BAŞKENT UNIVERSITY

INSTITUTE OF SCIENCE AND ENGINEERING

A PRELIMINARY STUDY FOR THE DEVELOPMENT OF A REAL-TIME STRATEGY GAME: “GALLIPOLI WARS”

R. EMRE ORÇUN

MASTER THESIS

2011

A PRELIMINARY STUDY FOR THE DEVELOPMENT OF A REAL-TIME STRATEGY GAME: “GALLIPOLI WARS”

“ÇANAKKALE SAVAŞLARI” GERÇEK ZAMANLI STRATEJİ OYUNUNUN GELİŞTİRİLMESİ İÇİN BİR ÖN ÇALIŞMA

R. EMRE ORÇUN

Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Department of Computer Engineering
at Başkent University
2011

Institute of Science and Engineering;

This thesis has been approved in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING** by the committee members.

Chairman (Supervisor)	: Prof. Dr. A. Ziya AKTAŞ
Member	: Assist. Prof. Dr. Tolga CAN
Member	: Assist. Prof. Dr. Mustafa SERT

Approval

This thesis is approved by the committee members on 24/01/2011

…../…../.........

Prof. Dr. Emin AKATA

Director of Institute of Science and Engineering

ACKNOWLEDGEMENTS

The author would like to acknowledge the contributions of the following groups and individuals to the development of this study.

Prof. Dr. Ziya AKTAŞ (supervisor), for his invaluable direction, assistance, and guidance that makes it easier to overcome the difficulties and to reach a conclusion…

Ufuk TUNÇER (software engineer), for his considerable contributions with his comments and suggestions...

ÖZ

“ÇANAKKALE SAVAŞLARI” GERÇEK ZAMANLI STRATEJİ OYUNUNUN GELİŞTİRİLMESİ İÇİN BİR ÖN ÇALIŞMA

R. Emre ORÇUN
Başkent Üniversitesi Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı

Bu çalışmada öncelikle bilgisayar oyunu geliştirme süreç ve teknikleri ele alınmıştır. Oyuncu, hikaye, kurallar, amaçlar, prosedürler, çatışma ve mücadele gibi oyunun tasarım öğeleri ve bunların oyun üzerine etkileri incelenmiştir. Bir oyunu oluşturan, grafik motoru ve grafik süreçleri (sabit işlev süreci ve esnek süreç), fizik motoru ve ilgili teknikler (çarpışma denetimi, ışın atma, vb.), oyun programlama (oyun mekanikleri, yapay zeka, senaryo oluşturma ve yönetme sistemleri), görsel içerikler (sahne, 3B modeller, gölgelendirme algoritmaları için 2B haritalar ve 3B animasyonlar) ve işitsel içerikler gibi bileşenler, geliştirme süreçleri ve yaklaşımları ile birlikte ele alınıp incelenmiştir. Bunlar, bir örnek durum çalışması olarak, Türk tarihinin bir parçası olan Çanakkale Savaşlarından bir kesiti konu alan gerçek zamanlı bir 3B strateji oyununa uygulanmıştır.

ANAHTAR SÖZCÜKLER: oyun geliştirme, oyun tasarımı, gerçek zamanlı strateji oyunu, 3B bilgisayar oyunu.
Danışman: Prof. Dr. A. Ziya AKTAŞ, Başkent Üniversitesi, Bilgisayar Mühendisliği Bölümü.

ABSTRACT

A PRELIMINARY STUDY FOR THE DEVELOPMENT OF A REAL-TIME STRATEGY GAME: "GALLIPOLI WARS"

R. Emre ORÇUN
Başkent University Institute of Science and Engineering
Department of Computer Engineering

In this study, game development processes and methodologies were examined first. Game design elements like players, story, rules, objectives, procedures, conflict and challenge, and their effect over gameplay were examined and described. Technical components including the render engine and rendering pipeline (fixed function pipeline and flexible pipeline), physics engine and physics related techniques (collision detection, ray-casting, etc.), game codes (for game mechanics, artificial intelligence, scenario creation and management) and artwork contents (game level, 3D models, 2D maps for shaders, skeletal animation and audio assets) that form a game were examined along with their development processes and approaches. They are all applied on a case problem, which is a part of Gallipoli Wars in Turkish history, as a real-time 3D strategy game.

KEY WORDS: game development, game design, real-time strategy game, 3D computer game.

Supervisor: Prof. Dr. A. Ziya AKTAŞ, Başkent University, Department of Computer Engineering.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

a	Acceleration
$\vec{F}$	Force
m	Mass
Σ	Summation
τ	Torque
V	Unit vector
2D	Two dimensional
3D	Three dimensional
AI	Artificial intelligence
ANZAC	Australian and New Zealand Army Corps
API	Application Programming Interface
BSP	Binary space partitioning
EMT	Emergency medical technician
FPS	First person shooter
FSM	Finite-state machine
GUI	Graphical user interface
HLA	High Level Architecture
IPR	Intellectual property rights
I/O	input / output
JPL	Jet Propulsion Laboratory
Lerp	Linear interpolation
LOD	Level of detail
MIT	Massachusetts Institute of Technology
MMORPG	Massively multiplayer online role-playing game
NASA	National Aeronautics and Space Administration
NOAA	National Oceanic and Atmospheric Administration
PDA	Personal digital assistant
PR	Public relations
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
QA	Quality assurance
RPG	Role-playing game

RTS	Real-time strategy
SDK	Software development kit
SDL	Scenario definition language
Slerp	Spherical linear interpolation
TnL	Transformation and lighting engine
XML	Extensible Markup Language

1. INTRODUCTION

1.1. What is a Game?

There are many definitions of the term "game". According to Maroney, game can be defined as "a form of play with goals and structure". According to Costikyan, “a game is a form of art in which participants, termed players, make decisions in order to manage resources through game tokens in the pursuit of a goal”. Zimmerman defined "game" as “an activity with some rules engaged in for an outcome” (Crawford, [1]).

According to game designer Crawford [1], “a game is a structured activity, usually undertaken for enjoyment and sometimes used as an educational tool. Games are distinct from work, which is usually carried out for remuneration, and from art, which is more concerned with the expression of ideas. However, the distinction is not clear-cut, and many games are also considered to be work (such as professional players of spectator sports games) or art.”

As noted by Crawford [1], “the key components of games are goals, rules, challenge, and interaction. Games generally involve mental or physical stimulation, and often both. Many games help develop practical skills, serve as a form of exercise, or otherwise perform an educational, simulational, or psychological role. The requirement for player interaction puts activities such as jigsaw puzzles and solitaire games into the category of puzzles rather than games.”

1.2. Computer Games

Computer games or video games are entertaining softwares that involve interaction with a user interface, generates audio-visual feedback and runs on a computer, a game console or an arcade machine.

Computer games have evolved from the early titles such as “Computer Space” or text based games like “Zork” to a wide range of games. Early games, even today’s modern games are commonly referred to as “video games” because in mid 70’s, the term “computer” was not widely used and early game consoles (arcade machines) were video like raster display devices which has had a very limited computational capability.

Computer games are created by one or more game developers, often with cooperation of other specialists (such as game artists, sound designers or musicians) and published by developer independently or through a third party publisher. Then they can be distributed on physical media like BDs (Blu-ray discs), DVDs and CDs, as Internet-downloadable software, or through online game distribution portals such as “Steam”. Computer games also require specific systems configurations in order to play such as a specific game console. PC games similarly require specific hardware capabilities such as a graphics processing unit (GPU) that has certain capabilities, certain amount of memory available for game or an Internet connection for online gaming.

1.3. A Brief History of Computer Games

As noted by Rollings and Morris [2], “the games industry has evolved much as any ecosystem does. The early proliferation of forms (games about making pizzas, running hospitals, dating, politics, and plumbing) has settled down to just a handful of distinct genres. Likewise, advances in technology seem to be converging towards a common point. On the latest generation of consoles, all games are starting to look pretty much identical in terms of graphical quality.”

The developments in computer games may chronologically be outlined as follows (Wolf, [3]):

In 1958, William Higinbotham’s “Tennis for Two” game was shown at Brookhaven National Laboratory in New York. The game demonstrated interactive control with an oscilloscope as the display device.

In 1962, Steve Russell’s “Spacewar!” was written at MIT. It later inspired Nolan Bushnell to develop “Computer Space” in 1971.

In 1966, Ralph Baer started the development of first game console that designed to work with standard televisions.

In 1971, Nolan Bushnell's "Computer Space" became the first coin-operated arcade game.

In 1972, Ralph Baer's "Magnavox Odyssey", the first game console, was released. Atari Inc. was founded by Nolan Bushnell and released its first arcade game named "Pong" which became the first hit game and helped the start of the game industry.

In 1973, the arcade video game industry took off, as many companies began producing computer games, including Chicago Coin, Midway, Ramtek, Taito, Allied Leisure, and Kee Games, which secretly owned by Atari for being able to sell games to different publishers.

In 1974, Kee Games's "Tank" had been the first arcade game to store graphics data on a ROM chip. Midway's "TV Basketball" became the first arcade game to use human figures, instead of blocks or vehicles.

In 1975, Midway's "Gun Fight" became the first arcade game that uses a microprocessor. Atari's "Steeplechase" became the first six-player arcade game, and Kee Games's "Indy 800" became the first eight-player game.

In 1976, General Instruments's "AY-3-8500" chip was released. It had all the circuitry necessary for a game. The Fairchild-Zircon "Channel F", the first cartridge-based game console was released. In the same year Atari's "Night Driver" was the first game that simulates a first-person point of view, though it did not have true 3D graphics. Atari's "Breakout" was released next in the same year (Wolf, [3]).

In 1977, the console game industry suffered its first crash, and many developers quit the industry. Atari's VCS game console (later renamed as 2600) was released. In Japan, Nintendo developed its first console game, “Color TV Game 6”. Kee Games’s arcade game “Super Bug” was released in the same year.

In 1978, Taito’s vertical shooting game “Space Invaders” was released and inspired many game developers. Atari’s arcade game “Football” was released in the same year.

In 1979, Vectorbeam was released the first fighting game named “Warrior”. Atari’s “Asteroids” and “Lunar Lander” were released in that year. Namco’s “Galaxian” became the first game that has color graphics. In the same year, Namco was released one of the most influential games, “Pac-Man”.

In 1980, Pac-Man was released in the U.S. Atari’s “Battlezone” became the first game that has true 3D graphics. “Ultima” became the first console game with 4-directional scrolling. “Star Fire” became the first arcade game to feature a high-score table.

In 1981, Nintendo was released its hit game named “Donkey Kong”. Arcade game industry reached a $5 billion figure in the U.S. (Wolf, [3]).

In 1982, Sega’s arcade game “Zaxxon” became the first arcade game that advertised on TV. Late in the year, another game industry crash, larger than the 1977 crash, began with the drop of income in arcade games (Wolf, [3]).

The crash in arcade game industry affected the console game industry. In 1983, Nintendo released “Famicon” game console in Japan. Atari’s “I, Robot” became the first commercial 3D game with filled polygons and flat shading. Atari’s vector game “Star Wars” was also released in the same year.

In 1984, Nintendo released the “Famicon” console in U.S and Canada. RDI released the “Halcyon”, which was a laserdisc-based game console. Alexey Pajitnov’s puzzle game, “Tetris” was released in USSR.

In 1985, Nintendo released a new version of its game console and named it as “Nintendo Entertainment System (NES)”, in the U.S. Its commercial success helped to bring an end to the industry crash. Nintendo also released one of the hit games named “Super Mario Bros”.

In 1986, “The Legend of Zelda” was released for Nintendo’s game console “Famicon”. Taito’s “Arkanoid” and “Bubble Bobble” was released in arcades. Sega released the “Sega Master System (SMS)” in the same year.

In 1987, “The Manhole” became the first computer game that released on CD-ROM. LucasArts’s “Maniac Mansion” became the first adventure game with point-and-click gameplay. “Shadow Land” became the first 16-bit arcade game. Incentive Software released “Driller”, a console game that has filled polygon 3D graphics. Squaresoft’s “Final Fantasy” was also released in the same year.

In 1988, Williams’ “NARC” became the first game that uses a 32-bit processor. Nintendo released “Super Mario Bros. 2” in the same year.

In 1989, Gottlieb’s “Exterminator” became the first game that uses digitized imagery for backgrounds. Two handheld game consoles were released: Nintendo’s “Game Boy” and Atari’s “Lynx”. The Sega Genesis game console was also released in the same year (Wolf, [3]).

In 1990, Maxis released “SimCity”, the first title of “Sim” games. Nintendo released “Super Mario Bros. 3”. Sega “Game Gear” was released.

In 1991, Nintendo released the “Super Nintendo Entertainment System (SNES)” in the U.S. Capcom released “Street Fighter II” arcade game. Sega released the console game “Sonic the Hedgehog”, the main character of which became Sega’s mascot. The CD-i (compact disc interactive) interactive multimedia player was released by Philips Electronics.

In 1992, Midway released its fighting game, “Mortal Kombat”, for arcades. Virgin Games released “The 7th Guest” which became the best-selling console game.

Sega released a 3d racing game named “Virtua Racing”. Virtuality released “Dactyl Nightmare”, which was an arcade game with a VR (virtual reality) headset and gun interface. In the same year, id Software released one of the most influential 3D games, “Wolfenstein 3D”.

In 1993, Cyan’s “Myst” was released and became the best-selling console game. id Software released its best-selling 3D game named “Doom” which was first title of “Doom” series that still continuing. Sega released a 3D fighting game named “Virtua Fighter”. Atari’s new console Jaguar was released in the same year.

In 1994, Sega “Saturn” and the Sony “PlayStation” were released. The “Computer Game Developers Association” was formed in the same year. Blizzard released its best selling real-time strategy game named “Warcraft”. SNK’s “Neo•Geo” game console released in the same year.

In 1995, The Sony “PlayStation” and Sega “Saturn” game consoles were released in the U.S. and Canada. Blizzard released “Warcraft II” in the same year.

In 1996, The “Nintendo 64” game console was released. Nintendo also released the “Virtual Boy”, which became the first game console that is capable of displaying stereoscopic 3D graphics out of the box. Digipen Institute of Technology at Washington became the first school to offer college degrees in video game development.

In 1997, The “Nintendo 64” game console was released in Europe and Australia. Cyan’s “Riven”, the sequel to “Myst”, was released. Sega released an arcade game with a skateboard interface named “Top Skater”. The MMORPG (Massively Multiplayer Online Role Playing Game) “Ultima Online” began in the same year.

In 1998, Konami released “Dance Dance Revolution” and the first games of its “Beatmania” and “Guitar Freaks” music game series. Sierra Studios released its best-selling game, “Half-Life”. SNK released the “Neo•Geo Pocket” handheld game console and Rockstar Games released its best-selling game, “Grand Theft Auto”.

In 1999, The “Sega Dreamcast” game console was released. The MMORPGs “EverQuest” and “Asheron’s Call” began in the same year. The first Independent Games Festival was held at the Game Developers Conference.

In 2000, Sony released PlayStation 2 game console. Nintendo sold its 100 millionth “Game Boy” console. Maxis released the best-selling PC game in history, “The Sims”.

In 2001, Microsoft’s game console “Xbox” was released. Bungie Studios released “Halo: Combat Evolved” for Xbox in the same year. Sega had left game console industry.

In 2002, The MMORPG “Sims Online” has begun. Sega released “Rez” for the “PlayStation 2”. Microsoft’s “Xbox Live” online gaming service has begun.

In 2003, The MMORPG “Star Wars Galaxies” has begun. Atari released “Enter the Matrix” in the same year of the second installment of the “The Matrix” movie. Cell phone producer Nokia released the “N-Gage” which was a mobile phone and a handheld game console.

In 2004, Nintendo released the “Nintendo DS” (dual screen) handheld game console. Bungie released “Halo 2” for Xbox and Microsoft Windows.

In 2005, Sony released the “PlayStation Portable” handheld game console in the U.S. and Canada. Microsoft released the “Xbox 360” game console.

In 2006, Ralph Baer was awarded the National Medal of Technology by the president of the U.S. for pioneering the innovations of game consoles and computer games. Nintendo released its award-winning game console named “Wii”. Sony released “PlayStation 3” game console in the same year.

In 2007, Blizzard Entertainments' MMORPG "World of Warcraft" has more than 9 million subscribers. The number of subscribers has exceeded 12 million as of October 2010.[1]

As it can be seen from this historical outline, game development has been playing an important role as being the driving force behind many technological achievements. Now, thousands of games, from the browser based mini games to hardcore games developed with a budget of millions of dollars, are being released every year and the life in computer games field still goes on.

1.4. Game Industry

The most important parameters of the computer games industry are quite recent and a good economic performance is expected. Historically, the multi-billion-dollar game industry is cyclical, with an economic peak about every six years or so, keyed to the release of new generation game consoles.[2] The boom in game sales isn't immediate, because it generally requires game developers at least a year to develop titles that take full advantage of the new hardware (Bergeron, [4]).

Bergeron [4] states that, assuming past performance is a predictor of the next boom, the top console makers, which are Sony, Nintendo and Microsoft, have positioned the industry for a boom in 2007 and 2008. Moreover, because the largely unsaturated and growing cell phone game market is likely to have its boom a few years later, it is predicted that the trough following the boom in console games may be higher than in previous years.

The game industry has reached $43 billion dollars in 2007 worldwide including hardware and software sales. In the United States, annual revenues were about $12.5 billion, which is bigger that the United States domestic box office revenues from the film industry (Fullerton et. al., [5]).

[1] http://eu.blizzard.com/en-gb/company/press/pressreleases.html?101007
[2] http://www.economist.com/node/1189352?story_id=1189352

The $12.5 billion figure revenue of the industry includes $5 billion for hardware sales. Game sales alone, which are a more equivalent comparison to box office revenues, bring in about $7.5 billion annually. This number is growing consistently and was expected to be at $10 billion by 2010. While game industry is large in volume and growing consistently, there is still a long way to go before catching the film industry in terms of domestic revenues. It might not take long before the games industry fills the gap, however, because the industry has remained strong, even in the recessions, and has grown steadily over the past decade. Figure 1.1. shows the growth of computer game sales (PC and console) in the United States since 2001 (Fullerton et. al., [5]).

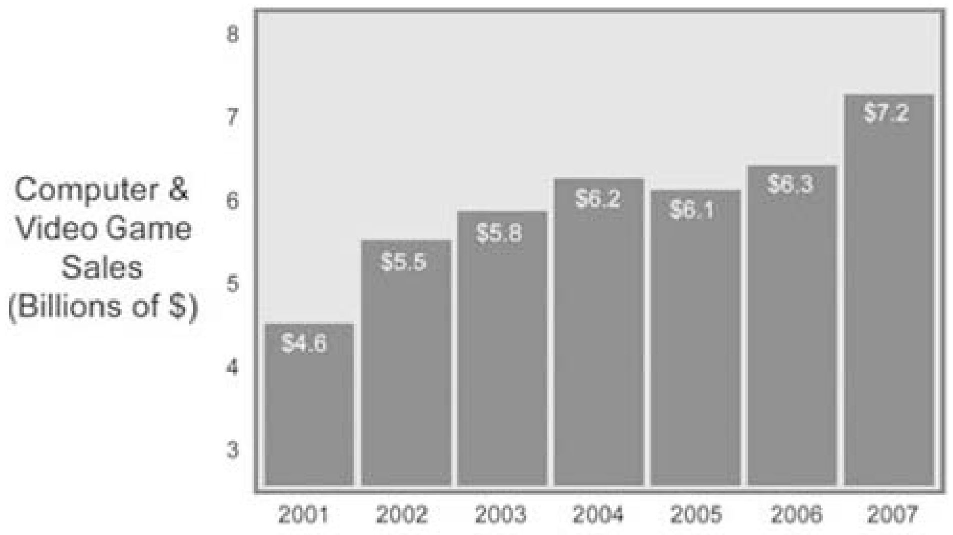


Figure 1.1. Computer game sales

Digital games have become a special form of entertainment since their introduction in the 1970's. Today, more than 60% of American people, which are about 145 million people, play games on a regular basis. The split between men and women players is reducing as well, with women players comprising 40% of all players. Women comprise 44% of the 45–54 demographic, and women actually outnumber men in the 25–34 demographic, mostly due to the recent growth of casual and family games. As generations of players have grown up with computer games, most have continued to play as they grow older: 26% of PC gamers are between 18 and 35 and 40% are over the age of 36. Figure 1.2. shows the majority of players have been playing games for more than 6 years (Fullerton et. al., [5]).

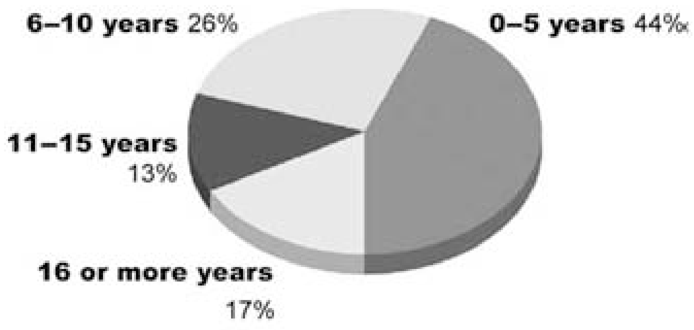


Figure 1.2. How long have gamers been playing games

1.5. Classification of Games

1.5.1. Core games

In general, discussion about computer games in both the press and politics revolves around titles found in the core games classification; historically, consisting of computer games developed for play on personal computers, dedicated game consoles or handheld game consoles. Core games are generally defined by scale of development efforts involved for their production, their intensity or depth of play and can include games across a wide range of genres. For example the “Monster Hunter” series for Wii, the “Doom” series for PC and console or “Mass Effect” for the PS3, all fall within the core games classification. Core games are sometimes considered demanding in their game-play and generally do not appeal to the casual gamer.[3]

1.5.2. Casual games

Casual games derive their name from their ease of accessibility, simple to understand gameplay and quick to grasp rule sets. Additionally, casual games strongly support the ability to jump in and out of play on demand. Casual games as a game genre existed years before the term became popular and include

[3] http://en.wikipedia.org/wiki/Computer_games

computer games such as “Solitaire”, “Hearts” or “Minesweeper” which can commonly be found on many versions of the Microsoft Windows operating system (Bates, [6]).

Gameplay examples of the games within this genre are finding hidden object, matching pairs or filling the gaps, “Tetris” and the “Mahjong” style games. Casual games are generally sold or hired through online distributors like “GameHouse” or provided for free play through web sites like “AddictingGames”.

Casual games are commonly played on PCs, mobile or hand held consoles. They can also be found on many web portals as browser based games or used for online advertisements in game form.

Casual games include adaptations of board games such as chess, backgammon, go, and solitaire. They also include easy-to-play, short-session games on the web, such as hearts and poker.[4]

Television game shows are also falls under this category, with the very popular Jeopardy, Wheel of Fortune, and Who Wants to Be a Millionaire? Players generally can participate these games quickly. They’re already familiar with the rules of the non-digital version of the game and expect to find same rules emulated. These games generally have simple user interfaces, which have little or no learning curve (Bates, [6]).

1.5.3. Serious games

According to Bates, serious games are games that are designed primarily to teach or educate or to gain some sort of ability to the player. Some serious games may even fail to be considered as a video game in the general sense of the term. Also, educational software does not typically falls under this category (e.g., touch typing tutors, language learning, painting etc.) and the primary distinction would appear

[4] http://en.wikipedia.org/wiki/Computer_games

to be based on the title's primary goal as well as target age demographics. Bates [6] also notes that these descriptions are more a guideline than strict rules.

Serious games are games generally developed for reasons beyond entertainment and as with the core and casual games may include works from any given genre, although some such as “exergames” (fitness and exercising games), educational games or simulations may have a higher representation in this group due to their subject matter. These games are generally designed to be played by certain professionals as part of a specific job or for skill set improvement. They can also be created for providing social-political awareness on specific subjects such as environmental issues (Bates, [6]).

One of the longest running serious of games in that category would be “Microsoft Flight Simulator” first published in 1982 under that name. The United States military uses virtual reality based simulations for training their soldiers, as do a growing number of first responder roles (e.g., police, fire fighter, EMT). One example of a non-game environment utilized as a platform for serious game development would be the virtual world of Second Life, which is currently used by several United States governmental departments (e.g., NOAA, NASA, JPL), worldwide universities (e.g., Ohio University, MIT and METU[5]) for educational and remote learning programs and businesses (e.g., IBM, Cisco Systems) for meetings and training.[6]

1.5.4. Educational games

Educational games are those that teach, but they can entertain at the same time. Sometimes called “edutainment”, examples of these educational games are “Oregon Trail” and “Reader Rabbit”. Typically, these educational games targets younger audience than most of the commercial products. Game designers works closely with educationalist and experts of the subject to ensure that the content is appropriate for the target audience (Bates, [6]).

[5] http://ocw.metu.edu.tr/file.php/82/FLE361_Syllabus2010.pdf
[6] http://en.wikipedia.org/wiki/Computer_games

Educational games are games which were developed for adults or older children and which have great potential in learning implications. For the most part, these games can provide simulations of different kinds of human activities and can allow players to explore a wide variety of social, historical or economic processes as well as scientific areas. In short, educational games are the games that are developed with a clear educational intent. In their publicity material, the developers of these games typically focus more on the 'fun' aspects of the games rather than their educational potential. This might be led some misjudges like the absence of educational intent. However, large amounts of information and educational materials might be found within the content or the nature of these games (for example, “Europa Universalis”, “Railroad Tycoon” and “Rails Across America”), suggesting that education was indeed very much in the minds of the developers.[7]

1.6. Game Genres

1.6.1. Adventure games

Adventure games are story-based games that usually rely on puzzle-solving and decision making to move the action along. They can be text-based (such as the early adventures like “Zork” and “Planetfall”) or graphical. They can be told from a first-person perspective, second-person (most text games in which the main character is referred as “you”), or third-person (Bates, [6]).

Generally, the games that fall in this category are not truly real-time games in terms of gameplay, if they are not a kind of hybrid of action and adventure genres. The player typically is allowed to spend as much time as he wants, but nothing happens within the game world until the player triggers an event by performing a certain action like solving the current puzzle.

The early adventure games were parser-based, accepting simple sentence commands from the keyboard. More modern adventures are point-and-click, in which the player indicates what he wants to do by moving the cursor around the

[7] http://en.wikipedia.org/wiki/Educational_software

screen and click on game objects. An active community of hobbyists still developing parser-based text adventures, but they are rarely published commercially. Players generally expect from an adventure game to have a broad and complex game world to explore supported with well constructed characters and an interesting story (Bates, [6]).

1.6.2. Action games

Action games are real-time games in which the player must react quickly to the events. The category is dominated by first person shooter games (FPS) such as “Doom”, “Half Life” and “Halo”.

The action-adventure hybrid, however, is often a third-person game, such as “Gothic”, in which the players can see the main character controlled by themselves from a distant point of view (Bates, [6]).

In action games, players have much more to do than just shoot and kill the opponents. Generally, the games that fall in this category considered far less cerebral than adventure, strategy, or puzzle games. Players are looking for the adrenaline rush of rapidly advancing action that calls for snap judgments and quick reflexes. Opponents can either be computer controlled characters or other human players connected to the game over a local area network or the Internet (Bates, [6]).

1.6.3. Role-playing games (RPG)

In role-playing games, the player generally controls a single character or a small group of characters and tries to accomplish a series of quests. Gameplay revolves around gradually increasing the abilities and strengths of these characters. Classical RPGs include “Ultima Online”, “Might and Magic”, and “Elder Scrolls” series.

Like an adventure game, an RPG offers a huge world to explore with a gradually developing story. Players expect to be able to manage their characters, all the way down to the weapons they carry and the armors for each part of their bodies as well as the disciplines they learn. Combat is an important element, by which the characters gain experience, strength and money to spend for new equipments. Fantasy RPGs (FRPs) also feature complex magical systems, as well as variety of mythological races that make up the player's party (Bates, [6]).

1.6.4. Strategy games

Strategy games require players to accomplish certain goals by managing a limited set of resources effectively. This resource management frequently involves deciding which kinds of units (e.g. military units) to produce and when to put them into action (Bates, [6]).

In the classic "Dune", for example, the player must continually decide which kind of units to produce, as different kinds of units has advantages and disadvantages over each other, and which kind of resources to collect and to allocate.

Older strategy games were typically turn-based as their board game counterparts like chess. The player could take his time as he made each decision, and the computer reacts only when the player indicated he finished his moves. Now, real-time strategy (RTS) games set the computer AI in motion against the player whether the player is ready or not. Multiplayer versions of RTS games substitute human opponents for the AI opponents. (Bates, [6]).

1.6.5. Simulations

Simulations, or "sims", are the games that intent to emulate the real-world conditions of a complicated system varying from the vehicles like jet fighters, helicopters, tanks to incorporated bodies like companies or foundations (Bates, [6]).

The more serious the simulation, the higher the premium that is placed on high fidelity to its real-world counterparts, especially with equipment controls. Players expect to spend hours for learning the mechanics of the machine, and they also expect a user manual to help them with the finer points. Less serious simulations, on the other hand, just let the player to experience a real-world situation without education intent. These are sometimes referred to as arcade simulations. Controls are simplified, players have less to learn in order to play the game, and their mistakes are punished less often (Bates, [6]).

1.6.6. Sports games

Sports games let players virtually participate in their favorite sport, either as an athlete or a coach. Prowess required for success in a real-world sport is not required in its computer-game counterpart. One of the things that players expect from a computer game is the opportunity to do things they cannot do in real life (Bates, [6]).

Sport games must accurately reproduce the rules and strategies of the sport. One gameplay session may cover an individual match, a short series, or an entire season. Sport games may focus on emulating an athlete's actions, on actually playing the game. Alternatively, they may approach the sport from the management point of view to allow the user to be a coach or general manager, sending in plays, making sponsorship agreements or trades (Bates, [6]).

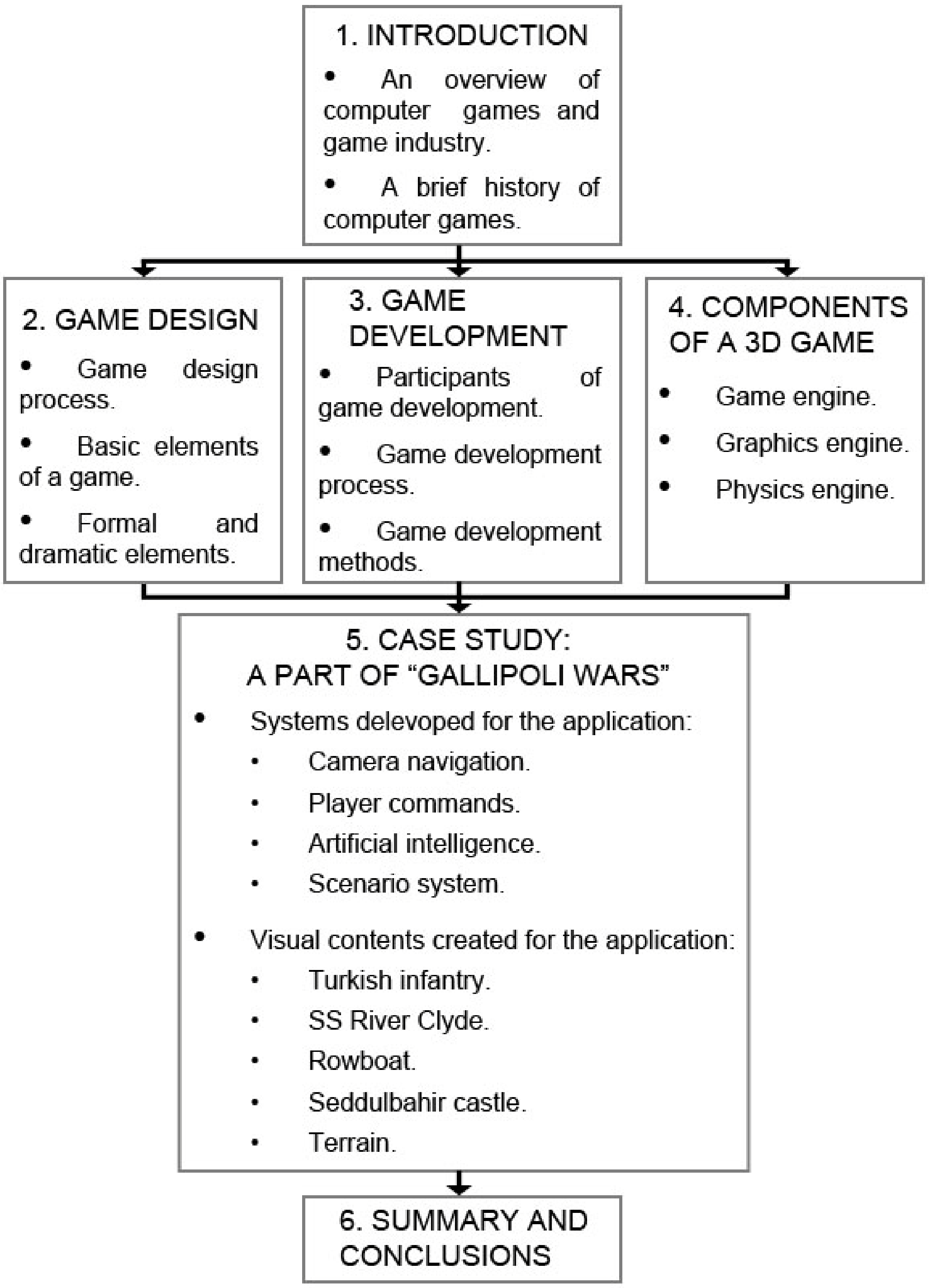
1.7. Organization of the Study

Game development is one of the most special fields that differentiating from overall software development in many ways. The most important reasons of this differentiation can be summarized as excessive use of audiovisual contents that requires an artistic point of view and the entertainment purpose of it, which requires creative approaches to design a good game. As a result, game development is studied in different perspectives.

