
delegate BuffCall( ThaumaPlayer SkillUser)
{
        //each skill has its own buff
}

delegate TimerCall( ThaumaPlayer SkillUser)
{
        //each skill has its own cooldown timer
}

//=============================================================
================
// Functions needed by all abilities.

/**
 * Function that calls the right skill and checks if it can be used.
 * */
function SkillCall(ThaumaAbilities Skill, ThaumaPlayer SkillUser)
{
        //check if player is dead or without a pawn
        if(SkillUser.Pawn==NONE || SkillUser.Pawn.Health <1 )
                `log(">>Abilities:No Valid Player");
        else
        {
                BuffCall = Skill.SkillBuff;        //call the skillbuff funtion of whatever ability is
needed
                TimerCall = Skill.SkillCooldown;    //call the timer of whatever ability is needed

                if (Skill.bIsAvailable)
                {
```

```
                        if(ThaumaPawn(SkillUser.Pawn).EnergyManagement(Skill.EnergyCost))
                        {
                                BuffCall(SkillUser);
                                Skill.bIsAvailable = FALSE;
                                TimerCall(SkillUser);
                        }

                }
                else
                {
                        SkillUser.ClientMessage("I can not use : "$Skill.AbilityName $", yet!");
                }
        }
}

/**
 * Function that handles the main action of the ability.
 * */
simulated function SkillBuff( ThaumaPlayer SkillUser)
{
        //each ability has its own buff.
}

/**
 * Function that calls the timer of the ability.
 * */
function SkillCooldown( ThaumaPlayer SkillUser)
{
        //each ability has its own timer.
}

/**
 * Function that handles the action of the ability after it ends.(Could also be done with states.)
 * */
function SkillEnded( ThaumaPlayer SkillUser)
{
        //each ability has its own ending conditions.
}

/**
 * Check if the ability is ready to use.(cooldown).for a global cooldown a different function is
needed.
 * */
function MakeSkillAvailable(ThaumaAbilities Skill)
{
        if(Skill == NONE || Skill.bIsAvailable == TRUE)
                `log(">>Abilities:this shouldn't happen");
        else
                Skill.bIsAvailable = TRUE;
}

DefaultProperties
{
}
```

B.1.3 ThaumaAbilitiesDefense

```
/**
 * DESCRIPTION :    Defensive abilities tier. Consisting of Armor, Regeneration
 *
  * Written by : Grigoris Nikiforakis for "Thauma" project, 2010 */


class ThaumaAbilitiesDefense extends ThaumaAbilities;


/**
 * The abilities this tier have are in this struct.
 * */
struct AbilitiesDefense
{
        var ThaumaAbilityArmor AbiArm;
        var ThaumaAbilityRegeneration AbiReg;
};

var AbilitiesDefense DefenseTier;


function AbilitiesConstructor()
{
        DefenseTier.AbiArm = new class'Thauma2.ThaumaAbilityArmor';
        DefenseTier.AbiReg = new class'Thauma2.ThaumaAbilityRegeneration';
}


DefaultProperties
{
}
```

B.1.4 ThaumaAbilitySpeed

```
/**
 * DESCRIPTION :    Speed - Player increases his movement Speed for a certain period of time
 *      Cooldown:  8
 *      Duration:  3
 *      Energy:  25
 *
 * Written by : Grigoris Nikiforakis for "Thauma" project, 2010 */

class ThaumaAbilitySpeed extends ThaumaAbilitiesNeutral;


var SoundCue SpeedUseSound;


simulated function SkillBuff( ThaumaPlayer SkillUser)
{
        if(SkillUser.Pawn != NONE )
        {
                SkillUser.SetTimer(Duration, FALSE, 'SpeedEnded');
                if( SkillUser.Pawn.Role < Role_Authority )
                        SkillUser.ServerSpeedBuff();
```

```
                else
                        SkillUser.Pawn.GroundSpeed =
ThaumaPawn(SkillUser.Pawn).MaxSpeed;
                if( SpeedUseSound != None && SkillUser.Pawn != None )
                        SkillUser.Pawn.PlaySound(SpeedUseSound, false, false);
                SkillUser.ClientMessage("My feet feel too light!");

        }
        else
                `log(">>AbilitySpeed:this shouldn't happen");
}

function SkillCooldown(ThaumaPlayer SkillUser)
{
        SkillUser.SetTimer(Cooldown,FALSE,'MakeAvailableSpeed');
}

function SkillEnded(ThaumaPlayer SkillUser)
{
        if( SkillUser.Pawn.Role < Role_Authority )
                SkillUser.ServerSpeedEnded();
        else
                SkillUser.Pawn.GroundSpeed = ThaumaPawn(SkillUser.Pawn).BaseSpeed;
        SkillUser.ClientMessage("My feet feel heavy again!");
}


DefaultProperties
{
        EnergyCost = 25
        Cooldown = 8.0
        Duration = 3.0
        bIsAvailable = TRUE
        AbilityName = "Speed"

        SpeedUseSound =
SoundCue'A_Pickups_Powerups.PowerUps.A_Powerup_UDamage_PickupCue'
}
```

B.1.5 ThaumaAbilityStun

