MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

GAME-ORIENTED TOOLS LAYERED ON UNITY 3D GAME ENGINE

LICENSE THESIS

Graduate: Virgil Emilian ANDREIEŞ
Supervisor: Prof. Dr. Eng. Dorian GORGAN

2013

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
Prof. dr. eng. Rodica POTOLEA

Graduate: **Virgil Emilian ANDREIEŞ**

GAME-ORIENTED TOOLS LAYERED ON UNITY 3D GAME ENGINE

1. **Project proposal:** *Short description of the license thesis and initial data*
2. **Project contents:** *(enumerate the main component parts) Presentation page, advisor's evaluation, title of chapter 1, title of chapter 2, ..., title of chapter n, bibliography, appendices.*
3. **Place of documentation:** *Example*: Technical University of Cluj-Napoca, Computer Science Department
4. **Consultants:**
5. **Date of issue of the proposal:** November 1, 2012
6. **Date of delivery:** June 4, 2013

Graduate: _____

Supervisor: _____

MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind autenticitatea lucrării de licenţă

Subsemnatul(a) _____, legitimat(ă) cu _____ seria _____ nr. _____
CNP _____, autorul lucrării _____
_____
_____
elaborată în vederea susţinerii examenului de finalizare a studiilor de licenţă la Facultatea de Automatică şi Calculatoare, Specializarea _____ din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea _____ a anului universitar _____, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

În cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data _____

Nume, Prenume _____

Semnătura

Abstract

dsad dskjdlajdklasj ask jdlasjdla dkjsald jkdasdkjalsd dksajldja

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction - Project Context

Modern day society is evolving extremely fast, making life easier, more comfortable and complete for each and every one of us. This is a direct consequence of the rapidly increasing rate at which technology, in general, is evolving. It takes only a moment to look around into the house and we can see how depend we are on various devices that make our life a little bit easier. From the earliest days of the Industrial Revolution and the discovery of the power of electricity and its use, to modern day telecommunication systems, computer systems and the Internet, technology has never ceased to evolve and lead to incredible breakthroughs, one after another. Computer Games and Video Games make no exception to this. As technology progressed, video games became more popular and their limits in today's society. Nowadays, video games have progressed so much that are now considered a standalone industry with annual revenues of 67 billion dollars in 2012 [Forbes] that rival the film industry. News reports forecast that global video game industry will reach 82 billion by 2017.[Forbes]. The forecast includes PC games, dedicated console hardware and software, dedicated portable hardware and software and games for mobile devices such as mobile phones, tablets, music players and other devices that run games as one of their features. The word game is most of the times associated with games like chess, checkers or card games, such as poker, blackjack, 21. These games have been present in our life from an early age till the end. Other games like Monopoly could create world of their own that would immerse the players into countless of hours were they would be able to orchestrate and manage a virtual business. On the other hand, games like Hangman and variations of Tic-Tac-Toe, Sudoku put emphasis on the memory and logical aspects of the human brain.Then, after the introduction of personal computers in our life and, of course, the Internet, we were amazed that these devices were able to run our favorite childhood games at the pressing of a mouse click or a keyboard. The first version of a "game" was introduced in the early 1950's when the first computer game with graphics was designed. It was a Tic-Tac-Toe game programmed on a computer that had a cathode ray display. Later that decade another game called "Tennis for Two" and it was played on an oscilloscope. The first video game programmed specifically for computer use was "SpaceWar" by Steve Russell in 1962, which later became one of the founders of "Atari Computers" In 1971 the

first arcade game was created, by Nolan Bushnell and Ted Dabney, being called Computer Space and it was largely based on Russell's game SpaceWar . A year later they started "Atari Computers" after releasing "Pong", and they re-released it in 1975 as a home game. It featured simple 2 dimensional graphics. The main idea of the game is to defeat your opponent in a simulated table tennis game. This is done by earning a higher score. These were some of the most important releases that have contributed to the development of this industry. As hardware and software evolved in time so did video games improving a lot since the earliest days of "Pong". Nowadays, video games are played on PCs and consoles. Their advanced software is responsible for offering the wide variety of game-plays, game worlds and gaming experiences. Although the "gaming industry" it is an industry, no doubt about it, it is still considered young craft. Games span from one place t another, and they are globally known. Video games are different and fall under different "genres". These are basically categories or in technical terms, applying different mechanics and set of rules so that the experienced is varied. Since there are a lot of different types of games on various platforms, there are just as many solutions to approach game design. An ideal solution does not exist in terms of software.

Artificial Intelligence in games is what attracts people to playing countless hours. It is very similar to real life. With each release, AI is becoming increasingly intelligent, and with the help increased processing powers of nowadays hardware it is becoming better and better. Another revolutionary constructs in modern video games is the use of Physics. This contributes to incredible realism in objects interaction. Vast improvements have been brought to the visual component of the Game. Game Graphics nowadays is probably the most impressive aspect of video games. Certainly it is the one that captures the eye first. The main reasons technology has opened the door for amazing graphics is through hardware upgrades. Consequently, with today's high speed hardware, more pixels can be displayed, instructions and shading to take place - sometimes on a massive scale. The early creations were produced by engineers. As time passed by and the video game became more and more complex, the development required people from many other disciplines (graphic designers, game designers, composers, writers). Although these various people are extremely important in the development process, the technology designed and created by the engineers influenced how much freedom and creativity every other contributor can have. They are responsible with creating the core and main functionality of each game. Because of this, video games today are compatible with PCs, mobile phones, consoles, tablets with their own unique OS. It would not be possible to create the impressive realistic graphics and sounds, without the tools and programs that allow these games to be created. Designing 3D graphics models for future games requires strong knowledge of mathematical concepts like quaternions, points, Cartesian systems, Euler angles, and vector and matrix math to render these graphics. Physics engines imply an understanding of different types of real world physics to be simulated. A lot of games require the existence of gravity, density, water and liquid viscosity to be rendered. Other games may contain specific physics needs. These games Fall in the category of sports, racing and fighting. On the other hand, most games require specific artificial intelligence. It is the logic behind

the set of actions of the player's computer-run opponents. Tools used in building games require support for scripting, building game levels, and importing and converting art. Another important aspect of an engineer is the porting aspect. Porting refers to converting game from one platform to another so that it can be run on both of them. Not once but many times has been said that video games are the cutting edge of new computer, interactive graphics and sound technology. Computer games, mobile games or video games are probably the pinnacle of computer programming, graphical modeling and visual design. They intersect a lot of areas and subfields that compose the Computer Science field. From a visual representation and graphics design models it is strongly connected to the subfield of Computer Graphics, Computational Geometry and Visualization and Image Processing. Regarding Game Mechanics and System Design we can clearly identify the fields of Programming Language Theory and Artificial Intelligence. Furthermore, if we bring into discussion the interactive nature and features of a video game, we are talking about Human-Computer Interaction and User Interface Design. The process also falls under the incidence of the steps phases that form the discipline of Software Engineering. The process of design and development of a video game is a long process that involves care and attention to a lot of factors. Technology in computer science has allowed free game development to be accessible to anyone. Tools and Game Engines such as Unity 3D allow developers, designers, engineers and programmers to create video games. As indies (aka individual, singular developers) are more and more frequent there has been an influx of game engines. There is a wide variety of paid game engines that are very performant and provide large capabilities for developing video games in short amount of time. Unity 3D represents a full pledged game engine that contains all the features, capabilities and functionality that anyone would need in order to create full 3D and 2D video games that are multi-platform. It has a free indie version as well as commercial license version. A game engine is nothing else that the system designated for the development of video games by providing a software framework that the developers make use of. Typically, it includes a physics engine, collision detection, sound, scripting, animation, artificial intelligence, networking tools, streaming, localization, scene graph, memory management and threading and last but not least, a rendering engine. Game design is still a young industry, but it is maturing fast. Computer games have been around for over 30 years. It is also very important to distinguish between the game creative design and game software design. The Game Creative Design provides the artistic vision behind the game. It also implies that software will be developed. On the other hand, Game Software Design handles the set features that the game will have and it revolves around implementing them. Because games, computer and video games are such a fascinating subject and like it or not are present in our daily life, I have decide to approach the area of Game Design and Programming and Production as my Bachelor's Degree Diploma Thesis.

Chapter 2

Project Objectives and Specifications

2.1 Area and Domain of Study

Game design and development represents a complex domain that intersects a lot of disciplines, areas and sub-domains of computer science. It cannot be said that game design and/or game development is a direct branch of computer science but rather a process that involves various tasks be completed by having as support those mentioned areas. Furthermore, it creates a final product, a game (computer, console game, mobile game) that is released on the market in order to make profit.

The most important areas that game design and development approaches are the following: Computer Graphics and Visualization, Artificial intelligence, Software Engineering, Algorithms and Data Structures and Programming Language Theory. Other areas are Computer Networks, Concurrent, Parallel and Distributed systems. Besides these vital areas there are some that relate to Project Management and Methodology, Marketing and Sales. Creating a game is a difficult task and requires the developer to be aware of.

Hence, during the analysis, design and implementation, the following areas will be approached with the main goal of achieving the main objective and the secondary objectives:

Computer graphics and Visualization represents a branch of computer science that focuses on the study and research of digital visual contents as well as manipulations of image data. This area is highly connected and correlated with other fields like computational geometry, image processing but it is mostly and most heavily applied in video games. That is why it holds such an important role in the game development process since it provides for the visual aspects, components, objects, models and interpretations of the game. During the game production, in game design, many artists submit various conceptual art and visual designs to be used in the upcoming game.

Artificial intelligence (short A.I) synthesizes goal-oriented processes(problem solving, decision making, adaption to environment, learning from environment and reproducing the knowledge accumulated). However, in video games, A.I is used for implementing algorithms, solutions, techniques and strategies that offer the "idea" that player characters

and NPC's(non-playable-characters) actually have a level of intelligence and knowledge and interact with the player from their own free will. It is centered on the illusion or appearance of intelligence so that it provides a better game-play and a better experience. Software Engineering represents a formal discipline that is dedicated to enhancing the quality of software development processes and the products that result from them.

Quality in a software product implies the fact that the product is also reliable, extensible and maintainable. If a software development process possesses quality it results that it is and engineering process and something that is open to continuous improvement. Creating and making games, in a lot of cases involves some of them most complex software development efforts to be found in any other discipline. Software Engineering starts with the primordial notion that it is important to learn from what you do and apply what you learn in and incremental manner. The categories that emerge after technique and knowledge is applied are the following: time, energy, efficiency, reliability, maintainability and appropriateness. Software Engineering practices: define a scope, provide an architecture, provide planning for each iteration and phase, test your product frequently for verification and validation and implement. Data Structures and Algorithms provide the most important tools in implementing, evaluating and optimizing solutions for the core game mechanism. It approaches such areas such as algorithm analysis, algorithm stating and optimization, handling data structures as well as handling computational geometry. Data structures and computational geometry are essential in the process of video game development since they are the fundaments of some of the principles that hold the game engines that create the games.

Programming language theory is a branch of computer science that deals with the analyzing, designing and implementing, classifying programming languages and their individual features. It is strongly connected with software engineering and linguistics. Basically, in the context of game design and development refers to selecting and studying the appropriate programming language that corresponds to the requirements of the system.

Computer Networks, concurrent, parallel and distributed systems along with databases and information retrieval are relevant if it is required from a video game to support multiplayer and it is required to have good, reliable and fast connections over the Internet.

Consequently, in order to develop a video game it is essential to approach most if not all of the above areas, domains and fields of computer science.

2.2 Theme Description

The main theme of the diploma thesis revolves around the video game design, development and production. The basic idea is that using a reliable well known and powerful game engine try to use its capabilities, functionalities and features to create with the help of a specific IDE and programming language some components and tools that facilitate various game implementation and offer a support for future designs and developments.

As stated in Chapter 2.1 there is a very large spectrum of fields, domains and sub-domains that are approached and they target computer science engineering as well as digital art, conceptual art and Project Management.

The most essential topics that are approached are:

Computer Graphics - the creation, modeling and texturing of graphical models, creating and assigning skeletons (armatures) and animations to these models, importing them in an adequate format and manipulating them via the game engine.

Artificial Intelligence - selecting the appropriate algorithms, techniques and strategies for the A.I. This can be done doing extensive research on what are the best solutions for the desired category of video game. There are AI solutions for various types of game types.

Graphical User Interface Design and Human-Computer Interaction, User Interface Design âĂŞ designing a graphical interface that conforms with the requirements of both the game and with its theme. Also handling the interaction between the system and the user, in such a way that it can attract him for future use.

Software Engineering occupies a pivotal role in this process. It offers methodologies and best practices for analyzing, designing, implementing software and it also ensures features like quality, affordability, maintainability, efficiency, time-efficiency. It's main concern, beside the actual manufacturing of a new software product, it deals with organization of the software. It is a basis for the development of any application, especially one as laborious and time-intensive like game design, development and production.

The programming language selecting for the actual implementation phase is largely based upon the actual game engine used as well as for constraints regarding data structures and algorithms.

2.3 Main Objective

- The main goal of this project is to study, design and implement a set of game-specific tools that facilitate easy and fast development of video games. Furthermore, these tasks should represent the basic mechanism and components that interact with the game engine.

- Therefore, this inter-dependence between the game engine and these tools results in a layered structure. In addition to this, achieving this task is done by making use of the capabilities, functionality, assets and plug-ins offered by Unity 3D Engine. Unity offers great support and learning platforms together with instructional, being very easy to understand and quick to learn.

- A very important sub-goal that emerges from the main objective represents the exemplification of how these tools work and interact with each other by the implementation of a small video game that of course can be subjected to further changes and improvements. This small video game should highlight the main idea behind

the tools created along with Unity's large variety of set of tolls. The result of this process will be a relatively small game. This game will need to comply with all of the requirements and fulfill all of the tasks that it is set up to do.

2.4 Secondary Objectives

The secondary objectives are derived from the main objective. These objectives should meet the SMART guide for project management. This mnemonic conforms in a broad sense with the following concepts: Specific, Measurable, attainable, relevant and time-bound.

- *Specific* means that it needs a specific goal to be clear and unambiguous. It should be said what is expected, why it is important, which attributes are the most important and why it is important.
- *Measurable* highlights the need to provide concrete criteria in order to measure progress towards obtaining the desired goal. It helps to stay on right path and achieve the final goal or the main objective.
- *Attainable* describes the importance of the goals to be realistic and attainable. Therefore, the goal should not be out of reach or below a standard performance. It answers the question: How can the goal be accomplished.
- *Relevant* means that choosing the goals that matter is of great importance since they drive the project forward.
- *Time-bound* stresses the importance of placing the goals within a specific time-frame and giving them a target date or a milestone.

Consequently, the main objective of this diploma project is divided into the following phases: Requirements, Analysis, Design, Implementation and Testing, and Evaluation Metrics. These phases highlight the main flow of the project and offer a clear logical separation of work for goal achievement.

These phases will be presented below in detail and following the description, it should clearly the 5 SMART acronyms.

Requirements

This represents the first and initial phase of the development process. The main problem of not having a specific set of tools for game development for better game implementation is clearly stated. Proper justification and explanation regarding the need for such a set of tools is given. This phase identifies functionality, performance levels, terms that need familiarization. In the context of this project it identifies the most important

tasks that need to be fulfilled by the tools. On the other hand it should correlate these tasks with the functionality that the game engine offers. Furthermore, it is essential to identify the most important elements that will arise from the development process of a game. Another important aspect is the problem statement. Identify both the functional and non-functional requirements along with constraints that arise. The requirements phase does not need to take a very long time and it should last no more than 2 weeks and they represent a foundation for the remaining phases of the development process.

Analysis

This phase represents filtering domain-specific literature, gathering essential information, extracting the aspects, elements, algorithms, models, strategies, patterns that are directly linked with the problem domain and leave behind unwanted material. Research and study is needed along with investigations on written papers books and other domain âĂŞ centric materials. Furthermore it is almost vital to read documentations, follow instructional programs, and get familiar with Unity 3D game Engine.

This is the base and foundation for creating our tools. Iterate through documents, manuals and books and extract, summarize and synthesize the aspects that are of vital interest for the project. During this stage it is also essential to have a clear understanding of the principles that stand behind game engine: localization support and scene graph require a high level understanding of mathematics, algebra, descriptive geometry and perspective and/or orthogonal space, physics engine and collision detection involve basic understanding of physical laws, rendering engine, animation implies and understanding of domains such as computer graphics and visualization, artificial intelligence demands study in artificial intelligence in video games, networking, streaming, memory management and threading require skills in the areas of computer networks, parallel and distributed systems and database management. Last but not least, it is important to have knowledge and skills in the programming language area (this will be treated in future Implementation chapter).

Analysis involves also documentation and research about the graphical modeling software tools that will be used for the creation of 3D models required in the game. Furthermore it is decided what graphical tools suit the solution better.

This phase is extremely relevant for the upcoming phases since it provides a general overview of how the system will function, it identifies the system's main components, packages, domain model, their roles and interactions with each other and also interaction with related systems and/or components.

This phase is also associated with providing an Use-Case model that maps the requirements into use cases and groups the relevant ones in scenarios. Therefore typical scenarios and atypical scenarios are presented. The main flow of the system is provided and described in detail

The time-frame associated with this phase is about 2 and a half months since it requires intensive study and research, training and familiarizing with the game engine, and also summarizing, synthesizing and providing a basic architecture of the system.

Design

The next phase represents the design of the tools and the video game. During this phase the both the requirements specification (2.4.1 Requirements) and the analysis specification (2.4.2 Analysis) are used in order to produce a detailed design of the game. It provides a detailed specification for each component and tool, describes in detail interfaces and functions. As previously stated all knowledge gathered and summarized is processed, and filtered to match fully the requirements. The system design is about the system's behavioral design affecting the design of the components that include all applicable items. After examining the software components and their interfaces it is required to build a physical model after these recognized methods. This physical solution describes the solution in concrete terms. The logical model defined in previous phases serves as a basis to the structure of the problem.

Firstly, during this phase the architecture of the system is established. It maps the requirements onto the architecture. The architecture defines the components behaviors and interfaces. This also offers details about programming languages and environment used, machines, package hierarchy, the architecture layering, factors regarding memory, platforms, artificial intelligence algorithms data structures, data types and it is associated with the "hard" engineering phase.

It is also essential to provide a test plan which is a result of taking the architecture and converting the typical scenarios. It also provides critical information and priorities for the future base, the Implementation phases (2.4.4 Implementation). The architectural design phase defines the software in terms of the major software components and interfaces.

Secondly, it involves the definition of the hardware, software and network components(if any). This involves the game engine, the software development tool and environment, programming language-specific descriptions. Ii also involves the description of the graphical modeling tools and editors. The graphical models are acquired, created, modeled, textured, fine-tuned. Adding to this, functional and technical requirements for system integration are presented.

Next, the phase should also relate to other similar works in the field and realize brief comparisons between the 2 systems, highlight advantages and disadvantages, propose other solutions and justify with valid arguments the proposed solution.

The result of the design phase should be: the system architecture, a menu design, general GUI design, user groups, class list, class diagrams, class attributes and class diagrams in UML(Unified modeling language), sequence diagrams in UML.

Last, this phase should be closely correlated with the previous stage (2.4.2 Analysis) and, most importantly with the following phase, the Implementation phase (2.4.4) Implementation.

Implementation

The implementation phase represents the phase that is somehow similar to a manufacturing process. The components are created either from scratch or using composition.

Using the requirements from phases 2.4.1 and 2.4.2 and the general system architecture, the GUI design, class lists and diagrams, class attributes and the UML models, it is needed to create exactly what it has been requested. On the other hand, it is essential to not forget about innovation and providing flexibility as well as well as risk management and mitigation. This phase emphasizes on such issues like quality, performance, baselines, debugging and libraries.

At the 2.4.3 step, the Design, the conceptual design and architecture is produced. This serves as a support for this phase whose main goal is developing and producing the actual product. Therefore, a robust architecture along with an understandable pan are highly correlated and equally important.

Firstly, it is required to use the designated graphical tool to create, model, fine-tune, convert the graphical components of the game. As known, the human mind is extremely visual and graphics is one of the most important attributes that set video games apart from other software products. The models are only means of creating the tools and take vital part in the implementation part of the game.

Secondly, the algorithms that are artificial intelligence-oriented needed to be written, translated and optimized to work in the actual context following the actual requirements. At the analysis and design phase they are presented as pseudo-code and they are converted into the required programming language. Other AI techniques and solutions are evaluated and integrated together with other components.

Thirdly, the game mechanics system is implemented. It deals with issues such as game-play, player input, player creation, player movement, handling events. The actual game rules system pays a pivotal role in this phase since rules and constraints are translated into actual code.

Finally, these created components and tools need to be integrated as they collaborate one with another. This is where the game engine takes the most important part. Having the capabilities to support and, moreover, encourage integration from high-level to low-levels, it should serve as a support for creating the actual implementation of the game through the tools.

The Class Model implementation represents the proposed solution for resolving the requirements stated in earlier phases. It does not mean, however that it is the only solution nor that it is the best solution in other contexts. It only represents the solution offered using the resources at our disposal. Selecting the appropriate high-level programming language is extremely important.

This particular stage should not take more than 1 month and a half.

Testing and Validation

The next phase represents testing the solution that is created during the previous phase. As it is very well known, quality and robustness is very important, and in some cases it is a deciding factor as to how well a software product has been engineered. Regardless if testing is done in a late stage or continuously, iteratively, testing relies on some major

aspects like internal unit, application and stress. They are offered from the perspective of the system provider.

Video games or computer games in particular are subjected to a lot of bugs and this problem is very frequent. Being able to reduce critical errors and bugs and also reduce as much as possible their total numbers is a matter of testing and validating a system. It is also essential that for each and every use case there must exist a test case to ensure that the game system meets all the requirements. If critical problems are uncovered, that component or subsystem is set back to the analysis and design phases where it is analyzed and offered a new design and again subjected to tests.