In this study, it is not intended to focus a specific part of game development. Instead, it is preferred to discuss game development as a whole with its all aspects, which can be broken down into design and development topics, and technical issues. After, providing a general overview in this introduction chapter, all of these aspects are discussed in different chapters, which are numbered from 2 to 4 (Figure 1.3.), as compilations from various sources.

Then, they all were applied on a case problem, which is a part of the development of a 3D real-time strategy game, named “Gallipoli Wars”. The application is covered in chapter 5, which includes the discussion about the development of various systems that are needed to create a real-time strategy gameplay and the creation of artistic contents. Chapter 5 also includes a very short historical overview about the story of the game, which is Gallipoli Wars (or Gallipoli Campaign) that took place at Gallipoli peninsula, between 1914 and 1916, during the First World War.

Summary of the overall study is given in chapter 6 with the discussion about possible extensions that can be done to take forward the application.



1. INTRODUCTION
• An overview of computer games and game industry.
• A brief history of computer games.
2. GAME DESIGN
• Game design process.
• Basic elements of a game.
• Formal and dramatic elements.
3. GAME DEVELOPMENT
• Participants of game development.
• Game development process.
• Game development methods.
4. COMPONENTS OF A 3D GAME
• Game engine.
• Graphics engine.
• Physics engine.
5. CASE STUDY: A PART OF "GALLIPOLI WARS"
• Systems delevoped for the application:
• Camera navigation.
• Player commands.
• Artificial intelligence.
• Scenario system.
• Visual contents created for the application:
• Turkish infantry.
• SS River Clyde.
• Rowboat.
• Seddulbahir castle.
• Terrain.
6. SUMMARY AND CONCLUSIONS

Figure 1.3. Organization of the study

2. GAME DESIGN

2.1. Introduction

Game design is one part of an extensive process that focuses the creation of a meaningful play (Salen and Zimmerman, [7]) and is the process of designing the rules of the game that let a meaningful play to emerge. Game design also covers design of gameplay, environment, and storyline as well as rules of the game (Rollings and Adams, [8]).

According to Rouse [9], the game design is the most important factor that determines the form of the resulting gameplay. The game design determines what actions players will be able to perform in the game environment and what consequences of those actions will occur. The game design determines what win or loss criteria the game will include, how the player will be able to control the game, and what feedbacks the game will provide to player, and it also determines how hard the game will be. In short, the game design determines every detail of how the gameplay will be (Rouse, [9]).

According to Brathwaite and Schreiber [10], "good game design is the process of creating goals that a player feels motivated to reach and rules that a player must follow as he makes making meaningful decisions in pursuit of those goals."

Brathwaite and Schreiber [10] argue that a good game design must be player-centric. Rather than demanding that the player do something via the rules, the gameplay itself should inherently motivate the player. According to Schell [11], games are made for players and players play games because of the experience created by the game designer.

2.2. Game Design Process

Having a good solid process for developing an idea from the initial concept into a playable and satisfying game experience is the key aspect of game design (Fullerton et. al., [5]).

Historically the game design process has been an intuitive and organic part of a relatively unstructured development practice. This worked well in the days when development teams were much smaller and the resources required to develop a computer game were much lower. The rising cost of developing computer games has increased the significance of the game design process to the extent that it's assuming ever greater importance, which led developers to formalize the process (McCarthy et. al., [12]).

One of the most common game design methods is iterative design, which is a play-centric design process. Iterative design is a method in which design decisions are made based on the gameplay experience gained with play testing and prototyping while it is in development (Figure 2.1.) (Fullerton et. al., [5]). In an iterative design approach, a simple version of the game is rapidly prototyped as early in the design process as possible. This prototype is not necessarily needed to have any of the aesthetic finishes of the final game, but must be represent its fundamental rules and core mechanics. It may not be a visual prototype, but must be an interactive one. This prototype is played, tested, edited, and played again to allow the designer or design team to make decisions on the successive iterations or versions of the game. "Iterative design is a cyclic process that alternates between prototyping, play testing, evaluation, and refinement (Salen and Zimmerman, [7])."

In general, a game design process consists of four major steps (Fullerton et. al., [5], Salen and Zimmerman, [7], and Schell [11]):

- Conceptualization
- Prototyping
- Play testing
- Refining

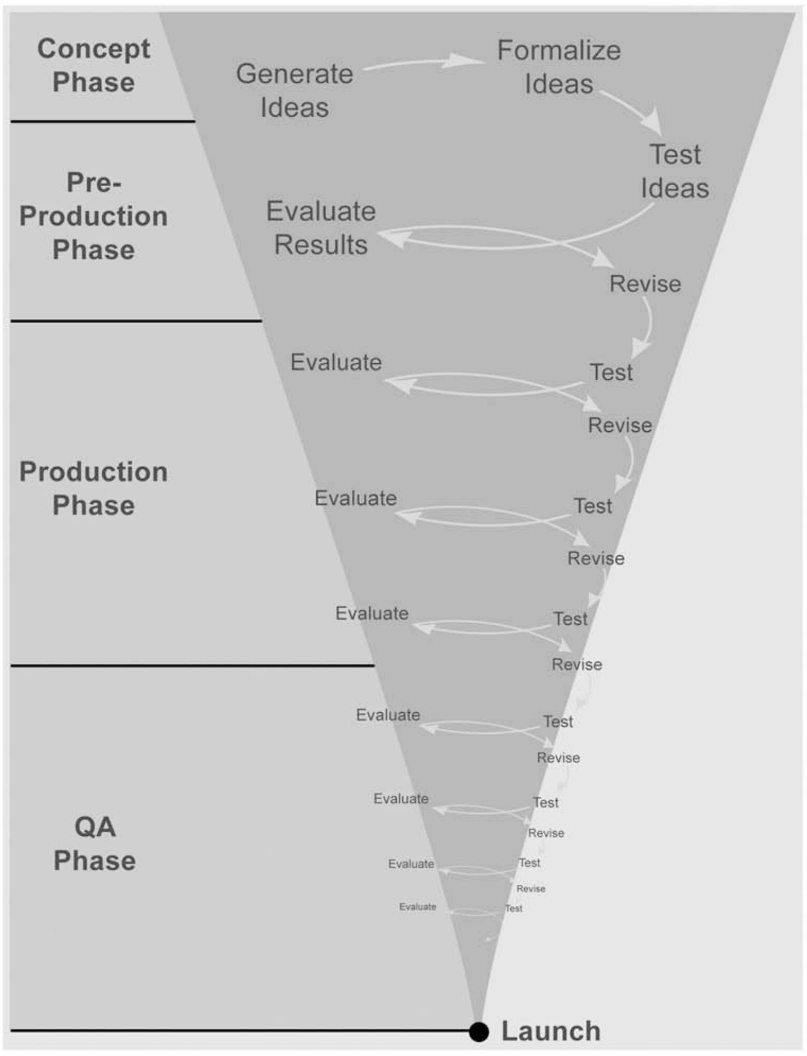


Figure 2.1. Iterative design process

Conceptualization generally considered as the most important part of determining the actual purpose of the entire game development project. Conceptualization step begins with the initial game idea and lasts until basic game elements (game mechanics, story, etc.) are shaped (Schell, [11]).

According to Fullerton et al. [5], “prototyping lies at the heart of good game design.” The word “prototyping” means creating the initial and functioning version of the formal system or a game that, while playable, includes only a rough approximation of the artwork, sound, and features. It can be thought as an immature model whose purpose is allowing to discuss about the game mechanics and to revise how they function.

Play testing can be thought as the design equivalent of debugging (Rouse, [9]) and is a feedback gathering activity that performed by the designers during the design process to gain feedbacks about how players experience the game. There are many ways to perform play testing, some of which are intuitive, informal and qualitative, and some are more structured and quantitative. But there is one thing that all forms of play testing have in common it is the end goal: Gaining useful feedbacks in order to improve the game design further (Fullerton et. al., [5]).

Fullerton, Swain and Hoffman [5] are offering an approach for refining process where the play testing process is broken down into several discreet phases so that in each phase, designers can focus on specific aspects of the design to perfect these aspects, and after then they moves to the next phase and repeat.

As Fullerton, Swain and Hoffman [5] noted, “games are dynamic and interrelated systems.” A change in one part of the system can completely change the players’ perception of another and this fact must be considered during the refining processes. What is important to take away from this process is the need to focus on the distinct goals of each phase and not trying to refine everything in the game all at once. Goal-based phases and a method to move through them is a good way to accomplish that.

2.3. Basic Elements of a Game

According to Schell, there are many ways to break down and classify the many elements that form the design of the game but the most important design elements of a game can be broken down into four categories as shown in Figure 2.2. (Schell, [11]).

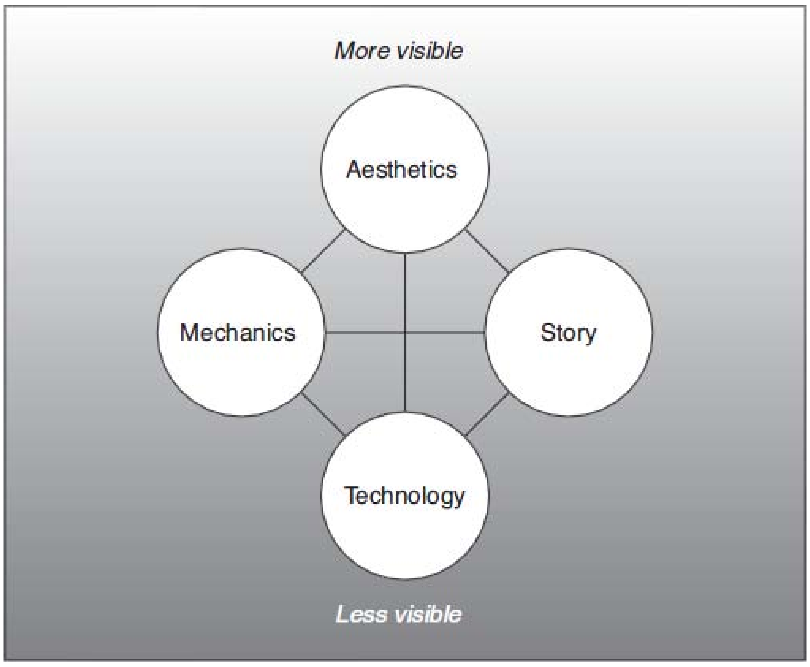


Figure 2.2. The four basic elements of a game

2.3.1. Mechanics

Mechanics are the rules and the procedures of a game. They describe the actual goal of a game, what players can and cannot do to achieve it, and what happens when they do them (Schell, [11]).

Beginning with mechanics or gameplay is one of the most common starting points for designing a game, especially for designer or developer driven projects. Game mechanics or the characteristics of gameplay are often the easiest core to understand, especially if these mechanics are similar to an existing game (such as “a racing game like Need for Speed”, “a flight simulator like Microsoft Flight Simulator”, “3D action-adventure like Tomb Raider”, “a first-person shooter like Doom”, etc.). These mechanics defines what is going to accomplish in terms of gameplay. What type of action game will it be? What aspects of action will be tried to capture for the player? With a more specific idea of what type of gameplay expected, the designer should start thinking about how that will impact the technology the game will require and what sort of story, the game will be able to tell (Rouse, [9]).

If we compare games with more linear entertainment experiences (e.g. books, movies, etc.), we saw that the linear experiences also involve technology, story, and aesthetics, but they do not involve mechanics, because of that reason, the mechanics can be considered as the thing that make a game a game. When choosing a set of mechanics, it is needed to choose the technology that can allow them, aesthetics that emphasize them clearly to players, and a story that supports game mechanics (Schell, [11]).

Depending on what type of mechanics that are aimed to be created for the player, it is needed to analyze which technologies that will require. Does the game require a 3D engine, or will a 2D engine be enough or even more appropriate? What sort of point of views will the player have in the game world? Will they be static or dynamic? Does the gameplay require the rendering of a large number of game objects moving around on the screen at once? Are the game worlds consisting of huge outdoor scenes or small indoor areas? All of these questions and many more need to be answered to understand what features that the game engine must have in order to support the game mechanics. The technology chosen to license or develop for the mechanics must also be able to run on the target platform, whether it is PC, console, or mobile. Technological feasibility analysis may end up limiting the scope of the gameplay and features (Rouse, [9]).

Game mechanics and the type of the game also limits what type of story can be told. An RPG can tell a much more complex and long story than an action game, or an action game can tell a more abstract story. Certain types of stories just not fit with certain types of gameplay. For example a detective story might not fit in a strategy game. Similarly, a romantic story or a story about diplomacy possibly not fit in a fast action game. After the decisions about the mechanics of the game made, it is needed to decide what sort of story is best suited to that gameplay (Rouse, [9]).

2.3.2. Story

Story can be defined as the sequence of events that occurs in a game. It can be branching and emergent as well as linear and pre-scripted as in the movies (Schell, [11]).

It is certainly possible that a game design may start with brainstorming about a story to tell, a setting to employ, or a set of characters to explore. Story may probably a less common starting point than technology or mechanics. Since there are many games that have no story, starting to design a game with thinking about a story first may seem unnatural. At the same time, a particular setting may inspire a game designer, such as a post-apocalyptic world or the “neo-noir” world of Frank Miller’s “Sin City” (Rouse, [9]).

The story to be told has a dramatic effect on the type of game the project will need to have. For example when the designer wants to tell the adventures of a group of friends in a fantastic world full of hostile creatures, a first-person shooter with cooperative multiplayer gameplay might be appropriate. Any sort of story that involves the player talking to variety of characters and taking “quests” from them might easily be addressed in a RPG style game. The story of the Second World War could be perfectly addressed in a strategy game. Bringing out the characteristic of Second World War can be the primary concern of the designer. Will the players have a general’s point of view? In that case, a gameplay that allow the tracking of tactics and logistics can be used. Or a story from the soldiers’ point of view can be preferred. Then mechanics that would allow the player to command the soldiers unit by unit can be appropriate. If character dialogues are playing an important part of a story, game mechanics that allow player to type sentences or select dialog choices can be preferred. The designer needs to decide what gameplay can allow the player to experience the story (Rouse, [9]).

When a story available to tell through a game, it is needed to choose mechanics that will both strengthen that story and let that story emerge. It is also necessary to choose aesthetics that help reinforce the ideas of the story, and technology that is best suited to the particular story that will come out of the game (Schell, [11]).

According to Rouse [9], the technology is limited by what the team is capable of accomplishing within the given time, but the story is limited only by the ability of the game designers to tell it.

2.3.3. Technology

When referring to “technology”, it does not mean “high technology” exclusively, but to any tools, materials and interactions that make a game possible such as paper and pencil, toys, or laser emitting devices. The technology used in a game enables it to do certain things as well as limits it from doing other things. The technology is the medium in which the mechanics will occur and aesthetics take place, and through which the story will be told (Schell, [11]).

Starting a game project that a large portion of the technology already developed is a common case. If the project is not the first project of the development team, then it is possibly that there will be an existing technology developed for previous titles. Even when a new engine will be used for the project, this generally means the use of an updated version of an older engine. As a result, the style of game best suited to the engine does not change significantly. Even when a game engine is being written from scratch for the project, it is likely that the development team is best equipped to create a certain type of engine, that may support indoor areas or outdoor terrains, 3D or 2D rendering, a complex physics simulation for procedural animations or something simpler. The developers may be interested in implementing certain lighting or rendering techniques, and may create an engine to accomplish these achievements. The designer is then tasked with designing a game that can highlight the sophisticated technology that is developed (Rouse, [9]).

Starting the game design with a certain technology requires that the game designer consider the type of game that will suit best to that technology like as starting with a desired gameplay dictates what type of technology is required. If the given technology is a 3D game engine, the designer will need to design a 3D game that takes place in 3D environment. If the engine has a sophisticated physics simulation, a game that uses the physics for puzzles and player movements may be designed. For such situations, the designer needs to consider how to use owned technology well, for it is surely going to be easier than modifying the engine (Rouse, [9]).

A technology-driven game is a game that is generally designed to show off a technological achievement like a new rendering technique implemented successfully. The “Quake” released in 1996, was a technology-driven game. There was not much game there, but there were technological achievements (such as the use of 3D game characters instead of pre-rendered sprites) which helped to sell the game engine to other game developers. The achievement can be about hardware instead of software. Console manufacturers often develop technology-driven games to introduce the features of their new hardware (Rollings and Adams, [8]).

The technology also limits the story can be told. Without a sophisticated parser, it is difficult to tell a story that depends communicating with game characters by typing. Without an engine that can handle outdoor terrains effectively, it is difficult to make a game about mountain climbing. Without good artificial intelligence and an optimized rendering engine that can handle large numbers of units on the screen at once, it is hard to make a game about massive battles between armies. As is the case in the application of the thesis, the game designer needs to consider how the story line will be communicated to the player through the engine that has licensed (Rouse, [9]).

2.3.4. Aesthetics

Aesthetics is how a game looks, sounds and feels. Aesthetics are an incredibly important aspect of game design because they are much more visible to players and more effective than other aspects in player’s experience. When there is a certain style or look that wanted players to experience, it is needed to use a technology that will not only allow the aesthetics become enable, but enrich them further. It is also necessary to design the mechanics that make players feel like they are in the world defined with style and look. A story that allows the aesthetics emerge at the right time and have the most impact may also necessary (Schell, [11]).

Art-driven games are not common. An art-driven game exists to show off an artist’s works, like as a technology-driven game exists to introduce a technological achievement. Art-driven games may be visually impressive, but they are possibly not that good in terms of gameplay because the designer has spent more time thinking about to show off his artwork than players’ experience. “Myst” can be considered as an exception of that. It is an art-driven game with strong gameplay and a well constructed story (Rollings and Adams, [8]).

2.4. Formal and Dramatic Elements of a Game

Formal elements are the major elements that define the structure of a game. Without them, games cannot be games. A game without players to play, without objectives to accomplish, without rules to obey, or procedures to follow, is not a game at all. Players, objectives, procedures, rules, conflict, challenge and story are the essence of games and a strong understanding of their effects and relations is the key for a good game design (Fullerton et. al., [5]).

2.4.1. Players

As previously noted, games are experiences designed for players, and they have no reason to exist without players. It is not as obvious how to structure the participation of the players to a game. This is one of the key questions that must be answered by designer. For instance, how many players will play the game? How many players will the game support? Will players have different roles? Will they play to beat each other or will they play cooperatively, or both? The answer of these questions will determine the basic aspects of the game (Fullerton et. al., [5]).

Generally games have identical roles for all players as in chess, backgammon or Monopoly. But games may have more than one role for players. In Mastermind, one player plays the role of code-maker, while the other plays as code-breaker.

Also, many team games, like football, have different roles for players (Fullerton et. al., [5]).

The role of the player also helps the player to understand what rules playing with and what he is trying to achieve. In Sierra Online's Police Quest series, for example, the player plays the role of a police officer. Real police officers cannot just shoot innocents. They have strict rules and regulations about in which situations they can use their guns. Tactical combat simulations in which players plays the role of a Special Forces soldier such as Rainbow Six and Counter-Strike also implement similar rules. When a player clearly knows what role he is playing, he can understand or predict the rules that he must obey or objectives he must try to accomplish (Rollings and Adams, [8]).

As previously noted, there are different roles for players in football, but these roles are well understood by the audience and the ones who want to play the game. But it is important to clarify the roles when a game takes place in a less familiar world. If the player's role is changing from time to time, it is essential that the player need to know why it changed in order to adapt quickly to the new circumstances (Rollings and Adams, [8]).

2.4.2. Rules

The rules are the most fundamental game mechanics. They define the space, the actions can be done, the consequences of these actions, the constraints on the actions, and the goals. They make possible all the mechanics and add objectives; which is one of the things that make a game a game (Schell, [11]).

The rules also define the challenges and obstacles that the players must try overcome. The challenges and obstacles, together with the actions that players can do to overcome them, forms the gameplay (Rollings and Adams, [8]).

Some of the questions might be asked about rules are: How can players learn the rules? How are the rules enforced? What rules suits in what situations? Rules are generally explained in the user manuals of board games. In digital games, they may be explained in the manual, or they may be implicit within the game itself. For example, a computer game may forbid actions without explicitly stating that fact. The game may stop the player from performing an action when it is attempted or the interface may not provide controls for that action (Fullerton et. al., [5]).

Rules have the following qualities (Salen and Zimmerman, [7]):

- They limit player actions.
- They are explicit and clear.
- They are shared by all players.
- They are fixed.
- They are obligatory.
- They are repeatable

Many of these qualities seem to be applicable for all softwares. For example, a computer code is explicit and clear. It is also quite precise, shared and repeatable: all players who buy a game in a store get the same code. Players certainly do not share same game experience, but from a formal point of view, the rules they follow are the same (Salen and Zimmerman, [7]).

The idea that rules are obligatory and fixed is true for non-digital games as well as the digital ones. The rules of a game do not change during a game session. There may be hacking, cheating attempts in games, but this does not change the fact that the rules of computer games are fixed and obligatory (Salen and Zimmerman, [7]).

It is important to think the rules in relation to players when designing them. Too many rules may make a game hard to understand and play. Too few rules may make it so simple and boring. Likely poorly defined rules may mislead players. Even if the game itself is tracking the proper application of rules, a player needs to clearly understand the rules or he may feel cheated by the consequences of certain rules (Fullerton et. al., [5]).

The first quality of rules, which is limiting player actions, is very useful. Rules restrict and stylize the actions of the players. The rules of a computer game are directly related to manipulating player behavior. How does the player take action? How does the game respond? Rules also enable the choice making. (Salen and Zimmerman, [7]).

2.4.3. Objectives

Objectives define what players will try to achieve within the rule set of the game. Objectives must be challenging, but achievable. The objectives of a game can determine the gameplay, in addition to providing challenge. A game in which the objective is to kill the opponent’s forces will have a very different gameplay from a game in which the main objective is spelling more or longer words (Fullerton et. al., [5]).

In some games different players have different objectives and some games allow players to select desired objectives from several alternatives. There may also be sub objectives that help players to achieve the main objective. In either case, the objectives must be considered carefully because it affects not only the formal elements of the game design but also the dramatic elements like the story. If the objective is well integrated into the story, the game can take on strong dramatic aspects (Fullerton et. al., [5]).

2.4.4. Procedures

Procedures are the methods of play and the actions players can follow when they are trying to achieve the game objectives. There are several types of procedures that are common in many games (Fullerton et. al., [5]):

- *Starting Action:* How to put a game into play?
- *Progression of Action:* Ongoing procedures after the starting action.
- *Special Actions:* Available conditional to other elements or game state.
- *Resolving Actions:* Bring gameplay to a close.

Both in board games and in computer games, procedures are explained in the user manuals. In computer games, procedures can be distinguished from rules much more easily because they simply are the things that explain the ways of using the input devices.

Some examples of procedures from “Super Mario Bros” game are as follows (Fullerton et. al., [5]):

- Select Button: Use this button to select the character you wish to play with.
- Start Button: Press this button to start the game. If you press it during play it will pause the game.
- Left Arrow: Walk to the left. Push button B at the same time to run.
- Right Arrow: Walk to the right. Push button B at the same time to run.

2.4.5. Conflict

Conflict occurs when the players try to achieve the objectives of the game within its rules. As mentioned previously, conflict is designed into the game by forming rules and procedures that prevents player accomplish their goals easily. Instead, the procedures offer inefficient ways toward achieving the game objective. While inefficient, these ways can challenge the players by forcing them to employ certain skills. The procedures can also offer a sense of competition, so that players may submit themselves to this inefficient system in order to gain the sense of achievement that comes from participating. Some examples of things that cause conflicts are as follows (Fullerton et. al., [5]):

- *Pinball*: Keep the ball within the field of play using only the left and right flippers.
- *Golf*: Get the ball into the hole in as few strokes as possible.
- *Monopoly*: Manage your properties and collect money from other players to become the richest player.
- *Quake*: Stay alive and beat the opponents.
- *WarCraft III*: Command your forces to beat your opponents and accomplish map objectives.
- *Poker*: Win the opponents by making good choices and bluff.

Conflict can be emerging in varying degrees of intensity. It can be considered as in deferent resolutions that certain amount of conflict required by a process. The conflict can be in high amounts into a short time, in that case there is an intense conflict, or it can be distributed over a long period of time, in that case there is a low intense. This distinction directly related to the design of the game. Intense conflict is more suitable for a short game while a long game require low intensity. Designers may easily fail to follow this simple rule and they may create games that have long-duration intensity in conflict. The monsters just keep coming without giving a break and the player is exhausted when the game finally ends. Less experienced players may enjoy in such intensity, but it is a bad idea to design a game that way. A good game design relies on more indirect, less intense forms of conflict (Crawford, [1]).

The three main sources of conflict in games are as follows (Fullerton et. al., [5]):

- Obstacles
- Opponents
- Dilemmas

Obstacles are one of the most common sources of conflict in both single and multiplayer games, though they are more important for single-player games. Obstacles may involve mental skills, such as the puzzles in an adventure game or may take a physical form, such as in the ice hockey match and billiards (Fullerton et. al., [5]).

In multiplayer games, the opponents are the primary source of conflict (Fullerton et. al., [5]). In “Counter Strike” there are two teams of human players that compete against each other. Also, in chess, conflict comes from playing against an opponent. In single-player games, conflict comes from opponents can be created via artificial intelligence.

Another source of game conflict is the choices that cause dilemma. An example of a dilemma in Monopoly is the choice of where to invest money, to buy a new property or to upgrade a property that is already owned. Another example of dilemma in poker would be whether to stay in or draw away. In both cases, players have to make choices that have positive or negative consequences. Dilemma is one of the important sources of conflict (Fullerton et. al., [5]).

2.4.6. Challenge

Dramatic elements, like challenge or story, surround the more abstract formal elements of the game design. They create sense of involvement for the players and enrich their overall game experience (Fullerton et. al., [5]).

Most of the players play games because of the challenge they offer. Games can amuse players over the play time and differently each time they play, while employing their minds in a different way than a book, movie, or other forms of entertainment. Games can direct players to think actively, to try out different things to solve problems and to understand a given game system (Rouse, [9]).

According to Rouse [9], people learn something when they face a challenge and then overcome it. It does not matter if that challenge comes from a math problem or a computer game. Challenging games can be educational experiences. Players can learn from games, even when learning is limited only with its context, such as how to navigate through a jungle, survive from a severe battle, or convince someone about a topic. In the best games, players learn things through gameplay that can be applied to real life, even if they do not realize it. This may mean that they can use their problem solving methods at work, use their improved spatial skills to rearrange their furniture in a better way, or even develop empathy skills through role-playing. Many players improve their abilities through the challenges provided.

All challenges take place in some kind of a defined context that sets the conditions under which the challenge is encountered. Sometimes that context is about a financial situation, in which case the conditions are defined by a contract. Sometimes the challenge is caused from a job in which a variety of factors defines what is allowed, expected, and required. Some challenges are physical in nature, in which case the nature of the challenge is defined by laws of physics (Crawford, [1]).

In some sense, challenge can be considered one of the key elements for an entertaining gameplay, since every game, at its heart, has a problem to be solved (Schell, [11]).

According to Fullerton, Swain and Hoffman [5], most people would agree that one thing that enjoys playing a game because of challenge and they do not simply imply that they want to be faced with objectives that are hard to accomplish. When players talk about challenge in games, they are actually talking about objectives that are satisfying to complete that require the right amount of effort to create a sense of accomplishment.

The psychologist M. Czikszentmihalyi set out to identify the elements of enjoyment by studying similarities and dissimilarities of experience across many different tasks and types of people. Based on his findings, he created a theory known as “flow,” which is illustrated in Figure 2.3.a. As can be seen in the diagram, there's a dynamic relationship between challenge and skills, frustration and boredom that creates an optimal experience for a person engaged in an activity (Fullerton et. al., [5]).

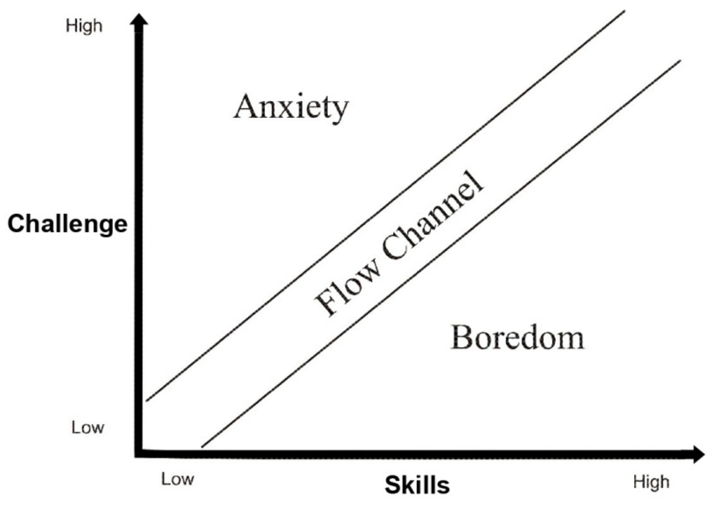


Figure 2.3.a. Flow: The relationship of challenge and skill levels

The vertical axis represents the degree of challenge an activity offers. The horizontal one represents the skills of a participant (Salen and Zimmerman, [7]).

Both factors range from low to high values, and as a players move to different positions on the chart, they navigating through different experiences of the activity. The narrow diagonal strip represents a potential flow state where challenge offered by the activity is in balance with skills of players. The outsides of this strip are the state of anxiety, in which the challenge exceeds the skills of players and the state of boredom, in which the skills of players outstrip the challenge offered (Salen and Zimmerman, [7]).

Csikszentmihalyi uses the example of someone learning to play Tennis (Figure 2.3.b.). When a Tennis player begins to study, she is likely at position 1 on the chart, possessing low skills, but also facing challenges appropriate to her abilities, meaning that she may enjoy the match. As she proceeds, however, she can move out of the flow. If her tennis skills exceed the challenge of her lessons, the result is an experience that cannot truly engage her, and she finds herself in position 2 (boredom). On the other hand, if the sense of challenge she feels from tennis is overwhelming, the result is a negative and frustrating experience, position 3 (anxiety). The tennis player can arrive at position 4 and regain the flow state only after finding a new balance between her skills and challenge (Salen and Zimmerman, [7]).

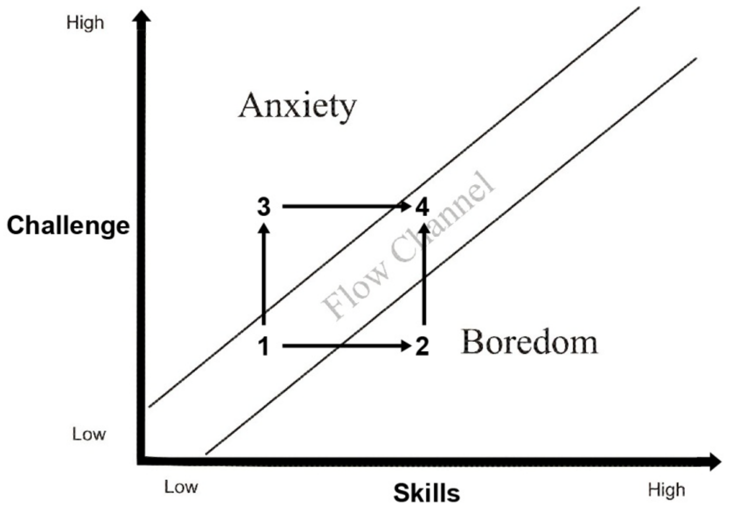


Figure 2.3.b. Learning to play tennis example

3. GAME DEVELOPMENT

3.1. Introduction

According to Manninen et. al. [13], game development can be considered as a specific form of software development where certain product and/or service is designed and developed. The outcome of the development (computer game, comprises of assets, audiovisual contents) which generally exist only in electronic format. Due to the heterogeneous nature of game assets, the development requires a team of skilled individuals from different disciplines working in collaboration.

3.2. Developer – Publisher – Consumer Model

The game design and development process is a tiring effort to combine multiple disciplines that has different characteristics. Today, the trend has been towards more specialization, whereby each business unit becomes considered as separate entities. Figure 3.1. illustrates the developer – publisher – consumer model (Grassioulet, [14]).

Developing computer games is a complex and expensive process. A developer's goal is to produce the highest quality game within the limits of the resources and time. The publisher's goal is to release a best-selling game while minimizing their risk by keeping costs low. There is a common interest in producing a successful product between these two parties, and also a conflict about how much money and time must be spent (Fullerton et. al., [5]).