```
/**
 * DESCRIPTION :    Stun - Next energy-ball attack stuns opponent if hit.
 *      Cooldown:  16
 *      Duration:  2
 *      Energy:  35
 *
 * Written by : Grigoris Nikiforakis for "Thauma" project, 2010 */

class ThaumaAbilityStun extends ThaumaAbilitiesOffense;


simulated function SkillBuff( ThaumaPlayer SkillUser)
{
        if(SkillUser.Pawn != NONE )
        {
                //Only client uses Stun. If server can use it too, some changes are needed in
Pawn's CheckForStun function.
```

```
                if( SkillUser.Pawn.Role < Role_Authority )
                        SkillUser.ServerStunBuff();
                SkillUser.ClientMessage("I am aiming for his kidney.");
        }
        else
                `log(">>AbilityStun:this shouldn't happen");
}

function SkillCooldown(ThaumaPlayer SkillUser)
{
        SkillUser.SetTimer(Cooldown,FALSE,'MakeAvailableStun');
}

//stun ended for the stunned player!
function SkillEnded(ThaumaPlayer SkillUser)
{
        SkillUser.SpeedEnded();
        ThaumaPawn(SkillUser.Pawn).bEnergyUsable = TRUE;
}


DefaultProperties
{
        EnergyCost = 35
        Cooldown = 16.0
        Duration = 2.0
        bIsAvailable = TRUE
        AbilityName = "Stun"
}
```

Bibliography

[1] Csikszentmihalyi, M. (1990). Flow: The Psychology of Optimal Experience. New York: Harper and Row

[2] Daniel Terdiman, Behind the Prototyping of 'Spore', http://news.cnet.com/8301-13772_3-10033443-52.html

[3] Ella Tallyn, Applying Narrative Theory to the Design of Digital Interactive Narratives with the Aim of Understanding and Enhancing Participant Engagement

[4] Epic's UDK engine, http://www.udk.com/features

[5] Fulton, B., & Medlock, M. (2003). Beyond Focus Groups: Getting More Useful Feedback from Consumers. Game Developer's Conference 2003 Proceedings, San Jose CA, March 2003. , http://mgsuserresearch.com/publications/

[6] Georgios N. Yannakakis and John Hallam, Capturing Player Enjoyment in Computer Games

[7] Jenova Chen, Flow in Games (and Everything Else), Communications of the ACM April 2007/Vol. 50, No. 4

[8] Jesse Schell CMU Professor, DICE 2010: "Design Outside the Box" Presentation, http://e3.g4tv.com/videos/44277/DICE-2010-Design-Outside-the-Box-Presentation/

[9] Koivisto Elina, Suomela Riku, Using Prototypes in Early Pervasive game Development, Sandbox Symposium ACM 2007

[10]Mark Griffiths, Video Games and Health, http://www.bmj.com/content/331/7509/122.full

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

[12] Romero, R. (2008). Successful Instrumentation: Tracking Attitudes and Behaviors to Improve Games. Presented at the Game Developer's Conference, San Jose CA, February 2008, http://mgsuserresearch.com/publications/

[13] The Agile Alliance, http://agilemanifesto.org/

[14] The Cabal: Valve's Design Process For Creating *Half-Life,* http://www.gamasutra.com/view/feature/3408/the_cabal_valves_design_process_.php

Useful References

[-] Articles on Game Development, www.gamasutra.com/ , www.gamespot.com

[-] Charles Platt , Interactive Entertainment article, http://www.wired.com/wired/archive/3.09/interactive.html

[-] Gamboa, M., Kowalewski, R., & Roy, P. (2004), Playtesting Strategies, Presented at the Game Developer's Conference, San Jose CA, March 2004. http://mgsuserresearch.com/publications/

[-] Game Design Course, Univestity of Utrecht http://www.cs.uu.nl/docs/vakken/gds/

[-] Game development Course, ECE 495/595; CS 491/591, University of New Mexico, http://www.cs.unm.edu/~angel/GAME/Game%20Development%20--First%20steps.pdf

[-] Gamer's Quest to save the World , McGonigal Jane : Gaming can make a better world, http://www.ted.com/talks/jane_mcgonigal_gaming_can_make_a_better_world.html

[-] HCI Course, Stauros Christodoulakis , Technical University of Crete

[-] Jim Thompson 's, The Computer Game Design Course, Thames & Hudson

[15] Michael Mateas and Andrew Stern, Natural Language understanding in Facade:Surface-text Processing, Best Paper Award, Technologies for Interactive Digital Storytelling and Entertainment (TIDSE), Darmstadt, Germany, June 2004 , http://www.interactivestory.net/papers/MateasSternTIDSE04.pdf

[-] Neils Clark, The Sensible Side of Immersion,

http://www.gamasutra.com/view/feature/4265/the_sensible_side_of_immersion.php

[-] Thauma TUC Version , http://rapidshare.com/files/430261715/UDKInstall-Thauma.exe

[16] Thauma Intro Video , http://www.youtube.com/watch?v=to7-pClGxLg

[-] Thomas Maarup, HEX thesis, http://maarup.net/thomas/hex/

[-] Wikipedia, The free Encyclopedia, www.wikipedia.com

[-] Zervoudakis Nikos, TUC UnderGraduate Thesis ,3D game development using a rendering engine