Evaluation Metrics

This last phase contains comparisons to other related games or other work. Since the application is largely based on the processing power of the processor and the GPU or Video Card it is essential to subject the software to performance metrics that evaluate from hardware and software perspective the amount of memory used, average frame-per-seconds, average frame-drops, amount of memory used by the GPU. It tackles such problems like rendering, pixel level operations, depth complexity, color depth, resolution, anti-aliasing, applying benchmarks (graphics, application, biases). Other issues such as lighting and animations need to be considered. Other primitive tests like fill rate, fixed rate and polygon rate need to be issued. Although there are numerous specific tools for providing these metrics singularly, the game engine already comes with some basic metrics for performance and scalability measurement that are optimized to work under the project's specifications, constraints and requirements. On the other hand it is quite important to relate to other related work that have been evaluated under the same conditions.

Project Specifications

Specifications in the context of a software engineering process is highly related with stating the requirements of the system. The main goal of the project, as stated in previous phases is the study, design and implementation of a set of game-specific tools that facilitate easy and fast development of video games. Furthermore, these tasks should represent the basic mechanism and components that interact with the game engine. The layered character of the design emerges from the main objective and goal. The game engine serves as a supporting tool that offers the capabilities for developing the required set of tools. One layer above, at a higher level, the tools components and subsystems are created. One layer above there is the actual game model implementation that makes use of these set of tools. The layer of abstraction between the tools and the actual implementation is not very clear but it was introduced in order to highlight that the actual game implementation is only one solution that these specific tools solved and that other solutions, i.e. game implementations are more than likely possible.

In most software engineering processes and methodologies, the technologies play an

important part but they are not a crucial part of the actual process. However, in the case of this project, technologies play an essential and crucial part.

The game engine selected for the tasks that are going to be fulfilled in this project is Unity 3D Engine. It represents a cross-platform game engine and IDE developed by Unity Technologies and it targets web plug-ins, desktop platforms and mobile devices. It contains all the features, capabilities and functionalities that are essential to create full 3D and 2D video games that are multi-platform. It has a free indie version as well as commercial license version.

Graphical modelling tools are also extremely important since they are the support for the creation of 3D models that will be later used in the game. Free modelling tools like Blender and paid ones like 3D Studio Max offer extremely powerful functionality and are compatible with Unity.

The selected game genre for the game implementation is the RPG (Role playing game). It is gaming genre where the player assumes the roles of fictional characters and take responsibility for these role in a narrative setting through structured decision making and character improvement and/development and actions are based upon a system of rules. Basically the player decides which action he is going to take. The game system will contain an artificial intelligence, an explore system, a combat system, an item management system and object interaction system and an attribute assignment system.

Video games/computer games are user-oriented pieces of software. The user is identified with the player in a video game. The player can create a new playable character instance, can spend available points on the attributes generating his desired version or class of character. After the assignment of attributes, he can create his own custom avatar/appearance. These player preferences are of course saved along with the version for later use so that later the player can load them. He can also delete his avatar and create a new one. The RPG genre is associated with a an open world that the created avatar can explore and perform different actions, interact with other objects or NPCs and enemy AI.

Main ideas: The application will be developed using Unity 3D Game Engine (pre-existent free engine), game is included in the RPG(Role-Playing-Game) genre and it is designated to PC platform although it can be ported onto multiple platforms.

Expected Results

Some of the most important aspects that will be treated throughout the project:

1. Studying the Unity 3D Game Engine and getting accustomed to the various features that this free tool offers, and finally create, design, develop and implement a game application that uses the functionalities of this free, open-source game engine.

2. Make use of 3D Graphic Modeling Tools such as 3D Studio Max and Blender for creating the 3D models needed for the game application. These models are compatible with Unity 3D and can be imported in Unity as such

3. Make use of a high-level programming language adequate for this situation that offers at the same time robustness and flexibility, but at the same time is supported by the Unity 3D Game Engine as a scripting language. Therefore C# is going to be used.

4. Develop simple but effective algorithms regarding the AI (Artificial Intelligence) that contribute to the application's development.

5. 5. Approach the fields of Human âĂŞ Computer Interaction, by developing a user-friendly appropriate user-interface (User Interface Design) that offers highly customizable features. The user/player has at its disposal an entire world to explore and can create a main character after its preferences.

6. Get through the phases of Game Design and Development Methodology (as well as Software Engineering).

7. Incorporate the Object Oriented Programming paradigms, principles and concepts in the development of the application.

8. Incorporate the Object Oriented Programming paradigms, principles and concepts in the development of the application.

9. It could be a standalone application or a web application.

The final application should be open and subject to further improvements and modifications, addition of levels, 3d objects, textures, customizable options, menus, controls, etc.

Project Timeline

Week	Period	Research	Implementation	Writing
1 - 8	29 oct - 22 dec	Bibliographic Study	None	Once every 2 weeks short report
14	11 feb - 17 feb	Bibliographic Study; Chosen technologies and tools	None	Once every 2 weeks short report
15	18 feb - 24 feb	Theoretical Background	None	Project Objectives and Specifications
16	25 feb - 3 mar	Theoretical Background	Creating 3D models	None
17	4 mar - 10 mar	Requirements	Creating 3D models	None
18	11 mar - 17 mar	Conceptual Architecture	Testing Unity 3D	Conceptual Architecture
19	18 mar - 24 mar	Analysis and Design	Creating 3D models	Detailed Architecture
20	25 mar - 31 mar	Analysis and Design	Implement basic classes	None
21	29 mar - 4 apr	Analysis and Design	Implement basic classes and A.I.	None
22	1 apr - 7 apr		Incremental and Iterative Implementation	Introduction
23	8 apr - 14 apr			Project Objectives and Specifications
24	15 apr - 21 apr		Implementation	Bibliographic Reasearch
25	22 apr - 28 apr		Implementation	Analysis and Theoretical Foundation
26	29 apr - 5 may		Implementation and fine tuning	Implementation
27 - 28	13 may - 26 may		Experiments, tests, evaluation	Testing and validation and conclusions

Table 2.1: Project Timeline

Chapter 3

Bibliographic research

In the previous chapters the main domain and area of interest were stated along with the theme description. This inevitably lead to the statement of the problem that this project is trying to resolve. As game design and development is somewhat a new and up and coming field, it clearly reaches beyond the actual computer science. It also intersects other fields like art, digital art, story generation and creation for the levels and it is considered to be the pinnacle of computer programming and computer graphics and visualization. Furthermore, its complexity is obvious and that is why it is considered in our days a true and powerful industry.

A variety of experts, professors, software engineers, graphical engineers, audio engineers, physics engine programmers, artificial intelligence programmers, network programmers have released numerous materials such as articles, books, papers, manuals, technical documentations, tutorials and training methodology. They shared their knowledge with the world so that game industry has become what it is today.

3.1 Game Development

John Flynt and Omar Salem, 2005, in [1] describes the process of creation and development of a video game through the vision of a software engineer. In this book the most important practices, methodologies and designs are emphasized making use of OOP, configuration management, UML diagrams, patterns, process improvements and so forth. The authors provide a clear systematic approach to video game development from a software engineer's perspective. They assume that a video game is nothing else than a software product like any other one and that it is subjected to software engineering methodologies, phases and management. It is stated that OOP (object oriented programming), C++, C# and Java are just tools of the trade. It tackles problems like: clear statement of the requirements, reuse, pattern applicability, risk analysis, iterating design, configuration management, development strategies, process improvement, planning and management and documentation elicitation. In conclusion, software engineering leads to better soft-

ware products, this book tries to teach readers on how to develop games according to a design and follow standardized approach to game development. The book also provides exercises that highlight or illustrate that show how software engineering practices can improve your game. The basic categories, phases and stagers are covered. The book is aimed towards architects, generalists, designers, programmers, software engineers and game developers seeking knowledge about standard frameworks for games. The three frameworks presented are: function, object-oriented, and patterned. Design documentation also plays an important role in the present book. It is a professional, formal engineering approach to game development.

Erik Bethke published [2] in 2003 this book having the goal to realize the discipline of game production in a formal, yet widely appealing treatment and that the book is aimed towards the making of digital interactive entertainment software âĂŞ games. This book gives you specific tools for the management of your game, methods to create a project plan and track tasks, an overview of outsourcing parts of your project, and philosophical tools to help you solve abstract production problems. It is stated that the subject of video game production has never been pproached by a lot of books, research studies or articles. On the other hand, art for video games, programming for games have been tackled numerous times and there is a significant amount of literature that focuses on these aspects. It can be associated with project management but it is said that project managemnet does not fully encompass the skills needed to manage game development. Therefore this book includes elements of project management, engineering discipline and the essential ingredients in game production.

Richard Rouse III, 2004, in [3] Game Design: Theory & Practice balances a discussion of the essential concepts behind game design with explanation of how to implement them in your current project. Detailed analysis of successful games along with concrete examples from personal experience are presented. He assumes some little familiarity with game design. In the beginning it focuses more on theory, what the gamers want, the idea of a game, the elements of game-play. As it progresses , it addresses more practical concerns like how to design levels for games, how to test them. There are numerous interviews regarding best practices in the domain and area. The author takes closer look at various games and deconstructs them in an attempt to explain what makes them well designed and good games.

Daniel Schuller in [3], C# Game Programming: For Serious Game Creation shows programmers how to write simple, clean, and reliable code step-by-step through the creation of a basic game. The game is built using C#, a high-level programming language, and OpenGL, an industry favorite for graphics display. You'll get an overview of the methods and libraries used to build good games, learn how to use those libraries and create your own, and finally build your own scrolling shooter game. You'll even find tips and information on how to develop your own game ideas and you'll have an excellent code base to work with. C# Game Programming: For Serious Game Creation provides you with all the information you need to take your game ideas from concept to completion. The book is constructed around the Tao C# framework, and any input (keyboard, mouse, joypad)

and output(OpenGL, screen, OpenAL sound). C#, unlike other programming languages or scripting languages, has grown into a generalized language available on several platforms. C# primary language for Unity3D and MonoTouch. And while some of the code in this book might be useful for those platforms, much of it will not, as they have their own I/O abstractions. While C# spans several platforms, this book is a Windows-only thing. The book is intended for seasoned game programmers that have experience in C# programming. It teaches the reader how to use OpenGL and C# together, how to use the latest versions of C# to write tighter, leaner code, how to create the fastest game loop for C# using a Windows Form, get useful game functions such as timers, 2D and 3D math, bitmap font rendering, accelerated 2D game programming, creation of small-time fast game engine, game development techniques like pragmatism and project management. Unit testing and animation are covered. The user learns how to make the game that he wants.

Roger E. Pedersen, 2009, in [4] Game Design Foundations provide and easy-to follow process for game designers to express their concepts and begin the development process. The author is a game designer and he explores the various game genres, discusses research processes, and offers explanations regarding the basics of artwork, programming, sound, scripting testing. The book is filled with extensive examples that highlight the game development process and the main ideas evolving it. The user achieves knowledge in researching a concept, finds about the tools needed for sound, 2D and 3D art animation. It also provides a variety of game engines available in the industry presenting each one's advantages and disadvantages. It teaches how to program, script, the basics of artificial intelligence, user interface and testing. Documentation is also covered. Learning how to provide important documents, executive summary and design document. The examples are dissected and analyzed for way to get your game idea and vision documented and ready for development.

Andrew Rollings, David Morris, in [5] Game Design Foundations, 2nd Edition covers a revision of the classic first edition. The author emphasizes that the book is not a programming book, but a design book. The authors talk about architecture, and pick apart some top games with state diagrams and sketches of class hierarchies, but that sort of content is in the minority. "Mostly, the authors provide informed opinions about bigger engineering decisions, such as the question of whether to use Microsoft DirectX or OpenGL, or how to spread processor cycles across artificial intelligence and rendering operations. They make frequent reference to successful (and failed) games, explaining why each might have worked out as it did. - David Wall". The topics covered are writing good games and the entertainment software. The main goal is starting from an idea, develop it into a product, with emphasis on architectural decisions (game-play and visual effects), implementation choices (languages, libraries, and algorithms), and team management.

Unity 3D Game Development

Will Goldstone, 2009, in [6] Unity Game Development Essentials approaches the world of game development with the Unity game engine. With no prior knowledge of game development or 3D required, this book teaches the reader from scratch, taking each concept at a time working up to a full 3d mini-game. Scripting is done using JavaScript and as the reader masters the unity development environment with easy to follow step-wise tasks. The book is also aimed towards designers and animator, this book represents a valid starting point since no prior knowledge of game production is required. The book does a terrific job introducing the engine's main concepts and workflow. Being constructed as a tutorial, the reader progresses, through the chapters of the book, each one of them presenting a set of features that make a game. Importing assets, generating particles, managing collisions, designing GUI's, sculpting terrains, foliage generation, sound placement, etc. It is a very good approach since the author assumes you do not know anything about the engine.

At the end of the book, the reader has a pretty good idea on how to improve the game. He learns the notions on how to implement game mechanics and new ideas on his own. It grasps the key concepts of unity like scripting, particles, object instantiation and physics. Throughout the entire book, the author gives you a great exercise filled with everything Unity-based. From the beginning up until the end the reader us working in the 3D world of unity. The essentials, as the author calls them are: learning how to use the Unity 3D interface, building a 3D island with terrains, volcanoes, importing 3D models from other programs, add interaction between the player and 3D objects having multiple mini-games, creating cameras to move a character around your 3D world, creating title screens to navigate your games, turning static 3D objects into interactive objects using animation and scripting and dynamic effects light sound, shadows and of course, lighting. In the beginning, it teaches the creation of terrains, in this kind, and island with different kinds of textures (sandy, rocky, grassy), different elevations (the volcano) and of course interactive games. From particle systems to ray casting, all of the main vocabulary you should know is discussed. Engines like Unity, take much stress off the programmer and it helps you focus on creating the tools for the game itself. Another focus is on how to use the IDE, which in this case is Unity's own dedicated IDE. The game engine has a lot of build in settings. As stated above, the book is meant for a wide audience, including designers, and the code is not hard to understand. The main benefit of the book is, however, that it gives the reader an introduction to the facets of Unity. The book starts by explaining what Unity offers in terms of built-in terrain tools along with explanations on how to use them. Then it creates a small mini-game based on the island, created entirely in Unity. Other features like sky-boxes and sound are approached. Then, the concept of player is introduced by giving an explanation of what players are and how they work. Parent-child relationships are explained and also the basics of scripting Unity projects in JavaScript and how to add in movement. Other aspects are introduced, like Ray Casting and Collision Detection, which are vital to any games. The HUD(Head-up Display), essential aspect of the game is shown along with how to use colliders like Triggers. Interactivity, in-game

hints and helps are discussed. Then , object instantiation in the 3D world and rigid body physics. All these concepts are then out into practice by creating various mini-games which allow readers to know how to link them together in a complete game. Particle systems are key to all special effects and are one of the more complicated aspects of game development and this is what is explained next. They are explained in general then how to add them to your projects by creating a fire with wood, rocks and matches. Texture are added and interactivity aspects to the menu. Animations and menu GUI are detailed.

Ryan Henson Creighton, 2010, takes a clear, step-by-step approach to building small, simple game projects in [7] Unity 3D Game Development By Example. It is focused on short and attainable goals so that the reader can take all the steps needed in obtaining a finished solution. This book is aimed towards the passionate and hungry for knowledge. It represents the fastest path from zero to a finished game using Unity 3D engine. Exactly as the title says, the development is done by example. It starts with a general overview of the controls in Unity that the user will be working with. After that it describes the 3D world and emphasizes on things like meshes, lighting and physics. It is not a version for the advanced, but rather an introduction to the capabilities, functionalities, interface and assets the Unity has to offer. It teaches how to handle different resource, import them and also manipulate them. It involves detailed description of the physics, particle, sound and rendering engine. As a scripting language, the author has decide to use JavaScript for the unity Scripts. Some advanced areas like quaternions, mip-mapping and prefabs are approached. Animation also holds an important place in this book. A big plus of the book is that it explains what does the code actually do and why it is written in that manner. The code is fully commented and explained why that line of code or piece of code is there at that point in time. It is a very useful book because it directs the reader to places where he can find help if needed. Throughout the book, the author recommended for the readers to explore around and tweak settings with referring to the Unity Script Reference website. The book grasps the concept of starting with small things and improving them as time progresses and the reader accumulates more skills and knowledge. The author goes into detail about his method to learn things and find new and relevant solutions. The research skills are one of the main goals of the book, since the author often emphasizes finding out and investigating on your own. The book also does a great job balancing theory and actual work, as in, implementation. It talks about concepts then applies them. Complex concepts like quaternion are not treated in detail, just introduced and briefly described on how will be used to create the game.

Sue Blackman, 2011, in [8] Beginning to 3D Game Development with Unity represents a perfect introduction to programming unity. It introduces key game production concepts in and artist-friendly way, and rapidly teaches the basic scripting skills needed with Unity. The book is aimed towards artists familiar with tools like 3ds Max or Maya who want to create games for mobile platforms, computers or consoles, but with little or no experience in scripting and game logic behind the actual development of games. It relates to adventure games in the style of Telltale Tales of Monkey Island, while also giving foundation and basis in design and game logic. The book is divided into three parts: the

first part of the book explains the logic involved in game interaction. After this, it help you create game assets through basic examples that help you expand gradually and build upon. The second part, the basics of an adventure FPS game are built, including reusable state management scripts, load/save functionality, robust inventory system, dynamically configured maze and mini-map. The third part, using 2D and 3D content, the user learns how to deal with challenges as the project progresses, acquire problem-solving skills and interactive design. By the end of the game the reader and user will be able to use Unity 3D game engine, already having learned the necessary workflows to utilize his own assets and resources. The factor of reusable assets and scripts and tools is a key foundation element for building future games. In conclusion, the reader learns how to handle characters, 3D object visibility, effects and other special cases, what is inventory logic and how to successfully manage it, how to create a test environment and gain control over action objects, functionality, state management, message texts, cursor control, object metadata and many more. It teaches you how to build interactive games that are available on a variety of platforms, how to handle variety of menus levels in your games development and it introduces the user to user interface fundamentals, scripting and more.

Another great source for familiarizing with Unity and creating your own game is by following online tutorials. David Lancaster, in [19] HowToMakeAGameInUnity3D.pdf offers a tutorial whose final goal is the making of a 2D top down survival alien shooter, along with Daniel Wilkinson. The game is free. The tutorial introduces the reader into the world of Unity and C# programming language, level design in unity 3D and how to create textures and art assets in the Gimp. The reader is required, though to have a basic understanding of the Unity 3D interface, which is very intuitive and easy. It is a very short but well made tutorial and it is based on game development only. Because of this, it is divided into 2 parts: the programming part and the level design part. The programming part contains the following features: character movement, where it defines the mechanics for player movement and motion. Then if follows with the character animation where it defines the animation clips associated with player types of movement. The programming part continues with enemy artificial intelligence. The enemy AI uses the movement and animation framework previously defined. Then it adds projectiles to the player character. Once the particle part is created, particle effects and damage is attributed. The additional development section hints the reader what he can further add to the game system to improve the content. This is where the second part of the tutorial begins, the level design part. Level design starts first with the objects placement: creating the terrain, texturing it, making it 2D. It also defines some basic guidelines and features that make a design bad or good. Therefore things that can make a good game design: balanced challenge and reward system, balanced win-loss system, immersive world, lots of interaction. Bad design refers to: road blocks, information that the player is required to know to play and it isn't communicated to him, unbalanced challenge-reward system, a consequence of losing which is too much to risk and a reward for winning that is too rewarding.

3.2 Game Design and Modeling

Kelly L. Murdock, 2011, 2012, 2013, in [9], Autodesk 2011 3ds Max Bible, Autodesk 2012 3ds Max Bible, Autodesk 2013 3ds Max Bible offers the only comprehensible reference-tutorial on 3ds max making it a great support for all users, starting from beginners to professionals. Autodesk 3ds Max represents a top animation, modeling software used by developers. It deals with issues like manipulating objects, modeling 3D assets, applying materials and textures, working with cameras, lighting and rendering, animating objects and scenes, working with characters, sculpting and modeling using procedural substance textures, creating custom character rigs using CAT, learn techniques regarding rigging, kinematics, assigning skeletons, working with bones and articulation points and skinning.

Jeffrey Harper, 2012, in [10] offers a book that is extremely popular with video game designers as well as architects. 3ds Max offers integrated 3D modelling, animation, rendering, and compositing tools designed to streamline production. Coverage includes drawing shapes with splines and editing objects on multiple levels, understanding the interface of the software, shaping primitives into more complex forms using modifiers. It also provides an excellent source of showing how to organize the work using layers and assigning names to objects, creating and adjusting different light types to emphasize parts of your model. In terms of designing , editing and using materials to add color, texture, and realism. Also deals with importing carious types of 2D and 3D data for the creation of more interesting and eye-candy scenes and/or complex projects. Last but not least it makes use of mental ray for rendering photorealistic images and render passes. Applying techniques that are used in real world of 3D Animation and Visual Effects and optimization in efficient work with polygon modelling and texturing. This book is extremely highly regarded among literature regarding this area of expertise.

Paul Steed, 2005, in [11] Modeling a Character in 3DS Max focuses from the point of view of a graphic designer rather than a programmer or a software engineer. This book is full of illustrations showing how pass through the phases of designing your own character using 3ds Max 7. It focuses on the 3 main steps of : design, modelling and texturing the graphical components and models used in a video game. He provides tricks, tips and techniques from the perspective of one of the most recognized 3D artists. It provides to the readers a professional level skill set. It basically emphasizes on the creation of a single low-poly real-time character from the conceptual level to texture mapping. Model with primitives, use extrusions and Booleans, mirror and reuse models, optimize meshes and polygons, create and apply textures and develop new shapes through the addition of modifiers.

Jonathan Williamson in [12] Character Development in Blender 2.5, 2011,provides the readers and users a support for creating believable character using Blender. Blender represents a free, open-source 3d animation package. It covers technical , theoretical and artistic aspects of character development, being an in-depth look at Blender's modelling tools. The information that is present in the book includes references to sculpting, mod-

elling, materials, rendering, animation. A very good instructional book, it is written in a tutorial style, covering step-by step tips and instructions. The book is divided into 5 parts and approaches poly-by-poly modelling, sculpting, box modelling, mapping, etc. In the beginning it will help the reader understand how the interface works, how to manipulate 3d objects in 3d space, meshes with the various modelling tools, modifiers, and modelling functionality that Blender offers. The main focus of the book : modelling the character, handling the workflow and learning various modelling techniques. Other parts are providing introduction to rendering, lighting of the character and the preparation of the character for texturing using normal maps.