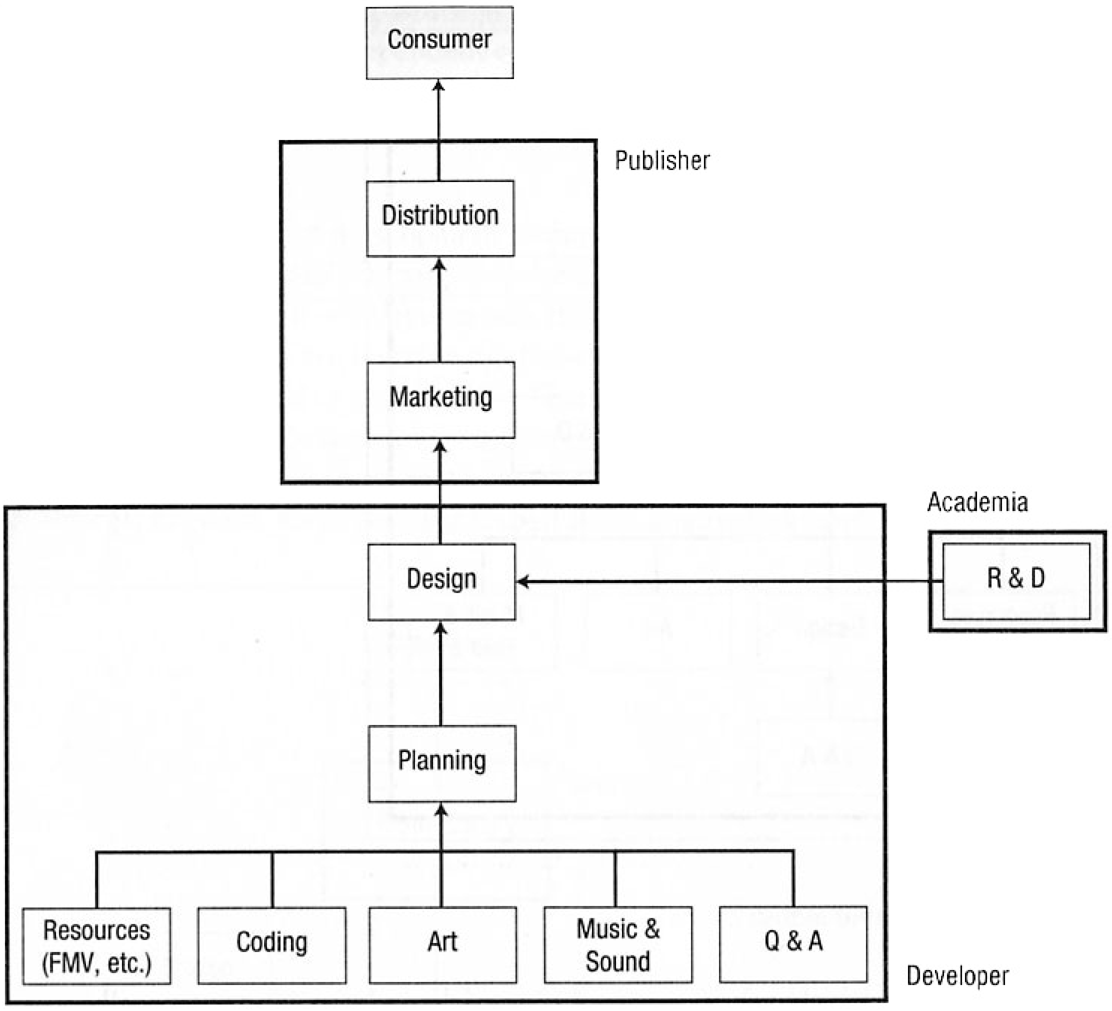


Figure 3.1. Developer-Publisher-Consumer Model

3.2.1. Developer

Game development is an area of work, which requires a high-level of motivation, commitment, and visionary innovation. While the work processes themselves do not differ greatly from those of software industry, the nature of the game development seems to especially attract people who enjoy playing games. Many game developers genuinely love their work. Game development could be, and have been, referred to that of film industry in terms of creativity and artistic components (Schell, [11]).

The game development can be considered as a risky and competitive profession. New developers (e.g., game studios) will be evaluated based on their game concept, abilities of their development team, competence of their production processes, capabilities of budget management and quality assurance. Therefore, production system needs to be perfect as whole. The most important thing for a game developer is having a substantial and strong set of reference productions, like the track record of previous titles developed by the team (Manninen et. al., [13]).

Game development can be broken down into the tasks of planning, design, programming, creation of audiovisual contents and testing. Historically these tasks have been done by small development teams and even by a single person. In the early days of the game development, the artistic creativeness and authorship were the only key aspects for success. Nowadays the development efforts have increased into a level where it is more suitable to consider it as a production factory. Generally a game studio designs and develops game in return for royalties or advances given by the publisher. Studios work for the publisher because generally they do not have the budget, distribution channels and marketing resources required to reach their game to a large audience. Therefore the developers are dependent to publishers. There may no stable income and if the project is canceled before release, the developer can be faced with being out of assets. The publisher also takes the intellectual property rights (IPR) most of the time. However, developer’s future may be dependent to hold on IPRs (Manninen et. al., [13]).

Game developers rarely sell their products to end-users directly. However, the developers usually involve players as focus groups interviewees and beta testers (Manninen et. al., [13]).

3.2.2. Publisher

The primary role of the publisher is funding the developers in order to make profit by publishing their games to the market. Publishers also take responsibility in manufacturing as well as marketing, PR, distribution, and customer support. Major parts of publishers are originally old developers. Also, bigger publishers own developers or have development teams in-house (Schell, [11]).

As noted previously, publishers act as a source for funding that can be in the forms of royalty advances and other development funds. With this type of funding model, the publisher undertakes the major part of the risk. In return, they also tend to take majority of the profits. The games business has ended a situation where leading publishers acquiring individual developers to become massive global enterprises that cover many areas of games industry and reaching a high market share (e.g. Electronic Arts). Publishers generally consider game development as part of their profit making activities. If a developer cannot deliver the game, they contract another one (Manninen et. al., [13]).

The duties and responsibilities of the publisher are generally specified in the publishing contract signed between the developer and the publisher. A commonly used negotiation tool is prototype or demo of a game, especially if the track record of the developer is not available.

While most game developers especially the small or new ones are at the mercy of the publisher. According to Maninnen et. al. [13]; some of the big developers are often aggressive publishers that surpass pure publishers with their status of being independence. This status, however, is not easy to reach.

3.2.3. Consumer

The role of the consumer in games industry is very different than in overall software business. The players are consuming games because of different reasons other than utilitarian purposes. Games business is considered as a part of the entertainment industry just like the film and music industries (Manninen et. al., [13]).

The traditional approaches based on supply and demand do not represent well the games industry. The consumers typically buy games based on their personal preferences. There may also be a very observable demand for a certain game, but marketing generally plays an important role in the formation of such demand. Even high quality and engaging games can stay unnoticed when there is not enough marketing effort. Game reviews, demo distributions, media publicity, hype and word of mouth advertising raise the attention of the consumers (Manninen et. al., [13]).

According to Manninen et. al. [13], individual game developers, especially the so called respectable developers, may provide some added-value to the consumer. However, as current trends indicate, the game itself or the previous versions of the game are the most affective to attract players. Whereas, the publishers preferring the games that have minimum risk and maximum profit. The consumers' purchasing behaviors also look similar. Unique and off-mainstream titles started to reach their audience lately, but these situations are still in minority.

The latest trends in games industry involve casual games and serious games as well as MMOs (massively multiplayer online games). The consumers of casual games have created different target audience and business model for publishers. While small games with minimized costs can attract consumers, it is very difficult to make profit from casual games since casual gamers are generally beware of spending money to games. Furthermore, the serious games sector seems to be the closest one to the overall software business even while its audience is slightly different. This trend indicates the increasing potential for innovative game developers whom are outside the mainstream game development (Manninen et. al., [13]).

3.3. Game Development Process

Currently, there is no single model for game development process that can act as a standard for the industry. Game developers follow their own adapted and modified processes. Manninen et. al. [13] argue that the average resources spent for the development of a game are generally between 2 to 10 million Euros of budget for 30-60 member development team working for up to 24 months. There may be huge variations in the figures. For example, the development costs of a simple mobile game are much lower than those of a game console.

The characteristics of a game development have many similarities with overall software development and film industry.

There is a logically reasonable trend towards considering game development more like the software industry, however, as Manninen et. al. [13] have noted, this cause some problems because software production processes tend to be:

- goal-oriented,
- cost effective,
- efficient in time management,
- heavily using production breakdown in to reusable components.

The conflict between the progressive nature of the game development and goal-oriented effective process may produce creative innovations, but generally suffers from unstable production management as well as including risky investments from the publishers' point of view (Manninen et. al., [13]).

The core aspect of the game development process is that a project is handled and approved in stages that are defined by milestones. Agreements between developers and publishers are generally done based on these milestones, and the developer is responsible to pay a predetermined amount when milestones are reached (Fullerton et. al., [5]).

The development process of the computer games can be broken down into six major stages (Figure 3.2.) (Manninen et. al., [13]):

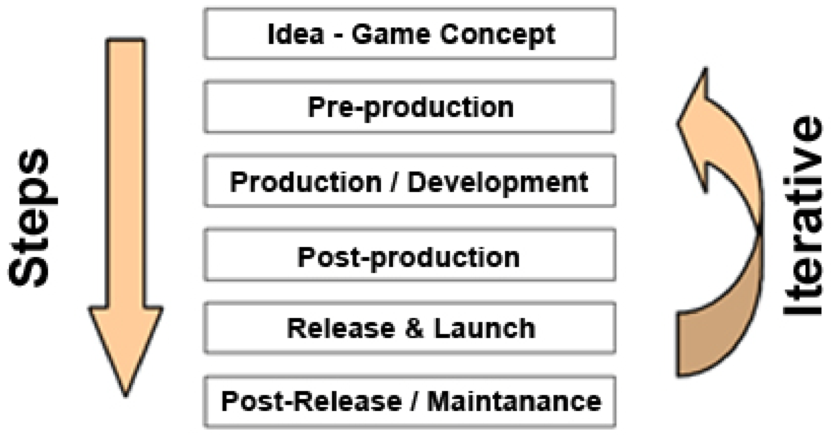


Figure 3.2. Major stages of game development

1- Idea - Game Concept Phase (Specification and planning): Consists of the formation of key features of the game including genre, target audience, concept documents, platforms, references, and the initial draft of the development plan. The concept design, from idea conception to pitching, is considered as the most creative part of entire process.

2- Pre-production Phase: Includes the game and level design issues like the story and scenario, game mechanics, aesthetics, development principles, object-oriented specifications, and the design of the game world (a city or a race circuit). Both game and level design are made for building a flexible prototype, allowing evaluations for a more realistic cost determinations and planning. The main idea of preproduction stage is to plan, to prototype, testing and evaluating everything possible before the actual development.

3- Production Phase / Development: Includes the development of all the different elements that the game intended to include like codes, models, sounds, videos, etc. and their integration to the whole game.

4- Post Production Phase (Validation and testing): Includes the validation through testing. This phase also consists of a QA (quality assurance) test that focuses on mechanics, gameplay, user interface, qualities of audiovisual contents, and ability to meet the market requirements. Only after the validation, developers get to the debugging process.

5- Release & Launch Phase: Includes delivery of the release to manufacturer.

6- Post-Release / Maintenance: Includes the development of patches and upgrades.

In addition to the development stages model, the production process generally involves heavily iterative tasks that generally cycle through pre-production, production and post production phases and may also cause the boundaries between stages to become less visible (Manninen et. al., [13]).

3.4. Game Development Methods

As previously mentioned, the game development has taken a slightly different route than the overall software development. Game development generally starts on small computers instead software development that mostly starts on large mainframes. One of the reasons of this difference was the need for game developers working around the limitations of the system. This difference is now, mostly, converged. This means that, both computer games and business softwares are written in similar ways. Both are now generally coded in the same languages like C++ or Java and either uses the application programming interface of the target operating system or some middleware and third-party libraries, or both (Rollings and Morris, [2]).

The main difference between the development of computer games and the softwares written for business is that the game development usually requires a correspondingly large amount of supporting material in the form of audiovisual contents. According to Bates [6], the discipline of software engineering offers formal approaches to design and development. Some of them are better suited to very small projects, with only one programmer and just a few thousand lines of code. Others are better suited to large projects, where there can be 50 or more developers and millions of lines of code. Game projects generally fall into a medium category, with teams of four to eight programmers and a few hundred thousand lines of code.

Given the size and nature of the game projects, the most appropriate lifecycle models are generally the classic waterfall, the modified waterfall, and iterative prototyping (Bates, [6]).

3.4.1. Waterfall Model

In waterfall model, the requirements of the project are completely defined first. Then the system architecture is designed, followed by detailed design, coding and debugging, and finally testing and validation. This development approach consists of progression through a defined set of processes sequentially and works best when everything is defined well from top to bottom and will not change during development. As already mentioned, a game development project is generally more chaotic than that. The game projects for which the waterfall model is appropriate are usually small casual games or series sequels in which the main game mechanics are already known, and other components of the game, like the game engine, are already set. If the only intent is doing a make-up, like adding new features and creating some new levels, to a game that already developed, than the pure waterfall model can be appropriate (Bates, [6]).

The waterfall steps are:

1. Concept
2. Requirements analysis
3. Architectural design
4. Detailed design
5. Coding and debugging
6. Testing
7. Maintenance

3.4.2. Modified waterfall model

The modified waterfall allows for overlapping of development steps. According to Bates [6], developers can begin coding some subsections without waiting the completion of detailed design, especially when those subsections are well understood and relatively independent modules. When working on a casual game, for example, designers can decide early on how the user interface will look like, and it can be developed anywhere along the process.

A typical genre that can be a good candidate for the modified waterfall model is the adventure games. Meaningful interaction with the characters and overall game environment is important for gameplay, but the designer generally cannot be sure how those environments will look like before the artists create them. Even though the main story and major puzzles are known before development starts, specific interactions, like detailed character dialogues, might have to wait until the designer can see the results of the artists' works (Bates, [6]).

3.4.3. Iterative prototyping model

According to Bates [6], rapid iterative prototyping is the best development model for the most of the game projects. The idea is to getting a rough prototype up and running quickly, testing it, keeping the strong elements (gameplay elements, features and artworks) and throwing out the weak ones, and then going back and repeat as needed.

The key aspect of this lifecycle model is refining the design continually, based on the evaluation of what have been built. At the beginning of each iteration, the team sits down to determine what needs to be done. The customer (in this case, the designer or the publisher's producer) writes out "use cases", abstract statements in spoken language of things that are wanted the game to do. The development team estimates how much time will be spent to these. If the customer learns that a particular request will take two weeks, and that 10 others could be done in one day each, he may decide he'd rather see those 10 features implemented first. The goal is to involve the customer in the development process, so that they take responsibility along with the development team and help for better decisions during the development of the game (Bates, [6]).

According to Bates [6], RTS (Real-Time Strategy) games are perfect candidates for iterative prototyping model. As the new units are created and features are added, unforeseen gameplay mechanics and strategies may emerge, some of them can be good and some of them can be bad. The bad mechanics can be thrown out while the good ones can be allowed to proceed.

Action games might be a better candidate for a mixed approach of the modified waterfall and the iterative prototype, especially for different tasks in the development process. The task of creating audiovisual contents suits to the waterfall model. Characters, weapons, and environments can all be designed, specified, and built in a sequential process. The task of building the game mechanics suits better to iterative prototyping, where the AI for the characters and the operation of the weapons are designed, coded, tested, and refined in a series of iterations (Bates, [6]).

4. COMPONENTS OF A 3D GAME

4.1. Introduction

As previously mentioned, games are dynamic and interrelated systems consisting of audiovisual contents and subsystems that are designed for specific tasks (Figure 4.1.). In this chapter, components that form a 3D game and their relations with each other will be discussed.

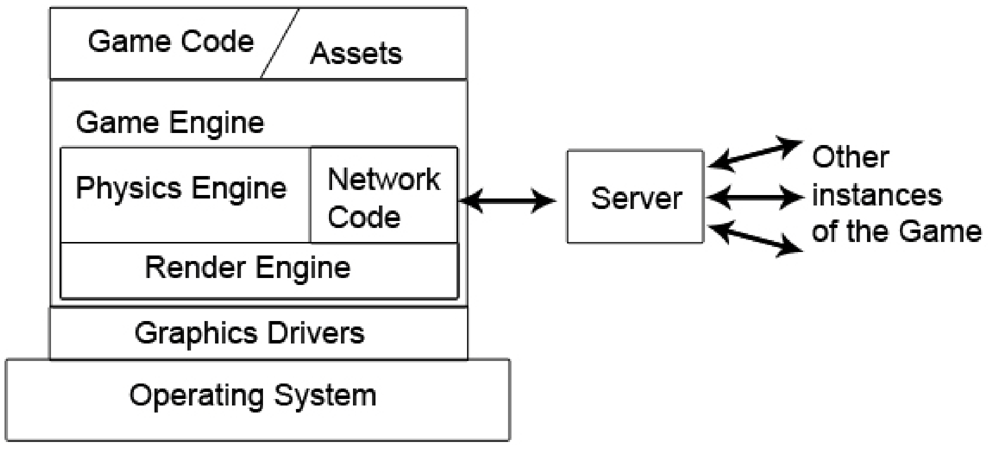


Figure 4.1. Simplified structure of a game

4.2. Game Codes and Audio-Visual Contents

While game engine part is about technical aspects that will be reused over multiple titles, the game codes or scripts are the part that create behaviors specific to a single game (White et. al., [15]). Game scripts define the gameplay and logic of the game. Behaviors of AI agents, actions and restrictions of the player controlled objects, user interface elements etc. are all within this context. On the other hand, rendering tasks or collision detection algorithms are not game specific, so they are within the game engine part. Game scripts also highly dependent to the game engine and API's used.

Game codes specific for a real-time strategy game will be discussed in detail in the following chapters of the thesis.

The asset pipeline is the creation path for all the game assets. For most assets, creation procedure starts at a creation tool and ends in game when final version of the asset is completed. There are a lot of asset types that a typical game needs, including models, materials, textures, animations, audio, and video. The asset pipeline involves obtaining the data needed from the creation tools, and then optimizes, modifies, converts, and outputs the final data that can be used by the game engine. A typical asset pipeline is described in Figure 4.2. (Lengyel, [16]).

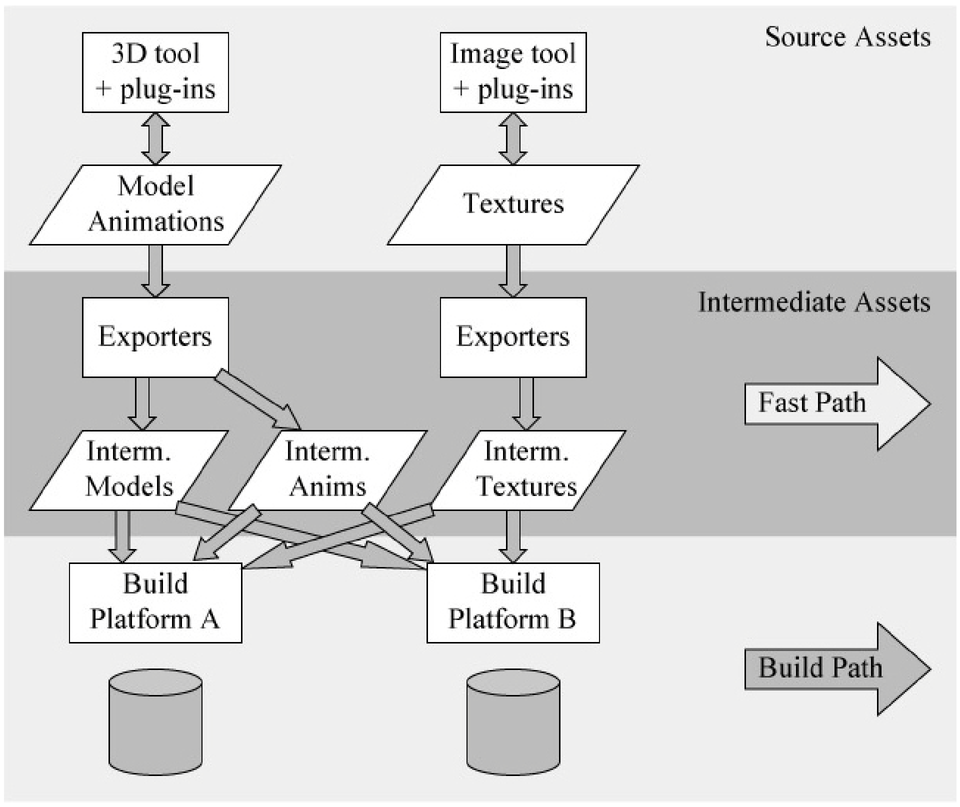


Figure 4.2. Asset creation pipeline

Source assets are created by designers, game artists and concept artists. The tools used to create the source assets are generally referenced as digital content creation tools. In general, source assets are created with a set of specialized tools: one for modeling, one for texture creation, one for animation data, and one for level building. Source assets must contain all the necessary information to build the game content. In other words, it should be possible to delete all the other assets except the source assets without losing the ability to rebuild the game content. Source assets are generally stored in tool-specific formats (Lengyel, [16]).

Final assets are the assets that are optimized for target platform. Generally, the artists has no choice in what the final format has to be due to the constraints caused by the programmers, the target platform, third party engine, or the publisher. Unnecessary information should be removed from the final assets, so all the data is pruned to its required minimum. Final assets have to be compiled and optimized for target platform just like source code. For instance, textures should to be stored in the format and size that can be used by the hardware efficiently, as texture compression algorithms and parameters need to be optimized according to the display capabilities of target hardware. Geometry data must also be compiled for target platform. Most platforms will only accept triangles, but the number of triangles per batch, the sorting order of the vertices, order of face vertex indices, the need for triangle strips, and the amount of data per vertex may have a big impact on the performance of the game. Limitations on the rendering capabilities of the hardware may also have an impact on the geometry data since it may be necessary to create separated geometries to enable the renderer to use these in different render passes to get the desired effect (Lengyel, [16]).

As previously mentioned, 3D objects created through modeling and texturing processes. Modeling involves the creation of a three dimensional representation of the real or imaginary objects and texturing fulfills the painting, or coloring, of the object with a required image. Texturing also involves the creation of other required maps that will be interpreted by shader algorithms to create visual effects over the models (Manninen et. al., [13]), like a “tangent-space normal map” can be used to create an illusion of bumpiness over the flat surfaces of the model.

The shape of a 3D geometry generally defined with set of vertices, edges and faces (Figure 4.3.a.). Vertices are just simple points (x,y,z positions) on 3D space. Vertices of a model are connected with lines known as edges. When three or more vertices connected with edges they create a closed figure, which is a polygon. The simplest polygon is a triangle. Modern 3D accelerated graphics processing units are designed to display and manipulate millions of triangles in milliseconds. Because of this capability, models constructed out of the simple triangles instead of the more complex ones (Finney, [17]).

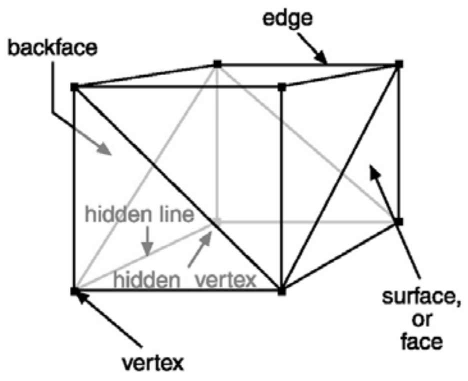


Figure 4.3.a. A simple 3D model

The most widely used structure for representing 3D models is face-vertex mesh. A six-sided box model and the face-vertex mesh structure are given in Figure 4.3.b. Because of the model consist of triangles, every face has three vertices. However, a vertex may have different number of surrounding faces. This structure also includes the information about the front and back sides of the faces. If the vertices of a face are lined up clockwise in the screen, according to their order in the face list, then we can say that the camera is looking towards the front side of the face. If they are lined up counter clockwise in the screen, than we can say that the camera is looking towards the back side of the face.[8]

Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15

example

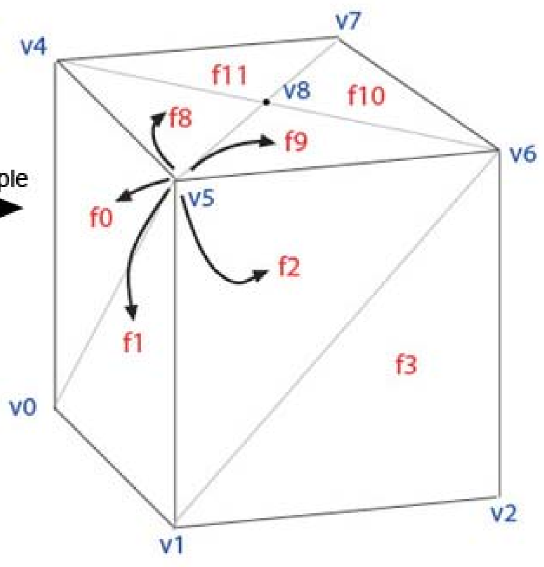


Figure 4.3.b. A widely used structure for representing 3D models

[8] http://en.wikipedia.org/wiki/Polygon_mesh

A texture is a 2D bitmap used to be drawn over the surfaces of a 3D object, to add detail and color upon the object. In order to apply textures to 3D objects, a system that describe where each pixel of the image will be drawn over which part of the model is needed. The system is called UV Coordinate Mapping (Figure 4.4.).[9] The U and V axes are analogous to the X and Y axes of a 2D coordinate system. Each UV coordinate consists of a pair of floating points (Finney, [17]). Generally, (0.0, 0.0) UV coordinate is the bottom-left corner of the texture and (1.0, 1.0) is the top-right corner.

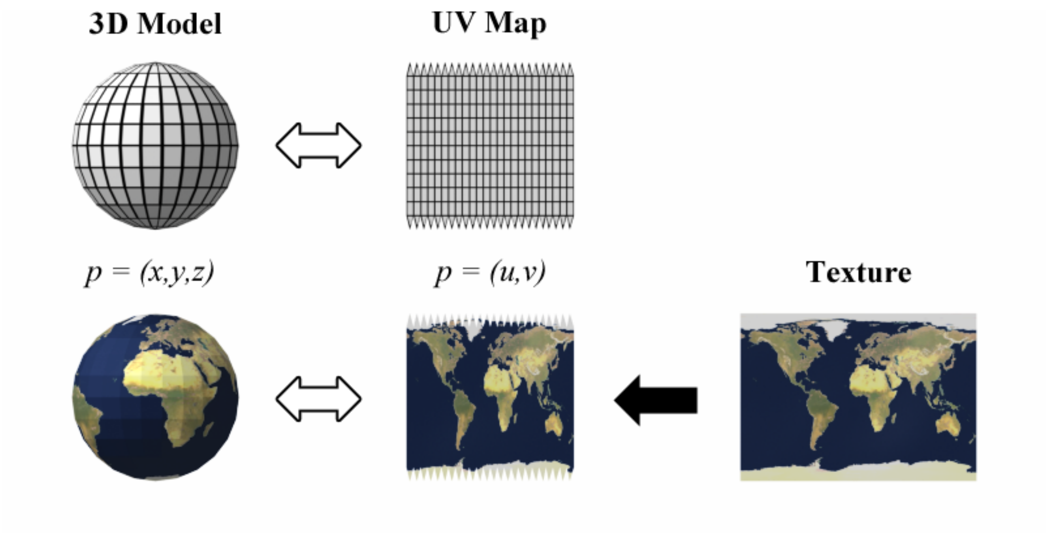


Figure 4.4. Usage of UV coordinates for texture mapping

3D objects can be broken down into three categories (Manninen et. al., [13]):

- Static Models
- Dynamic Models
- Effects

Buildings, plants, decorative items, elements of terrain, etc., can be considered as static objects. In other words, static objects are models that have no animation at all and/or have only decorative value. The player only sees these objects in the game but cannot interact with them (other than colliding). The production of these objects includes geometry modeling and texturing (Manninen et. al., [13]).

[9] http://en.wikipedia.org/wiki/UV_mapping

Characters, vehicles, different types of items used by characters, etc., may require the status of dynamic objects. If a model is created to be used rather than just being a static prop[10], it can be considered as a dynamic model. In other words, the player can interact with them. These objects also include the necessary information to be animated. The creation process of dynamic objects is a bit more complex than the static objects. Dynamic objects are constructed using modeling, texturing and rigging. While modeling and texturing are similar with the creation of static objects, rigging means the creation of animation mechanism, such as bone hierarchy, inside the 3D model (Manninen et. al., [13]).

Some of the effects seen in games, such as, smoke or fire, can also be created with 3D modeling tools. The effects, however, are generally created during the level design and may also be tightly combined with programming and/or texturing. It can be considered as either dynamic or static depending on the type of the effect. Effects in the game level like fire or smoke generally have not any purpose other than decorative or informative purposes. These effects can be created as animated sequences of 3D objects; however, they are usually created through particle systems that are capable of emitting and animating objects automatically (Manninen et. al., [13]).

The most important method to animate the 3D models is known as “skeletal animation” which is the use of “bones” to animate the model rather than editing and moving each vertex or face manually. Each vertex is affected by a bone or in most cases multiple bones. Therefore, a bone or node is simply a control point for a number of vertices. These bones are similar in concept to bones in human body, such as thigh or calf. When the bone moves, every vertex affected from it also moves, as shown in Figure 4.5., where, “N” and “k” represent the frame number or frame time. Even the movement of bones can affect other bones if there is a parent-child relation between them. This helps to animate the model appropriately, because movements in one portion of the model affect other parts, as it is in real life. Consequently, calculating the transformations for the individual vertices is required to work with the bones (Pipho, [18]).

[10] Prop: Theatrical property, commonly referred to as a “prop”, is any decorative object at the stage.

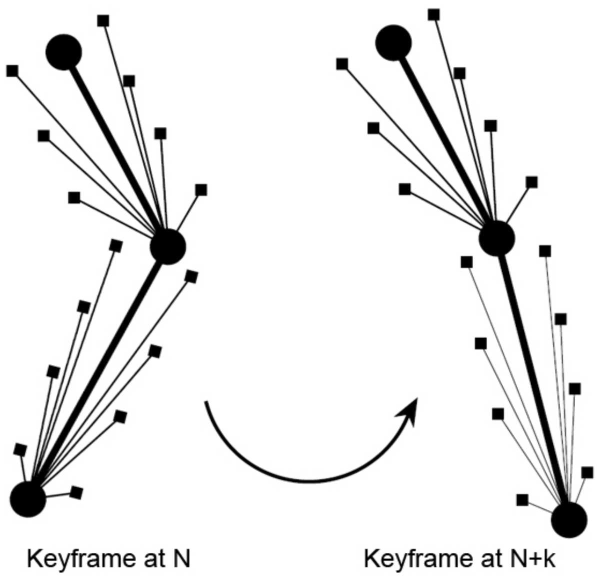


Figure 4.5. Skeletal animation

4.3. Game Engine

A game engine is a software system that provides the core functionalities to abstract the details (sometime platform-dependent details) of doing common game-related, but not game-specific tasks, like rendering, physics, and input, so that developers can focus on the details about their games (Ward, [19]).

A typical game engine handles graphical rendering and similar tasks, as well as handling additional tasks such as collision detection between game objects, physics simulation or networking. The most common component that game engines include is a graphics rendering system. One of the most important benefits that can be gained by using a game engine is platform abstraction. With platform abstraction the game can run in several platforms with little or no modifications in the source code (Manninen et. al., [13]).

According to Zerbst and Düvel [20], "an engine should be independent of the project and capable of working with other video game projects and non-game multimedia software projects without the need to modify the engine's code. In other words: The engine's code must not include any game-related code, and it has to be reusable."

For a long time, many game developers made their own game engines and kept that technology in house and used for their games with updating it as hardware improved and new versions were needed. Engines like “SCUMM” by LucasArts and “SCI” by Sierra, for example, powered most of the games that those companies released between late 1980s and mid 1990s. More recently, engines like “Quake Engine” which is the engine that used in the Quake series and the “Unreal Engine” are also started as in-house technologies, but they have evolved into middleware (Ward, [19]).

Over the past several decades, the cost of developing an in-house engine has grown significantly, and more developers have begun to specialize in developing either fully featured game engines or engine components to sell to other developers, rather than develop games. The middleware developers started to offer these products at very reasonable prices, and, for most game development studios, this created a very clear "build versus buy" decision. Why pay six programmers for a year to build a game engine when you can buy 90 percent of the features you want from a proven technology for less money and have it immediately? As a result, almost all components of a game engine became purchasable at a variety of prices, or obtainable for free from open source projects (Ward, [19]).

4.3.1. Game loop

In a graphical user interface (GUI), of the sort found on a Microsoft Windows, the majority of the contents on the screen are static. Only a small part of any one window is changing appearance actively at any specific moment. Because of this, graphical user interfaces have generally been drawn on screen with a technique known as rectangle invalidation, in which only the small portions of the screen where the contents have actually changed are re-drawn. Older 2D games used this method to reduce the number of pixels that needed to be re-drawn (Gregory, [21]).

Real-time 3D computer graphics are rendered in an entirely different way. As the camera moves in a 3D scene, the entire contents of the screen change continually, so the concept of invalid rectangles is no longer applicable. Instead, an illusion of motion and interactivity is produced in much the same way that in a movie by displaying a series of still images to the viewer in a rapid succession, which requires a loop. At its simplest a rendering loop is structured as follows in Figure 4.6. (Gregory, [21]):

```
while (!quit)
{
   // Update the transform of camera
   updateCamera();

   // Update positions, orientations and any other
   // visual states of dynamic objects in the scene
   updateGameObjects();

   // Render a still frame into an off-screen
   // frame buffer known as the “back buffer”
   renderScene();

   // Swap the back buffer with the front buffer,
   // or copy the content in the back buffer  into the front buffer,
   // in order to display  the most recently rendered image on screen.
   swapBuffers();
}
```

Figure 4.6. A simple rendering loop

A game is consists of many interacting systems, including I/O devices, rendering engine, animations, collision detection, resolution analysis, optional rigid body dynamics, networking, audio, etc. Most of the engine systems require periodic updating while the game is running. However, the rate at which these systems need to be updated varies from system to system. Animations typically need to be updated at a rate of 30 or 60 Hz, in synchronization with the rendering system. On the other hand, a physics simulation may actually require more frequent updates like 120 Hz. Higher level systems like AI, might only require to be updated once or twice per second, and they do not require being synchronized with the rendering loop at all. There are a number of ways to implement the periodic updating of game engine systems. Simplest way is using a single loop that updates everything in full synchrony. Such a loop is often called game loop or main loop because it is the master loop that updates every system in the engine (Gregory, [21]).