Paul Steed, 2005, in [11] Modeling a Character in 3DS Max focuses from the point of view of a graphic designer rather than a programmer or a software engineer. This book is full of illustrations showing how pass through the phases of designing your own character using 3ds Max 7. It focuses on the 3 main steps of : design, modelling and texturing the graphical components and models used in a video game. He provides tricks, tips and techniques from the perspective of one of the most recognized 3D artists. It provides to the readers a professional level skill set. It basically emphasizes on the creation of a single low-poly real-time character from the conceptual level to texture mapping. Model with primitives, use extrusions and Booleans, mirror and reuse models, optimize meshes and polygons, create and apply textures and develop new shapes through the addition of modifiers.

3.3 Game Enigne Architecture

Jason Gregory, in [13] covers the theory and practice of game engine software development. It covers a wide range of topics. The concepts and techniques are actual and are used in today's game studios. The examples offered are relevant, grounded on specific technologies. The main focus and emphasis is put on game engine technologies and architecture. It covers the main subsystems that compose a game engine with the algorithms and data structures, software interfaces and so on. The author created this book to be used in a course setting at college-level in the area of game programming and because of this it systematically introduces the notions. The tools are presented, the fundamentals of Software Engineering for games, mathematical concepts, low-level engine systems, graphics and motion and gameplay. The book brings together ideas, techniques and implementation through relevant examples that are not technology-centric. This book has been awarded in numerous academic settings and it represents one of the most important sources of documentation for this present project.

3.4 Game Mechanics

Tom Meigs, veteran game developer, 2003, in [14] offers a collection of tips and techniques from veteran game designers that work for important game development com-

panies like Sony, Blizzard, Disney, Activision. The book is aimed towards the planning, designing and creating game environments. He covers the foundation of game design including previsualization, level stubbing and layout, lighting, texturing, behavior scripting, using particles. He also explainds in detail each stage of game development and design. Hence, each chapter contains an interview with a game industry expert together with a case study from the author's own experinces at Sony PlayStation. Therefore, the following subjects are approached: devlopment of comprehensive previsualization process for a game project, creation of topographic maps, level planning and layout, building of game scenes using tools such as lighting, texturing, particles, effects and audio, tuning of specific camera details. It also offers valuable information about how to game design by game genre: sports games, fighting games, RTS (real-time-strategy), RPG(role-playing-games), FPS(First-Person-Shooter) and many others.he also emphasize on action events scripting using JavaScript, Python, Pearl C Sharp. It also provides examples on how to develop and design games for wireless device like mobile phones and PDAs, implements testing loops. He also provides examples on what mistakes should be avoided for efficient design.

Alex J. Champanard, 2003, in [15] describes some basic AI concepts that would be useful in developing games. He refers to neural networks, decision trees and genetic classifiers. It basically focuses on autonomous synthetic creatures together with the process of exploring techniques and theories that are centered to AI game development. Focus is on designing individual creatures, each one with unique set of abilities and skills. The chapters use demos and examples to make for a better understanding. It also offers examples from the very well known game F.E.A.R. and it considers it to be a "platform for experimentation". Movement is a topic that is approached in high detail. He explores the idea of navigation, game bots and movement, motion criteria, formalization of motion. It also includes such topics as knowledge representation, its formalism, the procedure for specification. Other subjects include steering behaviours and obstacle avoidance, rule-based systems. He synthesizes rule based systems with movement techniques offers relevant design and implementation modules. Other chapters include combat settings, player shooting skills, physics for prediction, perceptrons, aiming, target selection, fighting conditions, weapon selection and weapon choice, tactical decisions, scripting, classification and regression trees as a representation of decision trees, tree induction. Fuzzy Logic is dissected providing fuzzy logic conversions, set of logic, synthesize fuzzy logic and decision making, fuzzy rules. Other chapters contain genetic algorithms, biological evolution, classifier systems, adaptive defensive strategies with genetic algorithms. Learning Ai and emotive creatures, human/machine interaction, sensations, emotions and feelings. The most important topics are by far the FSMs (Finite State Machines) providing bases for most video games. Offers control logic, formal definitions, representation and simulation, non-deterministic state machines, combining state machines with fuzzy logic resulting in fuzzy state machines and probabilistic and hierarchic state machines, strategic decision making tactical intelligence and reactive learning strategies.

Matt Buckland, 2004, in [16] provides a comprehensive and practical introduction to the essence to Artificial Intelligence algorithms, techniques, solutions and strategies

that are employed by the game development industry. It leads the reader through the process of designing, programming and implementing intelligent agents for games. It makes use of the C++ language which is the most commonly used tool by artificial intelligence world when producing video games AI. The techniques and algorithms are graph theory, agent communication, individual and group steering behaviours, team AI, triggers, scripting FSMs(Finite State Machines), goal evaluation and arbitration, fuzzy logic, perpetual modelling and searching. It provides industrial strength solutions to difficult problems. The author guides the reader towards a robust foundation that is essential for real games. Each chapter is treated as a new fundamental game AI technology and expanding this idea into a fully formed solution with extensive code and clear examples. The book covers all important areas, including basic mathematics and physics through graph theory and scripting with Lua. It also intersects software engineering by making user of familiar and well-known design patterns. Although the book is not recent the examples provided are modern with efficient coding and highlights the techniques used in today's industry. The detailed examples can easily be incorporated in your own game. This book is a reference for Artificial Intelligence in the Game design and development process.

Ian Millington, 2009, in [17], Artificial Intelligence for Games, approaches the problem of improving the quality of AI in games. Because he brings his extensive professional experience, the numerous examples described by him from real games, he explores the underlying ideas through detailed case studies. In addition to this, he adds many techniques that have been lost or the industry doesn't use them. It also contains C++ source code and demonstration programs, and a complete commercial source code libraries of Ai algorithms and techniques. It provides exercises with each chapter, hence, it has a great value from an academic standpoint. It covers such problems like: AI-oriented game-play, casual games and behaviour driven AI. It is a comprehensive guide to artificial intelligence as it relates to games, it covers everything from simple character movement techniques, the mini-max search tree for board games, to complex problems in games like goal-oriented behaviour and path-finding. Besides the fact that the techniques are systematically presented, it also offers tricks for improving performance significantly, ex. Alpha-beta pruning. In the actual book the examples are written in a predefined pseudo-code which was standard throughout the whole book. Topics such as coordinated movement are tackled and expanded, path-finding, decision making(Decision Trees, Finite State Machines, Fuzzy Logic, Goal oriented behaviour and rule-based systems), tactical and strategic AI, Learning, board games etc. The author also provides examples of supporting technologies for implementing these techniques and algorithms: execution management(scheduling and level of detail), world interfacing(communication and event managers), tools and content creation(knowledge for path-finding and waypoint tactics, decision making and movement). It also provides solutions to how to design game AI based on game genre.

Mike McShaffry and David Graham, through [18] Game Coding Complete, 2nd edition 2003 and the later 4th edition 2012, offers a guide to developing commercial-quality games. The two veteran programmers examine the entire game development process and tackle all the unique challenges associated with creating a game. In this excellent intro-

duction to game architecture, you'll explore all the major subsystems of modern game engines and learn professional techniques used in actual games, as well as Teapot Wars, a game created specifically for this book. The updated version uses the latest versions of DirectX and Visual Studio, and it includes expanded chapter coverage of game actors, AI, C# editing, shader programming other important updates to every chapter. All the code provided us functional and tested. The book is full of best practices used in commercial games along with tips, tricks and advice. It approaches topics like game initialization and shutdown, game actor, controlling the main loop, loading and caching game data, programming input devices, user interface programming, game event management, scripting, game audio, 3D graphics basics, 3D vertex and pixel shaders, 3D scenes, collision and simple physics, network programming, Game AI introduction, multiprogramming, game editors in C#, debugging, and finishing.

Chapter 4

Analysis and Theoretical Foundation

4.1 Foundation and Fundamentals of Game Architecture

4.1.1 *Game* Definition

The general term "game" encompasses board games like chess and monopoly and card games. In academic circles, it refers to the "game theory". In [22], the author states that when used through the context of computer and console-based entertainment, the word "game" usually conjures images of a 3D virtual world featuring a humanoid, anima or vehicle as the main character under player control. The same book highlights that in the book "A Theory of Fun for Game Design", Raph Koster defines a game as "an interactive experience that provides the player with an increasingly challenging sequence of patterns which he or she learns and eventually masters."

4.1.2 Structure of a Game Engine

The line that separates a game engine from an actual game is very thin in most cases. Some engines make a distinction while in others it cannot be possible to separate them. The argument that differentiates a game from its engine is the engine's data driven architecture. The world of technology is ever-growing and faster computer hardware and specialized graphics cards, with more efficient rendering algorithms and data structures is beginning to soften the difference between game engines and graphics engines of different genres. Nowadays there are a lot of game engines that support game implementations for all game genres. Open source 3D game engines are provided for free.

A game engine usually has a tool suite and a runtime environment. Figure 4.1 illustrates the major components that construct a typical 3D game engine. Game engines are software systems. Therefore, they are built in layers. Upper layers depend on lower layers. Diagramn 4.1 highlights the architectural design and layout of a game engine.

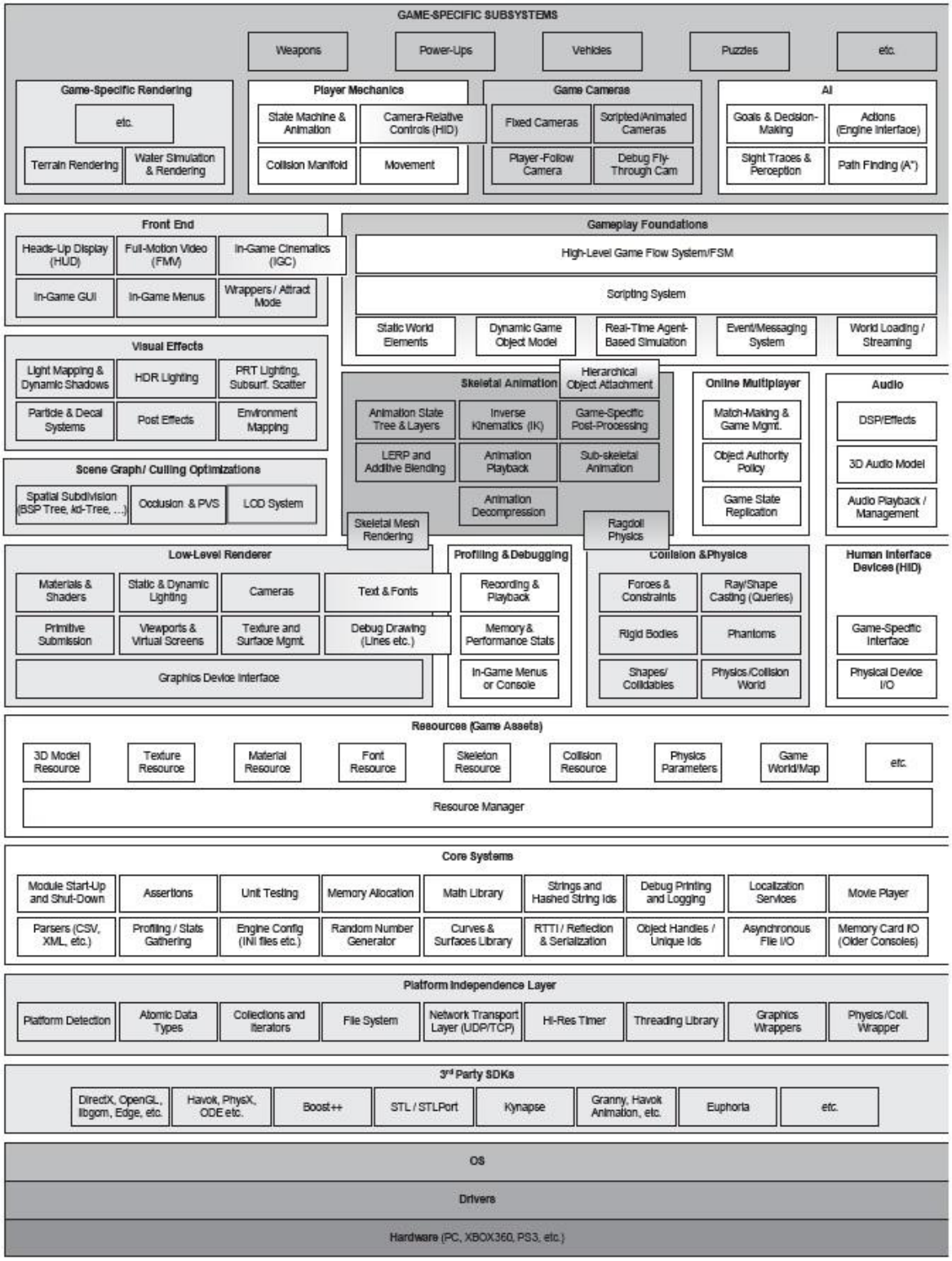

Figure 4.1: Run-time Game Architecture

Target Hardware

This layer represents the computer system or console on which the game will run. Platforms include Microsoft Windows, Linux-based PC, Apple iPhone and Macintosh (these are PCs) and Microsoft Xbox and Xbox360, PlayStation 2,3, PSP. In Figure 4.1 the Hardware Layer is highlight.

Figure 4.2: Hardware layer

Device Drivers

The next layer is the Drivers. Drivers are low-level software components that are provided by the OS or by the hardware producer and vendor. Their main function is to manage hardware resources and isolate the operating system and the upper engine layers from the communication details. The layers is depicted in Figure 4.2.

Figure 4.3: Drivers layer

Device Drivers

On a PC, the OS is running all the time. It basically orchestrates and coordinates the execution of multiple programs on a single computer. On a console, the OS is just a thin library layer that is compiled directly in the game executable. The OS layer is shown in Figure 4.4.

Figure 4.4: Operating Systems layer

SDKs and Middleware

Large majority of engines contain a number of third party software development kits and middleware. This is shown in Figure 4.5 The functional or more specific, the class based interface provided by and SDK is most of the times called an API(application programming interface). SDKs contain libraries that deal with: data structures and algorithms(ex. STL, Loki), graphics (ex. DirectX, OpenGL), Collision and Physics (Havok and NVidia's PhysX), Character Animation (Havok Animation, Edge), Artificial Intelligence (Kynapse).

Figure 4.5: SDK layer

Platform Independence Layer

Engines are usually designed to be capable of running on more than one hardware platform. This results in a Platform Independence Layer, like the one shown in Figure 4.6 . This layer is situated on top of the hardware, drivers, OS and SDKs and it isolates the rest of the engine from the majority of knowledge of the platform. In [22] it is stated that âĂIJby wrapping or replacing the most commonly used standard C library functions, operating system calls, and other foundational application programming interfaces (APIs), the platform independence layer ensures consistent behaviour across all hardware platforms. This is necessary because there is a good deal of variation across platforms, even among âĂIJstandardizedâĂİ libraries like the standard C library.âĂİ

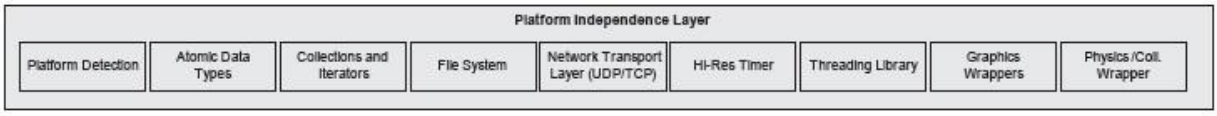

Figure 4.6: Platform layer

Core Systems

Every game engine contains a typical core systems. A typical core system layer is displayed in Figure 4.7 . Some very important core systems are the Assertions, Memory management unit, Math library, Custom data structures and algorithms.

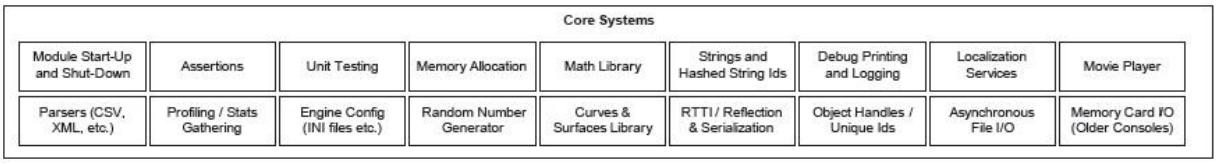

Figure 4.7: Core Systems

Resource Manager

This layer is present in every game engine and it is one of the most essential parts of it. It provides an interface for accessing any and all types of game assets and other engine input data. The Layer is presented in Figure 4.8.

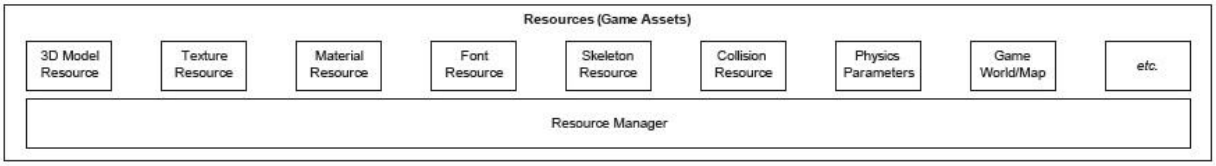

Figure 4.8: Resource Manager Layer

Rendering Engine

The rendering engine is one of the most important and largest and most complex components of any game engine. The fundamental designs are driven by the design of the 3D graphics hardware upon which they depend.

The components of the renderer are explained below and Figure 4.9 illustrates their structure . The low âĂŞ level renderer contains all the raw facilities of the engine (focuses on geometric primitives). The graphics device interface initializes SDKs such as DirectX or OpenGL. The other components cooperate to collect submissions of geometric primitives (meshes, lines, line lists, point lists, particles, terrain patches, text strings). The low-lever renderer manages the state of the graphics hardware and shaders (using material system and the dynamic lighting system). It also provides the abstraction with an associated camera-to-world matrix and 3D projection parameters.

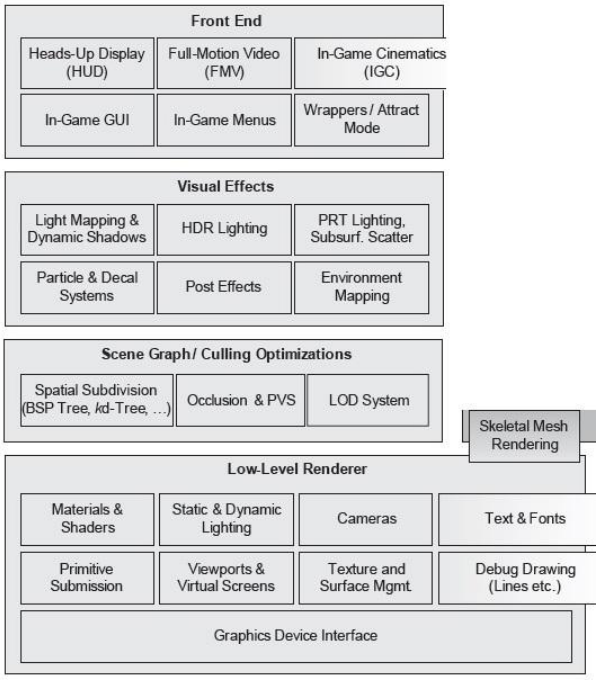

Figure 4.9: Rendering Engine Subsystem

A higher level component is needed to provide optimizations required to limit the numbers of primitives submitted for rendering, based on a form of visibility determination (frustum cull and special subdivision) . Visual effects contain particle systems, decal systems, light mapping and environment mapping, dynamic shadows, full screen post effects, etc. The Front End contain the game's HUD (heads-up display), in-game menus, consoles, development tools, a GUI (graphical user interface), FMV (full-motion video) and In-Game - Cinematics (ICG).

Profiling and Debugging Tools

Profiling the performance of a game is essential for developing a game. The layer, shown in Figure 4.10, provides these tools (debug drawing, in-game menu system, console, record and playback gameplay for testing and debugging purposes).

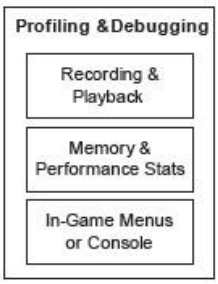

Figure 4.10: Profiling and Debugging Tools

Collision and Physics

Collision detection and physics are coupled together. Collision resolves the interpenetration of objects. Physics represents a realistic or semi-realistic dynamics simulation. They can be seen in Figure 4.11.

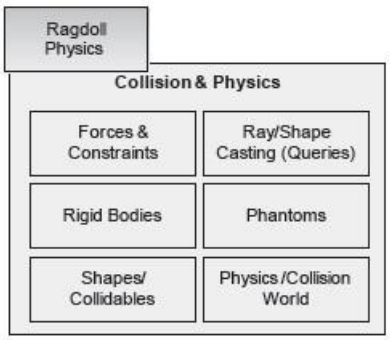

Figure 4.11: Collision and Physics

Animation

The basic five animation types used in animation systems are: sprite, texture animation, rigid body hierarchy animation, skeletal animation, vertex animation and morph targets.

Skeletal animation is the most widely used type. It permits a 3D character mesh to be posed by an animator using a simple skeleton.

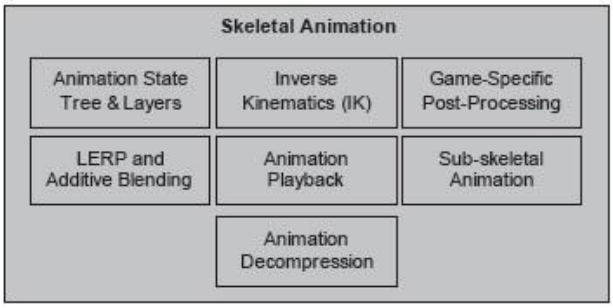

Figure 4.12: Animation component

Human-Interface Devices - HID

This Layer represents the interface through which the game processes the player inputs. It includes the keyboard and mouse, a joy-pad or other specialized controllers. This component is also called I/O component.

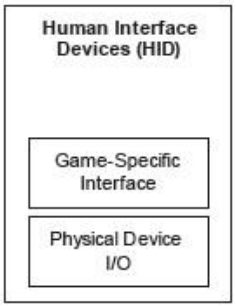

Figure 4.13: Human Interface Devices

Audio Component

Audio is just as important as graphics in the game engine. The layer is represented in Figure 4.14 from [22].

Gameplay Foundation Systems

Game-play is related to the action taking place in the game, the set of rules that govern the virtual world, abilities of the player characters - player mechanics and of other characters, goals and objectives. Gameplay is usually implemented in a native language of the engine or in a high-level scripting language or sometimes both. For this game there is the present layer. It introduces the notion of game world. This world contains both static and dynamic elements and are usually modeled in an object oriented manner. Typical types of game objects are: static background geometry (buildings, roads, terrain), dynamic rigid

bodies (rocks, chairs, etc), player character (PC), non-player-character (NPC), weapons, projectiles, vehicles, lights, cameras, etc. this game world model is linked to a software object model that contains the set of language, policies, features and conventions used to implement a piece of OOP software. The Game-play Foundation Systems are: Event System, Scripting System, Artificial Intelligence Foundations, etc. These components can be seen in Figure 4.15.

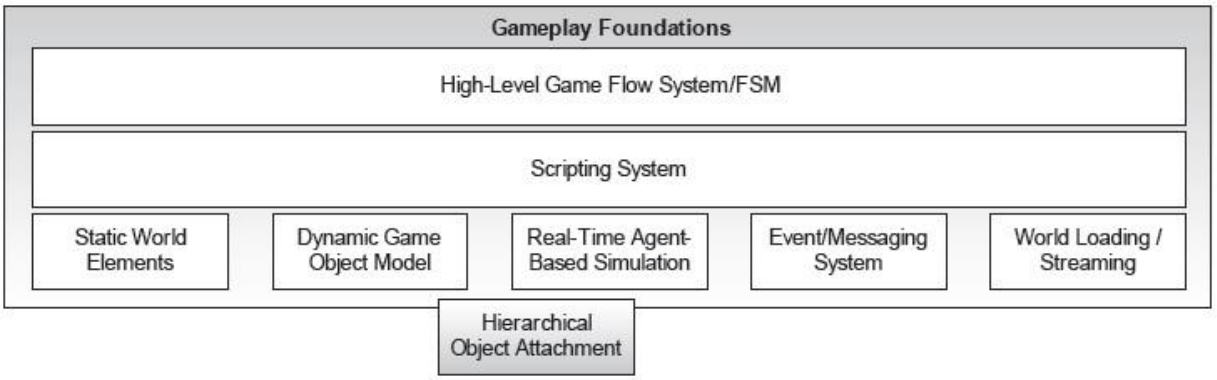

Figure 4.14: Gameplay Foundations

Game-Specific Subsystems

This represents the highest layer, being on top of the foundation layer and other low-level components. The game-play systems vary from programmer to programmer and they are game-specific. Figure 4.16. illustrates how they include mechanics for the player character, in-game camera systems, artificial intelligence for NPCs (Non-Playable Characters), weapon systems vehicles etc.

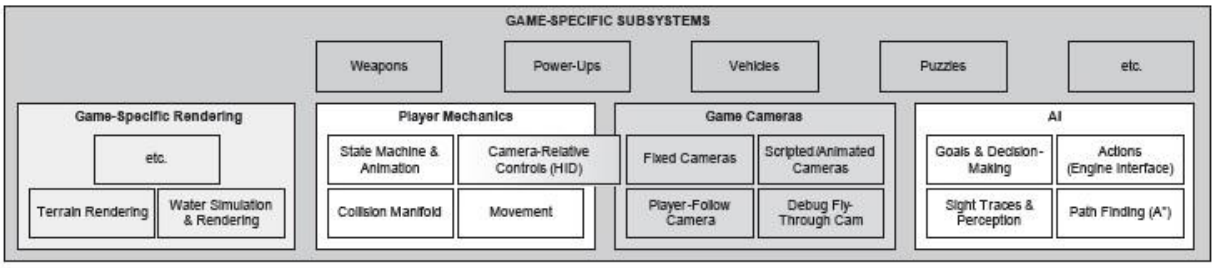

Figure 4.15: Game subsystems

4.1.3 Tools and the Asset Pipeline

Game Engines require a great amount of data. This data comes in various types: configuration files, game assets, scripts, etc. Because games are multimedia applications,

a game engine's input comes in a wide variety of forms. Some of them are: 3D meshes, texture bitmaps, animation data and clips, audio files. These are created by artists, designers and engineers and the tools that they use is called digital content creation (DCC) applications

The data formats used by the digital content creation (DCC) applications are almost never ready for direct use in-game. The data produced by the DCC applications is exported to a more accessible and standardized forma. After export, it needs to be processed before arriving at the game engine. The pipeline from DCC application to game engine is, in some cases, called asset conditioning pipeline and every engine has one.

The asset pipeline is presented below. In the present example the DCC tools are: 3ds Max and Maya for mesh creation, skeleton attachment, animation design and material modeling, Photoshop for textures creation, Houdini for particle effects and Sound Forge as an audio tool for sounds.

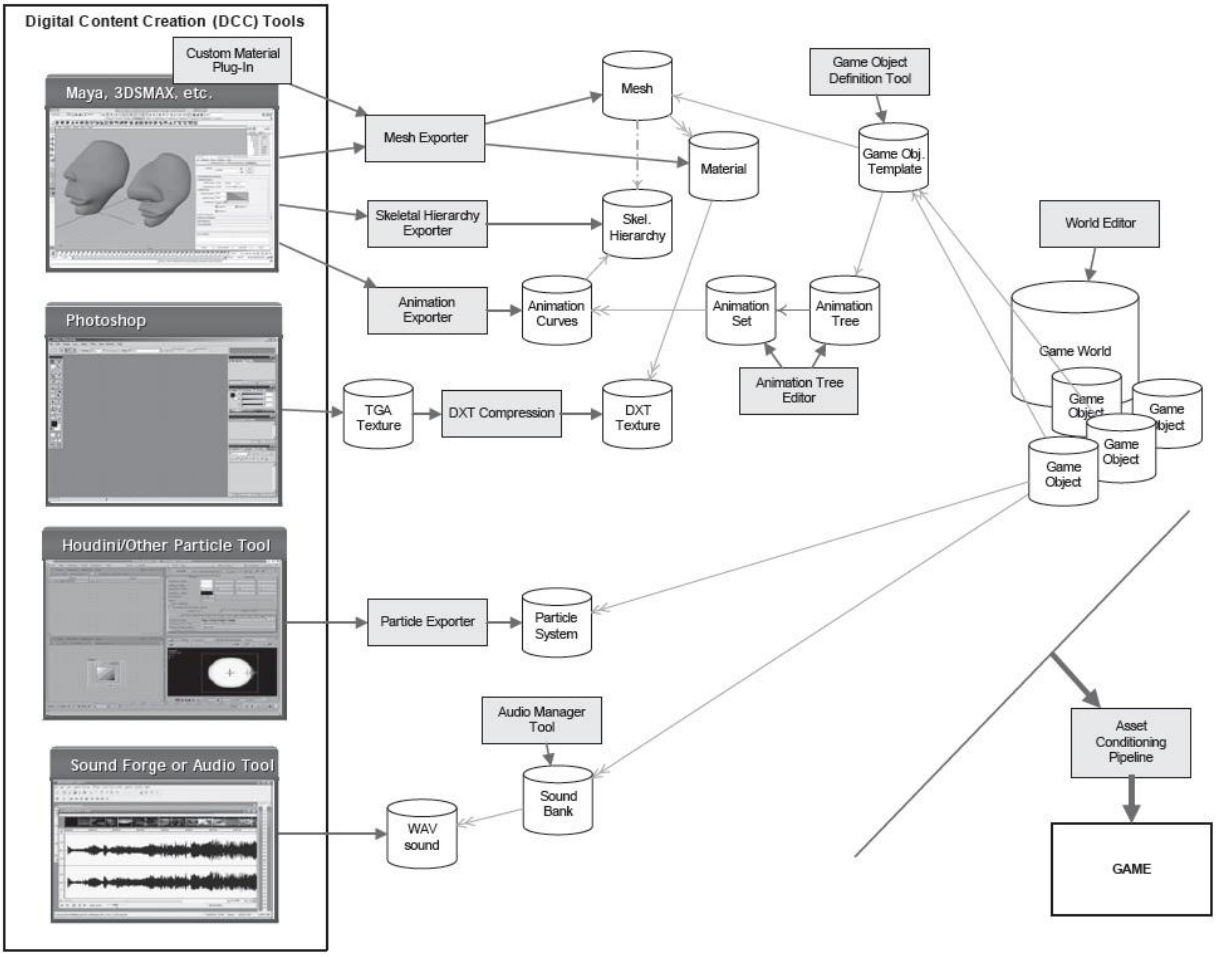

Figure 4.16: The Asset and Tools pipeline

4.1.4 Mathematics in a Game Engine

From another perspective, a game is a mathematical model of a virtual world simulated real-time on a computer. In consequence, mathematics is at the foundation of everything related to the game industry. All branches of mathematics are utilized to produce games, from algebra to trigonometry and calculus. However, the most kind of maths done by a game programmer is linear algebra (3D vectors and matrix math).

This branch of mathematics is very broad and deep. An overview of the mathematical tools needed in a typical game setting is presented in the following pages.

A lot of mathematical operations work equally well in 2D and 3D. this equivalence, however does not hold all the time. Some operations are only defined in 3D (ex. cross product).

The large majority of 3D games are constructed from 3D objects in a virtual world. A game is required to keep track of all the positions, orientations and scales of these objects, animate them in game world, and transform them into screens space so they can be rendered on screen. In games, almost always 3D objects are made up from triangles and their vertices are represented by points.

Points and Cartesian Coordinates

A point is a location in n-dimensional space (n=2 or 3 in games). The Cartesian coordinate system is by far the most common coordinate system employed in game creation. It uses 2 or 3 mutually perpendicular axes to specify a position in 2D or 3D space. Hence a point P is represented by a pair or a triple of real numbers, (P_x, P_y) or (P_x, P_y, P_z). Figure 4.18 illustrates a point representation in Cartesian coordinates.

In 3D Cartesian coordinates, there are 2 choices when arranging the perpendicular axes: RH, or right-handed and LH, or left-handed. The difference between them is the direction in which one of the 3 axes, X, Y or Z. Figure 4.19 depicts LH and RH. It is easy to convert from a LH to a RH and vice-versa. Simply flip the direction of any one axis, leaving the other axis alone. LH and RH apply to visualization only.

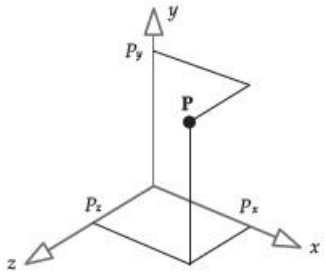

Figure 4.17: Point representation in cartesian coordinates

Vectors

A vector is a quantity that has both a magnitude and a direction in *n-dimensional space*. It can be visualized as a directed line segment that extends form one point, called tail, to another called head. A 3D vector can be represented by a triple of scalars (x, y, z) just like a point can be. A vector is just and offset relative to some known point and it can be moved anywhere in 3D space - as long as its magnitude and direction do not change. In the industry, the term "vector" is used to refer to both points (point vectors and) and to vectors in linear algebra sense (directional vectors). A vector can be represented as a sum of scalars multiplied by i, j, and k:

$$(5, 3, -2) = 5i + 3j - 2k \tag{4.1}$$

where i, j, and k are Cartesian basis vectors.

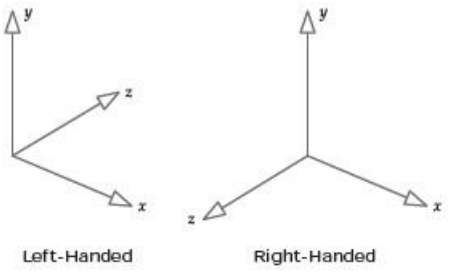

Figure 4.18: LH and RH

Vector operations:

- *Multiplication by a scalar* -

$$\mathbf{sa} = (\mathbf{s}ax, \mathbf{s}ay, \mathbf{s}ay)$$

- scales the magnitude or

$$\mathbf{aS} = \begin{bmatrix} \mathbf{a_x} & \mathbf{a_y} & \mathbf{a_z} & \mathbf{a_x} & \mathbf{a_y} & \mathbf{a_z} \end{bmatrix} \begin{bmatrix} \mathbf{s_x} & 0 & 0 \\ 0 & \mathbf{s_y} & 0 \\ 0 & 0 & \mathbf{s_z} \end{bmatrix} \tag{4.2}$$

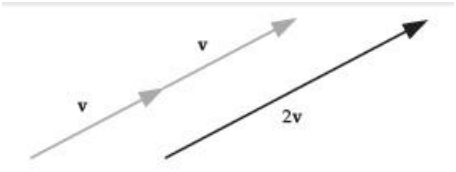

Figure 4.19: Multiplication

- *Addition and subtraction*

$$\mathbf{a} + \mathbf{b} = [(a_x + b_x), (a_y + b_y), (a_z + b_z)] \tag{4.3}$$

$$\mathbf{a} - \mathbf{b} = [(a_x - b_x), (a_y - b_y), (a_z - b_z)] \tag{4.4}$$

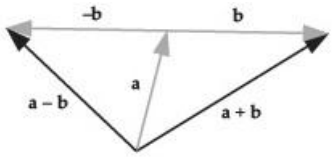

Figure 4.20: Addition and Subtraction

- *Magnitude*

$$|\mathbf{a}| = \sqrt{a_x^2 + a_y^2 + a_z^2}\sqrt{a_x^2 + a_y^2 + a_z^2} \tag{4.5}$$

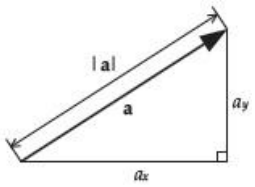

Figure 4.21: Magnitude

A unit vector is a vector with a magnitude of one. They are very useful in 3D Mathematics and game programming.

$$\mathbf{u} = \frac{v}{|v|} = \frac{1}{v}\mathbf{v} \tag{4.6}$$

A vector is said to be normal to a surface if it is perpendicular to that surface. They are very useful, in games, especially in computer graphics. A plane can be defined by a point and a normal vector. In 3D graphics, lighting, calculations make large use of normal vectors to define the direction of surfaces relative to the direction of the light rays.

Vectors can be multiplied and there are two kinds of multiplications.

1. The Dot Product:

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z = d \tag{4.7}$$

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{ab}\cos(\theta) \tag{4.8}$$

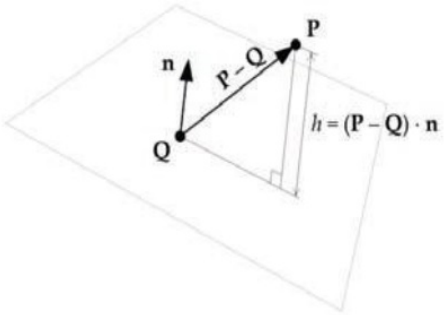

Figure 4.22: Enemy Position determination

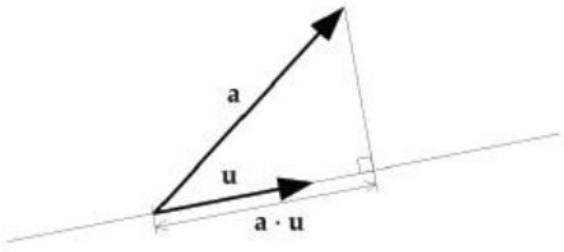

Figure 4.23: Dot product

The vector projection is illustrated in figure 4.23

It is very useful in game programming, for example, to find out if an enemy is in front of the player or behind him. A vector can be found from the player's position to the enemy position, by subtraction. Assume we have a vector pointing in the direction that the player is facing. This vector can be directly extracted from the player's model-to world matrix.

This valuable game application is depicted in Figure 4.22 . The dot product can be used to find the height of a point above or below a plane.

2. The Cross-Product or Vector-Product:

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)] \\ &= (a_y b_z - a_z b_y)\mathbf{i} + (a_z b_x - a_x b_z)\mathbf{i} + (a_x b_y - a_y b_x)\mathbf{k} \end{aligned} \tag{4.9}$$

Magnitude of a cross-product:

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\theta \tag{4.10}$$

Figure 4.24 depicts the cross product of vectors **a** and **b**.

The cross product can be applied in finding a vector that is perpendicular to 2 other vectors. They are also used in physics simulations. When a force is applied to an object it gives him a rotational motion i.f.f. is applied off-center. This rotational force is known as torque

$$N = r \times F, \tag{4.11}$$

where F is a given force, r is a vector r from the center of mass to the point at which the force is applied.

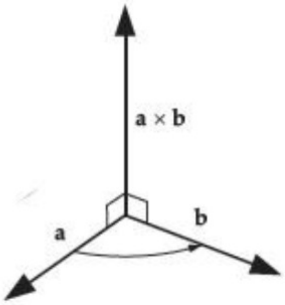

Figure 4.24: Cross product

Linear Interpolation of Points and Vectors

In games, we need to find a vector that is midway between 2 known vectors. A relevant example is a smooth animation of an object from point **A** to point **B** over the course of 2 seconds at 30 frames per second.

This implies that we need to find 60 intermediate positions between **A** and **B**. A linear interpolation is a simple mathematical operation that finds an intermediate point between 2 known points. It is also known as LERP. The operation for this specific example is defined below, where ranges from 0 to 1 closed interval:

$$\begin{aligned} \mathbf{L} &= \mathbf{LERP}(\mathbf{A}, \mathbf{B}, \beta)(1-\beta) + \beta\mathbf{B} \\ &= [(1-\beta)A_x + \beta B_x][(1-\beta)A_y + \beta B_y][(1-\beta)A_z + \beta B_z] \end{aligned} \tag{4.12}$$

From a geometrical perspective L is the position vector of a point that lies percent of the way along the line segment from point **A** to point **B**.

Matrices

A matrix is a rectangular array of m x n scalars. They are a convenient way of representing linear transformations such as translations, rotations and scale.

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix} \tag{4.13}$$

3 x 3 matrices can be viewed as 3D vectors. When all rows and columns are of unit magnitude, the matrix is called special orthogonal matrix or isotropic or orthogonal matrix. They represent pure rotations.

A 4 x 4 matrix, under certain constraints can represent arbitrary 3D transformations including translations, rotations and changes of scale. These types of matrices fall under the category of transformations matrices, and they are the most useful for game engineers. The transformations represented by a matrix are applied to a point via matrix multiplication.

Matrix multiplication represents the product of two matrices A and B: P=AB. If A and B are transformation matrices, then the product P is inevitably a transformation matrix that performs both of the original operations. For example if X is a scale matrix and Y is a rotation, then P = XY would scale and rotate the points or vectors to which it is applied. This is extremely useful in game programming for pre-calculation of a single matrix that performs sequences of transformations to a large number of vectors in an efficient manner.

Points and vectors can be represented as row matrices (1 x n) or column matrices (n x 1), n is the dimension of the space we are working with (most of cases 2 or 3). The choice is arbitrary but affects the order in which the multiplication is written.

The *Identity Matrix* is:

$$\mathbf{I_{3\times 3}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \mathbf{A}\mathcal{I} = \mathcal{I}\mathbf{A} = \mathbf{A} \tag{4.14}$$

The *Inverse Of a Matrix* X is another matrix X^{-1} that undoes the effects of a matrix X. If X scales objects to twice of their original size then X^{-1} scales objects to be half-sized. In the case of rotation, when we rotate X with 35 degrees then X^{-1} rotates with -35 degrees.

$$X(\mathbf{X^{-1}}) = (\mathbf{X^{-1}})X = \mathcal{I} \tag{4.15}$$

The *Transpose Of a Matrix* A is denoted A^T and it is obtained by reflecting the elements of the original matrix across the diagonal âĂŞ rows become columns and columns become rows.

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \tag{4.16}$$

$$\mathbf{A^T} = \begin{bmatrix} A_{11} & M_{21} & M_{13} \\ A_{12} & M_{22} & M_{32} \\ A_{13} & M_{23} & M_{33} \end{bmatrix} \tag{4.17}$$

A matrix can represent rotation in 2 dimensions:

$$[\mathbf{r'_x}\mathbf{r'_y}] = [r_x r_y] \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \tag{4.18}$$

This leads to a representation using a 3 x 3 matrix. Below is a rotation about the

z axis:

$$[\mathbf{r'_x r'_y r'_z}] = [r_x r_y r_z] \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.19}$$

To represent a translation we need a 4 x 4 matrix. The Translation matrix is:

$$\mathbf{r} + \mathbf{t} = [\mathbf{r'_x r'_y r'_z}] = [r_x r_y r_z] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = [(\mathbf{r_x} + \mathbf{t_x})(\mathbf{r_y} + \mathbf{t_y})(\mathbf{r_z} + \mathbf{t_z})\mathbf{1}] \tag{4.20}$$

This are the homogeneous coordinates (with w=1 always). The vast majority of 3D matrix math in game engines is done using 4 x 4 with four elements in homogeneous coordinates.

Any affine transformation matrix can be created a concatenation of a sequence or sequences of 4x4 matrices representing pure translations, rotations and scale operations, or pure shears. These are called atomic transformations.