One of the earliest games, "Pong", which is a kind of 2D table tennis game, only consists of two movable vertical paddles controlled by players and one ball, might be a good example of how a game loop might look like as in Figure 4.7. (Gregory, [21]).

```
void main() {
   initGame();
   while (true) // game loop
   {
      readHumanInterfaceDevices();
      if (quitButtonPressed())
      {
         Break; // exit the game loop
      }
      movePaddles();
      moveBall();
      collideAndBounceBall();
      if (ballImpactedSide(LEFT_PLAYER))
      {
         incrementScore(RIGHT_PLAYER);
         resetBall();
      }
      Else if (ballImpactedSide(RIGHT_PLAYER)
      {
         incrementScore(LEFT_PLAYER);
         resetBall();
      }
      renderPlayField();
   }
}
```

Figure 4.7. Game loop of a 2D table tennis game.

As it can be seen in Figure 4.7., when the game started, it calls initGame() to do necessary setups required by the graphics system, I/O devices, audio system, etc. Then the main loop is entered. The statement “while (true)” tells that the game loop will repeat forever, unless interrupted internally. The first thing to do inside the loop is reading I/O devices and check if the quit button is pressed. Next, the positions of paddles are changed slightly towards upward or downward based on I/O check. Then physics related functions are called to update the position of the ball in 2D space. After that some checks must be done to determine if one of the players had missed the ball. Finally renderPlayField() function draws the entire content on the screen (Gregory, [21]).

In game engines, rendering process generally implemented as a part of main game loop but there may be other independent loops for different tasks. Because of, duration of the rendering process heavily depends on current view at a specific moment; main game loop that includes rendering process becomes unpredictable from the point of iteration duration. Physics simulations, on the other hand, requires more stable and fixed duration on iterations to simulate physics objects accurately and more easily. Because of that reason, physics simulations better be apart from main game loop and handled within an independent loop (Figure 4.8.)

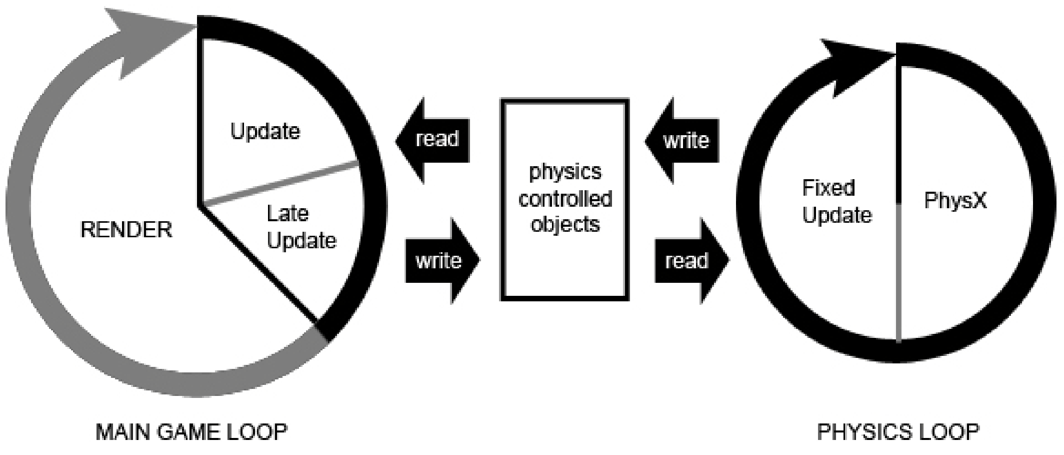


Figure 4.8. A simple representation of iteration cycles of the Unity game engine

In Unity game engine, main game loop handles majority of the low level and high level systems. It searches through the scene graph and calls Update() and

LateUpdate() methods of the game objects respectively. Than it starts the rendering process and waits until the display is updated. Nvidia’s (previously Aegia’s) “PhysX” physics engine is implemented as it runs independently from the main game loop. Physics loop generally iterates more than main game loop (for an averagely populated scene) at any time period. It searches through the scene graph and makes necessary calculations only for physics controlled dynamic objects. If any other physics calculations are specially needed, they can be handled within FixedUpdate() methods of game objects, which gets called right after PhysX. Both main game loop and physics loop can access and manipulate physics controlled objects directly.[11]

4.3.2. Graphics engine (renderer)

Game engines also include sophisticated systems for rendering the game environments. Each game can use a different approach to organize how the visual contents of the game will be modeled. This becomes increasingly important as games are becoming more realistic by using vast 3D environments, rich textures and forms, and other visual contents (Finney, [17]).

The 3D rendering engine is considered as one of the main components in most modern video games. Usually, 3D rendering engines handle both the software and hardware sides that are needed to function through 3D graphics APIs such as Direct3D and OpenGL (Lengyel, [16]).

At its lowest level, the main responsibility of a graphics engine is drawing the objects that are visible to the viewer. An engine programmer typically uses a graphics APIs such as OpenGL or Direct3D in order to develop a render engine that can draw the objects correctly. On some platforms, especially when there is no hardware acceleration or a standard API available, the developer might need to write the entire graphics system as it run on the CPU. The result is a software renderer. Although modern graphics processing units has a lot of power so there is no need to write software renderers for current systems. The ability to write

[11] http://unity3d.com/support/documentation/Manual/index.html

software renderers is still considered important. For example, on embedded systems with graphics capabilities such as mobile phones and handheld devices (Eberly, [22]).

The most complex engine is a 3D engine. A 3D engine needs to organize things in a suitable way for best performance. The object and scene management is one of the most important tasks. In a best case scenario, the user sends scene data to the engine and the engine takes care of tasks like organizing and structuring the data. After initialization, the user then needs to do only render calls and does not care about state switches and similar things. The engine catches those render calls to apply them with the most effective way in terms of performance (Zerbst and Düvel, [20]).

A 3D engine requires a system to know how the objects are laid out in the virtual world and how to keep track of changes in status of the models, their orientation, and other dynamic information. This is done using a mechanism called a scene graph, which is a specialized form of a directed graph. The scene graph keeps information about all entities within the game world in structures called nodes. The 3D engine searches through this graph, examining each node one at a time to determine how to render it. Figure 4.9. shows a very simple seaside scene with its scene graph. The nodes represented by ovals are group nodes, which contain information and point to other nodes. The nodes that represented by rectangles are leaf nodes. A node in a scene graph can be anything. The most common entity types are the camera, 3D models, materials, lights (or lighting information), fog, sky box and other environmental effects like particle emitters, and event triggers (Finney, [17]).

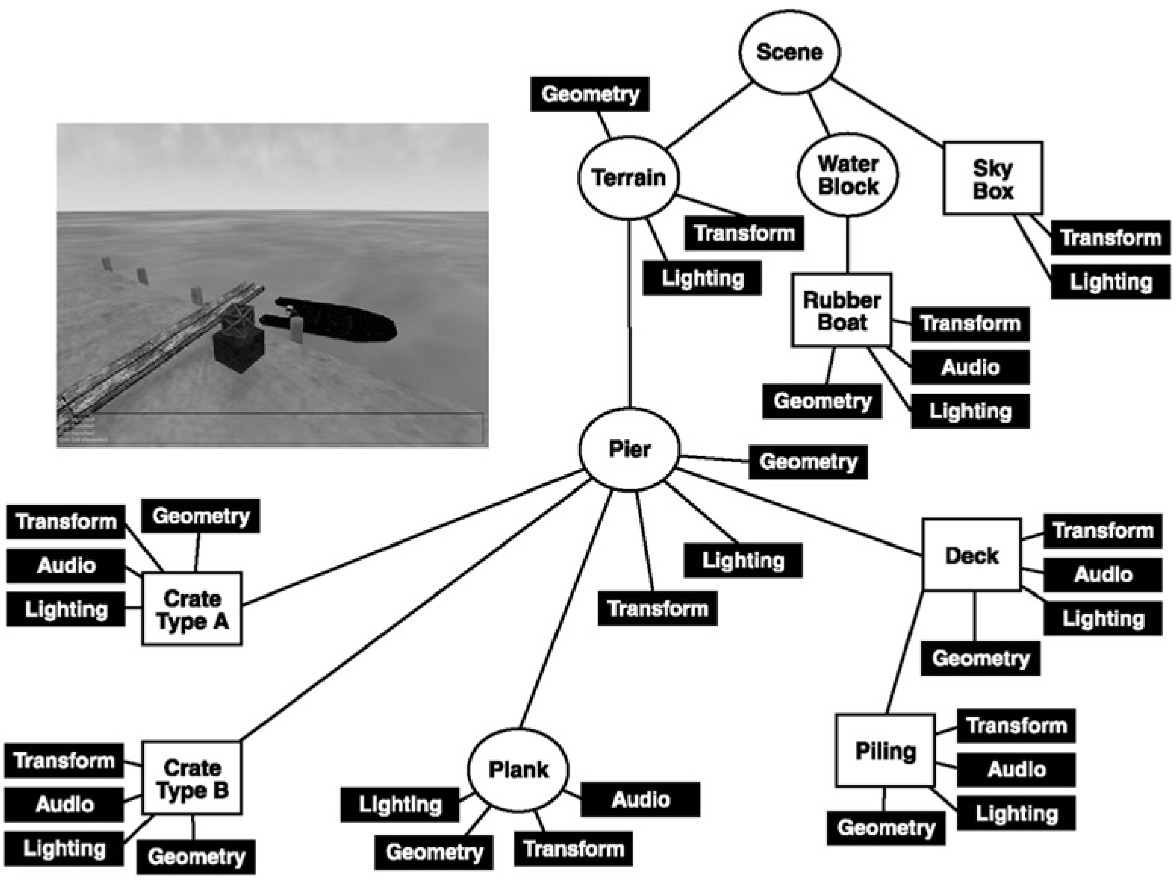


Figure 4.9. A simple scene graph of a seaside scene

a. Graphics pipeline

At the beginning of the graphics pipeline, there is some information in the form of vertex data. This information is sent to the vertex shader if there is one available. If a vertex shader is not available, then the information is sent to the transformation and lighting engine, or “TnL” for short. The TnL can be considered as the built-in system of the graphics adapter to transform the vertex data and calculate the lighting of the vertices (Zerbst and Düvel, [20]).

A hardware TnL engine on the graphics adapter provides a great boost in performance as compared to the standard TnL done in software. Today, every graphics adapter should be able to perform TnL functions. This TnL engine is said to be part of fixed-function pipeline. The fixed-function pipeline got its name from the fact that the programmer can set variables to the hardware influencing the outcome of only some of the calculations. This limitation represents a restriction in that there is no free access to the graphics hardware (to write new functions, for example). In contrast to fixed-function pipeline, a flexible pipeline in which shaders are used to directly access the GPU instructions and apply any function to the data (Figure 4.10.) (Zerbst and Düvel, [20]).

After the viewport transformation and the clippings are processed, current color of the pixel and per-pixel lighting in pixel shaders are taken into consideration. Only after this other tests can be performed on the pixels (for instance, the alpha test, the stencil test, and the depth test) and the pixels are sent into to the frame buffer (Zerbst and Düvel, [20]).

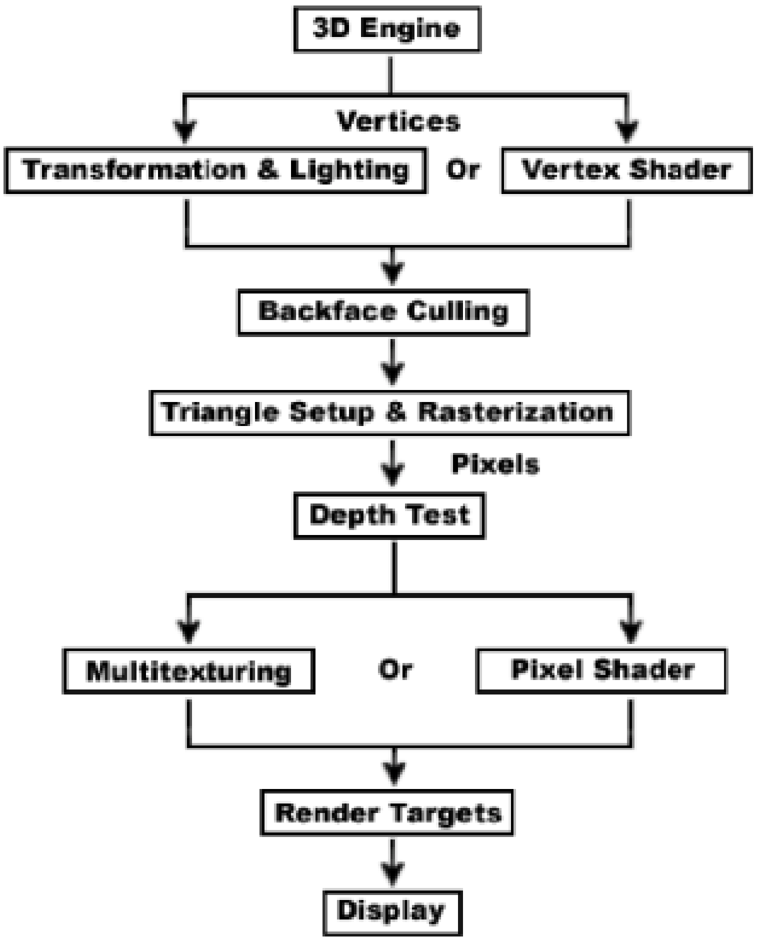


Figure 4.10. 3D graphics pipeline

A 3D pipeline can be decomposed into four stages (Sánchez and Dalmau, [24]):

- Visibility Determination
    - Clipping
    - Culling
    - Occlusion Culling
- Resolution Determination
    - LOD analysis
- Transform and Lighting
- Rasterization
    - Depth Buffer
    - Shaders

By combining these processes, a 3D scene can be drawn at a frame rate suitable for real-time games. A broad result spectrum can be achieved by refining the design of components and their relations with each other (Sánchez and Dalmau, [24]).

Clipping is the process of eliminating unseen geometry by testing it against a clipping volume, such as the view frustum (Figure 4.11.). If the test fails, we can ignore that geometry because we will know for sure it will not be seen. Obviously, the clipping test must be faster than drawing the primitives in order to provide a significant increase in performance. If it took longer to decide whether to draw or not to draw the primitive, clipping become a useless work (Sánchez and Dalmau, [24]).

The view frustum consists of six planes. A near plane sits close to the projection plane, which represents the screen of the monitor. No object closer than this near plane can be seen because it is behind the projection plane. On the opposite side there is the far plane, which simply represents the farthest distance at which the renderer allows objects to be visible (Zerbst and Düvel, [20]).

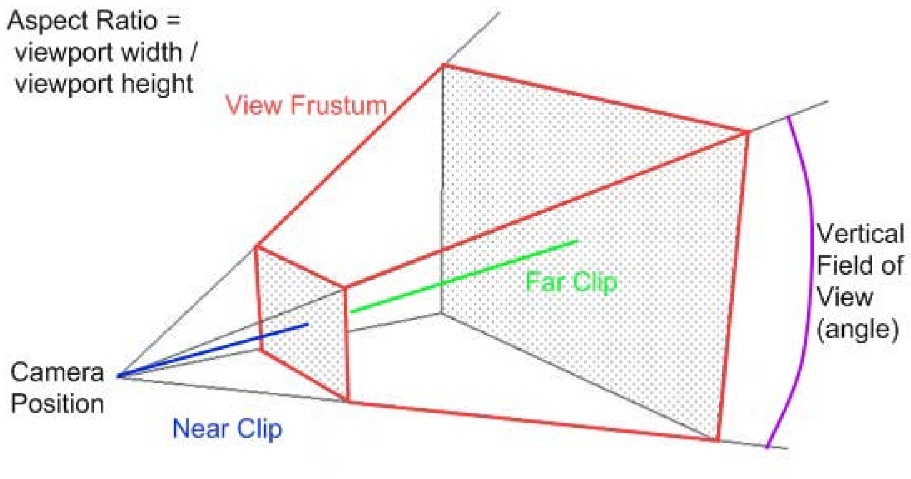


Figure 4.11. View frustum

Because of the highly dynamic nature of game environments and complex geometries of 3d models, visibility culling (view frustum culling) is handled simply by intersecting the view frustum with the bounding volume (Figure 4.12.) of the 3d geometry, instead of the geometry itself, and rendering each piece of geometry whose bounding volume is at least partly inside. This is a fast and efficient way of culling large amounts of geometry (Lengyel, [16]).

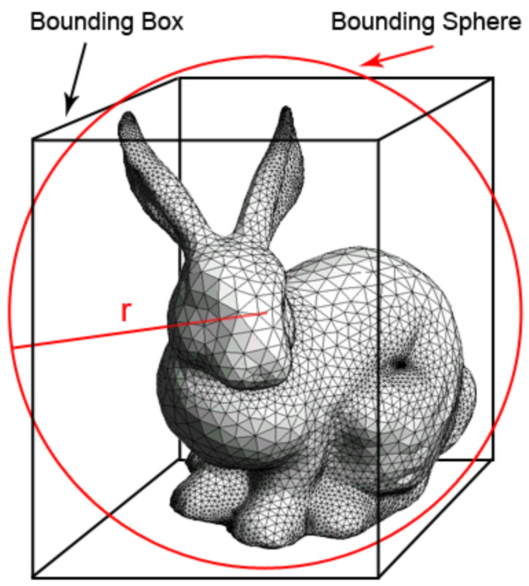


Figure 4.12. Bounding volumes of a 3D model

The term bounding volume refers to any three dimensional object that contains some other object. The simplest bounding volumes that used by game developers are spheres or axis-aligned bounding boxes. A bit more complicated one is an oriented bounding box. More complicated one is the convex hull shaped object (a convex polyhedron). In each case, the bounding volume tends to be convex. To get a more complicated bounding volume, several of them might be combined as a union of bounding volumes (Eberly, [22]).

Another important method for render optimization is using different resolution versions of the model known as “level of detail” (LOD). This method of computer graphics attempts to bridge quality and performance by changing the amount of detail needed to represent the model effectively and efficiently (Luebke et. al., [25]).

The fundamental concept of LOD is simple: when rendering, less detailed versions of the models are used for small, distant, or unimportant portions of the scene (Figure 4.13.). These less detailed representations typically consists of a selection of several versions of objects in the scene, each version has less amount of polygon and faster to render than the one before (Luebke et. al., [25]).

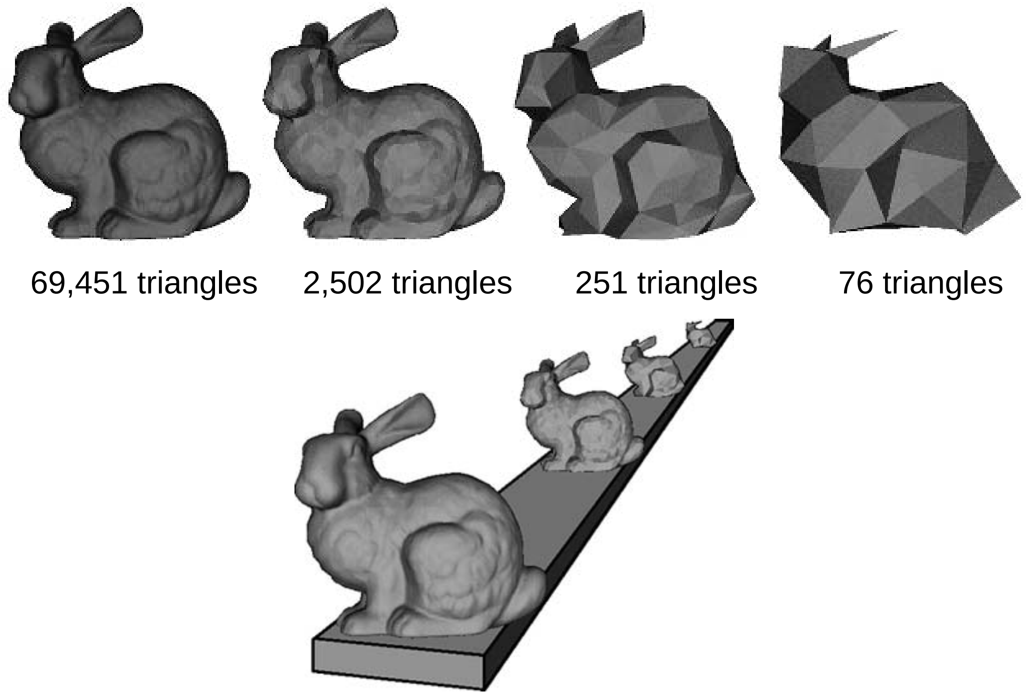


Figure 4.13. Levels of detail of a model

Another method of visibility determination is back face culling (Figure 4.14.), which is a triangle (or polygon) level operation that ensures that only triangles, facing the camera, are rendered. With back face culling enabled, any triangles that are facing away from the camera can be eliminated. The front side of a triangle is defined two ways. The first definition is that the vertices of the triangle are generally specified in counter-clockwise order. The front side of the triangle is also defined as the direction in which normal vector of the face is pointing (Harrison, [26]).

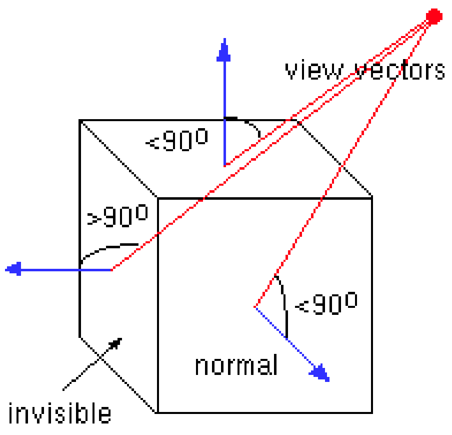


Figure 4.14. Back face culling

The transform stage is used to perform geometric transformation of the models in 3D space. This includes the familiar rotation, scaling, and translation, but also many other transforms. In its more general definition, a transform can be described as a 4x4 matrix, which left-multiplies incoming vertices in order to calculate transformed vertices. This stage also handles the projection of transformed vertices from 3D space into 2D screen space, using projection matrices. By doing so, 3D coordinates can be transformed into 2D coordinates, which are then rendered to the screen (Sánchez and Dalmau, [24]).

In computer graphics, the most important type of projection is perspective projection which makes the 3D scene look like the real world on the projection plane. The perspective projection also adds the depth information into the x and y component of the projected points (Zerbst and Düvel, [20]).

This projection works explicitly by drawing lines from the point of the viewer to each point inside the viewing frustum. The point is rendered onto the screen at the point at which this line hits the projection plane. As can be seen in Figure 4.15., the points that are more distant from the viewer require closer projection lines with a smaller angle than points that are closer to the viewer. Thus, the depth information is taken into account and distant polygons are projected into the smaller areas on the projection plane than polygons closer to the plane (Zerbst and Düvel, [20]).

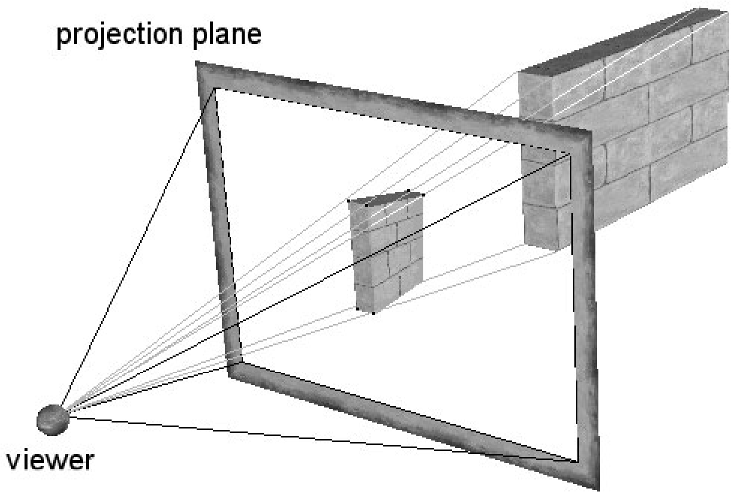


Figure 4.15. Perspective projection of a 3D scene onto a 2D projection plane

The lighting stage implements lighting calculations to increase the realism of the scene. In computer graphics, there are two main kinds of light: ambient light and direct light. Ambient light is the light that is present at each point in a 3D scene with the same intensity and can be considered as it coming from all directions with the same intensity. In nature, that kind of light stems from the direct light that is reflected all over the space on every object it hits. In other words, it can be thought as it makes each object an emitter of light for itself. This level of light existing all around is the ambient light (Zerbst and Düvel, [20]).

Direct light always comes from a specific light source whose position could be understood by looking at where its light falls. For direct light, the computer graphics have the three kinds of light sources which are as follows (Figure 4.16) (Zerbst and Düvel, [20]):

- Directional light
- Point light
- Spot light

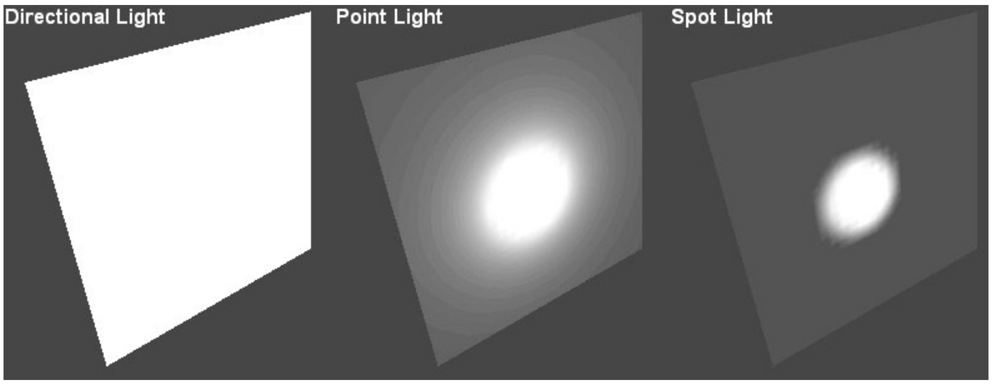


Figure 4.16. Lighting a gray rectangle by three types of light sources

Most current APIs and GPUs offer hardware lighting, so light sources and surface properties can be defined and illumination can be applied at a zero CPU hit. Unfortunately, only per-vertex calculations of lighting can be applied directly in today's hardware and the lighting values between vertices are then interpolated during rasterization (Sánchez and Dalmau, [24]).

Rasterization is the process by which the geometries are converted to pixel sequences on a CRT monitor (Sánchez and Dalmau, [24]).

When rendering two triangles that overlap each other in screen space, a method to ensure that the triangle that is closer to the camera will appear on top is needed. This could be accomplished by always rendering triangles in back-to-front order (painter's algorithm). However, as shown in Figure 4.17., this does not work if the triangles are intersecting one another (Gregory, [21]).

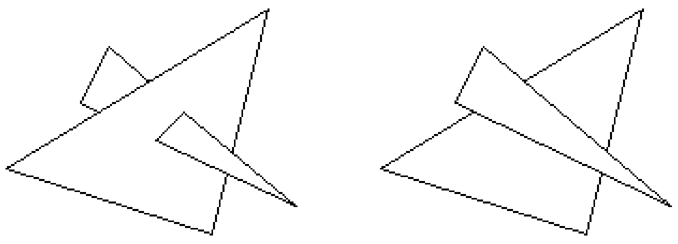


Figure 4.17. Intersecting triangles with (left) and without (right) depth buffering

In order to implement triangle occlusion[12] properly, independent from which of the triangles are rendered first, rendering engines use a technique known as "depth buffering" or "z-buffering". The depth buffer is a full screen buffer that typically contains 16 bit or 24 bit floating-point depth information for each pixel in the frame buffer. Every fragment has a z-coordinate that specify its depth into the screen. A fragment's depth value is stored into the corresponding pixel of the depth buffer. When another fragment from another triangle is drawn over the same pixel, the engine compares the depth values of the new fragment with the value within the depth buffer. If the new fragment is closer to the camera (if it has a smaller depth), engine overwrites the pixel in the frame buffer. Otherwise the fragment is ignored (Gregory, [21]).

Current generation graphics processing units are programmable by the programs that are called shader algorithms. The two types of shader algorithms are vertex shaders and pixel shaders. These programs provide more control over the rendering process than the fixed-function pipeline that is limited to the standard graphics API calls (Eberly, [22]).

A vertex shader calculates colors at the vertices of a triangle mesh by using vertex normals, textures, lights, and the standard lighting equations. The pixel shader then interpolates texture coordinates between the vertices and uses them with the texture image in order to assign colors to pixels in the rasterized triangles. Having the ability to write vertex and pixel shaders allows us to program the rendering behavior of the GPU and to have more creative special effects that cannot be done in the fixed-function pipeline (Eberly, [22]).

In summary, the lighting can be computed by a shader both at a vertex and pixel levels. A vertex shader computes per-vertex illumination. A pixel shader will compute lighting per pixel by interpolating pixels in between vertex samples, offering much better results. More complex lighting effects, like illusion of surface details through bump maps (normal maps) can also be done with shader algorithms (Sánchez and Dalmau, [24]).

[12] Occlusion culling is the process used to determine surfaces and part of surfaces that are not visible from the point of view (http://en.wikipedia.org/wiki/Hidden_surface_determination).

4.3.3. Physics engine

There are several fields of physics, from quantum physics, to fluid mechanics, but mostly, part of a specialized field, kinetics, or more accurately, classic mechanics used for games. Sir Isaac Newton was one of the philosophers who most contributed to this field. For this reason, this field is also referred to as Newtonian physics. The ideas and mathematics behind this field represent the real behaviors of the objects that do not travel at the speed of light (De Sousa, [27]).

Physically modeling objects is one of the aspects that remain the same even for the simplest games. Gravity is same, no matter whether it is a 3D racing game or a 2D platform game. Collisions always occur both in first-person shooters and with spaceships. Re-writing the same algorithms over and over again is a tiring, unneeded, and time-consuming task. There is no better way than developing or licensing a physics engine for a game (De Sousa, [27]).

Actual purpose of a physics engine for games is to create convincing illusion of real world dynamics. For games, it is not needed to calculate the most accurate results; instead, calculating approximate results in milliseconds is necessary to run the game real-time.

Basic concepts of game physics include, but are not limited to following terms:

- Time
- Position
- Mass
- Velocity
- Acceleration
- Force

A physics engine needs to measure both the duration of events and spatial distances of objects to determine the spatial relationship between objects and to create motion under force. Time is a quantity used to measure the duration of events and the intervals between them. Position, on the other hand, is a relative distance between objects. Generally, the term position is used for the distance of an object from origin in three-dimensional Cartesian space (McShaffry, [28]).

When two objects collide each other, how their momentum will change depends on their masses. Mass is the measurement of how much matter an object has. It can also be thought as a measure of body's resistance to motion or a change in its motion. Thus the greater a body's mass, the harder it will be to set it in motion or change its motion (Bourg, [29]).

All other measurements, such as force and velocity, are derived from different combinations of mass, distance, and time (McShaffry, [28]).

Linear velocity is the change of position of an object during a certain period of time. It is represented as vectors, which specifies how many units (depending on the measurement used e.g. meters or miles) an object moves per unit of time though a direction. Linear velocity can be calculated by the following formulas (De Sousa, [27]):

(current_position – previous_position) / duration (4.1.a)

or

(current_position – previous_position) / (current_time – previous_time) (4.1.b)

Similarly, angular velocity is the change in orientation in a period of time and can be calculated by (De Sousa, [27]):

(current_orientation – previous_orientation) / (current_time – previous_time) (4.2)

Similar to velocity, linear acceleration is the change of linear velocity during a period of time. The angular acceleration is the change of angular velocity during a period of time. Linear and angular acceleration can be calculated by following formulas respectively (De Sousa, [27]):

(current_velocity – previous_velocity) / (current_time – previous_time) (4.3.a) and

(current_angular_velocity – previous_angular_velocity) /
(current_time – previous_time) (4.3.b)

The center of mass is the point of an object from where all mass distribution is equal, frequently called the center of gravity. If there are no external forces, the exact center of mass of an object is the point that balance any object, stationary. It can be calculated by the sum of all the masses of all the points of an object multiplied by their positions, and then divide the total by the sum of all the masses, or the total mass (De Sousa, [27]). The equation can be shown as:

Center of Mass = Σ(Position i * mass i) / Σmass (4.4.)

In real life, center of mass can be calculated by selecting a set of distinct points of the object and use those points to find an approximate result or by using mechanical tools working with trial and error basis (De Sousa, [27]). In games, 3D objects are generally represented by collision volumes which are regular primitive shapes like sphere or box and physics engine uses these primitives to calculate the center of mass. It also can be defined manually like mass value.