Translation matrix translates a point by the vector t:

$$\mathbf{r} + \mathbf{t} = [\mathbf{r'_x r'_y r'_z 1}] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = [(\mathbf{r_x} + \mathbf{t_x})(\mathbf{r_y} + \mathbf{t_y})(\mathbf{r_z} + \mathbf{t_z})\mathbf{1}] \tag{4.21}$$

Rotation matrix, about the X-axis, Y-axis and Z-axis by the angle φ, θ, γ:

$$\mathbf{rotate_x}(\mathbf{r}, \varphi) = [\mathbf{r_x r_y r_z 1}] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - X - axis \tag{4.22}$$

$$\mathbf{rotate_x}(\mathbf{r}, \theta) = [\mathbf{r_x r_y r_z 1}] \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \cos\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - Y - axis \tag{4.23}$$

$$\mathbf{rotate_x}(\mathbf{r}, \gamma) = [\mathbf{r_x r_y r_z 1}] \begin{bmatrix} \cos\gamma & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} - Z - axis \tag{4.24}$$

The following matrix scales the point r by a factor of s_x, s_y and s_z:

$$\mathbf{r}S = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x r_x & s_y r_y & s_z r_z & 1 \end{bmatrix} \tag{4.25}$$

In computer graphics an object is usually represented by points, in which case, the object can be transformed by applying a transformation matrix to all of its vertices in turn.

A point is always expressed relative to a set of coordinate exes. The triplet of numbers that corresponds to a point changes numerically whenever we select a new set of coordinate axes.

Model Space

When an object is created using a modeling tool, the points positions of the triangles vertices that make the model are relative to a Cartesian coordinate system, called model space (object space or local space). This local object space is placed at a central location within the object, like the center of mass. The majority of game objects have an inherent directionality. The model space axes are usually defined with the object's natural directions on the model(Front, End, Left or Right).

World Space

World Space is a fixed coordinate space where the positions, orientations and scales of objects in the game world are expressed. This coordinate system connects all the individual objects together into a cohesive virtual world.

View Space

It is also known as camera space and it is a coordinate frame fixed to the camera. The view space origin is placed at the focal point of the camera. Any axis orientation scheme is possible, but a preferred version is the one with y-up with z increasing in the direction the camera is facing (LH).

Quaternions

Rotational representation that overcomes some of the problems of matrix rotation, the quaternion is a mathematical object and it looks like a four-dimensional vector, but it behaves differently. Quaternion $q = [qx, qy, qz, qw]$. The quaternion was developed initially as an extension to complex numbers. The unit-length quaternions, which are the quaternions that obey:

$$q_x^2 + q_y^2 + q_z^2 + q_2^2 = 1,$$

represent 3D rotations. A unit quaternion can be visualized as a 3D vector plus the

fourth scalar coordinate:

$$q = \begin{bmatrix} \mathbf{q}v & \mathbf{q}v \end{bmatrix} = \begin{bmatrix} \mathbf{a} \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \tag{4.26}$$

In [22] it is said that "A unit quaternion is very much like an axis+angle representation of a rotation(i.e., a four-element vector of the form $\begin{bmatrix} a & \theta \end{bmatrix}$). However, quaternions are more convenient mathematically than their axis+angle counterparts."

Quaternion operations

1. Multiplication

$$pq = \begin{bmatrix} (pSqV + qSpV + pV \times qV) & (pSqS - pV \cdot qV) \end{bmatrix} \tag{4.27}$$

2. Conjugation and inverse

   (a) Conjugate

$$q^* = \begin{bmatrix} -qVqS \end{bmatrix} \tag{4.28}$$

   (b) Inverse

$$q^{-1} = \frac{q^*}{|q|^2} \tag{4.29}$$

Because the quaternions the are sued in games are |q| = 1 always, the inverse and the conjugate are identical:

$$q^{-1} = q^* = \begin{bmatrix} -\mathbf{q}V & qS \end{bmatrix} for |q| = 1 \tag{4.30}$$

Rotational vectors with Quaternions is possible. Given a vector v the corresponding quaternion is:

$$v = \begin{bmatrix} v & 0 \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z & 0 \end{bmatrix} \tag{4.31}$$

To rotate a vector v by a quaternion q, first it has to be pre-multiplied by q and then post-multiplied by the inverse quaternion q-1 which is equivalent to a conjugate from previous relations.

$$v' = rotate(q, v) = qvq - 1 = qvq* \tag{4.32}$$

Quaternion Multiplication formula:

$$Fw = qFMq - 1q \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix} (q - 1) \tag{4.33}$$

Line, Rays and Line Segments

An infinite line can be represented by a point P0 plus a unit vector $\vec{u}$ in the direction of the line. A parametric equation of a line traces out every possible point P along the line

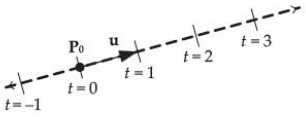

Figure 4.25: Line equation

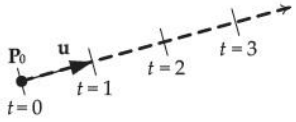

Figure 4.26: Ray equation

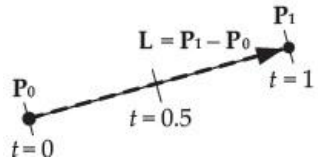

Figure 4.27: Segment equation

by starting at the initial point P0 and moving an arbitrary distance t along the direction of the unit vector $\vec{u}$. The infinitely large set of points P becomes a vector function of the scalar parameter t:

$$P(t) = P0 + tu, where - \infty < t < \infty \tag{4.34}$$

A ray is a line that extends to infinity in only one direction, P(t), t>=0. A line segment is bounded at both ends by P_0 and P_1.

Representations:

1. $\mathbf{P}(t) = \mathbf{P}_0 + t\mathbf{u}$, where $0 \leq$ t $\leq$ L, or
2. $\mathbf{P}(t) = \mathbf{P}_0 + t\mathbf{L}$, where $0 \leq$ t ≤ 1

Planes

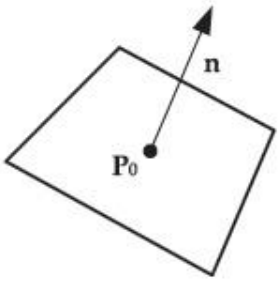

Figure 4.28: Point Normal Form

A plane is a 2D surface in 3D space, with the equation $Ax + By + Cz + D = 0$ and is satisfied only for the locus of points $P = [xyz]$.Planes can be represented by a point P0 and a unit vector n that is normal to the plane. This is sometimes called point-normal form, depicted in 4.28

Frusta

Illustrated in Figure 4.29, the frustum is a group of six planes that define a truncated pyramid shape. Frusta are commonplace in 3D rendering because they define the viewable

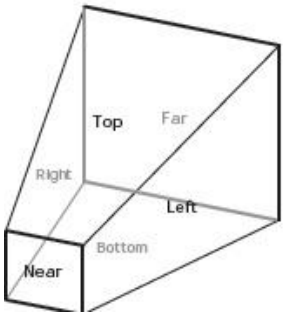

Figure 4.29: Frustum

region of the 3D world when it is rendered via a perspective projection from the point of view of a virtual camera. A convenient representation of a frustum is as an array of six planes, each of them being represented in point normal form. A basic idea for checking if a point resides inside a frustum is to determine whether the point lies in front or on the back side of each plane. If it lies inside all six planes, it is inside the frustum.

4.1.5 Resource Management

Every game is constructed from a wide variety of resources, called assets in most cases. They are materials, textures, shader programs, animations, audio clips, meshes, level layouts, physics parameters and collision primitives etc. The game's resources must be managed in terms of offline tools used for creation and in terms of loading and manipulation at runtime. This is where the resource manager comes into the scene. Every resource manager has 2 distinct but integrated components. A component manages the chain of off-line tools used for the asset creation and the other one, manages the resources at runtime, ensuring that they memory loaded in advance. They are unloaded from them memory when they are no longer needed.

In most of the cases of engines, the resource manager is a system or subsystem that is centralized and it manages all types of resources that are used by the game. In others, the manager is spread across a collection of subsystems. However, it doesn't matter how it is implemented, but how the resource manager takes on specific responsabilities and solves well-understood problems

The Off-Line Resource Manager contains: Revision Control for Assets, Dealing with Data Size components, the Resource Database, Resource Dependencies and Build Rules.

The Runtime Resource Manager has the responsability of ensuring that one copy of a unique resource exists in memory, managing the lifetime of each resource, handle loading of composite resources, maintaining referential integrity and memory usage, permitting custom processing on a resource, providing a single unified interface and handle streaming, or asynchronous resource loading.

4.1.6 Game Rendering

As games are real-time, interactive and dynamic objects, time holds and important role in a game. Time in a game engine comes in a variety of ways: real time, game time, local timeline of animation, the CPU cycles within a particular function.

As the camera moves about in a 3D scene, the whole contents of the screen or window change Continually. An illusion of motion and interactivity is produced in much the same way a movie produces it - by presenting to the viewer a series of sill images in a rapid succession.

This rapid succession of still images on-screen requires a loop. In real-time rendering applications, this is known as the render loop. In [22] the game loop has the following structure:

Algorithm 1: Updating Position and Orientation

```
while (!quit) do
    // Update the camera transform based on interactive inputs or by
    // following a predefined path
    updateCamera()
    // Update positions, orientations and any other relevant visual
    // state of any dynamic elements the scene.
    updateSceneElements()
    // Render a still frame into an off-screen frame buffer known as
    // the "back buffer".
    renderScene()
    // Swap the back buffer with the front buffer, making the
    // most-recently-rendered image visible on-screen. (Or, in
    // windowed mode, copy (blit) the back buffer's contents to the
    // front buffer.
    swapBuffers()
end
```

A game consists of many interactive subsystems, including, rendering, animation, collision detection and resolution, audio, rigid-body dynamics simulation, animation, rendering, device I/O etc. These subsystems require periodic servicing while the game runs. But the rate at which these subsystems need to be serviced varies from one to another. For example, animation needs to be updated at a rate of 30 to 60 Hz, in synchronization with the rendering system. Other high-level systems like AI might need service once or twice per second and they don't necessarily need to be in synch with the rendering loop. Game loops can be implemented in a number of different ways - but at their core, they usually boil down to one or more simple loops, with various embellishments. We'll explore a few of the more common architectures below. Game loop architectural Styles involve Windows message Pumps, Callback-Driven Frameworks, Event-based Updating.

Rendering Engine

Real-time 3D rendering is a very deep and broad topic. Real-time 3D graphics or three-dimensional scene involves the following steps:

A virtual scheme is described, in terms of 3D surfaces in some sort of mathematical form.

A virtual camera is positioned and oriented to produce the preferred and desired view of the scene. In most cases, the camera is modelled as an idealized focal point, that has an imaging surface that hovers some small distance in front of it, composed of virtual light sensors that correspond to the picture elements (pixels) of the target display device.

A large variety of light sources are defined. These sources provide all the light rays what interact and reflect off the objects in the environment and find their way onto the image-sensing surface of the virtual camera. The visual properties of the surfaces in the scene are described. This defines how the interaction of each surface with light is produced. Each pixel in the imaging rectangle is associated with the color and intensity of the light rays that converge on the virtual cameraâĂŹs focal point through the pixel. These are calculated by the rendering engine. Real-time rendering engines perform the steps listed above repeatedly, displaying rendered images at a rate of 30, 50, 60, 90 frames per second (FPS) to provide the illusion of motion. Therefore, the rendering engine has at most 33.3 ms to generate each image (for a 30 FPS). Usually much less time is available.

A real-world scene is composed of objects. Some objects are solid, others are amorphous, but every one of them occupies a volume of 3D space. Representations used by Rendering engines are most commonly are triangle meshes.

The Rendering Pipeline

In real-time game rendering engines, the high-level rendering steps described above are implemented using a software/hardware architecture known as a pipeline. A pipeline is just and ordered chain of computational stages each with a specific purpose, operating on a stream of input data items and it produces a stream of output data. Each stage of a pipeline is defined in such a way that it can operate independently of the other stages. Therefore it can be parallelized very easy.

The high level stages in our pipeline are:

Tools stage (offline). Geometry and surface properties (materials) are defined.

Asset conditioning stage (offline). The geometry and material data are processed by the asset conditioning pipeline (ACP) into an engine-ready format.

Application stage (CPU). Potentially visible mesh instances are identified and submitted to the graphics hardware along with their materials for rendering. Geometry processing stage (GPU). Vertices are transformed and lit and projected into homogeneous clip space. Triangles are processed by the optional geometry shader and then clipped to the frustum.

Rasterization stage (GPU). Triangles are converted into fragments that are shaded, passed through various tests (z test, alpha test, stencil test, etc.) and finally blended into the frame buffer.

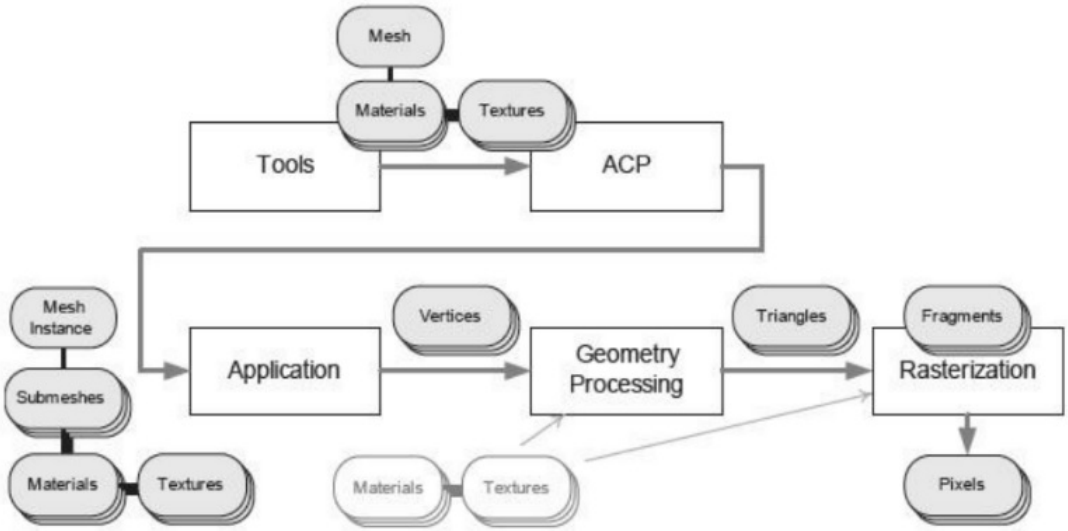

Figure 4.30: SDK layer

4.1.7 Gameplay Systems

A game is defined by its gameplay. Gameplay can be defined as the overall experience of playing a game. The term game mechanics defines this idea more concretely- the set of rules that govern the interactions between the various entities in the game. On the other, hand it also defines the objectives of the player, success or failure criteria, player characterâĂŹs abilities, the number of non-player entities that exist in the game world and their interaction. Lastly, it defines the overall flow of the gaming experience as a whole. The designs of a gameplay vary from one genre to another and game to game.

Game World

Most video games take place in 2D or 3D game world. This world is comprised of numerous elements of discrete nature. These elements fall under two categories: static elements and dynamic elements. Static elements are: the terrains, buildings, roads, bridges, everything that does not move or interact with the gameplay in an active way. Dynamic elements contain: characters, vehicles, particle emitters, weaponry, floating power-ups and health packs, collectible objects, dynamic lights, splines etc. Gameplay in general is concentrated around the dynamic elements of a game. They are concerned with updating locations, internal states and orientations because of the fact that they are elements that change over time.

Game state refers to the current state of all dynamic game world elements, as a whole.

Static geometry is a static world element which is most of the times defined in a

tools like 3ds max and Maya.

Dynamic Elements: Game Objects (GOs)

The dynamic elements of a game are usually defined in an OOP manner. Being a natural and intuitive approach, it maps very well to the designer's notion of how the world is constructed. Visualization of characters, vehicles and other dynamic objects is more handy. They are created and manipulated in the world editor. From the programmer's side of things, it is easy and natural to implement dynamic elements as largely autonomous agents at runtime.

Other terms for game objects are entities, actors or agents.

From an object-oriented design perspective, a game object is in essence a collection of attributes (current state of the object) and behaviors (how state changes over time in response to events. The instances of a particular type share the same attribute schema and same set of behaviors., but the values of the attributes vary from instance to instance.

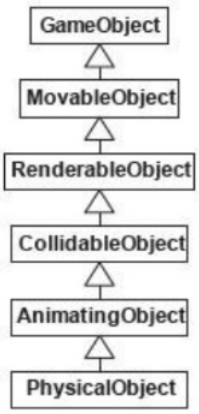

Figure 4.31: GO inheritance

Game Object Models

The term game object models, in the context of game development is used to describe the facilities provided by a game engine in order to permit dynamic entities in the virtual game world to be modelled and simulated. Furthermore, a game object model is a specific object-oriented programming interface.

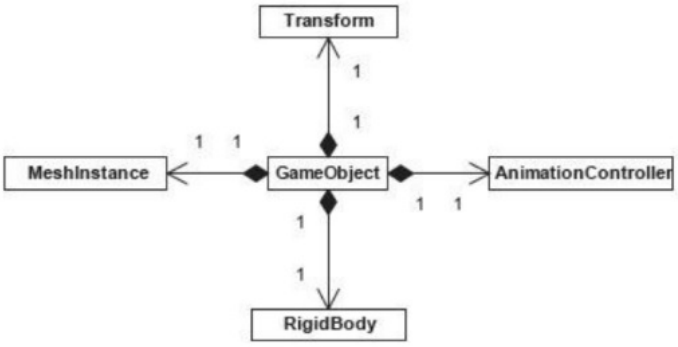

Figure 4.32: GO composition relation

4.2 Artificial Intelligence Algorithms and Techniques

4.2.1 Movement Algorithms

A major component of a large variety of character-based game is the artificial intelligence (AI). In its lowest levels, an AI is usually founded in technologies like basic path-finding, knowledge of the environment, movements, etc. On top of these foundations the control logic is implemented. A character control system determines how to make the character perform specific actions such as: navigation, locomotion, weapons using, driving vehicles, taking cover, interactions with other entities like AI, and so on. Above the character control the AI can have goal setting and decision making, emotional state, group behaviors) or abilities to learning from previous mistakes and adapt to the changing environment. As previously stated the term AI is loosely used in video games. It does not represent artificial intelligence in the true sense. In a game all that matters is the player's perception of what it is going on.

For the present project and application at hand, the following artificial intelligence techniques and algorithms are used: Finite-State-Machines, Movements and Decision Making.

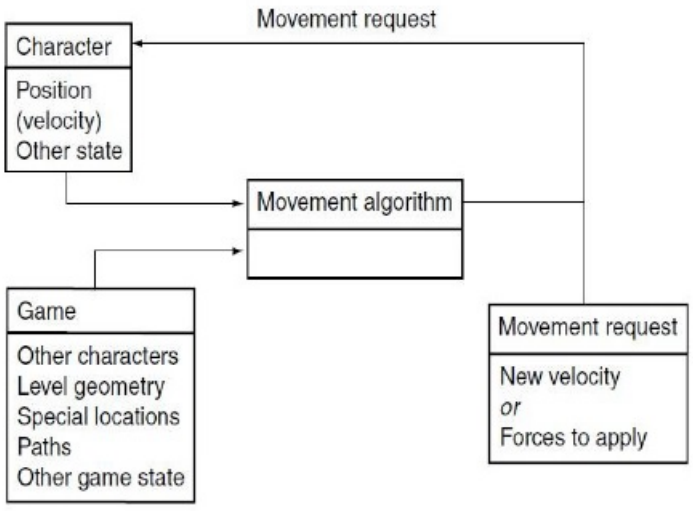

Figure 4.33: The movement algorithm

Algorithm 2: Kinematics - Updating Position and Orientation

```
update(steering, time):
    // Update the position and orientation
    position += velocity * time + 0.5 * steering.linear * time * time
    orientation += rotation * time + 0.5 * steering.angular * time * time
    // and the velocity and rotation
    velocity += steering.linear * time
    orientation += steering.angular * time
    // and the velocity and rotation
    velocity += steering.linear * time
    orientation += steering.angular * time
end

getNewOrientation(currentOrientation, velocity):
    // Make sure we have a velocity
    if velocity.length() > 0 then
        return arctan(−static.x, static.z)
    else
        return currentOrientation
    end
end
```

Algorithm 3: Seek and Flee Algorithm

```
Class Seek:
    // Holds the maximum acceleration of the character
    maxAcceleration
    // Returns the desired steering output
    getNewOrientation():
        // Create the structure to hold our output
        steering = new SteeringOutput()
        // Get the direction to the target steering.linear = target.position -
          character.position
        // Give full acceleration along this direction
        steering.linear.normalize()
        steering.linear *= maxAcceleration
        // Output the steering
        steering.angular = 0
        return steering
    end
end
```

Algorithms for 3D Movement

The face behaviour makes a character look at its target. It delegates to the align behaviour to perform the rotation, but calculates the target orientation first. The target orientation is generated from the relative position of the target to the character. It is the same process used in the **getOrientation** function for kinematic movement. The pseudo-code is below:

In [18] Ian Millington's version states that "A kinematic algorithm simply gives

the direction to the target, you move in that direction until you arrive, where upon the algorithm returns no direction: you've arrived." Figure 4.34 depicts the relation between the algorithm the character, and the game model. The algorithm pseudo-code is presented below. It is divided into two parts. First part represents the position and orientation definition and the updating of the two. The second part represents the algorithm.

Algorithm 4: Facing 3D

```
Class Face3D(Align3D):
    // The base orientation used to calculate facing
    baseOrientation
    target Overridden target
    // ... Other data is derived from the superclass ...
    // Calculate an orientation for a given vector
    def calculateOrientation(vector):
        baseZVector = new Vector(0,0,1) * baseOrientation
        // Get the base vector by transforming the z axis by base orientation (this // only
          needs to be done once for each base orientation, so could be cached between calls).
        if ( baseZVector == vector ) then
            // if If the base vector is the same as the target, return the base quaternion
            return baseOrientation
        if ( baseZVector == -vector ) then
            // If it is the exact opposite, return the inverse of the base quaternion
            return -baseOrientation
        else
            // Otherwise find the minimum rotation from the base to the target
            change = baseZVector x vector
        end
        // Find the angle and axis
        angle = arcsin (change.length()
        axis = change
        axis.normalize()
        return new
        Quaternion( cos (θ/2), sin (θ/2) * axis.x, sin (θ/2) * axis.y, sin (θ/2) * axis.z)
    end
    getSteering():
        // 1. Calculate the target todelegate to align and Work out the direction to target
        direction = target.position - character.position if ( direction.length() == 0 ) then
            // Check for a zero direction, and make no change if so return target
        // Put the target together
        Align3D.target = explicitTarget
        Align3D.target.orientation = calculateOrientation(direction)
        // 2. Delegate to align
        return Align3D.getSteering()
    end
end
```

4.2.2 Finite-State-Machines

State Machines

In a state machine each character occupies a state. Normally, actions or behaviors. are associated with each state. So as long as the character remains in that state, it will continue carrying out the same action.

The states are connected together by transitions. Each transition leads from one state to another, the target state, and each has a set of associated conditions. If the game the character changes state to the transitionâĂŹs target state. When a transitionâĂŹs conditions are met, it is said to trigger, and when the transition is followed to a new state, it has fired.

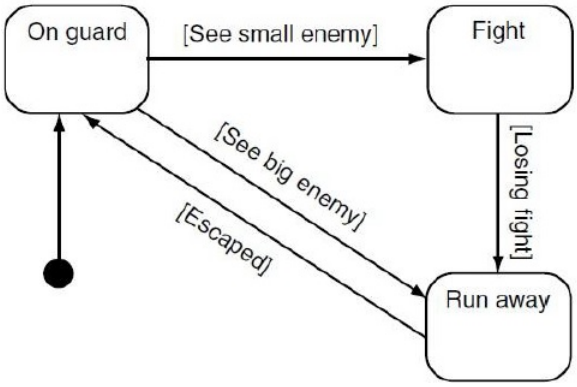

Figure 4.34: The Game FSM as seen in

Finite State Machines

In game AI any state machine with this kind of structure is usually called a finite state machine (FSM). This and the following sections will cover a range of increasingly powerful state machine implementations, all of which are often referred to as FSMs.

This causes confusion with non-games programmers, for whom the term FSM is more commonly used for a particular type of simple state machine. An FSM in computer science normally refers to an algorithm used for parsing text. Compilers use an FSM to tokenize the input code into symbols that can be interpreted by the compiler.

The Game FSM

The basic state machine structure is very general and admits any number of implementations. For the project at hand the FSM implemented is the Hard-Coded FSMs. In a Hard-Coded FSM, the state machine consists of an enumerated value that indicates which

state is currently occupied, and a function that checks if a transition should be followed. The algorithm is taken from [18] and is presented below:

Algorithm 5: Finite State Machine

```
Class MyFSM:
    // Defines the names for each state
    enum State: PATROL
                DEFEND
                SLEEP
    // Holds the current state
    myState
    update():
        if ( myState == PATROL: ) then
            if ( canSeePlayer(): ) then
                myState = DEFEND
            if ( tired(): ) then
                myState = SLEEP
        else if ( myState == DEFEND ) then
            if ( NOT canSeePlayer() ) then
                myState = PATROL
        end
        else if ( myState == SLEEP ) then
            if ( not tired() ) then
                myState = PATROL
        end
    notifyNoiseHeard(volume):
        if ( myState == SLEEP AND volume > 10 ) then
            myState = DEFEND
    getAction():
        if ( myState == PATROL ) then
            return PatrolAction
        else if ( myState == DEFEND ) then
            return DefendAction
        end
        else if ( myState == SLEEP ) then
            return SleepAction
        end
end
```

4.2.3 Decision Trees

Decision trees are fast, easily implemented, and simple to understand. They are the simplest decision making technique that exists. They are used extensively to control characters and for other in-game decision making, such as animation control. They have the advantage of being very modular and easy to create. Implementations vary from animation to complex strategic and tactical AI.

Decisions in a tree are simple. They typically check a single value and don't contain any Boolean logic (i.e., they don't join tests together with AND or OR). Depending on the implementation and the data types of the values stored in the character's knowledge,

different kinds of tests may be possible. A representative set is given in the following table, based on a game engine I've worked.

Figure 4.36 highlights an example decision trees and the pseudo-code is below:

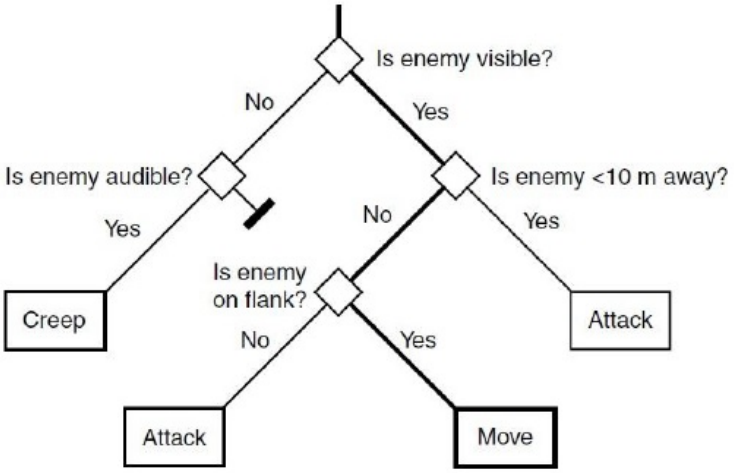

Figure 4.35: Decision Trees Example

Algorithm 6: Decision Tree

```
class DecisionTreeNode:
    def makeDecision():
end
class Action:
    def makeDecision():
        return this
end
class Decision (DecisionTreeNode): trueNode, falseNode, testValue
    def getBranch():
    def makeDecision():
end
class FloatDecision (Decision): minValue, maxValue
    def getBranch():
        if ( maxValue >= testValue >= minValue ) then
            return trueNode
        else
            return falseNode
        end
end
class Decision (DecisionTreeNode):
    def makeDecision():
        branch = getBranch()
        return branch.makeDecision()
end
class MultiDecision (DecisionTreeNode): daughterNodes, testValue
    def getBranch():
        return daughterNodes[testValue]
    def makeDecision():
        branch = getBranch()
        return branch.makeDecision()
end
```

4.3 Game Engine Technologies: Unity 3D Game Engine

Unity is a open-source and paid cross platform game engine and IDE (Integrated Development Environment) that targets web plug-ins(such as Web Player or Flash player), desktop platforms(Windows, Mac) and mobile devices (iOS and Android). It is known to be a development ecosystem having a powerful rendering engine that is fully integrated with a set of intuitive tools and rapid workflows with the scope of creating interactive 3D content. The engine is written in C++. Unity supports integration with 3ds Max, Maya, Softimage, Blender, Modo, ZBrush, Cinema 4D, Cheetah3D,Adobe Photoshop, Adobe Fireworks and Allegorithmic Substance. Changes made to assets created in these products automatically updates all instances of that asset throughout the project without needing to manually reimport.

Graphics engine uses Direct3D (Windows), OpenGL (Mac, Windows), OpenGL

ES (Android, iOS), and proprietary APIs (Wii). Support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects.

ShaderLab language for using shaders, supporting both declarative "programming" of the fixed-function pipeline and shader programs written in GLSL or Cg. A shader can include multiple variants and a declarative fallback specification, allowing Unity to detect the best variant for the current video card and if none are compatible, fall back to an alternative shader that may sacrifice features for broader compatibility.

Built-in support for Nvidia's (formerly Ageia's) PhysX physics engine, (as of Unity 3.0) with added support for real-time cloth on arbitrary and skinned meshes, thick ray casts, and collision layers.

Game scripting comes via Mono. Scripting is built on Mono, the open-source implementation of the .NET Framework. Programmers can use UnityScript (a custom language with ECMAScript-inspired syntax), C# or Boo(which has a Python-inspired syntax). Starting with the 3.0 release, Unity ships with a customized version of MonoDevelop for debugging scripts. Unity also includes Unity Asset Server - a version control solution for all game assets and scripts, using PostgreSQL as a backend, audio system built on FMOD library, with ability to playback Ogg Vorbis compressed audio, video playback using Theora codec, a terrain and vegetation engine, supporting tree billboarding, Occlusion Culling with Umbra, built-in lightmapping and global illumination with Beast, multiplayer networking using RakNet, built-in pathfinding navigation meshes. Information was taken from [23] and [24].

$$NewCalculatedCost/Qty = \frac{ExistentCalculatedCost/Qty}{ProductionDocumentQty * \mathbf{CostingLotSize}} \tag{4.35}$$

Chapter 5

Detailed Design and Implementation

5.1 Proposed Solution

5.1.1 Layered Structure and Design

The solution to achieve the main objective of the project will be presented below. As previously stated, this project regarding the generation of game specific tools, and subsequently the implementation of a small game depicts a layered structure. The bottom layer is represented by the Unity 3D Engine Development Platform. A brief introduction to this powerful tool was made above. Using Unity's capabilities, functionalities and assets, the next layer above that contains the game-oriented tools and subsystems will be mapped onto Unity. At the highest level we have the actual game implementation along with the 3D models and the set of rules that govern the game, i.e. object and entities interaction, collision and physics implementation and structuring, level design, artificial intelligence algorithms high level implementation, etc. Figure 5.1 depicts this structure.

5.1.2 Conceptual Architecture

The Use Case Model offers relevant and detailed information regarding the behaviors of the system or application that is in the course of development. As stated in earlier chapters and sections, the use case model identifies the requirements of the game application. The behavior description is given in terms of functionality so that the main goals are achieved. Therefore, use cases define the functionality of the behaviors that arise from the requirements and describe the results. In terms of how the system functions internally, use cases do not touch this subject.

Although there are three actors involved in the model, the emphasis will be put onto the user, the player of the game, because he is the most important. The most relevant and descriptive 3 use cases are defined below.

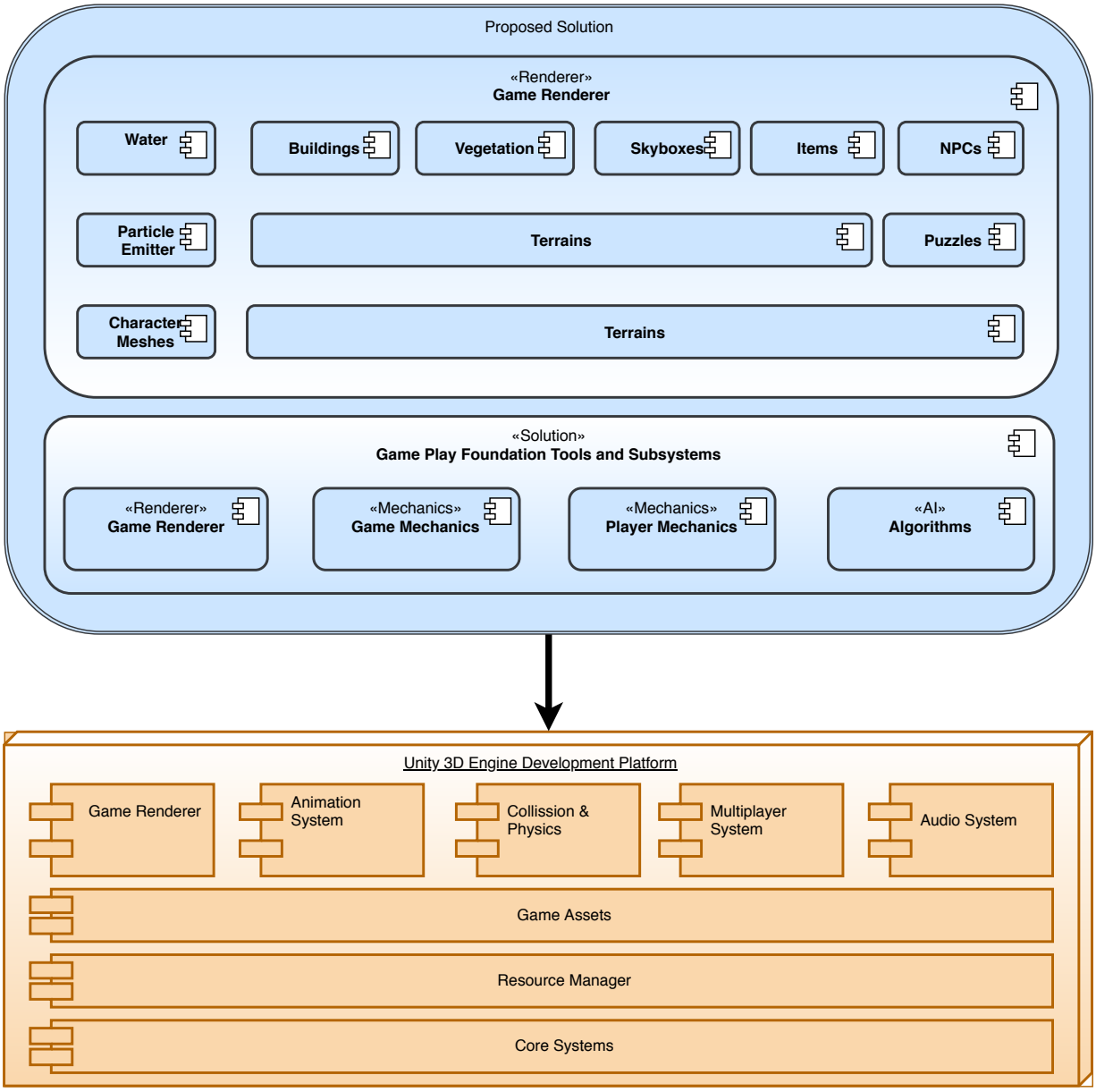

Figure 5.1: Hardware layer

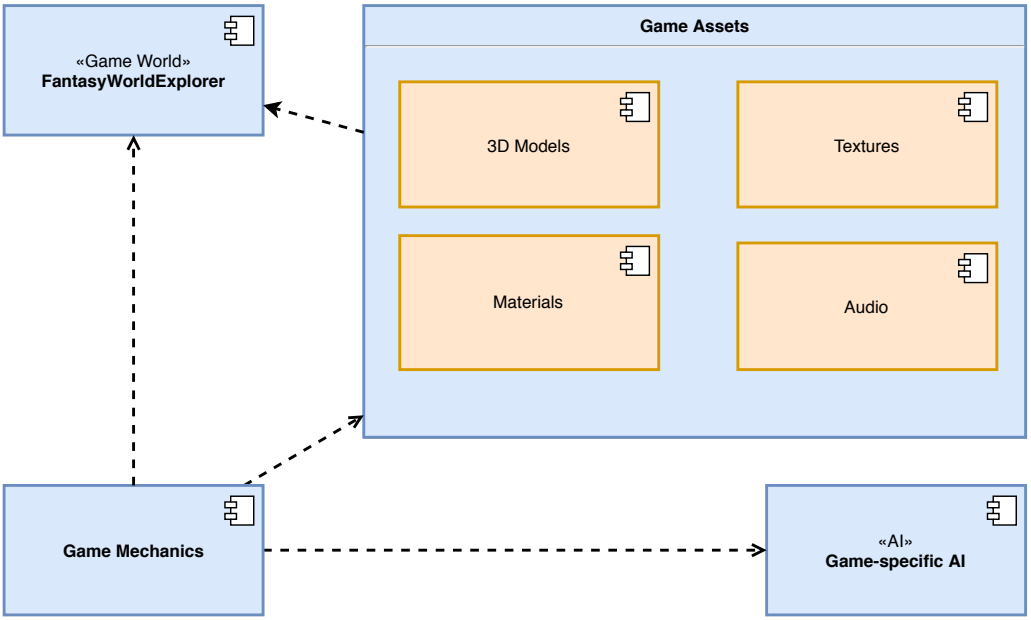

Figure 5.2: High-Level Conceptual Architecture

5.1.3 Use Case Model

The actors scenarios and use cases are identified. The most important three use-cases will be described in the format specific for writing use cases. The use-cases are:

Use case Number 1: generate the player avatar and Player preferences Level: user-goal Primary actor: user/player Main succes scenario: the user inputs the name and spends; the system saves the data in the register

Use case Number 2: customize the player avatar Level: user-goal Primary actor: user/player Main succes scenario: the user customizes the player character by adding a specific haircut, face, scale change, hair color; system saves the preferred data in the register and loads it into the next level

Use case Number 3: interact with the chest object Level: sub-function of the play game use - case Chapter 4 48 Primary actor: user/player Main succes scenario: in the first level scenario, the player opens the chest and selects the desired weapons and items; system adds the items in the player inventory.

The use case diagrams is the representation of the userâĂŹs interaction with the system, depicting the specifications of the above use-cases. They are illustrated in Figure 5.3 and 4.40.

5.2 Detailed Design

As stated in previous chapters, the main objective of the project is to design and implement a particular set of game-specific tools and components layered onto Unity Game

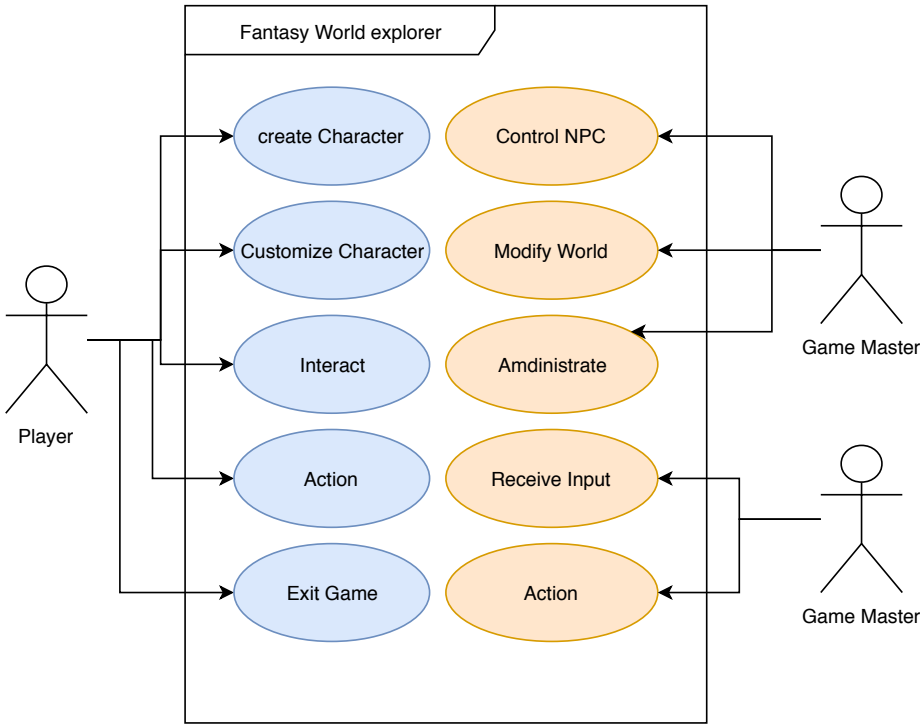

Figure 5.3: Use Case Diagram

Engine. The most important components of the application will be: the Game Mechanics, The AI System, The game Assets and of course, UnityâĂŹs functionality and capabilities through its powerful engines.

The Game Mechanics component represents the set of rules created to produce the game-play of the game. There is a wide variety of game mechanics. This design will contains some of the basic ones.

The game mechanics work in relation with the Game Assets or game resources. The Game Assets or Game Resources are the graphical and audio components created for the game. They can be graphical 3D models, particle effects, textures, maps, materials and audio effects and sounds.

The AI (Artificial Intelligence) System represents the techniques and algorithms used to produce the illusion of the existence of intelligence in the behavior of non-player characters (NPCs) in interacting with the player character or with each other.

The last component is represented by the actual game itself, the parameters for object placements, camera controls, game objects creation, customization and placement, level and scene design along with setting up the player input system, the inventory management. In the following sections, a more detailed version of the architectural design of the game application will be presented using more specific components and tools.

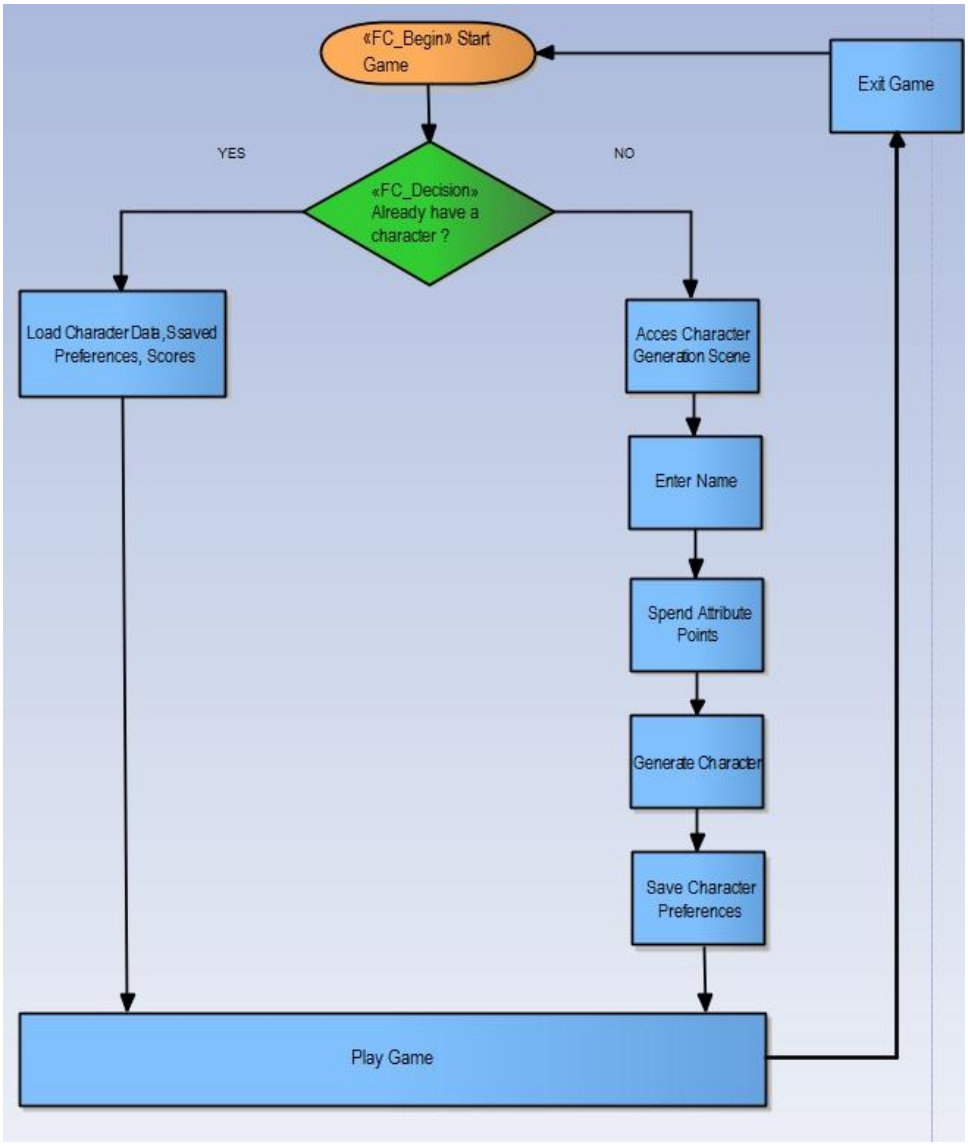

Figure 5.4: Flow Diagram

5.2.1 Application Components - Unity

Detailed Design

A stated in previous chapters, the main objective of the project is to design and implement a particular set of game-specific tools and components layered onto Unity Game Engine. The most important components of the application will be: the Game Mechanics, The AI System, The game Assets and of course, Unity's functionality and capabilities through its powerful engines.