The key equation of motion in physics is the basic force equation and its derivatives. In order to make an object move from one location to another in a realistic manner, a force must be applied to the object (Harrison, [26]). The term “force” is actually used for “linear force” which is a force that affects the center of mass of an object. When a force applied to an object, it exerts a linear effect to the object usually resulting in acceleration of the object, and thus a change of velocity (Eberly, [23]). The equation can be shown as:

Force ($\vec{F}$) = Mass (m) * Acceleration (**a**) (4.5)

When a force applied to an object, it results in both a linear force and in an angular force. An angular force is a force that only rotates the object without changing its position. This type of force is called torque. To calculate the torque, the first thing to find is the distance vector from the point, the force applied, to the center of mass of the object. This is called the arm of the force. After that the cross product of the arm with the force used to find the final torque (Eberly, [23]). The equation can be shown as:

$$\text{Torque } (\tau) = \text{arm } (r) \times \text{Force } (F) \tag{4.6}$$

A physics engine requires a collision detection system to detect the constraints for solving the Newtonian equations of motion. In fact, a physics engine can be based on a much broader foundation, including determine the Lagrangian equations of motion by using known constraints. Usually the reduction of dimensionality via known constraints provides equations of motion that have better stability when solving mathematically, especially in physical systems that model frictional forces (Eberly, [22]).

a. Collision detection

Even the simplest 2D game needs a collision detection system. “Breakout” (developed by Atari in 1976) is a typical example of a simple 2D game. A ball bounces off walls, bricks, and the paddle and the core of the game experience is depends on a 2D collision algorithm. It is almost impossible to design a game without having at least a basic collision, except text based adventure games like “Zork” (McShaffry, [28]).

The primary purpose of a physics engine’s collision detection system is to determine whether the objects in the game world have come into contact. To answer this question, it is needed to represent each logical object by one or more geometric volumes. These volumes are generally quite simple primitives, such as spheres, boxes, and capsules. However, more complex volumes can also be used. The collision system determines whether or not any of the volumes are intersecting at any given moment in time. So a collision detection system is actually performs geometrical intersection tests (Gregory, [21]).

From the point of view of detecting intersections efficiently, geometrically and mathematically simple volumes are preferred. For example, a rock might be represented by a sphere for collision purposes; the hood of a car might be represented by a rectangular box; a human character might be represented by a collection of capsules (pill shaped volumes). Ideally, a more complex shape must be preferred only when a simpler representation is proved being insufficient in order to achieve the desired behavior in the game (Gregory, [21]).

Spheres are the simplest type of primitives for collision detection systems. A sphere can be represented using only four scalar (three for center point position and one for the radius), so they are quite cheap to store. Furthermore, spheres are invariant under rotations, which make them the most efficient kind of collision primitive for rigid bodies (Bergen, [31]).

Due to their simplicity, collision detection of spheres can be done with simple math. Two spheres A and B intersect, if the distance between their centers c_A and c_B is not more than the sum of their radii r_A and r_B (Figure 4.18.a.) (Bergen, [31]):

$$A \cap B \neq \emptyset \equiv \|c_A - c_B\| \leq r_A + r_B \qquad (4.7.a)$$

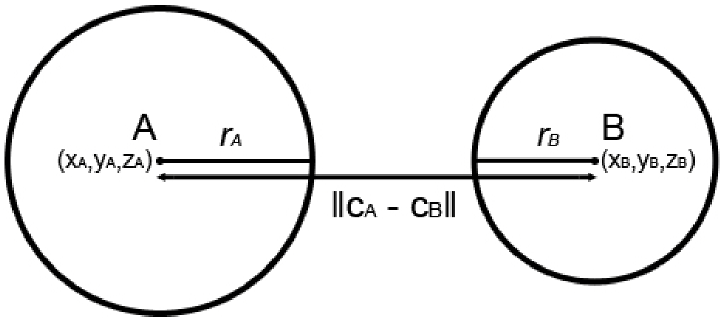


Figure 4.18.a. Collision detection between two spheres

Because of calculating the distance between two given points requires square root operations, which takes more time to compute than primitive arithmetic operations such as additions and multiplications, the expression would be faster to compute if written as (Bergen, [31]):

$$A \cap B \neq \emptyset \equiv \|c_A - c_B\|^2 \leq (r_A + r_B)^2 \qquad (4.7.b)$$

For Microsoft XNA Game Studio, the code for this algorithm would be as Figure 4.18.b.

```
if (Vector3.DistanceSquared(posA, posB) <= Math.Pow((radA + radB), 2))
   return true;      // Collision detected
else
   return false;     // No collision
```

Figure 4.18.b. Collision detection algorithm between two spheres

The "DistanceSquared" method bypasses square root operations, but provides same information when used to compare with square of the sum of radii.

b. Ray casting

One of the most important concepts in computer graphics is known as ray casting, which actually is an intersection test like the collision detection. Almost every 3D application uses ray casting techniques, including the ones that does not have any interaction between objects, because the rendering a 3D scene to a 2D screen is also done with ray casting (McShaffry, [28]).

From a mathematical point of view, a ray is nothing more than a part of line. A line is a rather one-dimensional object continuing from positive to negative infinity at some angle and some location in space. A ray is similar to a line, except that it has a specific starting point, and it continues through a specific direction for a finite length. The line segment is tested against the collidable objects in the scene to determine the possible intersections (Gregory, [21]).

Ray casting is used heavily in games. For example we might want to know whether character A has a direct line of sight to character B. To determine this, a directed line segment from the character A to character B can be cast. If the ray hits character B, we know that A can see B. But if the ray strikes some other object before reaching character B, we know that the line of sight is being blocked by that object (Gregory, [21]).

Ray casting is a perfect solution when collision detection for a small and fast object is needed. A collision detection that depends on intersection of collusion volumes may fail for a bullet, because in that kind of systems, intersections are tested for an exact moment in time and repeated again when the positions of the objects are updated. However, a bullet may move too much more than its size until next check, and the collision probably be missed. With ray casting, the collision between a bullet and a collidable can be detected properly (Zerbst and Düvel, [20]).

Ray casting is also needed to do the picking of 3D objects, which actually means to be able to select objects by firing rays at them. In a 3D editor, for example, user may want to click on the screen and select the object the cursor is pointing at. 3D applications like CAD tools also use ray casting methods to provide that kind of features (Zerbst and Düvel, [20]). Ray casting against a spherical collidable can be done as (Figure 4.19.) (Eberly, [22] and Bergen, [31]):

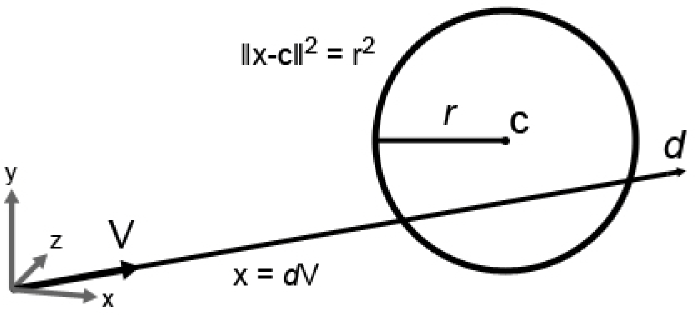


Figure 4.19. Ray – sphere intersection test

Equation for a sphere has center c, and radius *r*:

$$\|x-c\|^2 = r^2 \tag{4.8}$$

Equation for a ray, starting from origin, through the unit vector V to *d* distance long:

$$x = dV \tag{4.9}$$

Equations can be combined as:

$$\|d\mathrm{V} - \mathrm{c}\|^2 = r^2 \tag{4.10.a}$$

$$d^2\mathrm{V}^2 - 2d(\mathrm{V} \cdot \mathrm{c}) + \mathrm{c}^2 = r^2 \tag{4.10.b}$$

$$d^2\mathrm{V}^2 - 2d(\mathrm{V} \cdot \mathrm{c}) + \mathrm{c}^2 - r^2 = 0 \tag{4.10.c}$$

When solving the quadratic equation for *d*, two possible solutions can be shown as:

$$d = \frac{(\mathrm{V} \cdot \mathrm{c}) \pm \sqrt{(\mathrm{V} \cdot \mathrm{c})^2 - \mathrm{V}^2(\mathrm{c}^2 - r^2)}}{\mathrm{V}^2} \tag{4.11}$$

As it is known that, V is a unit vector and $\mathrm{V}^2 = 1$, the equation can be simplified as:

$$d = (\mathrm{V} \cdot \mathrm{c}) \pm \sqrt{(\mathrm{V} \cdot \mathrm{c})^2 - \mathrm{c}^2 + r^2} \tag{4.12}$$

Because of the principal square root, a solution can be defined only if the discriminant of the equation ($(\mathrm{V} \cdot \mathrm{c})^2 - \mathrm{c}^2 + r^2$) is equal or greater than zero. If the value is less than zero, than no solution exist, which means ray do not intersect with the sphere. If the value is equal to zero, than ray intersect with the sphere at single point, which also means ray is a tangent of sphere. If the value is greater than zero, than there are two solutions exist, which means ray intersect with the sphere at two points (Eberly, [22] and Bergen, [31]).

5. CASE STUDY: A PART OF “GALLIPOLI WARS”

5.1. Introduction

The Gallipoli Wars (also known as Gallipoli Campaign and known as Çanakkale Wars in Turkey) took place at Gallipoli peninsula in northwest Turkey from 1914 to 1916, during the First World War. The naval attack on Gallipoli was a joint operation by the British Empire (United Kingdom of Britain and Ireland with Australia, New Zealand, British India and dominion of Newfoundland) and France (France with French West Africa). Russia also participated to the joint naval force, but symbolically with one ship, which was one of Russia’s biggest cruisers, named “Askold”. Actual purpose of the operation was to take Ottoman Empire out of war and open a sea route to Russia by capturing the capital city of İstanbul through Dardanelles (Gallipoli Strait) and Sea of Marmara (Figure 5.1.a.) to gain control over both Dardanelles and Bosphorus (İstanbul Strait). The attempt failed with heavy casualties on both sides (Yılmaz, [32], Özakman, [33], Fewster et. al., [34] and Haythornthwaite, [35]).

The assault opened on 19 February 1915 with the long-range bombardment of the outer forts. Nine ships, including the British’s new battleship HMS Queen Elizabeth, which was the most powerful battleship floated until that day, engaged the forts for nearly eight hours but inflicted little real damage due to bad weather. When attack resumed on 25th of February, the forts were practically destroyed and their soldiers forced to withdraw. The way was now clear for the Allied fleet to enter the straits and attack the next line of forts. In İstanbul, preparations were made to evacuate the sultan, his court, the Treasury and the leading military and civilian authorities. Confidence was so high within Britain’s War Council that papers were circulated discussing what peace terms might be offered. On 4th of March, Admiral Carden presented his expectations to Churchill, stating that the fleet can be able to arrive in İstanbul within fourteen days (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

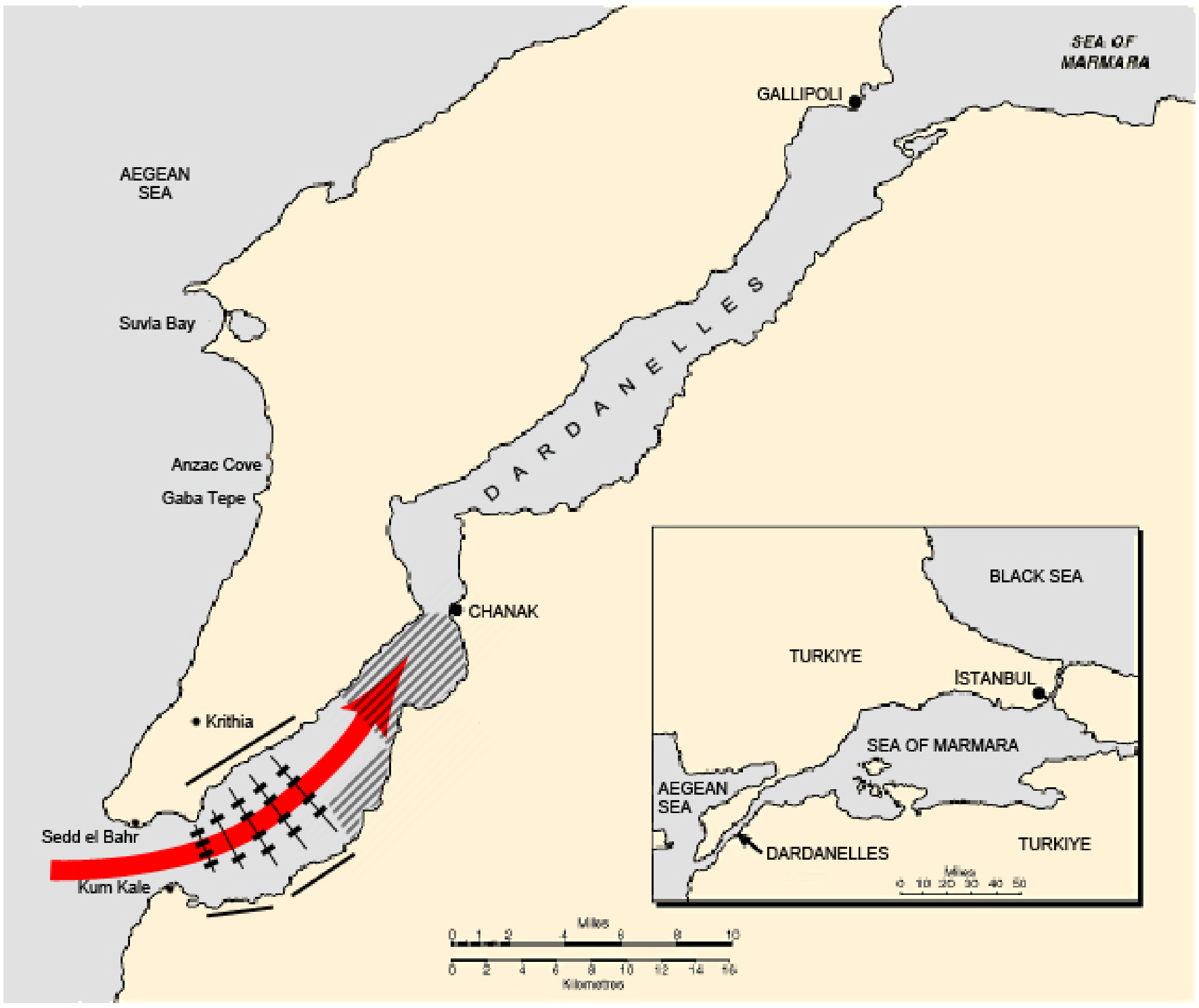


Figure 5.1.a. Initial attempt to pass through Dardanelles on 18 March 1915

The actual attempt to pass through Dardanelles started on 18th of March. Eighteen battleships supported by a group of cruisers and destroyers moved within close range of the shore guns. Simultaneously, trawlers started to scan the minefields. At first, the attack went well for the Allies and the forts were silenced by noon, but then French ship Bouvet hit one of the mines, laid previous night by Turkish minelayer Nusret, and sunk with its entire crew aboard. Then three British Battleships, HMS Irresistible, HMS Inflexible and HMS Ocean sustained critical damages because of gun fire and mines. HMS Inflexible managed to escape with other heavily wounded ships like French battleships Suffren and Gaulois. By late afternoon, one third of the fleet had been sunk or put out of action, yet no ship had even reached the narrows (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

Unexpected losses stopped the Allies to try any further attempts to force the straits by naval power alone and it was decided to land the ground forces to secure Dardanelles. After preparations done, the landing plan put into action on 25th of April. British soldiers landed on southwest tip of the Gallipoli peninsula under the command of Major-General Aylmer Hunter-Weston, designated from east to west as S, V, W, X and Y beaches. The Anzac (Australian and New Zealand Army Corps) troops landed on a small cove at north of Gaba Tepe (later named as Anzac Cove) to advance across the peninsula for possible reinforcements. The French made a diversionary landing at Kum Kale across the straits to keep Turkish forces on Asian shore busy (Figure 5.1.b.) (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

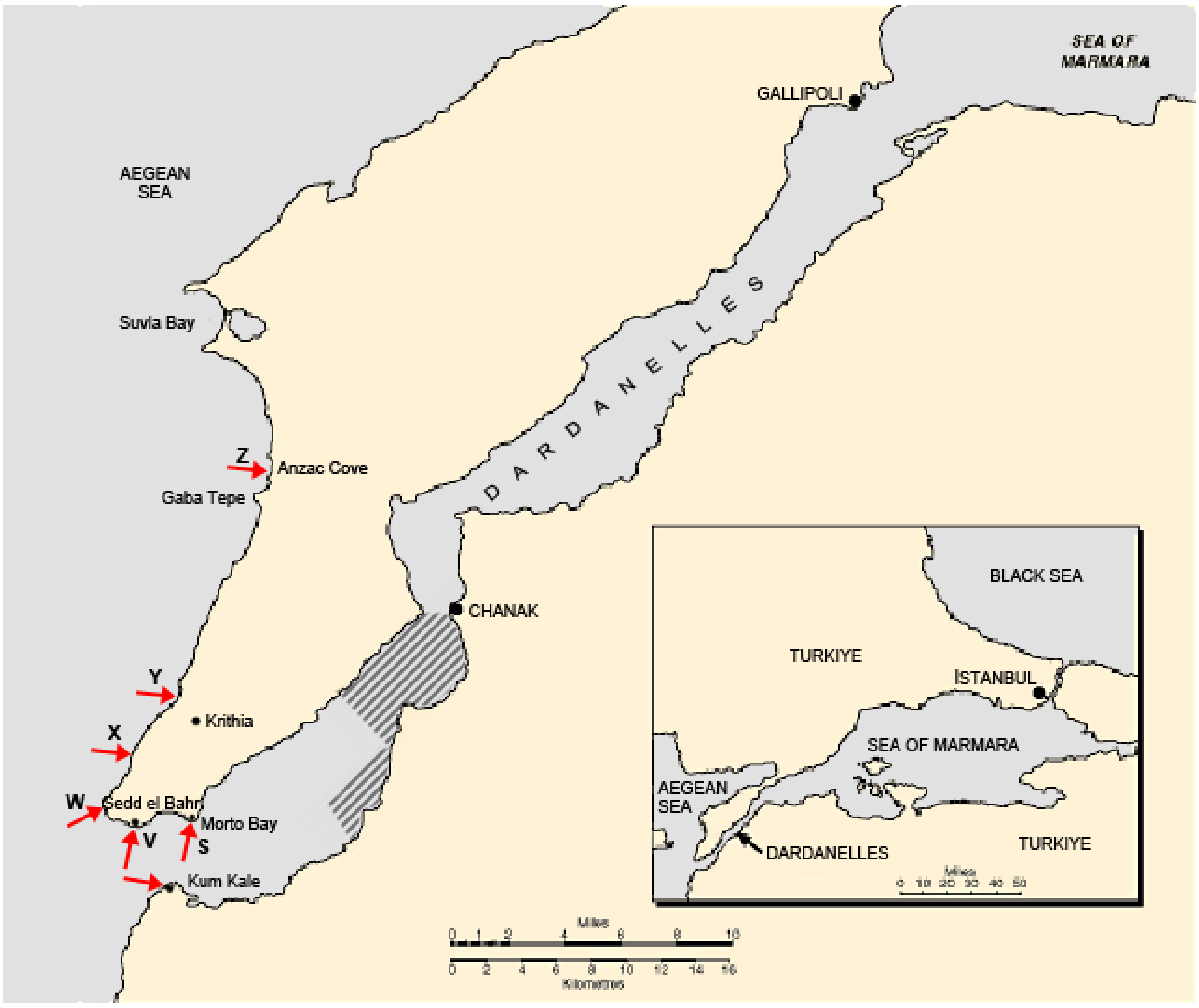


Figure 5.1.b. Landings on 25 April 1915

The main landings were the British landings at V Beach, beneath the old Seddülbahir fortress, and at W Beach at west. Turkish defense at W beach was weak, but the Lancashires[13] hardly overwhelmed the defenses with heavy losses, 600 killed or wounded, out of a total strength of 1,000. At V Beach the Royal Dublin Fusiliers[14] landed from open boats, the Royal Munster Fusiliers[15] and Royal Hampshires[16] were landed from SS River Clyde, which was a converted collier[17] used as a Trojan horse for landing the troops. Turkish defenders held their fire until the boats reached to the shore. When they opened fire, they caused heavy casualties due to the overcrowding of the boats. River Clyde safely carried 2,000 men to shore, when sally ports opened on each side of the ship and troops came running out, the Turkish soldiers turned all their fire against the River Clyde as they have already beaten off the boats. Out of the hundreds of soldiers to land, only 21 men made it onto the beach. When British canceled the landing by noon, thousands of soldiers were stuck inside the River Clyde and they waited until nightfall without making another attempt to get ashore from the ship (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

The landings at the three other beaches were made good with very few casualties. In fact, only four Turks were found near Y beach. S and X beaches were small landings on the flanks of the main V and W Beaches respectively. The troops landed at these beaches were the reserve divisions and they had no immediate objectives of their own, other than to secure their positions (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

The purpose of the Anzac landing was to overtake the Turkish forts that controlled the passage of the Dardanelles straits, but because of strong current, Anzacs have landed to a small shore surrounded by high grounds at 1.5 miles (2.4 km)

[13] The Lancashire Fusiliers was a British infantry regiment of the British Army (http://en.wikipedia.org/wiki/Lancashire_Fusiliers).
[14] The Royal Dublin Fusiliers was an Irish Infantry Regiment of the British Army (http://en.wikipedia.org/wiki/Royal_Dublin_Fusiliers).
[15] The Royal Munster Fusiliers was a regular infantry regiment of the British Army (http://en.wikipedia.org/wiki/Royal_Munster_Fusiliers).
[16] The Royal Hampshire Regiment was a British Army line infantry regiment (http://en.wikipedia.org/wiki/Royal_Hampshire_Regiment).
[17] Collier is a historical term used to describe a bulk cargo ship designed to carry coal, especially for naval use by steam-powered warships [http://en.wikipedia.org/wiki/Collier_(ship_type)].

north of their planned landing area. The troops were met by rifle and machine gun fire but the casualties remain relatively light. For eight month period, very severe battles occurred in the frontline of the Anzac. In certain times, opposing forces came very close to each other, as close as 25 feet (8 meter) on some locations, but battlefield remained almost same with the ground captured on the first day of the landing (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

Landings had continued on following months and reinforcements had kept coming, but the initial landings on 25th of April have a significant importance as the beginning of land wars. Severe battles happened until the end of the year, including (Yılmaz, [32], Özakman, [33], Fewster et. al., [34] and Haythornthwaite, [35]):

- 28th of April, First Battle of Krithia ("Birinci Kirte Muharebesi" in Turkish)
- 6th of May, Second Battle of Krithia ("İkinci Kirte Muharebesi" in Turkish)
- 4th of June, Third Battle of Krithia ("Üçüncü Kirte Muharebesi" in Turkish)
- 28th of June, Battle of Gully Ravine ("Zığındere Muharebesi" in Turkish),
- 6th of August, Battle of Sari Bair ("Birinci Anafartalar Muharebesi, Sarı Bayır Harekatı" in Turkish),
    - Battle of Krithia Vineyard ("Birinci Anafartalar Muharebesi" in Turkish)
    - Battle of Lone Pine ("Kanlısırt Muharebesi" in Turkish),
- 8th of August, Battle of Chunuk Bair ("Conk Bayırı Muharebesi" in Turkish),
- 21st of August, Battle of Scimitar Hill ("İkinci Anafartalar Muharebesi" in Turkish)
- 21st of August, Battle of Hill 60 ("İkinci Anafartalar Muharebesi" in Turkish)

For the most of the countries, the Gallipoli Wars is remembered as just another name in a long list of World War I battles, but for Turks, Australians and New Zealanders, Gallipoli meant a significant event for the birth of their national consciousness. Gallipoli has a special place in their national memory. Each year, Australians and New Zealanders remember their war dead on 25th of April, the date Anzac troops first landed on the Gallipoli peninsula. The occasion is set aside as a public holiday, with veterans' marches and memorial services held in large cities and small towns across the land. In Turkey, the annual commemoration centers around the decisive victory won over the British and French fleet on 18th of March. It is not a public holiday but senior government and military figures attend special ceremonies in and around Çanakkale. Turks also remembers the wars for launching the career of Mustafa Kemal, the young officer who went on to become Atatürk, the savior of Turkish nation and founder of modern Turkey (Yılmaz, [32], Özakman, [33] and Fewster et. al., [34]).

"Gallipoli Wars" real-time strategy game was thought to cover all of the wars that took place at Gallipoli between 1914 and 1916, during the First World War. However, it is not possible to develop a full length 3D strategy game without a team effort that will last for years. Therefore, a single-level prototype, covering the British landing at V Beach (at southwest tip of Gallipoli peninsula) on 25th of April, was developed within the scope of this study. The purpose of this case study is to demonstrate the major game mechanics and functionalities that must be developed and contents that must be created for a typical 3D real-time strategy game.

5.2. Unity Game Engine

When choosing a game engine for a project, there are many things which must be taken into account. What kind of game will be developed heavily affects the game engine preference. It is a matter of comparing game design vs. game engine feature support. The engine should support the main elements the game design but not necessarily all of them. So for example if the game is heavily based on good physics simulation, then choosing an engine that supports it already or an engine that is easy to add the required feature to it will reduce the required development time significantly (Manninen et. al., [13]).

Today, there are many game engines available for both professional and independent developers. Some of them are:

- Unreal Engine 3 (by Epic Games)[18]
- Torque 3D (by Garage Games)[19]
- Unity (by Unity Technologies)[20]
- Visual3D.Net (by Realmware Corporation)[21]
- Gamestudio (by Conitec Datensysteme GmbH)[22]
- Delta3D (by The MOVES Institute at Naval Postgraduate School)[23]

Many advantages and disadvantages can be found when these game engines compared to each other, in terms of feature sets, documentation and licensing options. Unreal Engine 3 for example, is a very robust and proven game engine used by many companies including well known game developers, but it is an expensive engine with high percent royalty fees. On the other hand, Torque 3D, Unity, Visual3D.Net and Gamestudio are much more affordable engines which have almost similar features (Table 5.1.). Delta3D developed by the US Navy can be good option for HLA (High Level Architecture) based military simulations or other projects that is requiring low level access.

As it can be seen in the Table 5.1., which is prepared according to the features lists and documents obtained from various sources[24], four of the engines have almost similar features, but Unity engine is preferred for this study because of having better documentation and user friendly built-in editors. Despite the lack of advanced debugging tools, like breakpoint or stepping (as of v2.6.1), having a version available for free of charge and already implemented Nvidia PhysX physics engine can be considered as a plus for this engine.

---

[18] http://www.unreal.com/technology.php
[19] http://www.torquepowered.com/products/torque-3d
[20] http://unity3d.com/unity/engine/
[21] http://www.visual3d.net/game-engine/features
[22] http://www.conitec.net/english/gstudio/3dgs7.php
[23] http://www.delta3d.org/article.php?story=20051209133127695&topic=docs
[24] Documents are obtained from the official developer websites. The feature sets are obtained from both official developer websites and the 3D engines database at DevMaster.net (http://www.devmaster.net/engines/)

Table 5.1. A comparison chart of four game engines

	Unity	Torque 3D	Visual3D.NET	Gamestudio
Programming Language	C#, Mono, Boo, Javascript	C++, Torquescript	C#, VB.NET, Python, Lua	C++, C-Lite
Built-in Editors	Scene Editor, Terrain Editor, Custom editors	Scene Editor, Terrain Editor, GUI Editor	Scene Editor, Terrain Editor, GUI Editor	Scene Editor
Scene Management	Occlusion Culling	Portals, Occlusion Culling, LOD	Portals, Occlusion Culling, LOD	BSP Trees, LOD
Rendering	Fixed-function, Flexible, Render to Texture, GUI	Fixed-function, Flexible, Render to Texture, GUI	Fixed-function, Flexible, Render to Texture, GUI	Fixed-function, Flexible, Render to Texture, GUI
Lighting	Per-vertex, Lightmap	Per-vertex, Lightmap	Per-vertex, Lightmap	Per-vertex, Lightmap
Shadows	Projected Planar	Projected Planar	Projected Planar	Projected Planar
Texturing	Multitexturing, Mipmapping	Multitexturing, Mipmapping	Multitexturing, Mipmapping	Multitexturing, Mipmapping
Animation	Skeletal, animation blending	Skeletal, Per-vertex, animation blending	Skeletal, Per-vertex, animation blending	Skeletal, Per-vertex, animation blending
Terrain Engine	Hightmap based, Continuous LOD	Hightmap based, Continuous LOD	Hightmap based, Continuous LOD	Hightmap based, Continuous LOD
Physics	Nvidia PhysX	Rigid body physics	Rigid body physics	-
Networking	Client-Server, UDP-TCP	Client-Server, UDP-TCP	Client-Server, UDP-TCP	Client-Server, UDP-TCP
Sound & Video	2D and 3D sound, Streaming audio and video	2D and 3D sound, Streaming audio and video	2D and 3D sound, Streaming audio	2D and 3D sound
Documentation	Excellent	Very Good	Poor	Good

5.3. Design and Development of Case Study

5.3.1 RTS camera

As previously mentioned, game mechanics describe the goal of a game, what players can and cannot try to achieve it, and what happens when they try. Both design and the implementation of game mechanics vary from game to game, but there are also very similar mechanics, common for every games of same genre. A camera of a 3D RTS (real-time strategy) game is a very good example to these mechanics. A 3D strategy game can be about wars in a futuristic world or in medieval Europe, the gameplay may or may not include production or resource gathering. It does not matter. They both need a camera which allows players easily navigate through the 3D game world to command units. The camera mechanics of these games are generally designed in similar ways. Actually, almost every 3D application shares similar mechanics when it comes to camera. The most important reason of this is to offer players what they are familiar with, and not forcing them to learn and get used to new things when it is not really necessary.

It is assumed that the RTS camera in Gallipoli Wars must have following features:

- Horizontal movement
    - In all directions (360°) via dragging cursor to screen borders
    - In eight directions via arrow keys on keyboard by default
    - Varying movement speed according to zoom amount

- Orbital rotation around focus point
    - Unlimited horizontal rotation (360°) around global y-axis (vertical axis) via clicking and holding mouse wheel while dragging cursor horizontally
    - Limited vertical rotation (0° to 90°) around local x-axis (horizontal axis) via clicking and holding mouse wheel while dragging cursor vertically

- Zoom-In and Zoom-Out
    - Smooth zoom (with acceleration and deceleration) via mouse wheel

- Collision avoidance against terrain
    - Changing camera height according to terrain topography

Camera features are implemented via CameraMovement class, details of which can be found on Appendix A, can be summarized as:

```
using UnityEngine;
using System.Collections;

// MonoBehaviour is the base class for every classes, that instances of which must be
// assigned to a game object as a component. Callback methods of these objects are
// triggered by the engine.
// The only instance of CameraMovement is assigned to cameraContainer game
// object, which is  the root node of the camera system and focus point of the camera

public class CameraMovement : MonoBehaviour
{
   // Fields (for movement, rotation, zoom, screen resolution, camera and
   // camera related game objects)
```

Figure 5.1.a. Class definition part of CameraMovement class

```
   // Start method is used for initialization.
   // Start is only called once in the lifetime of the instance and called just before any
   // of the Update methods is called the first time

   public void Start()
   {
     // Get screen resolution of current system.
     // Initialize resolution related variables.
     // Find camera (and camera related game objects) in the scene.
     // Initialize other variables.
   } // END OF Start
```

Figure 5.1.b. Initialization method of CameraMovement class

```
// Update method is called every frame, if the MonoBehaviour is enabled.
// It is the most commonly used method to implement any kind of game behavior.
public void Update()
{
   // ZOOM
   mouseWheelValue = GetMouseWheelValue();
   if (mouseWheelValue != 0.0f)
   {
      this.ZoomCamera(mouseWheelValue);
   }
   // MOVE
   if (isCursorOnBorders() && !isCameraRotate)
   {
      MoveCameraWithMouse();
   }
   else if (Input.GetButton("MoveForward") || Input.GetButton("MoveBackward")
            || Input.GetButton("MoveLeft") || Input.GetButton("MoveRight"))
   {
      if (Input.GetButton("MoveForward"))
      {
         MoveCameraWithKeys (Vector3.forward);
      }
      else if (Input.GetButton("MoveBackward"))
      {
         MoveCameraWithKeys (Vector3.back);
      }
      if (Input.GetButton("MoveLeft"))
      {
         MoveCameraWithKeys (Vector3.left);
      }
      else if (Input.GetButton("MoveRight"))
      {
         MoveCameraWithKeys (Vector3.right);
      }
```

Figure 5.1.c. Update method of CameraMovement class

```
        // SHIFT
        // Check camera position and terrain topography beneath camera and
        // shift camera to an appropriate position if necessary.

        // ROTATION
        if (Input.GetButtonDown("CameraFunction"))
        {
            mousePosionOnPressCameraFunction = Input.mousePosition;
            isCameraRotate = true;
        }

        if (Input.GetButtonUp("CameraFunction"))
        {
            isCameraRotate = false;
        }

        if (isCameraRotate)
            RotateCamera();
    }  // END OF Update

    // Private methods (MoveCameraWithMouse, RotateCamera, ShiftCamera etc.)

}  // END OF CameraMovement CLASS
```

Figure 5.1.d. Update method (continuation) of CameraMovement class

5.3.2. Player commands

Navigating the camera easily is an essential feature for a RTS game, but it is not enough to be able to play the game. A player needs additional features such as being able to select units and giving them orders like move or attack. Except some RTS games for consoles, these features are used with procedures, which can be done with mouse actions.

Gallipoli Wars has the following features for commanding the units:

- Unit selection
    - Selecting a unit by clicking on (equivalent of picking in computer graphics)
    - Selecting a group of units by clicking and holding the left mouse button and dragging to draw a selection rectangle

- Unit commands
    - Move command by clicking on terrain while there is one or more units selected
    - Attack command by clicking on an enemy unit while there is one or more units selected

These features are implemented via PlayerCommand and UnitManager classes, details of which can be found in Appendix B and C, can be summarized as:

```
using UnityEngine;
using System.Collections;
public class PlayerCommands : MonoBehaviour
{
   // Fields

   // OnGUI is called for rendering and handling GUI events.
   // It is called whenever an event (like user inputs or rendering events) occurs.
   // This means OnGUI might be called several times per frame.
   public void OnGUI()
   {
      // Draw selection rectangle on GUI layer with diagonal from the position
      // (screen position) first clicked to the current position of the cursor
   }  // END OF OnGUI
```

Figure 5.2.a. Class definition and GUI method of PlayerCommands class

```
    public void Update()
    {
        if (Input.GetButtonDown("Fire1"))  // Left mouse button pressed
            Mouse1Down(Input.mousePosition);
        if (Input.GetButtonUp("Fire1"))  // Left mouse button released
            Mouse1Up(Input.mousePosition);
        if (Input.GetButton("Fire1"))  // Mouse dragged
            MouseDrag(Input.mousePosition);
    }  // END OF Update

    private void Mouse1Down(Vector2 mousePosition)
    {
        // Get cursor position, which may become needed if player drags the mouse to
        // draw a selection rectangle.
    }
    private void Mouse1Up(Vector2 mousePosition)
    {
        // Cast a ray from camera through cursor position to check if player clicked on
        // one of selectable units. If so, then clear current selection and select the unit.
        // If player clicked on one of the enemy units, then selected units must attack it.
        // If player clicked on terrain, then selected units must move that position.
        // (see MoveSelectedUnitsToPoint and AttackTarget methods of UnitManager)
    }
    private void MouseDrag(Vector2 mouseCurrentPos)
    {
        // Casts four rays from camera through the four corners of the selection
        // rectangle to calculate the projection of the selection rectangle from screen to
        // terrain and tells UnitManager to select the units within this area
        // (see SelectUnitsInArea method of UnitManager)
    }
    // Other private methods
}  // END OF PlayerCommands CLASS
```

Figure 5.2.b. Update the other methods of PlayerCommands class

UnitManager singleton, which can be considered as a layer between PlayerCommands and units, is designed for holding and managing the collections of selectable and selected units and sending messages (like attack or move) to units which are selected.

```
using UnityEngine;
using System.Collections;
public class UnitManager : MonoBehaviour
{
   // Selectable and selected units collections
   // Other fields

   public void AddUnit(GameObject gameObject)
   {
      // Adds unit (gameObject) to the selectable units collection.
      // (When scene loaded, all units send a message to add themselves to the
      // collection.)
   }
   public void RemoveUnit(GameObject gameObject)
   {
      // Removes unit (gameObject) from the selectable units collection.
      // (Units send a message to remove themselves when they die.)
   }
   public void AddSelectedUnit(GameObject gameObject, string unitTag)
   {
      // Adds unit (gameObject) to the selected units collection.
      // (PlayerCommands calls this when player make a selection operation.)
   }
   public void RemoveUnitFromSelectedUnitList(GameObject gameObject)
   {
      // Removes unit (gameObject) from the selected units collection.
   }
   public void ClearSelectedUnitList(string unitTag)
   {
      // Removes all units from the selected units collection.
   }
```

Figure 5.3.a. Class definition and unit management methods of UnitManager class

```
   public void SelectUnitsInArea(Vector3[] points)
   {
      // Selects units within the rectangular area defined by four corner points.
      // For every unit in the selectable units collection, it creates four vectors from the
      // position of the unit to the position of the one of four corner points. If sum of the
      // angles between four vectors is equal to 360°, unit becomes selected.
      // (See MouseDrag method of PlayerCommands)
   }

   public void MoveSelectedUnitsToPoint (Vector3 destinationPoint,
                                         UnitMove.MoveCommand command,
                                         string unitTag)
   {
      // Sends a message for every unit in selected units collection to set them in
      // move state. (See Mouse1Up method of PlayerCommands)
   }

   public void AttackTarget (GameObject target, string unitTag)
   {
      // Sends a message for every unit in selected units collection to set them in
      // attack state. (See Mouse1Up method of PlayerCommands)
   }
   // Other private methods
} // END OF UnitManager CLASS
```

Figure 5.3.b. Unit command methods of UnitManager class

5.3.3. Finite-state machine

After translating player inputs into orders for units successfully achieved, then the response to these orders and the other behaviors of units must be programmed. As programming the behaviors of units is in the scope of game AI, it must be implemented with one of the AI techniques. There are many techniques for creating the artificial intelligence or illusion of intelligence, like genetic algorithms and neural networks, but in games, the most widely used one is finite-state machine (FSM) (Bourg and Seeman, [30] and Buckland, [36]).

A finite-state machine is a device, or a model of a device, which has a finite number of states that can be in at any time and can operate on input to either make transitions from one state to another or to cause an action to take place. A finite-state machine can only be in one state at any time (Buckland, [36]).

Finite-state machines have been around for a long time, but they are still quite common and useful in modern games because they are relatively easy to understand, implement, debug and expand. They are also computationally cheap because they essentially follow hard-coded rules (Bourg and Seeman, [30]).

The idea behind a finite-state machine is to decompose an object's behavior into easily manageable states. The light switch on a wall, for example, is a very simple finite-state machine. It has two states: on and off. Transitions between states are made by the input from a finger touch. By flicking the switch up it makes the transition from off to on, and by flicking the switch down it makes the transition from on to off (Buckland, [36]).

There are many ways of implementing finite-state machines. A naive approach is to use a series of if-then statements or the slightly better mechanism of a switch statement. As more states and conditions are added, these sort of structures can easily become complicated and difficult to understand or debug. A better mechanism for organizing states and managing state transitions is known as state transition table (Table 5.3.3.). This table can be queried by an agent at regular intervals, enabling it to make any necessary state transitions based on the events happened on the game environment. Another alternative is an architecture known as state design pattern. In this approach there is no separate structure or class for transitions, instead transition rules are embedded within states (Buckland, [36]).

Table 5.2. A simple example of state transition table

Current State	**Condition**	**State Transition**
Runaway	Safe	Patrol
Attack	WeakerThanEnemy	RunAway
Patrol	Threatened AND StrongerThanEnemy	Attack
Patrol	Threatened AND WeakerThanEnemy	RunAway

In Gallipoli Wars, a different approach is used to make the state machine usable for both player-controlled and computer-controlled units. To achieve this, major transition procedures and rules are separated from the states. Player controlled units are changing their states according to player inputs, which handled with PlayerCommands and UnitManager classes as shown in Figure 5.4. Computer controlled units, on the other hand, are changing their states according to a mission based system, which will be covered in next section.

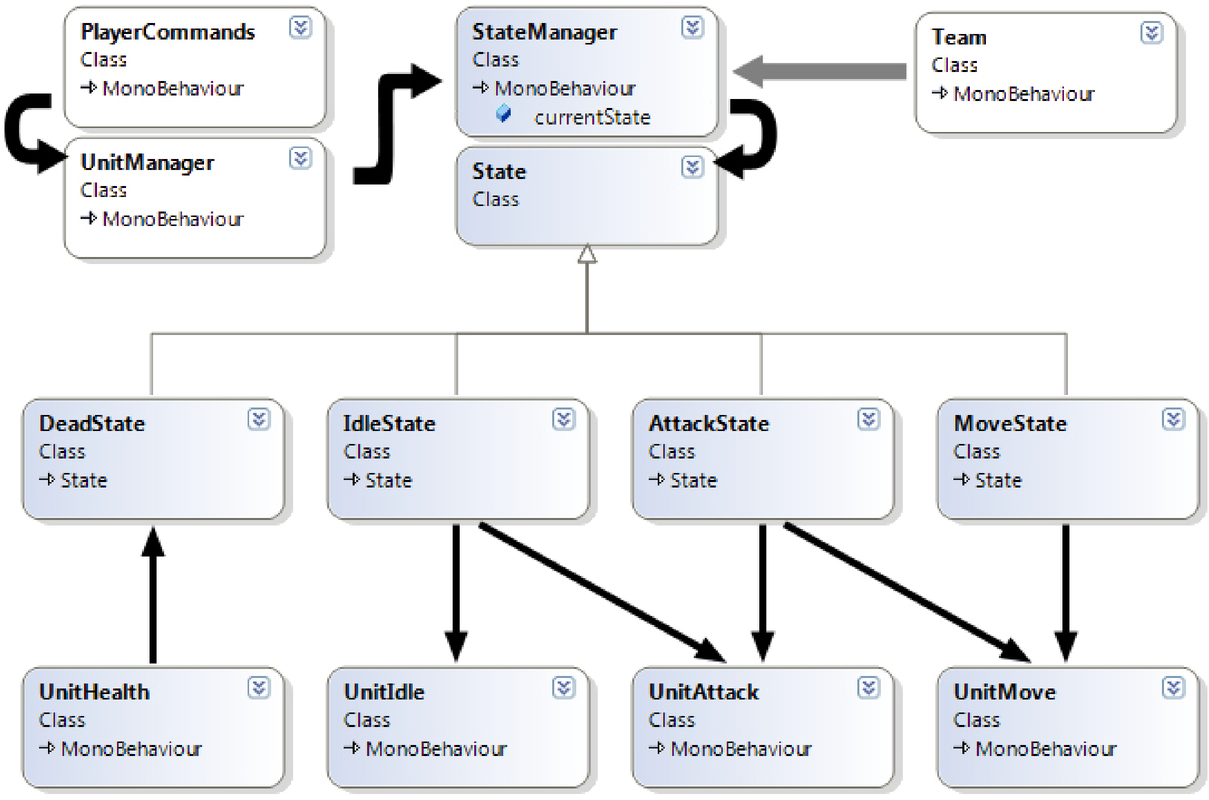


Figure 5.4. Finite-state machine used for units in Gallipoli Wars

In Gallipoli Wars, every unit has an instance of StateManager class, which is responsible for holding and executing current state, making state transition procedure by calling “OnExit” and “OnEnter” methods of the states and turning back to idle state when another state completed its job. As previously mentioned, major transition rules are separated from the state machine, but one simple transition rule is implemented within the StateManager, which is, if current state sets its “isActive” field as false (for example, when an attacker unit destroy its target or when a moving unit arrive its destination), then StateManager creates an idle state and sets it as current state. In this way, units have the ability to return to idle state per se. Other intrinsically triggered transition rule is switching to dead state when unit’s health drops to zero. It is implemented within the UnitHealth class. Because of other transitions are declared from outside, a very simple StateManager class, which is given in Figure 5.5.a. and Figure 5.5.b., became adequate for managing states.

```
using UnityEngine;
using System.Collections;
public class StateManager : MonoBehaviour
{
    public State currentState;

    public void Start()
    {
        this.currentState = new IdleState(this.transform.gameObject);
        this.currentState.OnEnter();
    }
```

Figure 5.5.a. Class definition and initialization method of StateManager class

```
    public void Update()
    {
        if (currentState.isActive)
            currentState.Execute();
        else
        {
            if (!(currentState is DeadState))
                this.ChangeState(new IdleState(this.transform.gameObject));
        }
    }
    public void ChangeState(State nextState)
    {
        currentState.OnExit();
        currentState = nextState;
        currentState.OnEnter();
    }
}  // END OF StateManager CLASS
```

Figure 5.5.b. Update and state transition methods of StateManager class

State machine consists of four states, which are IdleState, MoveState, AttackState and DeadState. Each of them is inherited from State base class (Figure 5.6.).

As it can be seen in State base class, states have three inherited methods, which are OnEnter, OnExit and Execute. OnEnter and OnExit methods are calling from StateManager, only when a transition occurs. They are used for initialization and finalization purposes. Execute method is calling from StateManager, in each frame, and it is used for implementing the actual behavior of the state.

```
using UnityEngine;
using System.Collections;
public class State
{
   public bool isActive = true;
   public GameObject go;
   public virtual void OnEnter()
   {
      throw new System.NotImplementedException();
   }
   public virtual void OnExit()
   {
      throw new System.NotImplementedException();
   }
   public virtual void Execute()
   {
      throw new System.NotImplementedException();
   }
}  // END OF State CLASS
```

Figure 5.6. State base class

State classes are given in Appendices. What they do can be summarized as follows:

- Idle State
    - Searches for nearby enemies
    - Listens for “under attack” event
    - Attacks to enemies (without chasing them) if there are any found
    - Performs other tasks (playing animations, sounds, etc.)

- Move State
    - Creates checkpoints
    - Moves towards the destination
    - Performs other tasks (playing animations, sounds, etc.)

- Attack State
    - Moves towards (and chases) the target
    - Attacks to the target
    - Performs other tasks (playing animations, sounds, etc.)

- Dead State
    - Replaces the unit with its dead version

As it can be seen, states have some similar abilities. For example both idle state and attack state have attacking ability and both move state and attack state have movement ability. To prevent code duplication and to simplify states, the functionalities that are needed to create the defined behavior (like applying Slerp, which is shorthand for spherical linear interpolation, to slowly rotate the unit towards destination direction or applying force to put the unit in motion etc.) are moved to separate classes (UnitMove, UnitAttack, and UnitIdle) and only the part that defining the actual behavior is left in states. In other terms, what does a state do, is defined in state classes and how they do it, is implemented in separate classes.

5.3.4. Scenario system

As it is known, a scenario is a sequence of events or actions. Many games and simulations require a kind of scenario system to trigger predefined events when certain conditions are met. This may include triggering the actions of game objects (e.g. opening of a door when a puzzle solved), changing game world parameters (e.g. weather condition), pausing game to play a cutscene (a live action video or animation) or loading the next level when current level has finished.

There are many ways to implement a scenario system. According to the specifications of the application, an XML (Extensible Markup Language) based scenario definition language (SDL) can be used with required parsing, processing and interpreting functionalities. For example, distributed simulations generally require offering scenario creation and management abilities to end-users. A simulation trainer demands to be able to create custom scenarios and to manipulate them at runtime with event injection (generating unscheduled events during the execution) functionalities (Topçu and Oğuztüzün, [37]). For Gallipoli Wars, there are no such requirements, but it is clear that a practical scenario system may provide many advantages during development and test. The scenario system for Gallipoli Wars is not based on a scenario definition language; instead it is designed to gain an advantage from the abilities of the engine's editors, like object instantiation by drag and drop or field initialization through dropdown lists.

The scenario system is designed in an extensible manner but only used for executing series of missions, which can be thought as operational plans for computer controlled armies. Missions are started from ScenarioManager object and they have a sequential list of objectives for teams (Figure 5.7.). Currently, there are three types of objectives:

- ProceedToDestination
    - Controls teams to move along given path
- Wait
    - Controls teams to hold their positions for a given duration
- AttackTargets
    - Controls teams to chase and attack given enemies

As shown in Figure 5.7., a mission also holds a group of teams and directs them through the objectives, via adding or removing them to the teamCollection field of the objectives. As previously mentioned objectives list of a mission is a sequential list, which means teams follows them in order from beginning to end. When a team completed its objective, its mission removes it from the collection of its current objective and adds it to the collection of next objective. If a team comes across enemy units during an objective, mission removes the team from its current objective, creates an encounter object and adds the team to it. Encounter objects are like objectives, but they are created and destroyed at runtime. Encounter object controls the team until the threat is eliminated. After then, mission object adds the team back to its last objective.

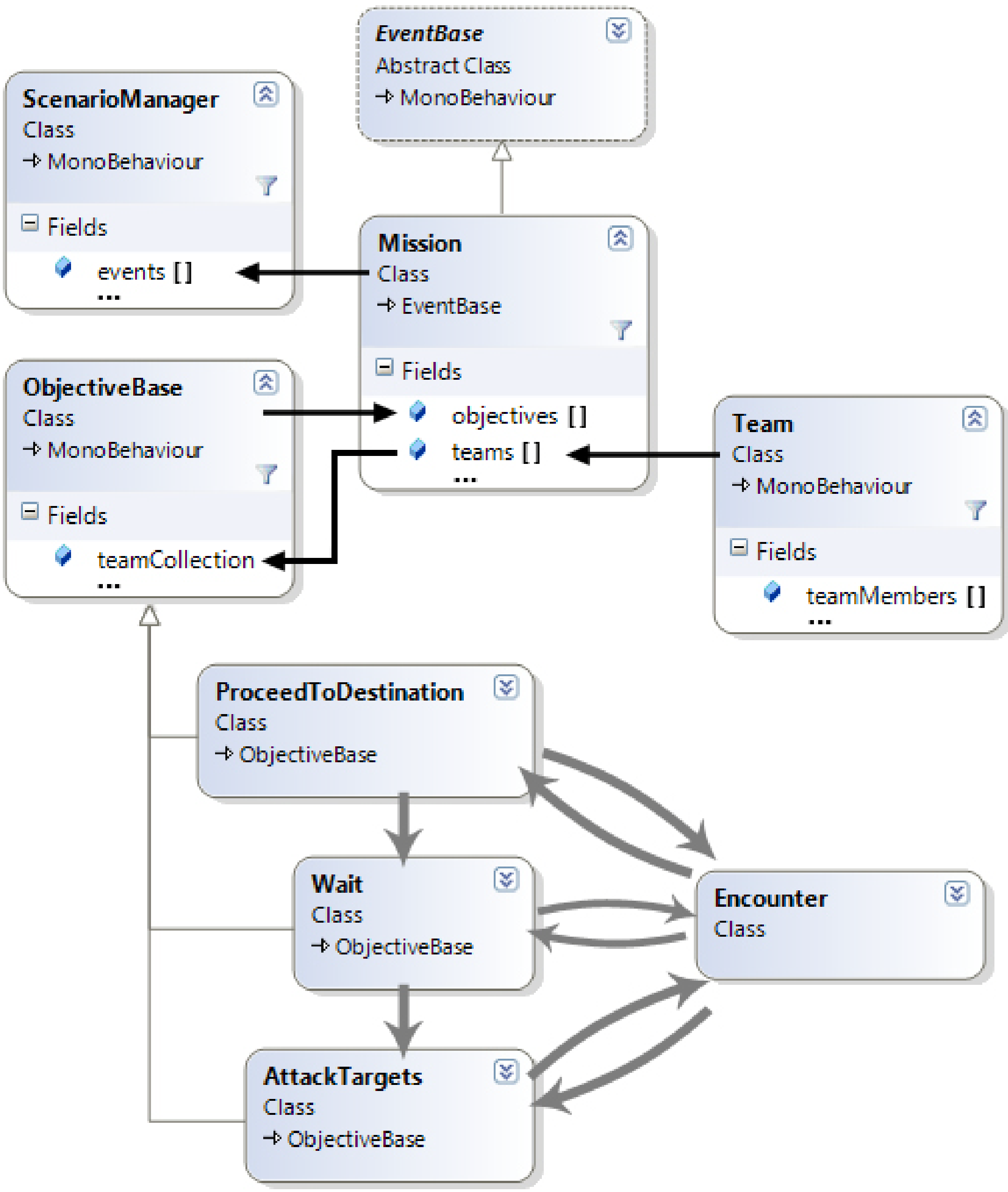


Figure 5.7. Scenario system in Gallipoli Wars

As previously mentioned, scenario system is designed to take advantage of the capabilities of engine's editors. Therefore, scenario definition procedure is heavily depending to the use of these editors. Each scene must have a scenario, which consists of an instance of the ScenarioManager class and one or more instances of event class (Mission), objective classes (ProceedToDestination, Wait and AttackTargets) and Team class. Each of them can be instantiated by dragging from project pane to scene or hierarchy pane (Figure 5.8.a.).

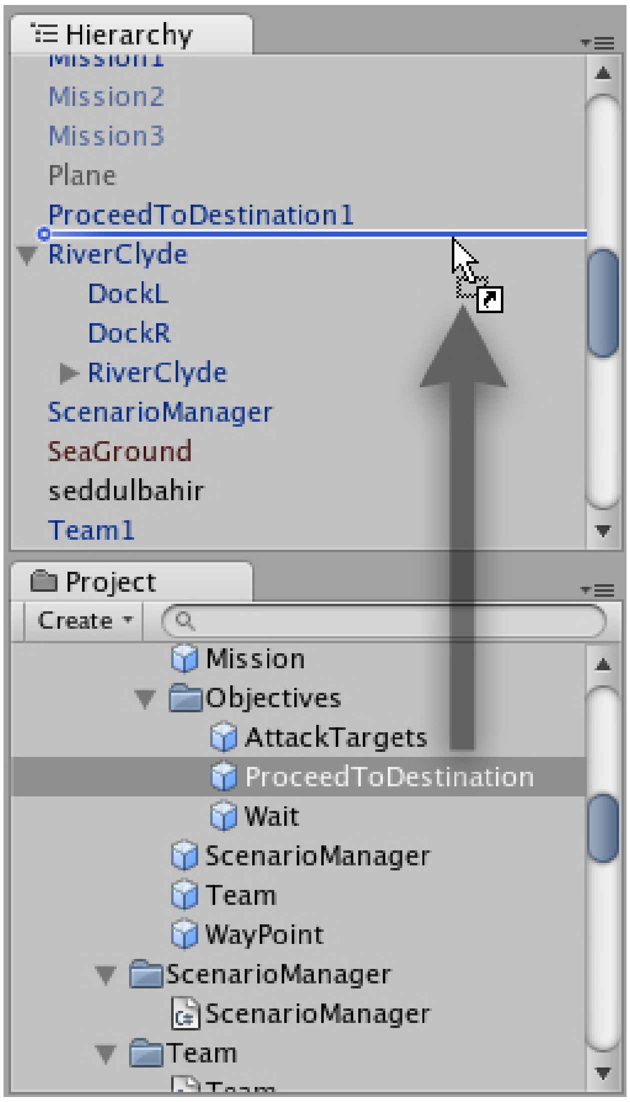
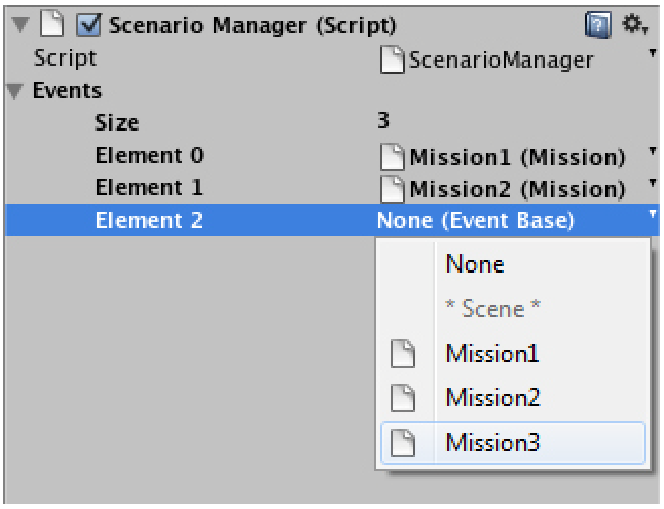


Figure 5.8.a. Instantiating an objective

After necessary scenario objects are instantiated, they must be connected as they form a system like the one shown in Figure 5.7. Only mission instances added to Events array of the scenario manager can be started and executed. Therefore, array size must be set equal to the amount of mission instances and they must be added by selecting from the dropdown list as shown in Figure 5.8.b.



Scenario Manager (Script)

Script ScenarioManager

Events

Size 3

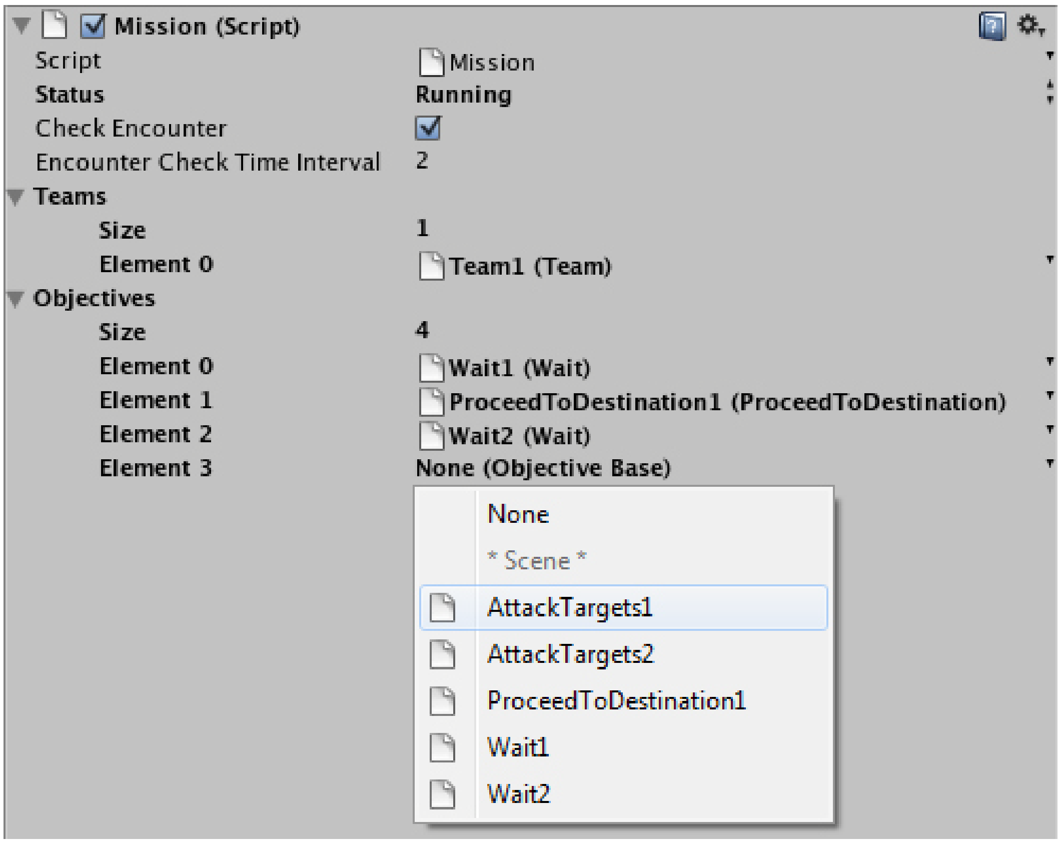
Element 0 Mission1 (Mission)

Element 1 Mission2 (Mission)

Element 2 None (Event Base)

Figure 5.8.b. Adding missions to scenario manager object

Similarly, teams and objectives arrays of a mission object must be initialized as shown in Figure 5.8.c. The objectives array of a mission defines the objectives must be accomplished by teams sequentially. When a mission started, each team added starts to first objective and goes to next one only after current objective completed successfully. Therefore, ordering of objectives is important for missions.



Mission (Script)

Script Mission

Status Running

Check Encounter ✓

Encounter Check Time Interval 2

Teams

Size 1

Element 0 Team1 (Team)

Objectives

Size 4

Element 0 Wait1 (Wait)

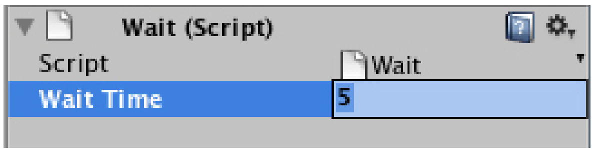
Element 1 ProceedToDestination1 (ProceedToDestination)

Element 2 Wait2 (Wait)

Element 3 None (Objective Base)

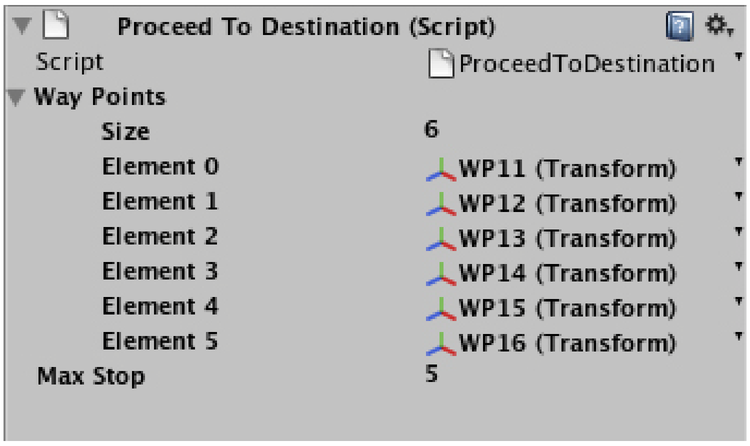
Figure 5.8.c. Adding teams and objectives to a mission object

Team instances have an array named members, which must be initialized by adding units. Objective instances, on the other hand, have different fields according to their purposes. Wait objectives only have a float field for waiting duration in seconds (Figure 5.8.d.). Proceed to destination objectives have an array of sequential transforms (empty game objects that only have position, rotation and scale properties) as waypoints to be followed by teams (Figure 5.8.e.). Attack to targets objectives have an array of enemy units to be chased and attacked by teams (Figure 5.8.f.). Both of these array fields can be initialized as before.



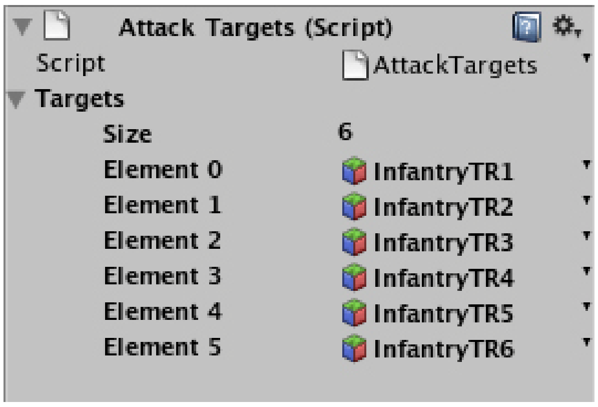
Wait (Script)
Script Wait
Wait Time 5

Figure 5.8.d. Setting waiting duration of a wait objective



Proceed To Destination (Script)
Script ProceedToDestination
Way Points
Size 6
Element 0 WP11 (Transform)
Element 1 WP12 (Transform)
Element 2 WP13 (Transform)
Element 3 WP14 (Transform)
Element 4 WP15 (Transform)
Element 5 WP16 (Transform)
Max Stop 5

Figure 5.8.e. Adding waypoints to a proceed to destination objective



Attack Targets (Script)
Script AttackTargets
Targets
Size 6
Element 0 InfantryTR1
Element 1 InfantryTR2
Element 2 InfantryTR3
Element 3 InfantryTR4
Element 4 InfantryTR5
Element 5 InfantryTR6

Figure 5.8.f. Adding targets to an attack targets objective

5.4. Visual Contents of Gallipoli Wars

As previously noted, game development includes the creation of audio-visual contents. Although it is very important to support visuality with sound effects and music, there is no audio content created or used for this study, except a gunshot sound.[25] On the other hand, several visual contents, including 3D models, animations and 2D images were created. Details of these are given in this section.

5.4.1. Turkish infantry

A 3D infantry model, that is reflecting the characteristics of Turkish soldiers of the period, has been modeled, textured and animated by using several 2D and 3D tools. Modeling workflow varies according to requirements of the game and the model but a sophisticated workflow for a realistic game character consists of following steps:

- Modeling the base mesh: Creating a rough, low-poly mesh (3D geometry consists of several thousands of triangles);
- Sculpting (digital sculpting): Creating high-poly mesh (3D geometry consists of hundreds of thousands of triangles or more) by detailing the base mesh;
- Retopologizing: Creating the final low-poly mesh that represents high-poly mesh as close as possible;
- UV mapping: Preparing UV layout data of the final model;
- Generating maps: This step includes the creation of maps that are generally obtained from 3d models by using specialized tools or algorithms. Some examples are:
    - Height maps (e.g. Bump map, normal map): Contains the information of elevation;
    - Ambient occlusion map: A Map that contains the information of light distribution over the surfaces of the model;
    - Reflection map: Rendered images of surrounding environment;

[25] Gunshot sound sample courtesy of Gezortenplotz, taken from “The Freesound Project”, which is a collaborative database of Creative Commons licensed sounds (http://freesound.iua.upf.edu/).

- Creating maps: This step includes the creation of manually painted maps. Some examples are:
    - Texture map (Diffuse map or color map): Contains the information of color;
    - Specular Map: Contains the information of shiny and matte areas of the model;
    - Opacity map (Alpha map or transparency map): Contains the information of transparent, translucent and opaque areas of the model;
- Rigging: Process of preparing the skeletal structure and the skin weight table, which is a data that contains the information of vertex-bone relation;
- Animating: Process of creating animations like walk-cycle, run-cycle, idle etc.

There are many modeling tools available to create the base mesh, including Autodesk's 3D Studio Max, Maya and Softimage, Maxon's Cinema 4D, NewTek's Lightwave. Base character mesh is created with 3D Studio Max. Modeling progress and how vertex, edge and triangle counts have gradually increased can be seen in Figure 5.9.a.