The Game Mechanics component represents the set of rules created to produce the game-play of the game. There is a wide variety of game mechanics. This design will contains some of the basic ones. The game mechanics work in relation with the Game Assets or game resources.

The Game Assets or Game Resources are the graphical and audio components created for the game. They can be graphical 3D models, particle effects, textures, maps, materials and audio effects and sounds.

The AI (Artificial Intelligence) System represents the techniques and algorithms used to produce the illusion of the existence of intelligence in the behavior of non-player characters (NPCs) in interacting with the player character or with each other.

The last component is represented by the actual game itself, the parameters for object placements, camera controls, game objects creation, customization and placement, level and scene design along with setting up the player input system, the inventory management. In the following sections, a more detailed version of the architectural design of the game application will be presented using more specific components and tools.

As described in the previous chapter, Unity represents very potent development ecosystem: "a powerful rendering engine fully integrated with a complete set of intuitive tools and rapid workflows to create interactive 3D content; easy multiplatform publishing; thousands of quality, ready-made assets in the Asset Store and a knowledge-sharing Community" [23]. Next, the most important classes of the Unity API will be presented and described. These components are essential in the creation of the tools and, consequently, for the game implementation that follows form this. A very general view of the structure of a Unity 3D project is illustrated in the figure below, in Figure 5.1.

...

Unity's architecture may very well be seen as a MVC (Model-View-Controller) architecture with the structural design depicted in Figure 5.2.

...

In Figure 5.3, it can be seen the hierarchical structure of the objects created using the most important classes.

...

Some of the most important classes and components that will be used for the tools, game design and implementation are: GameObject, Component, Renderer, Camera, Animation, Behaviour and Transform. Their functionality will be described briefly and their diagrams will be illustrated as well. The diagrams are taken from [24].

GameObject Class

The GameObject class represents the base class for all entities in Unity scenes and it inherits from class object, which is its superclass. It aggregates the following classes: Transform, Rigidbody, Renderer, Collider, ParticleEmitter, Camera, Animation, Light, Constant Fornce, AudioSource, NetworkView, GUIEmement and HingeJoint. Game objects are the containers for all other Component classes. All of the objects in our game are inevitably and inherently gameObjects. Although they do not add characteristics themselves, they hold the Components, that are the actual classes that implement the functionality: ex. gameobject.AddComponent(ClassName). GameObjects also have a Tag, a Layer and a Name tags are used to assign a certain preferred name so that it can be accessed quicker. Layers can be used in rendering situations. The structure of the class is presented in Figure 5.5, taken from [24].

Component Class

The main functionalities of the Component class are the broadcasting of messages to listeners and the fact that it manages the scene Component instances that are registered. The pieces of the scene are connected and wired through the Component class. It composits all components into itself, therefore it is also the base class, the superclass for all other Components. In other words, the component class is the base class for everything that is attached to GameObjects. The UML class diagram of the Component API for Unity 3D is presented in Figure 5.5

Renderer Class

The renderer class renders the materials of the current mesh and for achieving this, it makes use of the Material, Vector4, Matrix4x4 and Bounds classes. They correspond to mesh rendering, lightmap tiling, transformation and bounding volume to the renderer. It basically presents an object on the screen. The renderer class is presented in Figure 5.6.

Behaviour Class

This class inherits from the component and it activates and deactivates the component. The class UML diagram is depicted in Figure 5.7.

Transform Class

The Transform class is responsible with dealing with its own position, scale, rotation and translation, but also contains the Transform children. It is implemented via the Composite Design Pattern. The main functionalities of this class are: the addition and removal of children, queries among these children, applying such operations as Translation and Rotation, operations which were explained in previous chapter, in their pure mathematical form. On the other hand, some very important mathematical-oriented classes,

such as Vector3, Quaternion and Matrix4x4 are used by this class. These notions were also explained in previous chapter in their mathematical and theoretical form. The parent property of the parent Transform will automatically add the child Transform to the parent. The root property returns the top level Transform that is attached to the actual Scene. Figure 5.8 illustrates the Transform class's attributes and relations with other important classes.

Animation Class

The Animation component is composed in the gameObject as the animation property. Every GameObject uses this instance to control it's registered AnimationState instances.

5.2.2 Deployment

The deployment diagrams are used to illustrate the topology of the physical components involved in a system where the software is deployed. For this present game application we will have 2 variants: both running on a PC, but one running as a standalone desktop application, and the other is integrated in a HTML page and it requires internet connectivity and the Web Player plug-in provide by Unity Technologies. The deployment diagrams is depicted in Figure 5.11.

5.2.3 Game-specific A.I.

In the previous chapter, a description of the Hardwired Finite State Machine was made from an analytical standpoint. At this phase, we will create our own version of a Finite State Machine System, corresponding to the Player Movement packages and AI, Mob and Enemy AI. Also, a customized version of decision trees, movement techniques and Finite State Machines will be depicted in the following UML diagrams. These Finite State Machines, Decision Trees and Flowcharts are the graphical representation of how the basic AI elements will be implemented in the later Implementation section. Each finite state corresponds to some action that is produced by player input, based upon certain contextual variables and parameters. From a logical standpoint, these packages form the Game AI component, which is one of the most important and relevant in game development and game programming since it models the set of rules for the actual game-play and it creates the illusion that there is some sorts of intelligence capable of interacting with the userâĂŹs character or avatar. Figure 5.12 depicts a general form of how the enemy AI will interact with the player character. It describes 7 main states: Idle, Init, SetUp, Search, Attack, Flee and Retreat. These states will make use of the movement capabilities and techniques that are offered by the Movement Package. Figure 5.13 illustrates the algorithm that combines both Finite State Machines (FSM) and decision trees for designing the player mechanics.

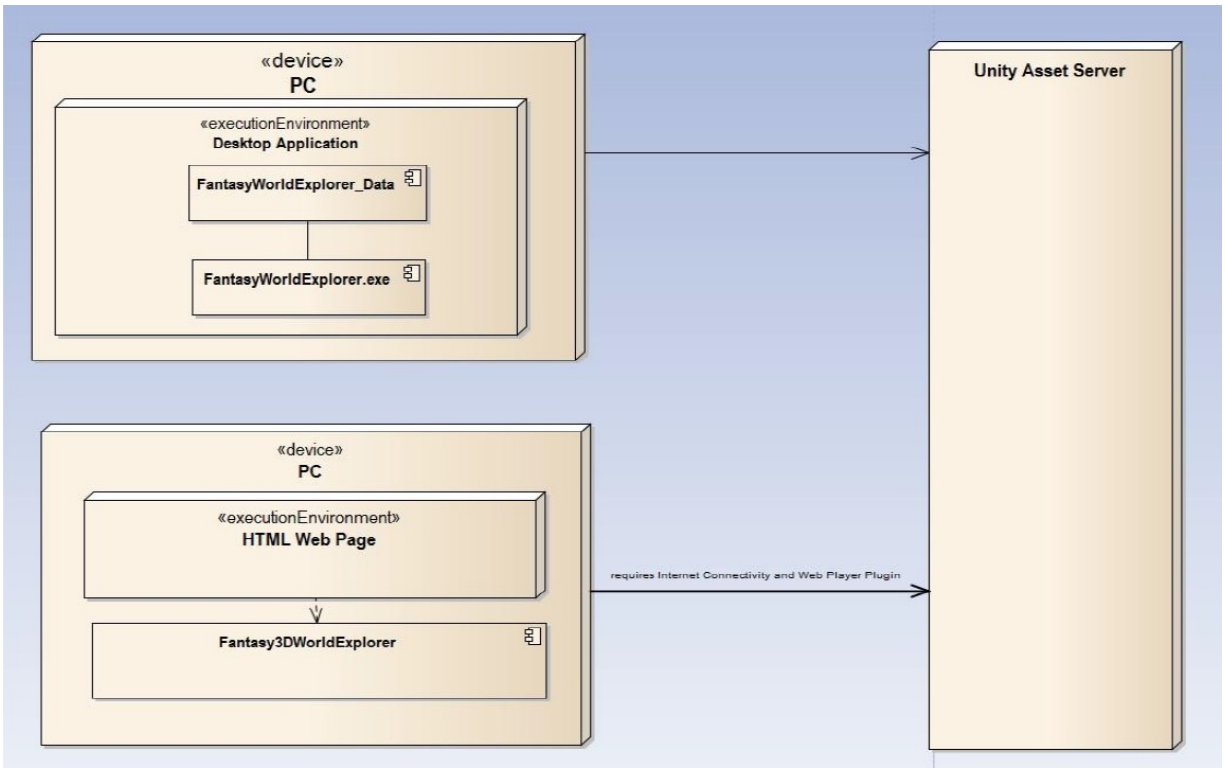

Figure 5.5: Component Diagram

Some functionality from this algorithm will be used for the design and implementation of the enemy AI.

The FSM combination with decision tress is implemented in the Movement package and in the AI package.

5.2.4 Component Design

In earlier sections, the main components were described as being the Game Mechanics, the Game Assets, the AI System and the actual game implementation. A more detailed version of these components along with the packages used is presented.

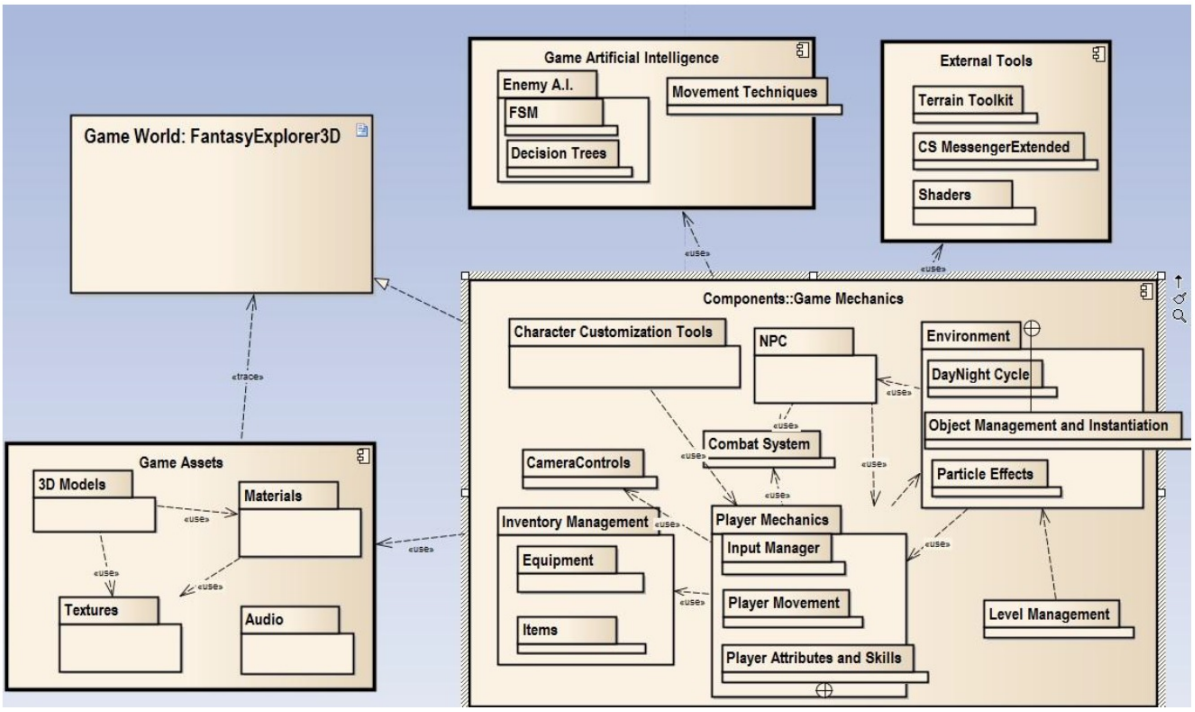

Figure 5.6: Deployment Diagram

5.2.5 Package Design

In terms of interaction between packages, the following packages are identified: Camera Controls, Player Character, DayNight Cycle, EditorGUI, HUD, Items, Mob, AI, Movement, PlayerCharacterCustomization and Utility. The Package Diagram is presented in Figure 5.10.

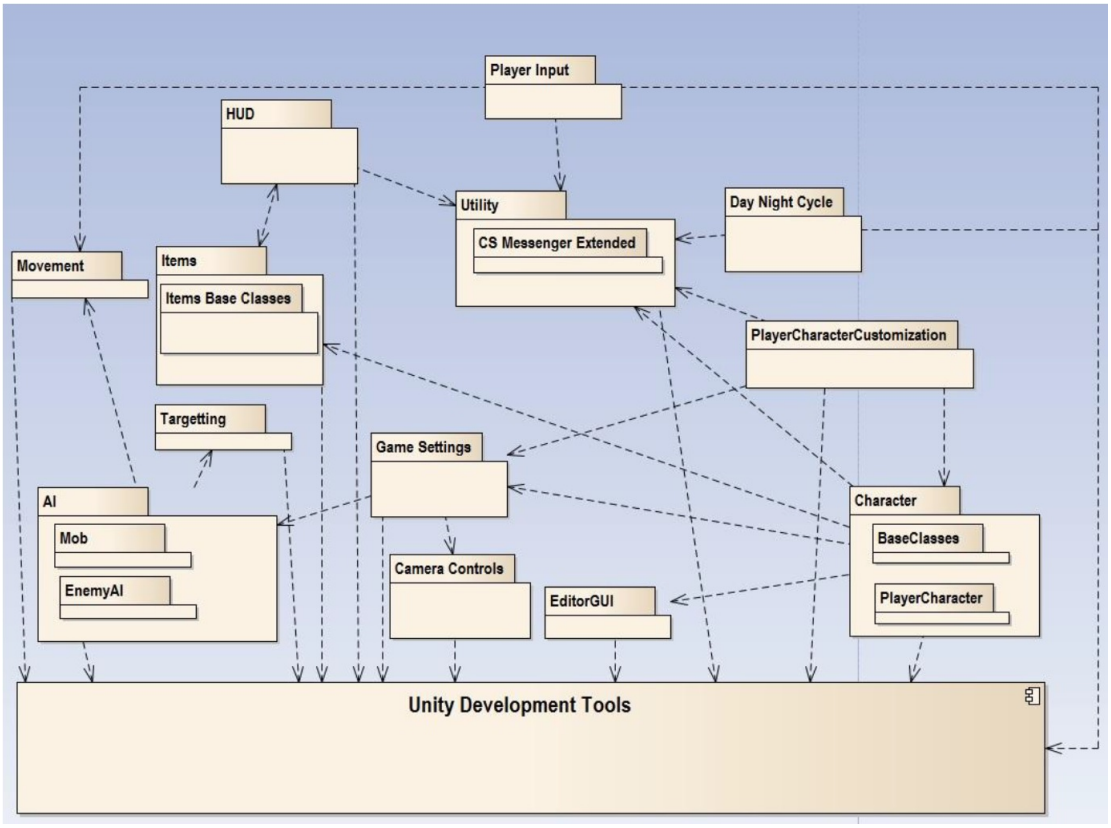

Figure 5.7: Package Diagram

5.2.6 Class Design

The Class Design represents a solution to the initial requirements and to the entities and components that arose during the analysis phase. In Figure 5.14 and 5.15, the package and component diagrams were presented highlighting the detailed design of the components and the interactions between them. The Class design represents a solution to creating, designing and implementing the set of game tools. Although the packages and components do not appear in the class diagram it is very easy to deduce them due to the modular structure of the Class Design. The modular design, based on multiple components and packages is a very important aspect in creating efficient and scalable software applications. Furthermore, the class model was designed based upon software engineering principles and it satisfies the High Cohesion and Low-Coupling Design Principles. On the other hand, the Class Model also contains the classes, components and structures that Unity offers Figure 5.14 depicts the Class Diagram of the application. Regarding the design patterns used, a very important class is the PC «PlayerCharacter» class. This class was created as a singleton. Since this class represents the player character/avatar, it is essential to hold only one instance for this class. This pattern was used for the centralization of internal and external resources. It also represents a global point of access to themselves. The class is responsible with instantiating itself. It also makes sure that it creates no more than one instance. In this case the class, The PC.cs represents a class for configuration of player preferences and accessing resources in a shared mode. Figure 5.15 illustrates the UML diagram of the PC singleton class.

5.3 Implementation

The analysis and design elements produced at previous chapters and subchapters is transformed into code. The implementation of the Game tools also involves the graphical implementation. The graphical implementation is concerned with the design, modeling, texturing and creation of 3D game models, textures, materials, meshes, particle effects, audio sound effects and audio files and so on. They are created using specific tools like the ones explained at section 4.1.2. Taking this into account, the following DCCs(Digital Content Creation) tools were used: 3ds Max 2012 Trial, from Autodesk Media and Entertainment was used for making some of the 3D models, meshes, animations which will be later imported into Unity. The formats for the models were ".max" and ".obj". Blender 2.5, by Blender Foundation, which is a free open-source 3D computer graphics product, was also used for creating some 3D models, UV unwrapping, texturing, rigging and skinning, particle effects animation and rendering. Blender products are fully compatible with Unity, and that is why it is a very powerful tool. The formats for the files were: ".blend", ".obj" and ".fbx". Adobe Photoshop CS3, from Adobe Systems was used for texture creation and editing and producing the required formats for assigning textures to materials or meshes. The textures are saved in: ".tga", ".bmp" and ".jpeg" formats. Last but not least, we have Unity 3D game engine which was presented in detail in previous chapters represents the

engine architecture that offers the functionality for developing the tools and the interactive game. The main components of the application are : the Game Mechanics, the Game AI System, the Game Assets and Resources and, of course, the actual Game implementation. The final version of the game represents the composition of all the modular components created, along with player input definition and handling. The packages identified are: Camera Controls, Player Character, DayNight Cycle, EditorGUI, HUD, Items, Mob, AI, Movement, PlayerCharacterCustomization and Utility. There are also predefined packages free on Unity's Asset Server that can be used. The external packages that were used in the present project application are: the Terrain Toolkit, sued for generating a wide variety of terrains by applying various modifiers and filters, the CSMessengerExtended used for easy message passing between classes, adding and removing listeners to certain events. In terms of GUI a free GUI skin was used called Necromancer GUI. For game testing and evaluation, a free class called DetectLeaks in order to monitor some graphical aspects of the game. The Shader package contains code for a SkyBoxBlended shader. At this chapter, each package along with the most important, relevant and essential classes will be explained, along with some important functions, procedures, attributes and algorithms. In Figure 5.16 it can be seen how the tools and technologies cooperate and how they are integrated with Unity.

5.4 Technologies used

5.5 Evaluation Metrics

Regarding the evaluation metrics used, the application is based on the processing power of the processor and the GPU or Video Card and it is essential to subject the software to performance metrics that evaluate from hardware and software perspective the amount of memory used, average frame-per-seconds, average frame-drops, amount of memory used by the GPU. It tackles such problems like rendering, pixel level operations, depth complexity, color depth, resolution, anti-aliasing, applying benchmarks (graphics, application, biases). Other issues such as lighting and animations need to be considered. Although there are numerous specific tools for providing these metrics singularly, the game engine already comes with some basic metrics for performance and scalability measurement that are optimized to work under the project's specifications, constraints and requirements. On the other hand it is quite important to relate to other related work that have been evaluated under the same conditions. The game was evaluated using FRAPS and MSI Afterburner for checking the hardware and software resources usage along with FPS average. The FPS average was 450 FPS using both tools. By adding more an more elements and graphical models, the frame rate will drop considerably, but for now it works very well.

Chapter 6

Testing and Validation

In [7] it is stated that software testing constitutes one of the major aspects of software development. It involves investigating software to discover its defects, which come in two general types. One type of defect arises when you have not developed your software according to specification. Another type of defect arises when you have developed software so that you add things to it that the specifications have not anticipated. From one perspective, your game fails to do what you expect it to do. From the other perspective, the game does things that you do not expect or want it to do. To eliminate defects, subject the game to a variety of tests. Among these tests are those involving components-modules, integration, and the system as a whole.

As figure 6.1 illustrates, testing works from the software specifications and creates a test case. The test case establishes a way that the tester can work from the specifications to test the behaviour of the software within a strictly defined context. Using information derived from the specifications, the tester defines initial and resulting conditions for the test case. Figure 6.2 illustrates three concepts that are central to testing. Although testing involves many other things, understanding what verification, validation, and exploration entail helps you grasp the basic reasoning that motivates the development of different approaches to testing.