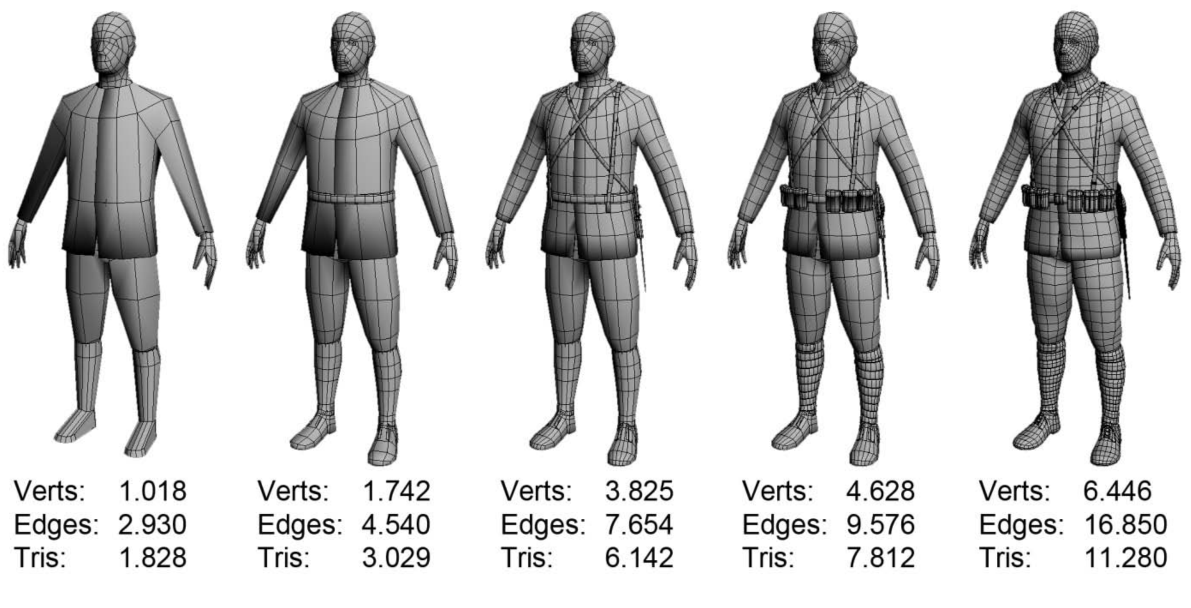


Figure 5.9.a. Modeling the base mesh

After the creation of base mesh has been completed, the model has been transferred into a digital sculpting tool, Autodesk's Mudbox, for adding further details by subdividing (dividing each polygon into four quadrants) several times and sculpting the high-poly mesh. Some of the other digital sculpting tools are Pixologic's ZBrush and Sculptris, Luxology's Modo and Nevercenter's Silo. Subdividing and sculpting progress and how vertex, edge and triangle counts have rapidly increased can be seen in Figure 5.9.b.

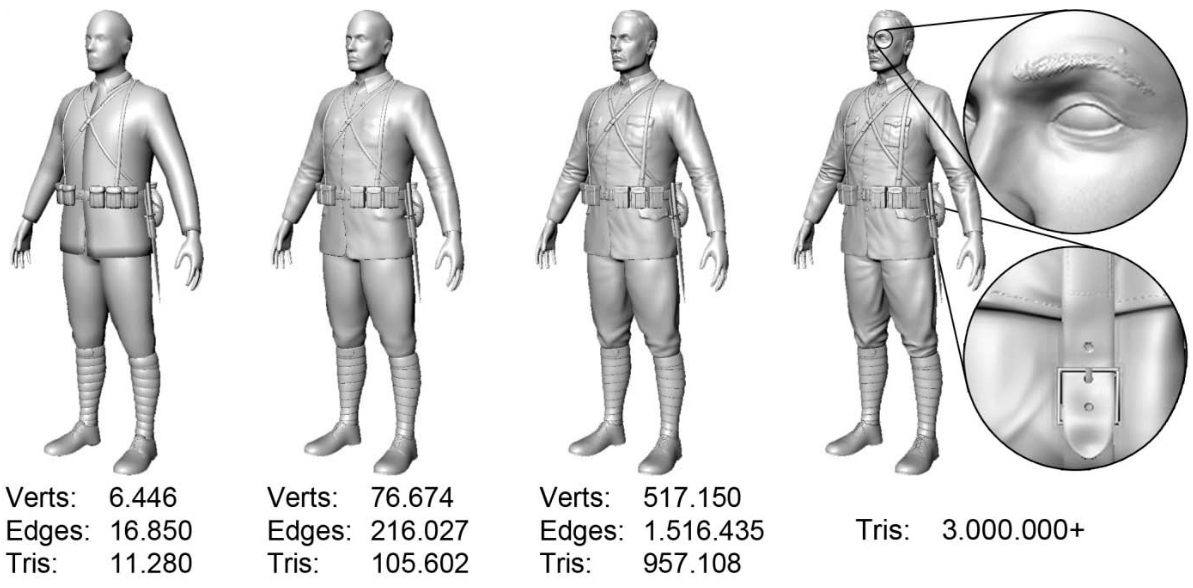


Figure 5.9.b. Sculpting high-poly mesh

Because of the limitations of current hardwares, high-poly models that have millions of triangles cannot be rendered effectively within a real time game environment. A low-poly model which has a polygon count around a few thousand, like the base mesh, must be used. But the base mesh modeled for sculpting, which means it consists of homogeneously dispersed quadrilateral polygons to be subdivided properly within a sculpting tool and it is not modeled to be animated properly. Topology (layout of edges and edge loops that create the polygon flow) of a model heavily affects the results of skeletal animation. For example joint areas like knee or elbow must have regular and dense edge loops to avoid disturbing deformations (Robson, [38]). In addition, topology of the base mesh is not reflecting the details, like garment folds, added during sculpting.

In order to create a proper low-poly mesh easily and quickly, a new method known as “retopologizing” can be used. It is not much different than classical modeling, but while adding or moving vertices in 3D space, they are automatically repositioned as they wraps the surface of another model called “reference mesh”.

There are few tools available for retopologizing. Pixologic’s ZBrush could be used as the method originally introduced by Pixologic as a feature of ZBrush 3.1 released in 2007, but Pixelmachine’s TopoGun, a dedicated retopologizing tool, is used (Figure 5.9.c.).

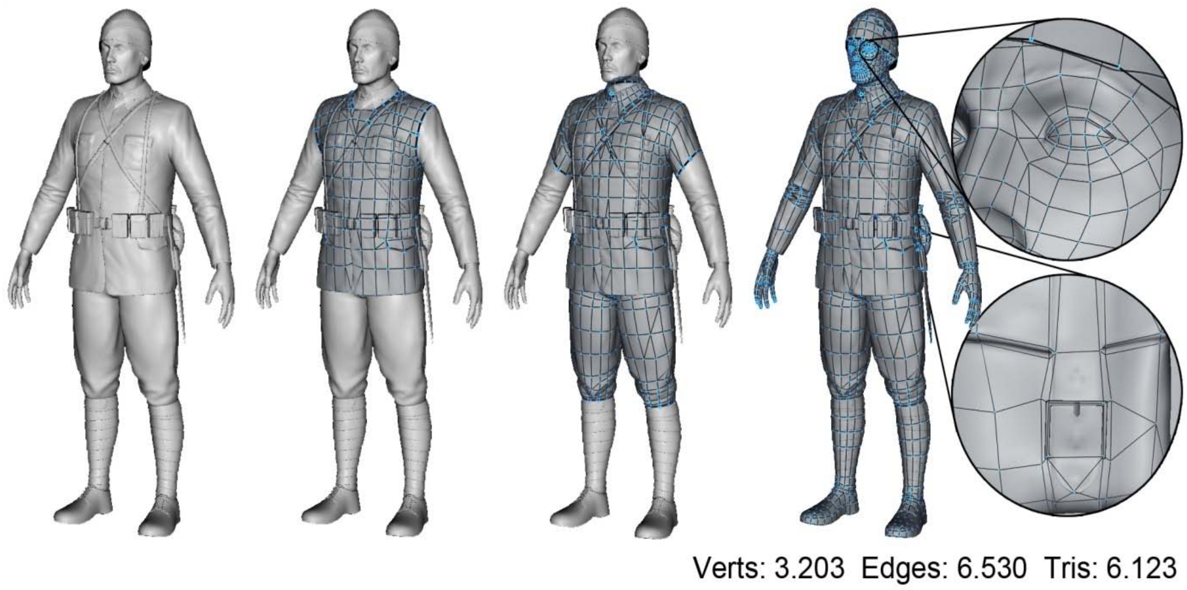


Figure 5.9.c. Retopologizing

After creating an appropriate simplified mesh, the UV layout should be prepared. A UV map can either be generated automatically or made manually, but methods that combine both are generally preferred because of providing ease and flexibility. In addition to dedicated UV mapping tools like Polygonal Design’s Unfold 3D, most of the 3D modeling packages are capable of UV mapping. Turkish infantry character has been UV mapped with 3D Studio Max. The result can be seen in Figure 5.9.d.

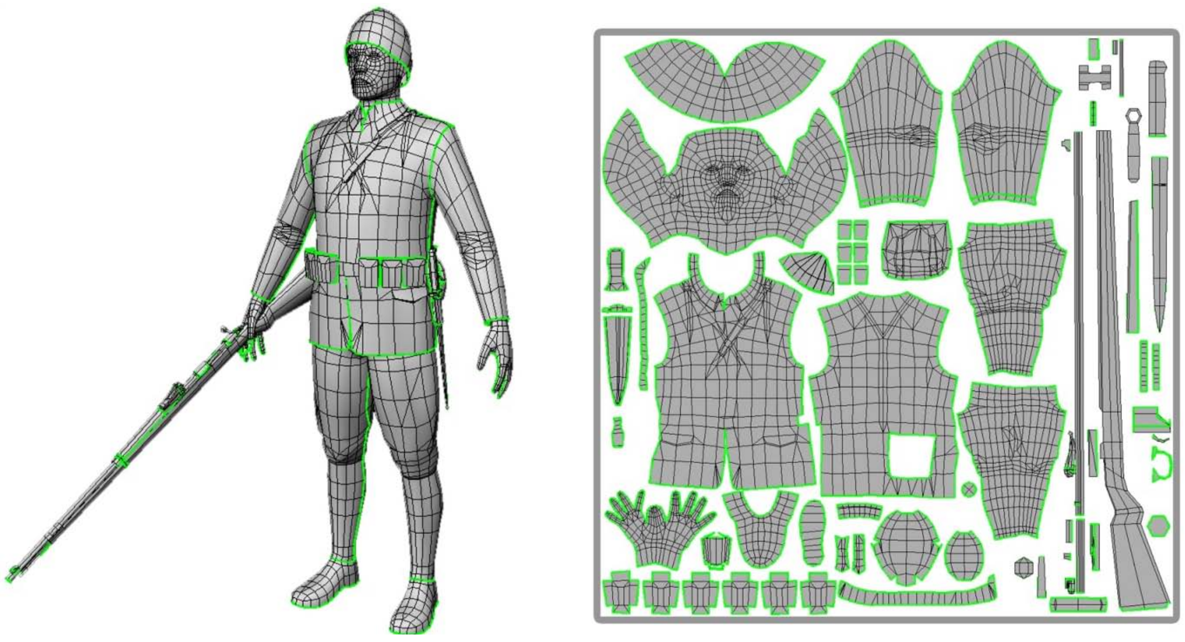


Figure 5.9.d. UV Mapping

After UV mapping is done, any required maps can be generated. Most of the 3D modeling and digital sculpting softwares have the ability to generate various maps, but a dedicated tool named xNormal (developed by Santiago Orgaz & Co.) was used to generate a normal map, which is an RGB image[26] that is used for creating details over a simplified mesh by manipulating the shading of the surface (Ahearn, [39]). A common method for creating a normal map depends to the comparison of elevation differences between the surfaces of low-poly and high-poly versions of same model (Figure 5.9.e.).

Result of using normal map can be seen in Figure 5.9.f. where each of three Turkish infantry models has equal amount of vertices, edges and polygons. The increase in processing time in return of using normal map shading is much lower than using a high-poly mesh to achieve the same result.

[26] An RGB image is a colored image that consists of three channels: Red, Green and Blue. Each of these channels contains a grayscale image of the same size as the colored image.

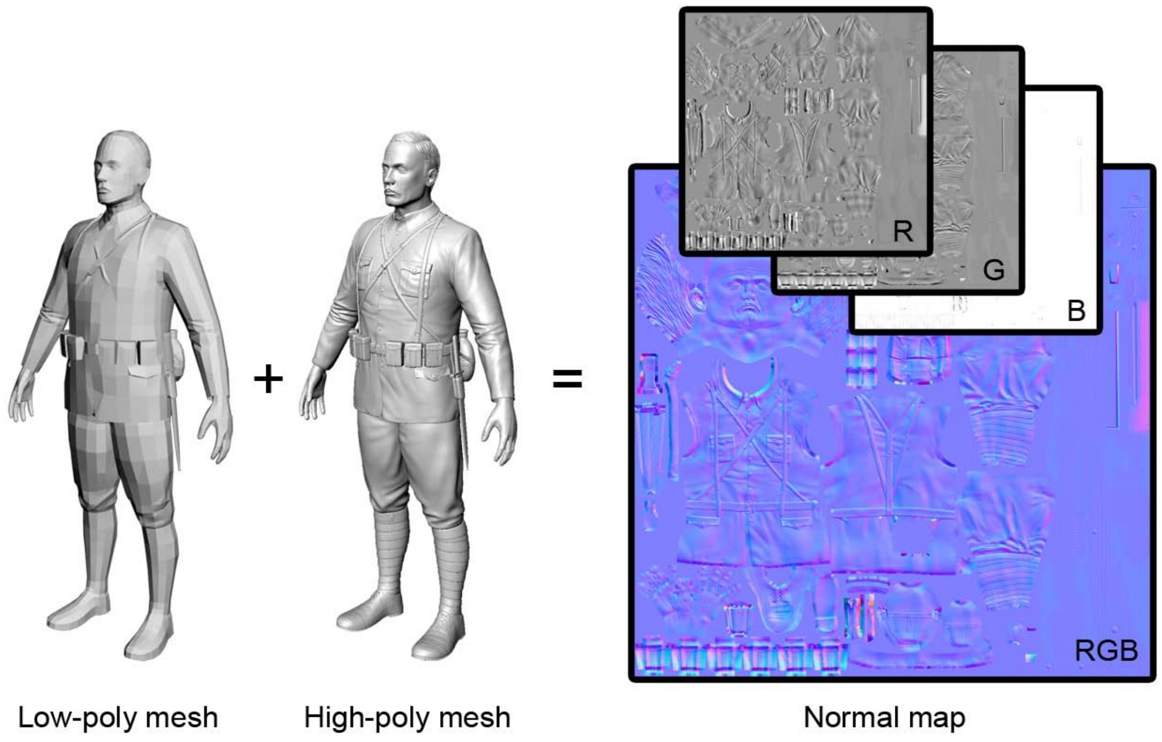
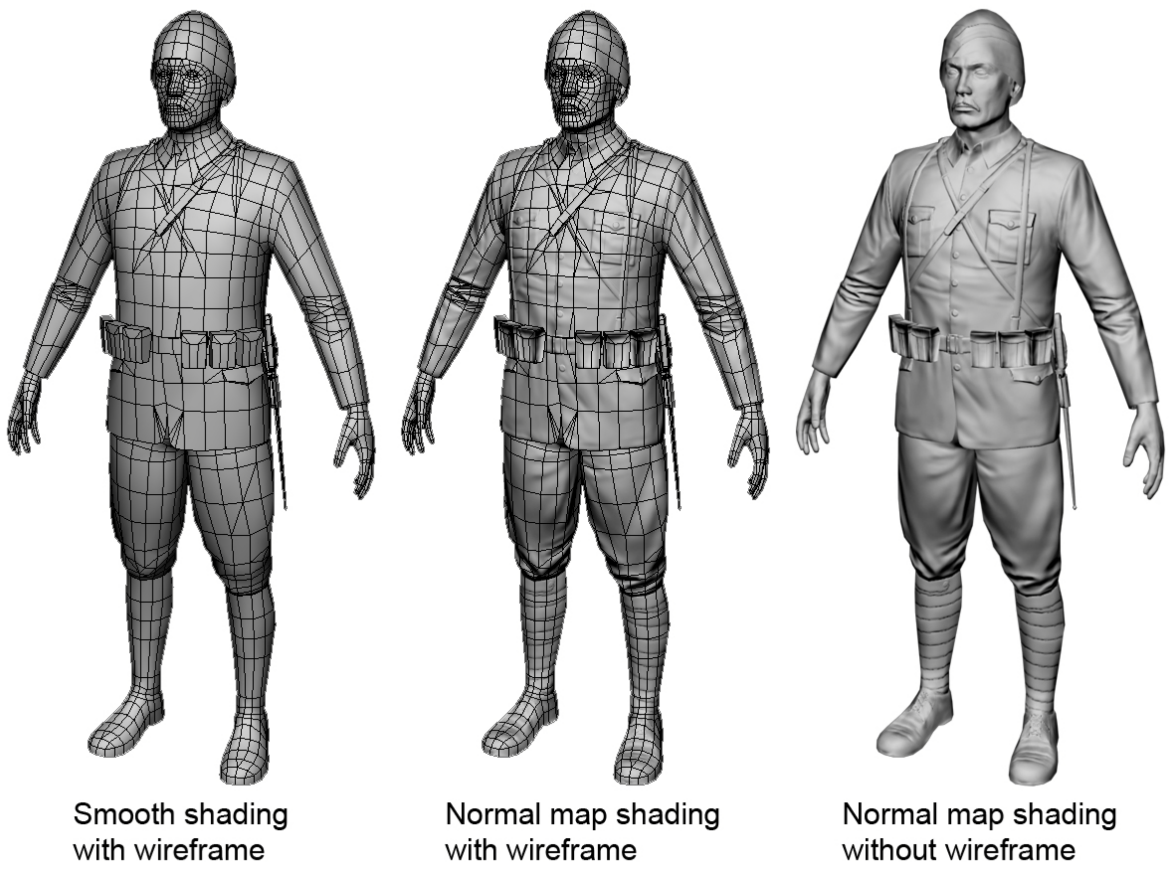


Figure 5.9.e. Generating normal map from high-poly and low-poly meshes



Smooth shading with wireframe
Normal map shading with wireframe
Normal map shading without wireframe

Figure 5.9.f. The effect of normal mapping

Other required maps for Turkish infantry model are texture map and specular map. Texture map is an RGB map used for colorizing a model. Specular map is a grayscale map used for simulating different types of materials (e.g. wood, metal, plastic) by altering the brightness value.

Since these maps cannot be generated from 3D geometries, they have to be painted manually. Texture and specular maps are painted with Adobe Photoshop and stored in an RGBA image[27], where RGB channels are used for texture map and the alpha channel is used for specular map (Figure 5.9.g.).

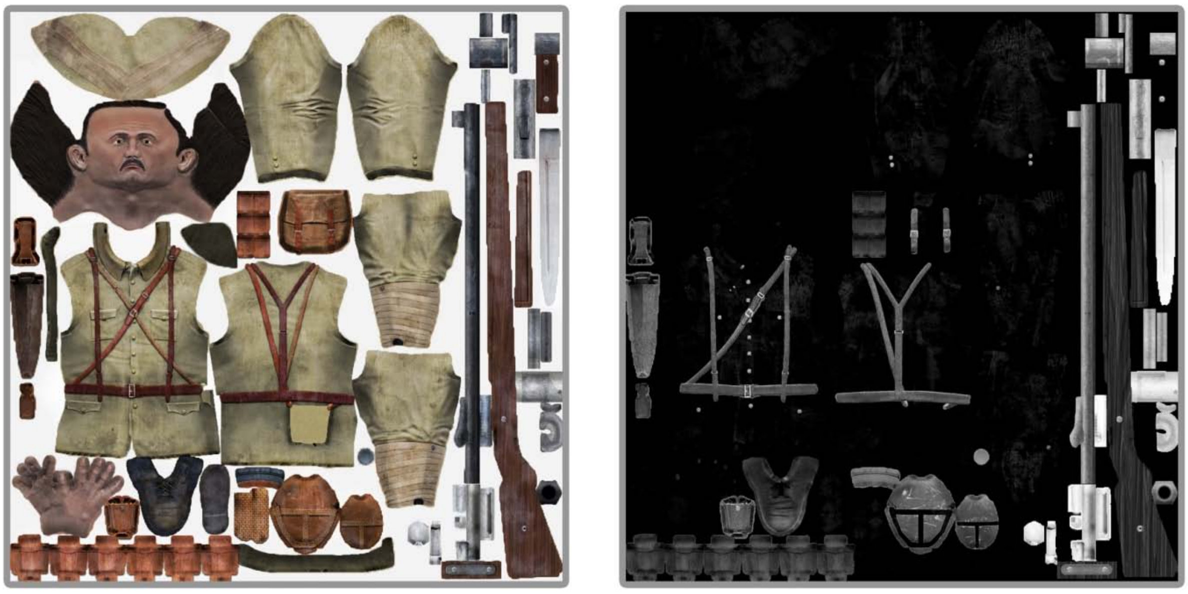


Figure 5.9.g. Texture (left) and specular (right) maps

The result of normal, texture and specular mapping can be seen in Figure 5.9.h.

[27] An RGBA image is a colored image with extra information. It consists of four channels: Red, Green, Blue and Alpha. The Alpha channel is normally used for storing the opacity information.

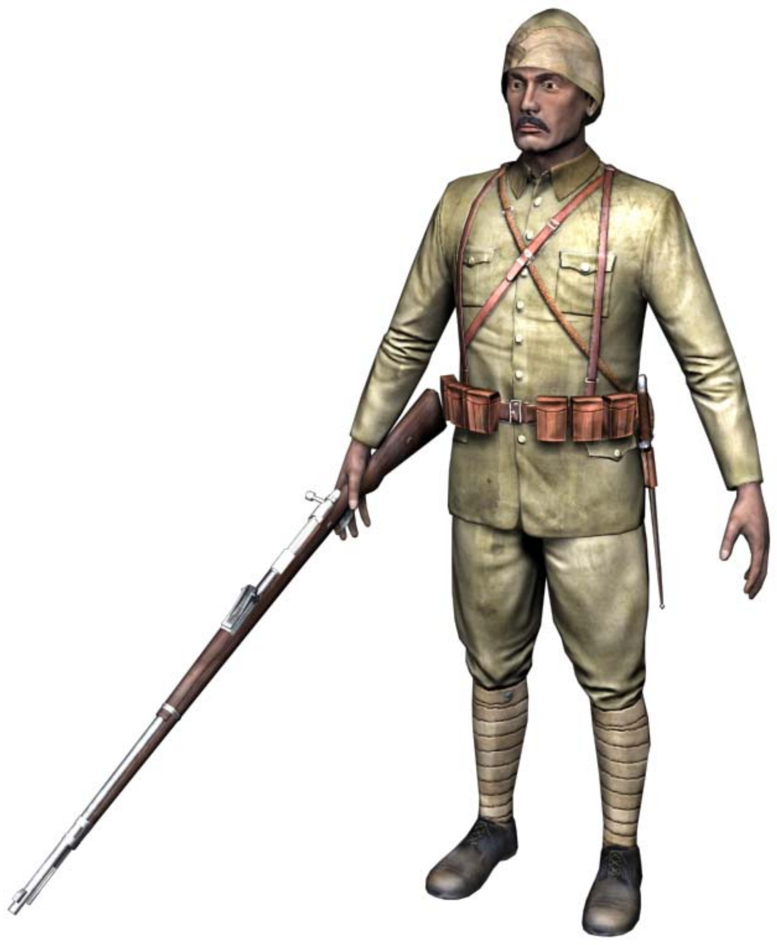


Figure 5.9.h. Turkish infantry model with texture, specular and normal maps

After preparation of the maps, rigging needs to be done to become able to animate the model. As previously mentioned, rigging step consist of preparation of an appropriate bone hierarchy (skeleton) and the skin weight table which is necessary to transfer bone animations to vertices of the model. 3D Studio Max was used for rigging the Turkish infantry character, but any other 3D animation tool could be used for these tasks. Bone hierarchy and skin weight table can be seen in Figure 5.9.i and Figure 5.9.j respectively.

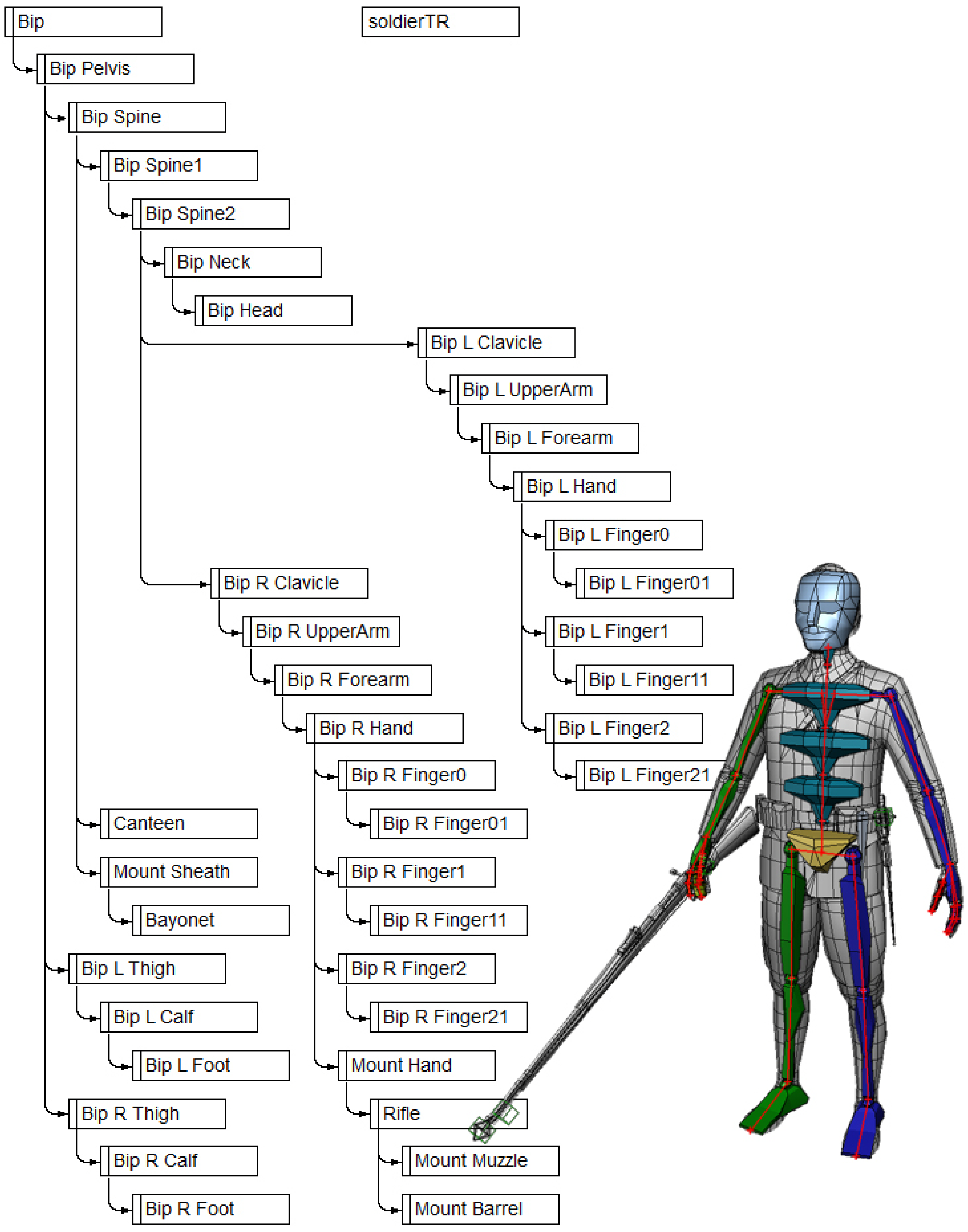
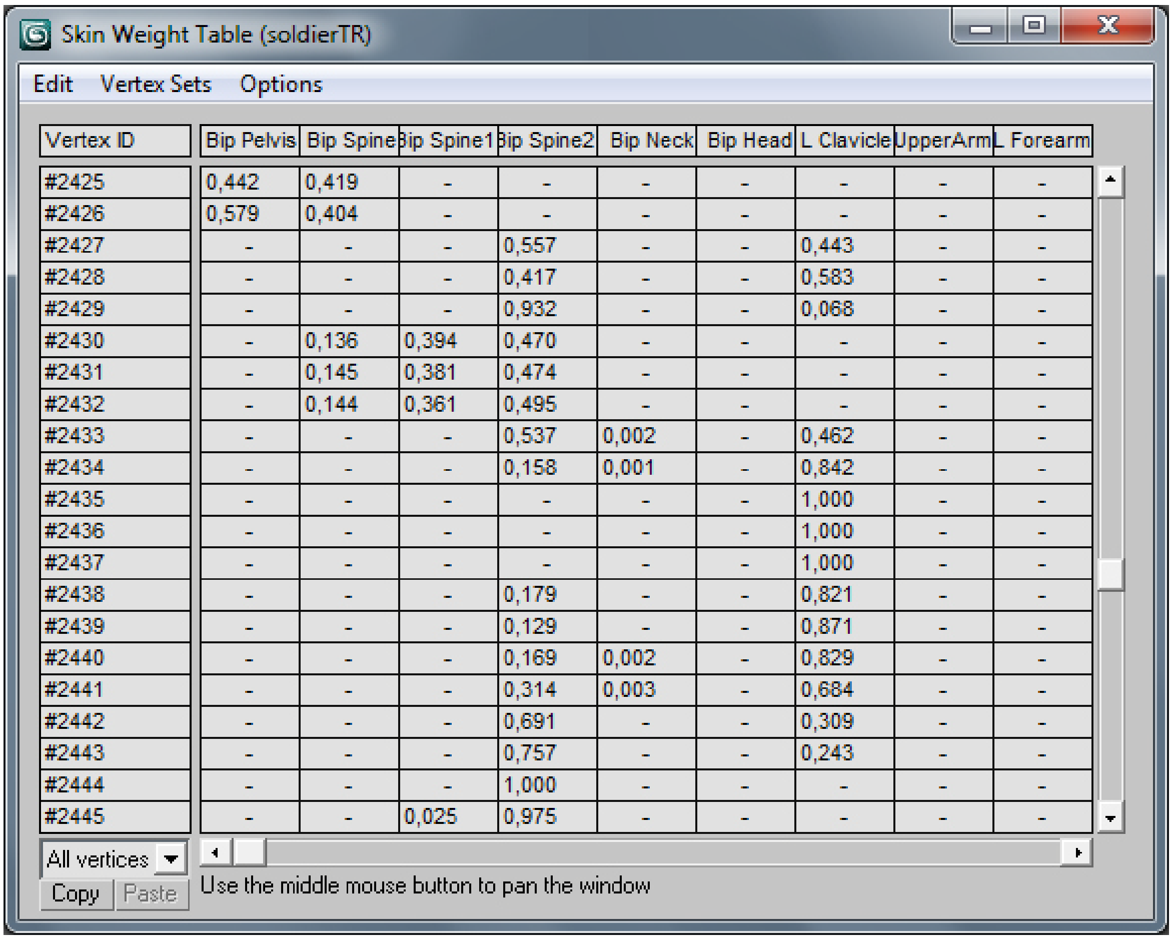


Figure 5.9.i. Bone hierarchy of the Turkish infantry character



Vertex ID	Bip Pelvis	Bip Spine	Bip Spine1	Bip Spine2	Bip Neck	Bip Head	L Clavicle	UpperArm	L Forearm
#2425	0,442	0,419	-	-	-	-	-	-	-
#2426	0,579	0,404	-	-	-	-	-	-	-
#2427	-	-	-	0,557	-	-	0,443	-	-
#2428	-	-	-	0,417	-	-	0,583	-	-
#2429	-	-	-	0,932	-	-	0,068	-	-
#2430	-	0,136	0,394	0,470	-	-	-	-	-
#2431	-	0,145	0,381	0,474	-	-	-	-	-
#2432	-	0,144	0,361	0,495	-	-	-	-	-
#2433	-	-	-	0,537	0,002	-	0,462	-	-
#2434	-	-	-	0,158	0,001	-	0,842	-	-
#2435	-	-	-	-	-	-	1,000	-	-
#2436	-	-	-	-	-	-	1,000	-	-
#2437	-	-	-	-	-	-	1,000	-	-
#2438	-	-	-	0,179	-	-	0,821	-	-
#2439	-	-	-	0,129	-	-	0,871	-	-
#2440	-	-	-	0,169	0,002	-	0,829	-	-
#2441	-	-	-	0,314	0,003	-	0,684	-	-
#2442	-	-	-	0,691	-	-	0,309	-	-
#2443	-	-	-	0,757	-	-	0,243	-	-
#2444	-	-	-	1,000	-	-	-	-	-
#2445	-	-	0,025	0,975	-	-	-	-	-

Use the middle mouse button to pan the window

Figure 5.9.j. Skin weight table of Turkish infantry character

Animations for Turkish infantry character have been created by key framing, which is an animation technique depending on changing the animatable properties (e.g. position, rotation) of objects (or bones) only at certain frames. The transition between them calculated at run time via interpolation algorithms. All of the animations have been made as they can loop continually, which means the pose at last frame is same with the pose at fist frame. The animations made for Turkish infantry character are as follows (Figure 5.9.k.):

- Walk: Walking loop.
- Run: Running loop.
- Idle 1: Looking around while standing.
- Idle 2: Looking around while kneeled.
- Shoot 1: Aiming, firing and reloading while standing.
- Shoot 2: Aiming, firing and reloading while kneeled.