6.1 Module Testing and Unit Tests

Verification:

Testing can be viewed as verification, validation and exploration. The set of requirements specifies the game. They represent the use cases, in plain text and use case diagrams and establish how the game is expected to behave. Testing whether the game meets the requirements through use cases is checking and verifying if the product is in accordance with the requirements. Validation refers to determining the correct responses and answers of the system, interpreting the feedback. It is usually done through a validation test. It usually aims towards data types correctness and result correctness after a

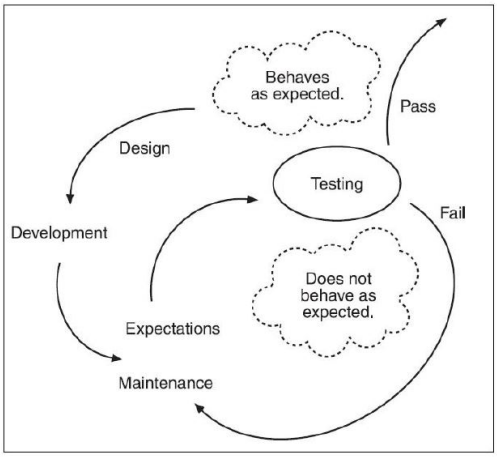

Figure 6.1: Art of testing 1, taken from [1]

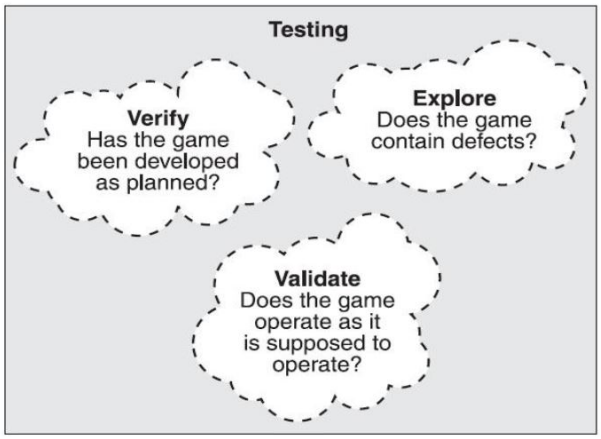

Figure 6.2: Art of testing 2, taken from [1]

certain functionality was applied. If the result is according to a standard then it is said that the software possesses validity.

Also known as component testing phase, modular testing and unit testing is concerned with the testing of the smallest piece of software for which a separate specification and requirements exist. They typically dynamic white-box tests. Software can be executed in a target system, an emulator, simulator or any other test environment. Therefore for each module and component created for the implementation of the game modular testing was defined.

The main classes aforementioned in the design and implementation phase contain several unit tests. However not all the implementation artifacts contain unit tests although the best best practice advises that every piece of code can suffer functional, syntactic and semantic changes and hence it is imperative that the software engineer adheres to the highest standards of clean code quality.

The Mob Targeting System is tested separately. The following screen shots suggest a transition from a target to another:

Next, the Day Night Cycle was tested, to highlight the transition between day and night and the earth rotation. In our case the light source revolves around the terrain:

The Camera Controls package deals with player character camera placement, target selection, parameterization of walk and run distance and height, the height and rotation damping along with X and Y speed. Other components tested were the character animations, AI and Movement.

6.2 Validation and Integration Testing

As presented in previous diagrams integration testing encompasses all unit and module test previously made. Therefore modules, components, packages and units are grouped together to confirm that they work as a whole in the system. They have to meet the requirements and they are merged together. This approach is also called system testing. In the following screenshots and captions the basic flow of the game is presented, from character loading, deletion, to character stats generation, player customization and launching and instantiating him in the created world.

The saved attributes, skills and vitals data is saved in the register on Windows. The following image illustrates that the introduced data has been correctly save in this register. By doing this, the boundary testing was achieved by testing the inputs and observing the outputs. Figure 6.11 illustrates that the saved data corresponding to the player preferences, is correctly saved into the Windows register. Unity saves the player preferences in the windows register.

On Windows, PlayerPrefs are stored in the registry under `HKCU\Software\companyname\productname` key, where company and product names are the names set up in Project Settings.

Chapter 7

User's manual

7.1 Installation Description

7.1.1 Hardware resources

To determine the minimum, recommended and optimal hardware resources for the game, it is necessary to test the game on as many computers as possible, to determine the appropriate values. The present iteration does not represent a final solution, that is why these values are not final. Along with the evaluation metrics that Unity 3D engine provides, the following conclusions arise:

Minimum System Requirements:

1. Supported Operating Systems: Microsoft Windows XP.
2. Processor: single core CPU, 1.5 GHz.
3. GPU (Graphical Processing unit) or Graphics Card: 128 Mb Memory.
4. HDD (Hard-disk-drive) space: 100 Mb of Memory PC standalone version and 20Mb for the WebPlayer version.
5. Peripheral attachment: keyboard and mouse.
6. Internet connection required(minimum 50 Mbps)

Recommended System Requirements:

1. Supported Operating Systems: Microsoft Windows Vista, Microsoft Windows 7, Microsoft Windows 8 on 32/64 bits
2. Processor: dual core CPU, 2.6 GHz.
3. GPU (Graphical Processing unit) or Graphics Card: 512 Mb Memory.

4. HDD (Hard-disk-drive) space: 100 Mb of Memory PC standalone version and 20Mb for the WebPlayer version.

5. Peripheral attachment: keyboard and mouse.

6. Internet connection required(minimum 100 Mbps) [1]

Optimal System Requirements:

1. Supported Operating Systems: Microsoft Windows Vista, Microsoft Windows 7, Microsoft Windows 8 on 64 bits

2. Processor: Core i5 CPU M 520, 2.4 GHz.

3. GPU (Graphical Processing unit) or Graphics Card: 1 GB Memory.

4. HDD (Hard-disk-drive) space: 100 Mb of Memory for the PC standalone version and 20Mb for the WebPlayer version.

5. Peripheral attachment: keyboard and mouse.

6. Internet connection required(minimum 100 Mbps). [2]

System Requirements for Unity development: 1. Windows: XP SP2 or later; Mac OS X "Snow Leopard" 10.6 or later. Note that Unity was not tested on server versions of Windows and OS X. 2. Graphics card with DirectX 9 level (shader model 2.0) capabilities. Any card made since 2004 should work.

System Requirements for Unity-made Content: 1. Windows XP or later; Mac OS X 10.5 or later. 2. Pretty much any 3D graphics card, depending on complexity. 3. Online games run on all browsers, including IE, Chrome, Firefox, and Safari, among others

Regarding software resources needed the game was created for: PC standalone and for WebPlayer. From the perspective of the simple user:

1. If he wants to play the PC standalone version, he simply needs copy the Resource folder onto his machine and run the .exe file. He is not required to install any additional software. 2. If the user wants to play the WebPlayer version, he is required to have 2 things: an Internet connection and the Unity WebPlayer plug-in installed.

It is very simple to install the plug-in. Simply run the file with the extension .unity3d and it will automatically verify if the plug-in is available. If not it asks the user to download it. After the download, the user installs the plug-in and the game will run in a .html web page.

[1] Only for the Web Player version. For the PC standalone version the internet connection is not required.
[2] Only for the Web Player version. For the PC standalone version the internet connection is not required.

7.1.2 Deployment and Installation

From an engineer or programmer point of view, regarding further development, it is required to install Unity 3D engine. The preset game implementation was realized using the 4.0.0 version of the game engine. To import the project simply select File -> Open Project and go to the Project folder and select. Unity will import all of the assets and from now on it depends on his skills and abilities.

For graphical modeling and animation the tools used were 3ds Max 2011 and Blender 2.5. Blender is very easy to install and it can be found on the web site.

7.2 User manual

A software product requires an user manual that describes how the end user will be able to interact and user the game. This manual refers to the small game implementation. The game is relatively small and basic, but playable. It can be categorized as an RPG (Role-Playing-Game) style game. The following steps are provided as a guide towards end-users.

Step1. Player launches the application. If he does not have a created player character then he will need to fill in his name and spend the attribute and skill points available. Through the relatively intuitive GUI, users spend their attribute points. When they feel that they have what they want they press the "Next" button to lead them to the following scene. The saved attributes, skills and vitals data is saved in the register on Windows, as stated in previous chapter. Figure 7.1 illustrates all the attributes that can be spent by the user. They are: Might, Constitution, Nimbleness, Speed, Concentration, Willpower and charisma. The way in which the user decides to spend these points affect, in the end the skills and attributes. In Figure 7.2, if the user already has a created player character he is prompted with the following scene. This is where he chooses whether he wants to load the character or delete it. If he wants to delete it he is returned to the character generation scene where he needs to create his preferences.

Step2. After the character generation and preferences saving, the following scene is loaded. The user needs to customize his player instance. He can change skin color, add haircut, select a color for his hair, choose a face, rotate the character change the scale of his character. This data is also saved in the register. The graphical user interface elements associated with this scene are presented in the figures below. Figure 7.3 illustrates the ability to change the hair style of the player character along with the color. There are a wide variety of colors to choose from: black, light brown, dark brown, red, white and blond. The arrows provide passing from a hair style mesh to another. When the user is happy with his selection, he can move on to the next customization control. Figure 7.4 represents the controls associated with the face selection of the character. The user has the ability to change from a variety of face meshes for his character.

Figure 7.5 defines the controls for character rotation in the scene. The following controls, in figure 7.6 below and 7.7 define the ability to change the scale of the character

(height, width) and the skin color (from 3 different skin tones). The scale controls are very easy to use because they are actually 2 sliders: vertical and horizontal. The skin changers are 3 cubes each: The above illustrations represent the various player design by modifying the controls: scale controls, skin tone controls, face changing controls, hair style and colors controls. Future additions and improvements may involve a wide variety of controls for customization.

Step3. After user has decided to stick with the final version of his character mesh he has to press th "Next" button. Then the first level of the game is loaded. This level corresponds to instantiating the saved player into the open world. The following illustrations and screen-captions are different contexts and scenarios from in-game play. It can be seen that they fulfill the specified requirements and represent the basic mechanism for a small-sized game. In the following chapters, aspects regarding further development and on-going improvements are approached.

Figures 7.14 and 7.15 illustrate a screenshot for the interaction with the "Loot Chest" object. The user selects the weapon classes he wants from the chest. Figure 7.15 displays the player's "swim" ability. It also highlights the ability feature that can be selected from the "Character Window". The following two pictures show the interaction between player character singleton and enemy AI.

In the end, for the moment, besides the PC Windows OS standalone application build, the present iteration of the software will be deployed using UnityâĂŹs WebPlayer plug-in described in previous sections. The Web Player version is integrated in a .html.

Chapter 8

Conclusions

8.1 Achievement Summary

The present project is an engineering project who's main objective is the development of tools and the creation of a small video game. The phases through which it has evolved are a clear testimony and proof of this fact. Furthermore, the benefit of approaching Game Design and Development from a Software Engineering standpoint is the fact that it encourages the creation and production of a better software product.

Quality is one of the crucial factors of the process. Following a standardized approach, a well-thought design and a well-acclaimed pre-established methodology inevitably leads to quality software. This project was build following these principles and the result was a good well-defined design that meets all of the specified requirements. The set of tools along with the scripts and the small video game model are the main objective. The secondary objectives are iterating through the SE stages in achieving the main objective.

As [7], John P. Flynt and Oamr Salem, states, employing software engineering techniques lead to the improvement of the game development process. Object-Oriented Programming and programming languages, like C++, C#, Java or scripting languages like JavaScript and Lua are just tools of the trade or a mean of realizing a goal.

These standardized engineering solutions and techniques have systematized the phases of creating a project. Therefore, based upon these phases, the achievements are going to be grouped into three parts: theoretical and research achievements, design and implementation achievements and testing, evaluation, verification and performance achievements. These achievements combined represent the final and the most important and relevant achievement: the completion of the diploma thesis and the study and research of an up and coming domain and area that has reached the level of an industry.

The theoretical and research analysis achievements are: acquiring the appropriate literature (books, articles, papers, instructional videos, tutorials etc), then, getting accustomed with the vocabulary and concepts, information analysis, selection and extraction. These project-centric information are synthesized and resumed. Based on this, the functional requirements are produced. Non-functional requirements and constraints are also

defined.

Design and implementation achievements: starting from a conceptual architecture establish a final version of the architecture of the game, establish main components involved, the relationship between components, packages and classes, package definition, based on the functional requirements and the use case model provide a domain model with the most important classes involved and object interaction. As far as implementation is concerned, the classes, data types and packages are translated into code. The code was written in C#.

Such a complex project implies testing and validation at every step. Each new class created, class modified, addition of new packages, toolkits, graphical components, plugins, assets, level management change, is vital and it ensures that the game has not lost functionality, been subjected to bugs or errors and that it retains is main functionalities performing exactly like it is required. The actual proof of these achievements is the actual game implementation. The game is functional and has as little bugs and errors as possible, hence the project is functional and it meets the requirements.

From a technology point of view, the main achievements are acquiring the basic knowledge and skills required for using the Unity 3D Game Engine and its capabilities and IDE. Unity's IDE is MonoDevelop.

3ds Max 2011 and Blender 2.5 are graphical modeling, animation tools that are very important for models that are going to be used by the game. These tools are compatible with Unity.

The actual game content and it's achievements: the game mechanism (object interaction, camera controls, input manager, assets and resource manager), artificial intelligence techniques and algorithms (FSMs, decision making and movement strategies), level system and organization (object placement, creation and localization), set o game rules. The main achievement is the integration of these components and tools with Unity engine in the creation of the game model.

From a graphical standpoint: computer graphics and visualization, human-computer interaction and graphical user interface design, the achievements are: the models created, bough or acquired, the textures created and mapped, the materials, particle systems and animations for the models. The end result is a simple and basic running game model implementation built upon the tools that are developed onto Unity's framework and it meets the defined requirements

8.2 Result Analysis

Analyzing the result of the project, it clear that the there are a lot of achievements and objectives that have been fulfilled. Some of them were essential since they targeted the system as a whole, others were added to provide multiple functionality to produce a more complete game model. A very important characteristic of the game is that was created based upon the modularity of the tools and scripts. Of course, Unity's capabilities played

a crucial role in providing the core support for the creation of these tools. Modularity as a very. This results in easier to find out and fix problems, errors and bugs that occur. Furthermore, the problems can be isolated and dealt with. For example, the CSMessengerExtended package, the DayNightCycle, HUD (Heads-Up-Display) system, the Player Character Package along with Camera Controls, the Game Rules System along with Item and Inventory Management, of course the Artificial Intelligence and Mob AI system, most of them are modular components and can be further used. This leads to the following main advantage.

The ability to reuse the tools and scripts is what makes them extremely relevant in the context of software engineering. Of course, the game implementation, along with the graphical models, camera controls, level design, input manager etc. are just a solution. With the support of these tools, various other game ideas, genres can be developed. On the other hand, not only video games can be created. For example they can represent a very powerful platform for e-learning, teaching young children various school subjects in an interactive way through a small game. Other uses can be in the emerging area of augmented reality. Unity's large functionality supports plug-ins and libraries that intersect with augmented and virtual reality.

Of course, from a critical view, there aren't only advantages to a certain solution. One of the main drawbacks is the fact that it is very hard to handle both the graphical models and the actual game design and development. Creating, animating, texturing, rigging a 3D model takes a lot of time and this is where work breakdown must come into place. In the actual industry, there are specialized teams of engineers and artists that handle this job. It is very difficult for one person to handle all this work. That is why, some models were specifically created for this game, others were bought and others were acquired.

Other disadvantage is the fact that the Artificial Intelligence System does not encompass huge amounts of functionality. It could be more refined and fine-tuned for more complete versions of the game. It could involve other techniques, improvements of the present one's to contribute to a more realistic interaction between human avatar and the world and environment. For now, and for the basic version of the game, the present AI works very well and meets the requirements.

For now, the only version of this game is for Windows operating systems on PC. But it can, of course, be ported onto different OS(iOS and Android) and platforms: consoles, mobile devices(mobile phones and tablets).

It is my opinion that the achievements by far surpass the drawbacks and that the end product result is a clear testimony and proof of this fact.

8.3 Further Development and Furture Improvements

Regarding further development and improvements, the present project can be expanded in a wide variety of areas. This solution does not represent a full, industry-specific

video game, but rather a basis for developing such applications with Unity 3D. Like in all engineering cases, this is not the only solution that can be employed for resolving these types of problems. It is however a valid one and it has the main advantage that is software engineering-driven. As the game implementation represents only the skeleton for further improvements and development, it is important to always try to think in matter of quality, efficiency and reuse.

First, the game AI (Artificial intelligence) can be improved so that it contributes to more realism in the human-computer-game system interaction. Various techniques like path-finding, alpha - beta pruning, agent communication, goal evaluation and arbitration, fuzzy logic, team AI, and rule-based systems. For the existing application it meets the requirements and is functional.

Secondly, there can be dedicated and specific graphic models for the game. A wide variety of models from: more player character meshes and NPCs (Non-playable-character) to items, objects and environmental elements, vegetation, architectural elements (houses, bridges, castles etc.), weaponry, particle effects (magic, weather elements etc.).

Thirdly, developing a better and well-thought story and narrative. Story-telling and the narrative aspect of a video game are essential for quality, immersive game-play. Unfortunately, the application at hand does not have a well-defined, super interesting story. The elaboration of the narrative and the story is possible.

Next, more levels can be added, and through these levels, the player can experience different aspects of the game. Good level design means that each level has relevant and interesting content that keeps the player's attention. He needs to be rewarded. The story needs to progress through these levels until it reaches its goal. The present game implementation has the basic levels required for this type of game genre. Other levels can of course be created an linked to existent ones. Unity offers very easy to use integration.

Other improvements refer to the production and development on a large variety of platforms. At the present time, the product is functional on Windows OS as a standalone application and using Unity's Web Player Plugin it can run on other OS but it does not run natively on iOS, MacOS, Windows Phone or Android. In the future, it is possible to create for as many OS as possible. Gaming industry relies mostly on the fact that the games can run on multiple platforms. Further development would mean porting the product on consoles, mobile devices as well as personal computers.

Last but not least, a great aspect of gaming is the multiplayer component. One of the main disadvantages of the current solution is that it does not implement a multiplayer system. However, through Unity this is entirely possible. The game engine offers a paid server for deployment. Of course that there are other free servers where the multiplayer system can be deployed. Multiple players can connect and play together. The present gaming genre is perfect for multiplayer. This is probably one of the most important improvements that can be added to the overall architecture of the system.

As a final thought, this project has made me understand the most important principles of software, starting from the research and domain specific literature, followed by analyzing, designing and implementing my own solution. Research, study and getting

accustomed with the development tools was hard work and took a lot of time, but, in my opinion it was worth it. I have learned how to use Unity's interface and Development Environment, I have learned to use Blender. 3ds Max is not an unknown for me and I think that switching between Blender and 3ds max was actually not hard. Furthermore, I have improved my coding skills and abilities in C# programming.

On the other hand, I have learned how write technical and engineering documentation about the project, using predefined principles, methodologies and phases.

The game development process is hard, time-consuming ongoing process that requires a lot of thought and work. Although there is a wide variety of tools for creating the games, most of the work has to be done from a human perspective; creativity, inventiveness, engineering and programming skills.

This present has contributed immensely to my experience and objective of becoming a computer science engineer.

Bibliography

[1] J. Flynt and O. Salem, *Software Engineering for Game Developers*, ser. Software Engineering Series. Thomson/Course Technology, 2005. [Online]. Available: https://books.google.pl/books?id=92loQgAACAAJ

[2] E. Bethke, *Game Development and Production*, ser. Wordware game developer's library. Wordware Pub., 2003. [Online]. Available: https://books.google.ro/books?id=m5exIODbtqkC

[3] R. Rouse, *Game Design: Theory and Practice, Second Edition*. Jones & Bartlett Learning, 2010. [Online]. Available: https://books.google.ro/books?id=tGePP1Nu_P8C

[4] R. Pedersen, *Game Design Foundations*. Jones & Bartlett Learning, 2009. [Online]. Available: https://books.google.ro/books?id=0flChljb9IIC

[5] A. Rollings and D. Morris, *Game Architecture and Design: A New Edition*. New Riders Games, 2003.

[6] W. Goldstone, *Unity Game Development Essentials*. Packt Publishing, 2009.

[7] R. H. Creighton, *Unity 3D Game Development by Example BeginnerâĂŹs Guide*. Packt Publishing, 2010.

[8] S. Blackman, *Beginning 3D Game Development with Unity: All-in-One, Multi-Platform Game Development*, 1st ed. USA: Apress, 2011.

[9] K. Murdock, *Autodesk 3ds Max 2014 Bible*, ser. Bible. Wiley, 2013. [Online]. Available: https://books.google.ro/books?id=lQLqAQAAQBAJ

[10] J. Harper, *Mastering Autodesk 3ds Max 2013*, ser. Autodesk official training guide. Wiley, 2012. [Online]. Available: https://books.google.com.bo/books?id=mUHi_KJTKGkC

[11] P. Steed, *Modeling a Character in 3DS Max*. Jones & Bartlett Learning, 2010. [Online]. Available: https://books.google.ro/books?id=w_KLe1AylhEC

[12] J. Williamson, *Character Development in Blender 2.5*. Course Technology, 2011. [Online]. Available: https://books.google.ro/books?id=osZvCgAAQBAJ

[13] J. Gregory, *Game Engine Architecture, Second Edition*, 2nd ed. USA: A. K. Peters, Ltd., 2014.

[14] T. Meigs, *Ultimate game design*. New York: McGraw-Hill/Osborne, 2003, includes index. [Online]. Available: http://www.loc.gov/catdir/description/mh051/2004297509.html

[15] A. Champandard, *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*, ser. NRG Series. New Riders, 2003. [Online]. Available: https://books.google.pl/books?id=ZpuR8GnBSGcC

[16] M. Buckland, *Ai Game Programming by Example*. USA: Wordware Publishing Inc., 2004.

[17] I. Millington and J. Funge, *Artificial Intelligence for Games, Second Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009.

[18] M. McShaffry and D. Graham, *Game Coding Complete*, 4th ed. Delmar Learning, 2012.

Appendix A

Relevant code

```
 /** Maps are easy to use in Scala. */
object Maps {
  val colors = Map("red" -> 0xFF0000,
                   "turquoise" -> 0x00FFFF,
                   "black" -> 0x000000,
                   "orange" -> 0xFF8040,
                   "brown" -> 0x804000)
  def main(args: Array[String]) {
    for (name <- args) println(
      colors.get(name) match {
        case Some(code) =>
          name + " has code: " + code
        case None =>
          "Unknown color: " + name
      }
    )
  }
}
```

Appendix B

Other relevant information (demonstrations, etc.)

Appendix C

Published papers