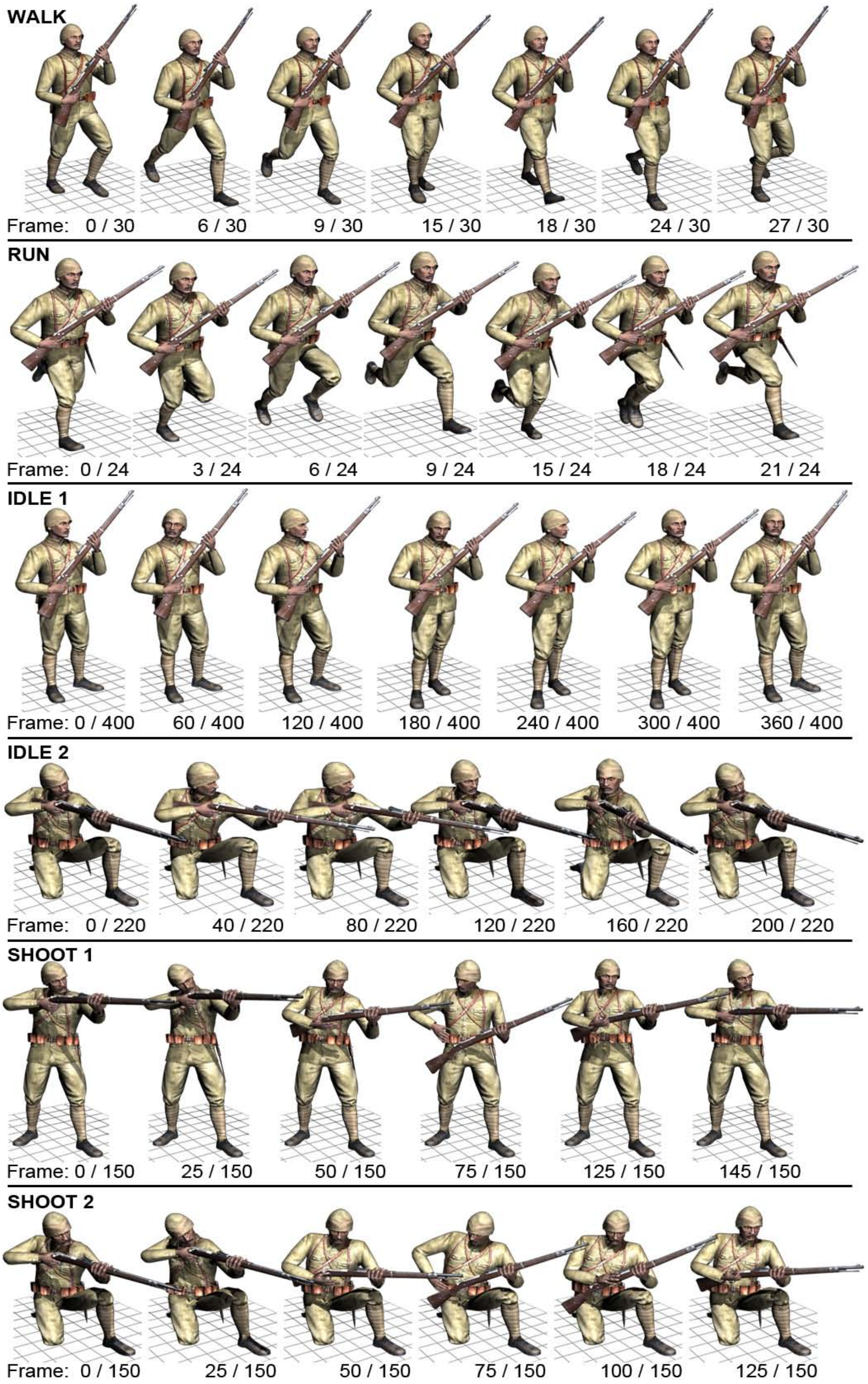


Figure 5.9.k. Animations of Turkish infantry character

5.4.2. SS River Clyde

As noted previously, some of the English troops had tried to land from a converted collier named SS River Clyde, beached near to the ruins of Seddulbahir castle on 25 April 1915. SS River Clyde model was made as a static model. Modeling and mapping processes of the ship are similar with Turkish infantry character except sculpting and retopologizing processes were bypassed by using Nvidia's normal map plugin for Photoshop to generate the normal map from texture map (Figure 5.10.a. and Figure 5.10.b.).

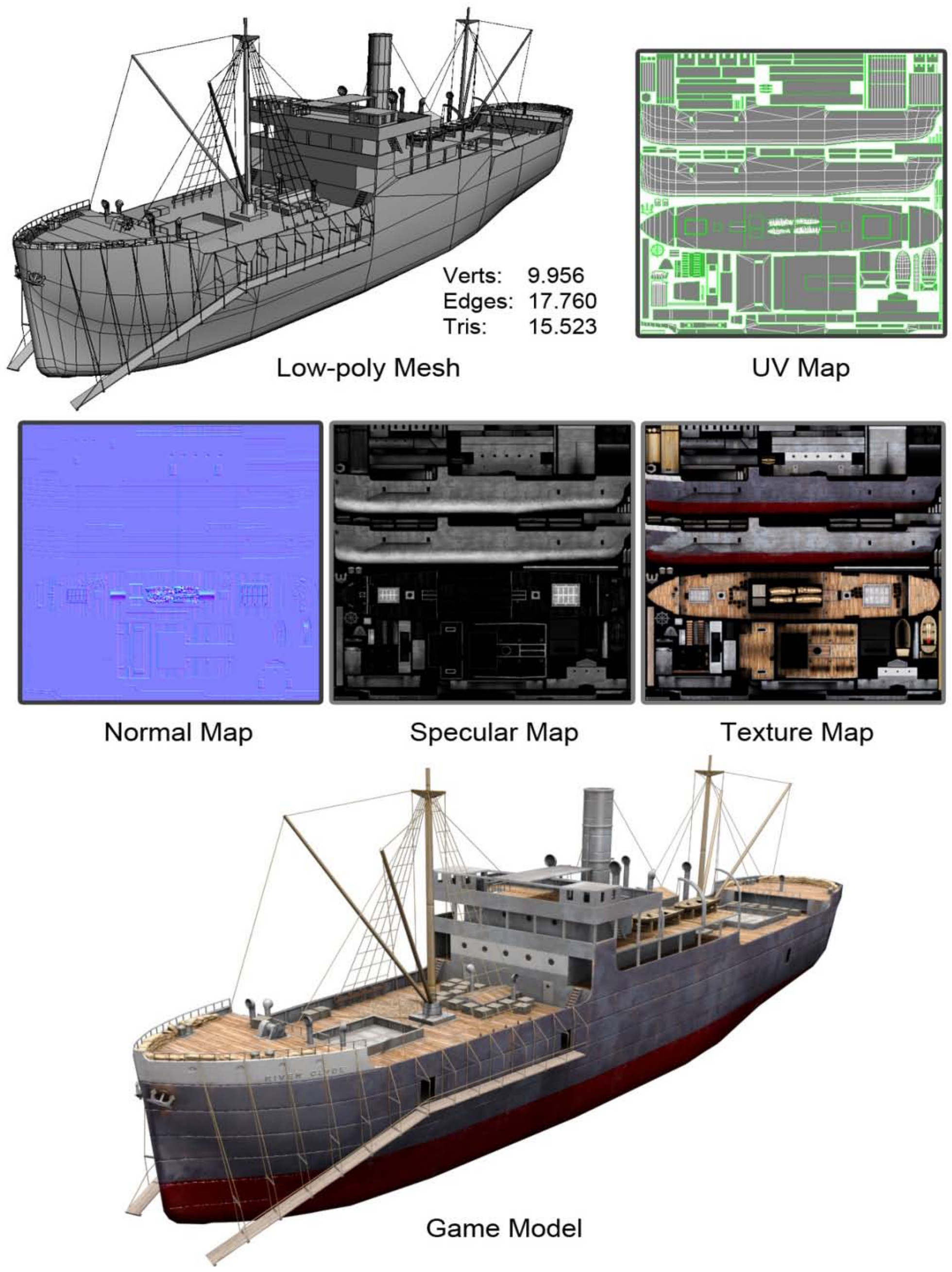


Figure 5.10.a. 3D model and maps of SS River Clyde

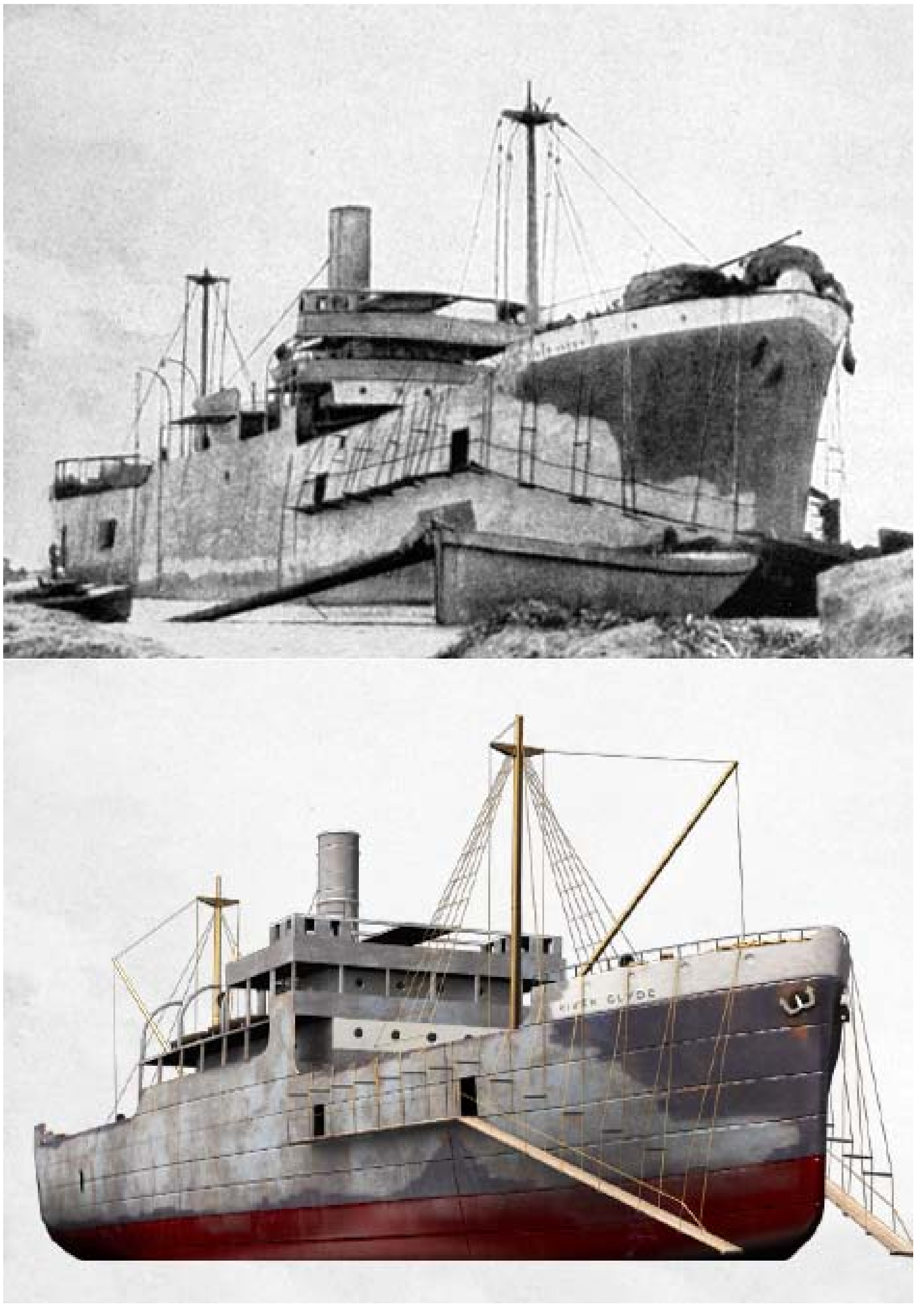


Figure 5.10.b. Photo (top) and 3D model (bottom) of SS River Clyde

5.4.3. Rowboat

Another static model made for Gallipoli Wars is a generic rowboat representing the rowboats used by English troops for landing along with SS River Clyde collier. The rowboat model and its texture and normal maps were made with same techniques used in SS River Clyde model except no specular map was used due to all parts of the rowboat are made from same material, which is wood, so there is no need to use different glossiness levels to visualize different materials (Figure 5.11.). Several copies of the model were placed to the beach where English troops have landed.

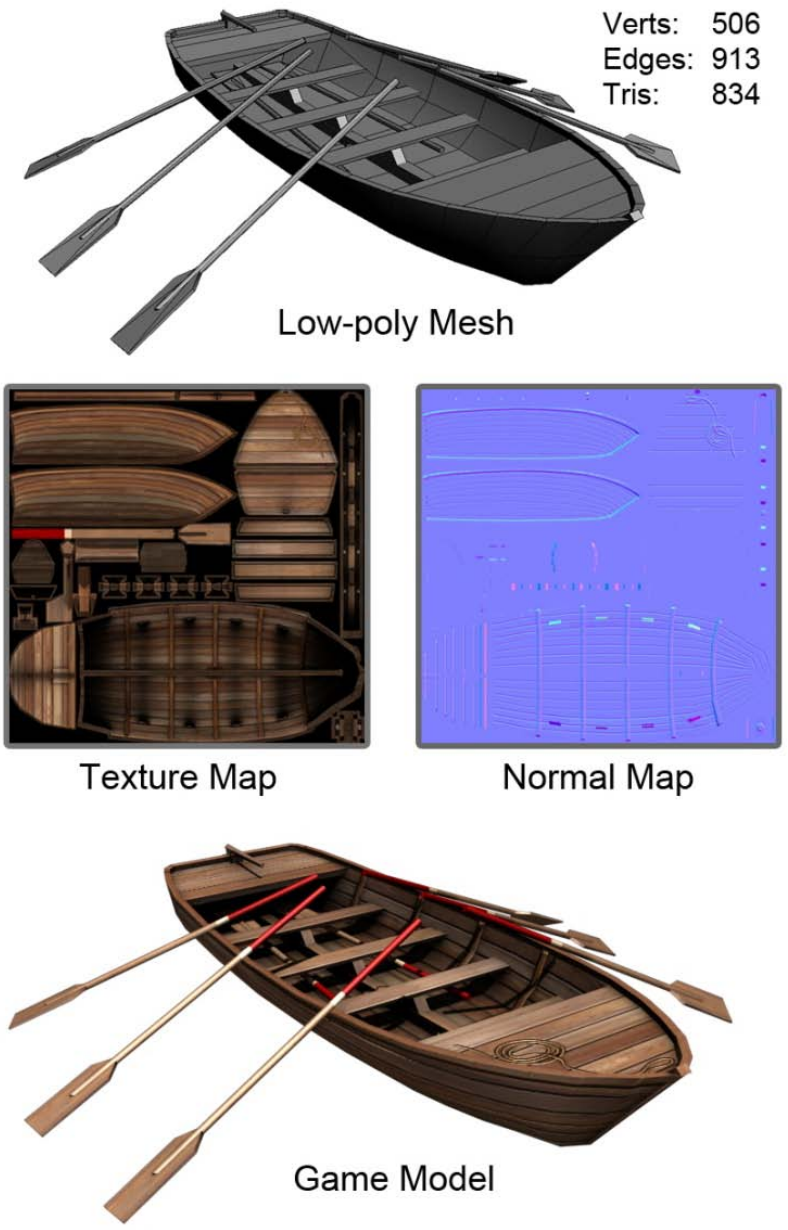


Figure 5.11. Low-poly model and maps of a rowboat

5.4.4. Seddulbahir castle

Remains of Seddulbahir (Sedd el Bahr) castle are the most noteworthy landmark of the V Beach landing zone at southwest tip of Gallipoli peninsula. Seddulbahir castle model was made as a static model. Modeling and mapping processes of the castle are exactly same with rowboat model (Figure. 5.12.a. and Figure 5.12.b.).

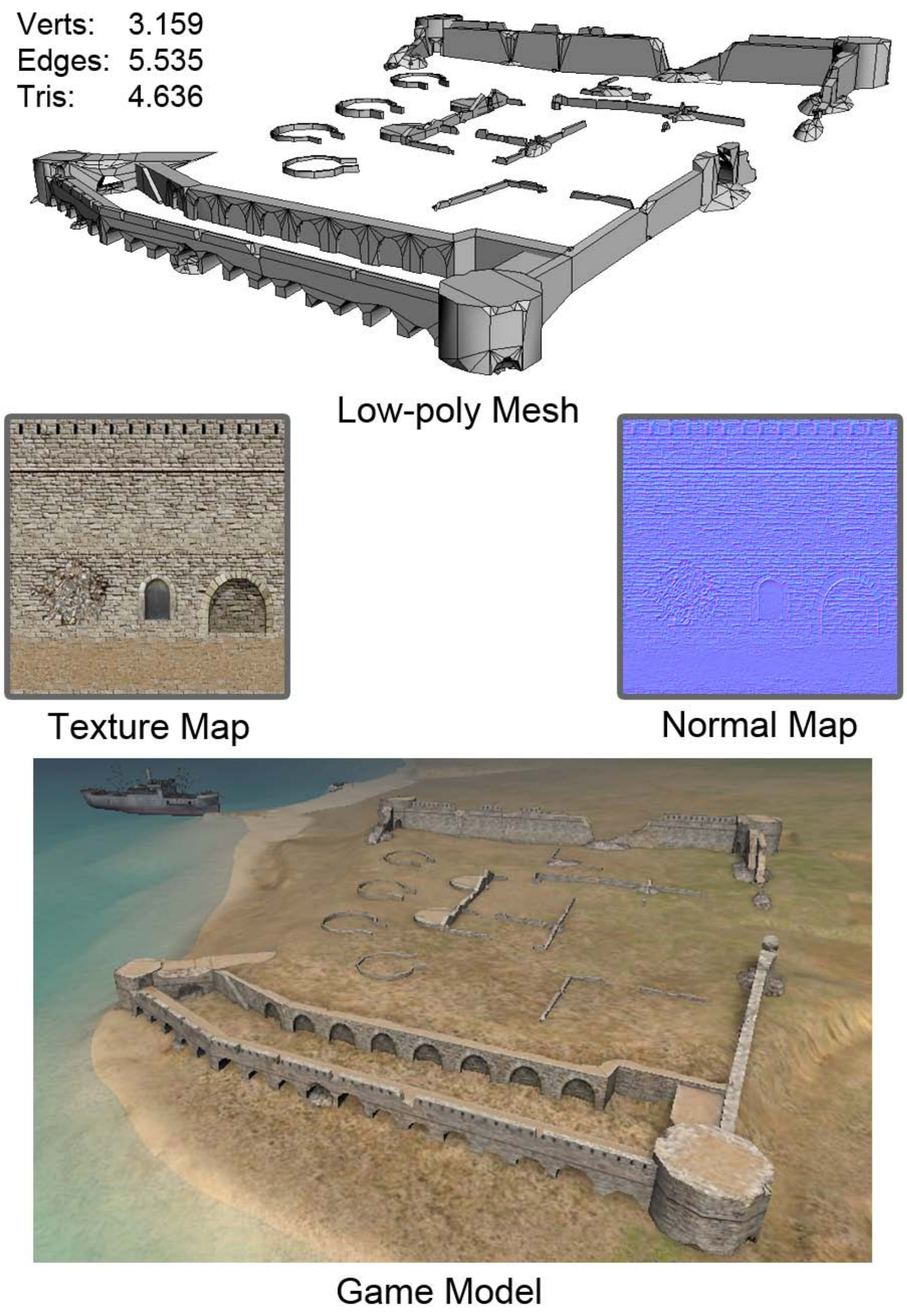


Figure. 5.12.a. Low-poly model and maps of Seddulbahir castle

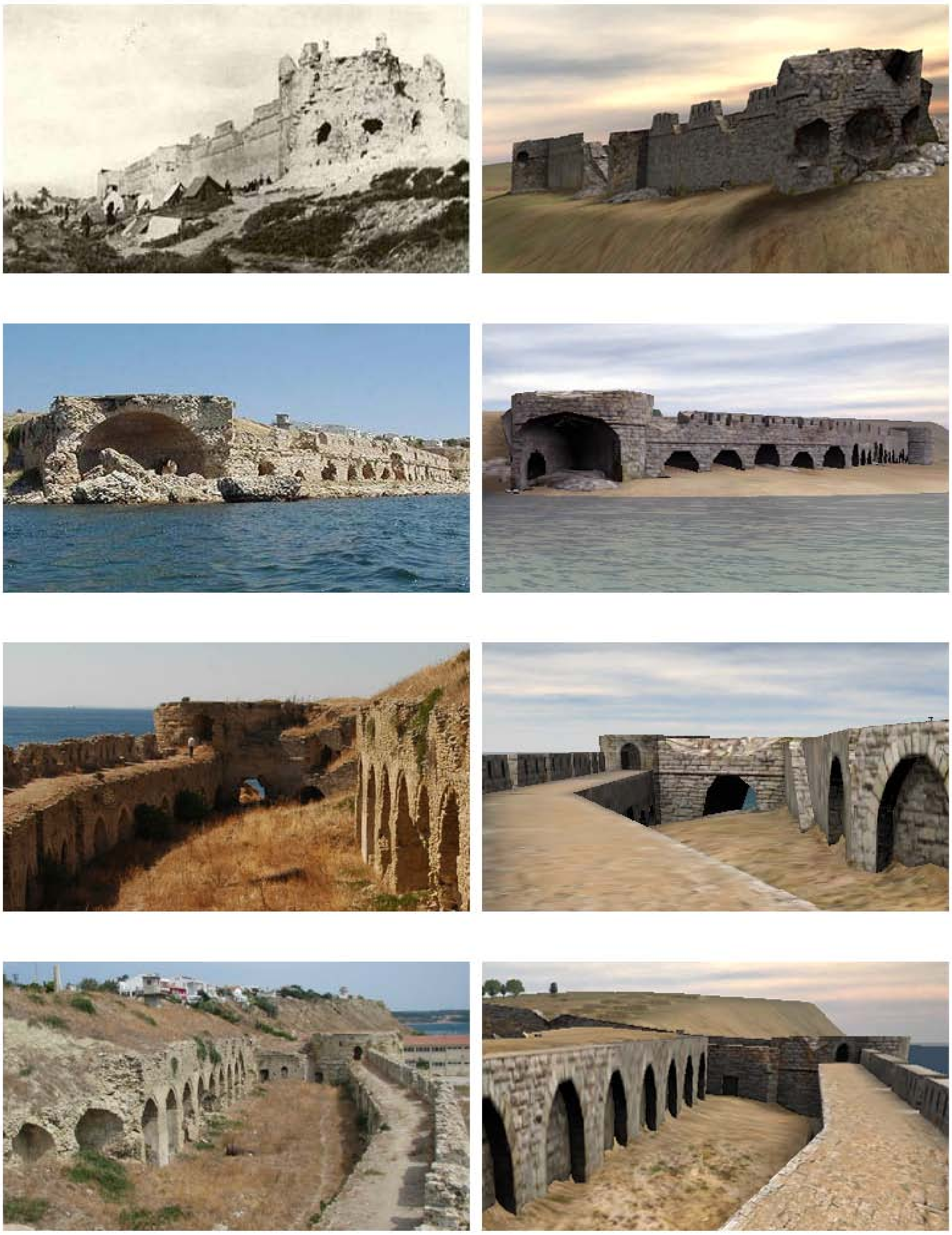


Figure 5.12.b. Photographs (left) and 3D model (right) of Seddulbahir castle

5.4.5. Terrain

An outdoor scene is intended as the game level, which means a 3D terrain model is needed. Unity game engine has a built-in terrain editor, which is capable to create height map terrains, where the elevation data is stored as a grayscale image. The way it works is that each gradient of gray represents a different height where lighter colors represent higher elevations (Feil and Scattergood, [40]). The height map is painted manually with built-in terrain editor (Figure 5.13.a.).

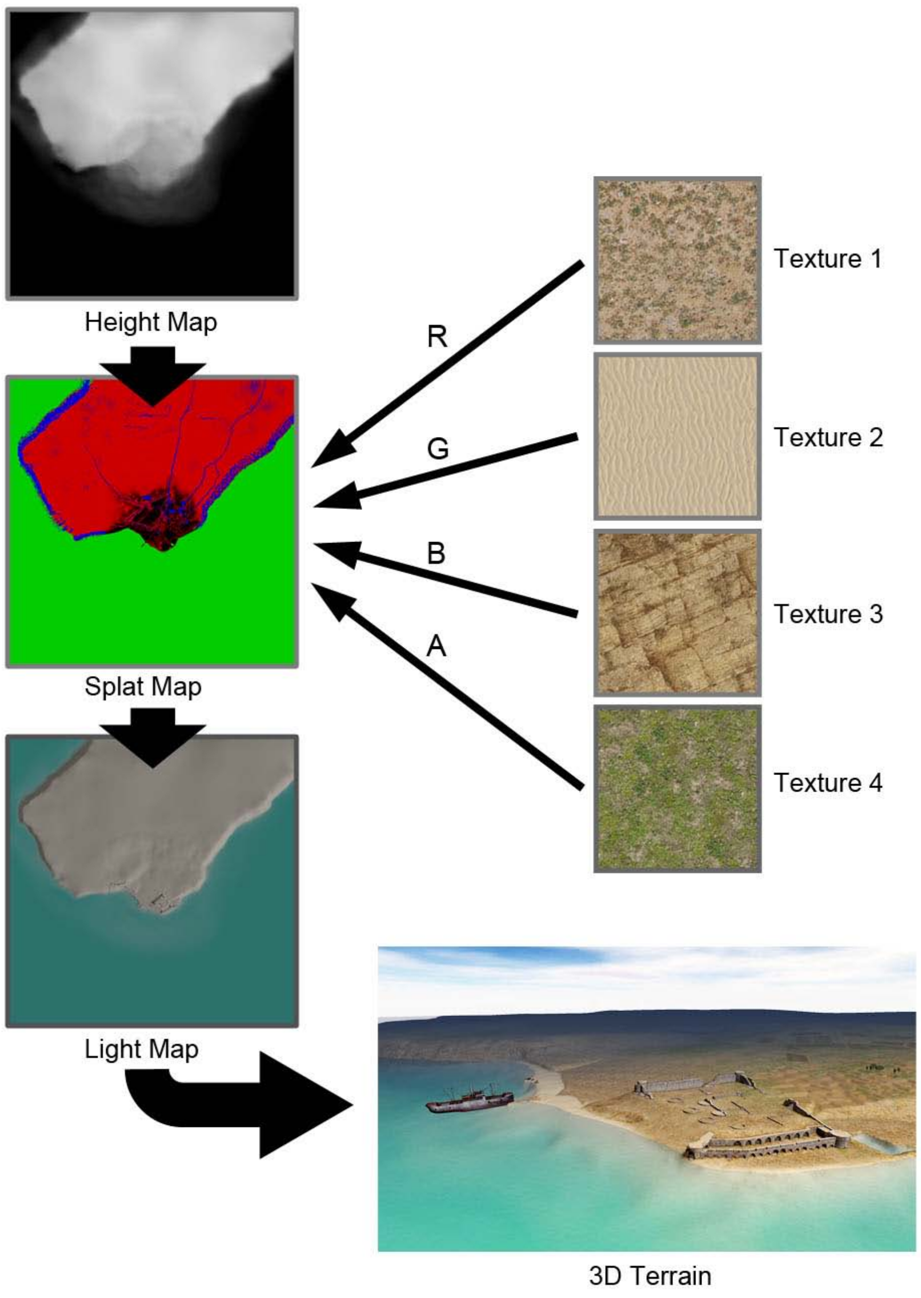


Figure 5.13.a. Height map terrain of southwest tip of Gallipoli peninsula

Since the terrain data is stored as a height map, the 3D geometry required for rendering is generated at runtime with dynamic LOD, where polygon dense and sparse areas of the terrain are changing according to the viewpoint. This approach optimizes the rendering of the terrain, but also makes the use of UV mapping for texturing very hard, due to changing topology. Also texturing square kilometers of terrain with a single texture map covering the whole terrain will result without any detail. A seamless texture can be used as it repeat over the terrain, but this time it will produce repeating patterns and unnaturally homogeneous look. A different approach, well suited for texturing the square kilometers of terrains is available, which is blending several repeating textures by using splat maps. A splat map is an RGBA raster image where each of four channels has a grayscale image that is used to store the opacity value for a certain texture. Therefore, up to four textures can be blended with a single splat map (Figure 5.4.5.). Finally an RGB light map was added for static shadows and underwater colors. Built in terrain editor is used for generating light map of terrain. Generated light map was edited for adding the underwater color (Figure 5.4.5.).

The resulting terrain without vegetation can be seen in Figure 5.4.6. Terrain assets provided by Unity Technologies are added for populating the terrain with vegetation and rocks.[28]

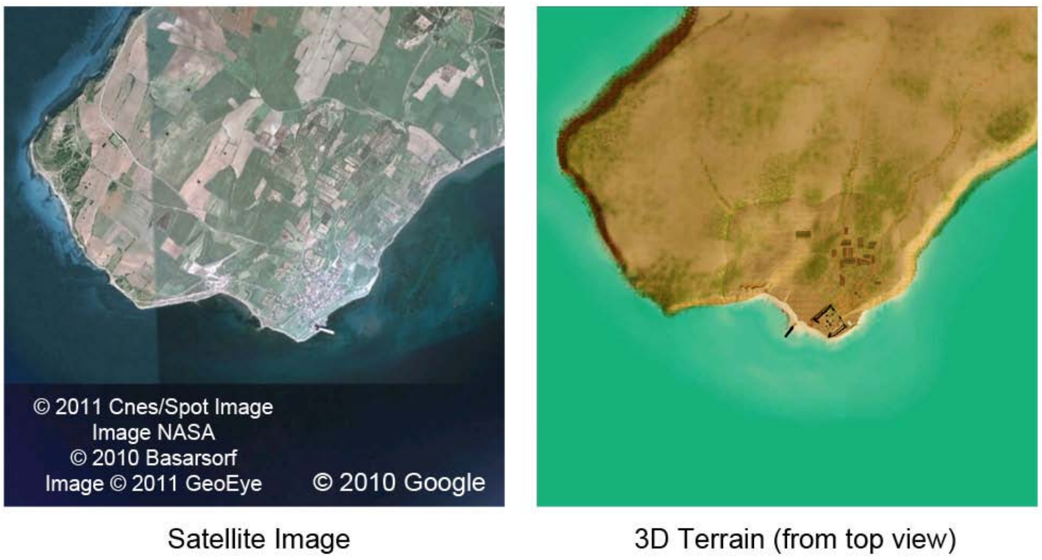


Figure 5.13.b. Satellite image and 3D model of southwest tip of Gallipoli peninsula

[28] http://unity3d.com/support/resources/assets/terrain-assets

5.5. Gameplay

As a result of the studies so far, a playable prototype of the “Gallipoli Wars” game has emerged. The prototype is consisting of only a single playable level, which is covering the British landing at V Beach (southwest tip of Gallipoli peninsula). The aim of the level is repulsing the British attack by commanding the Turkish soldiers.

At the beginning of the level, player sees the Turkish soldiers awaiting commands in the ruins of Seddülbahir castle (Figure 5.14.a)

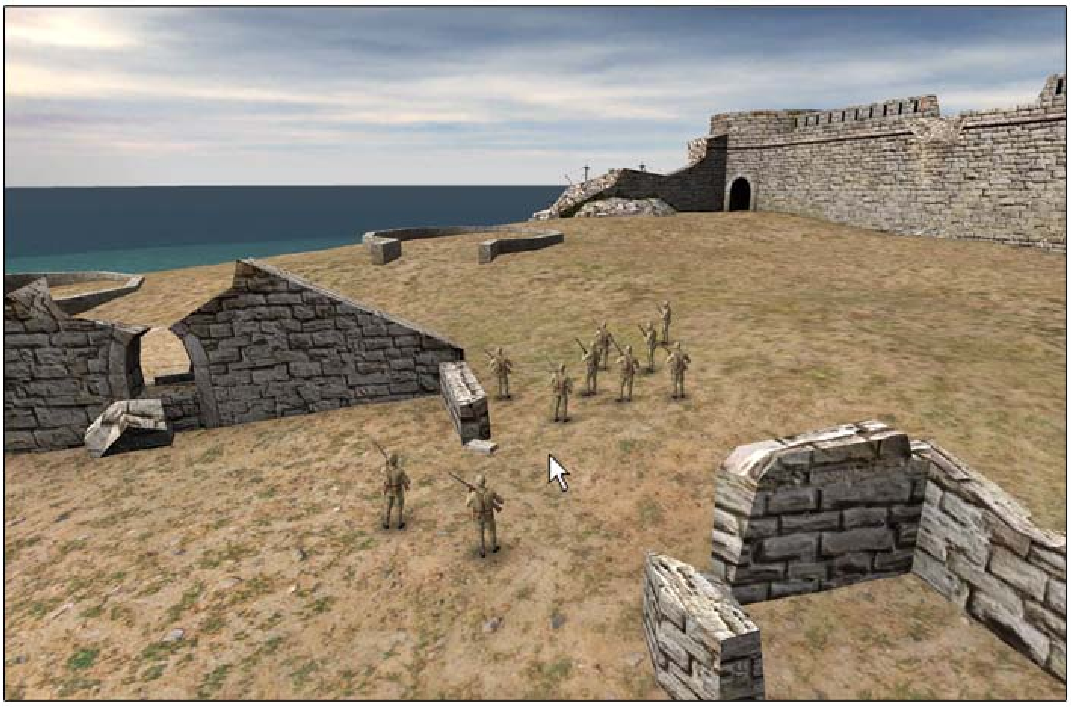


Figure 5.14.a Turkish soldiers in the ruins of Seddülbahir castle

Player can select soldiers individually by clicking on or as a group by drawing a selection rectangle (Figure 5.14.b.).

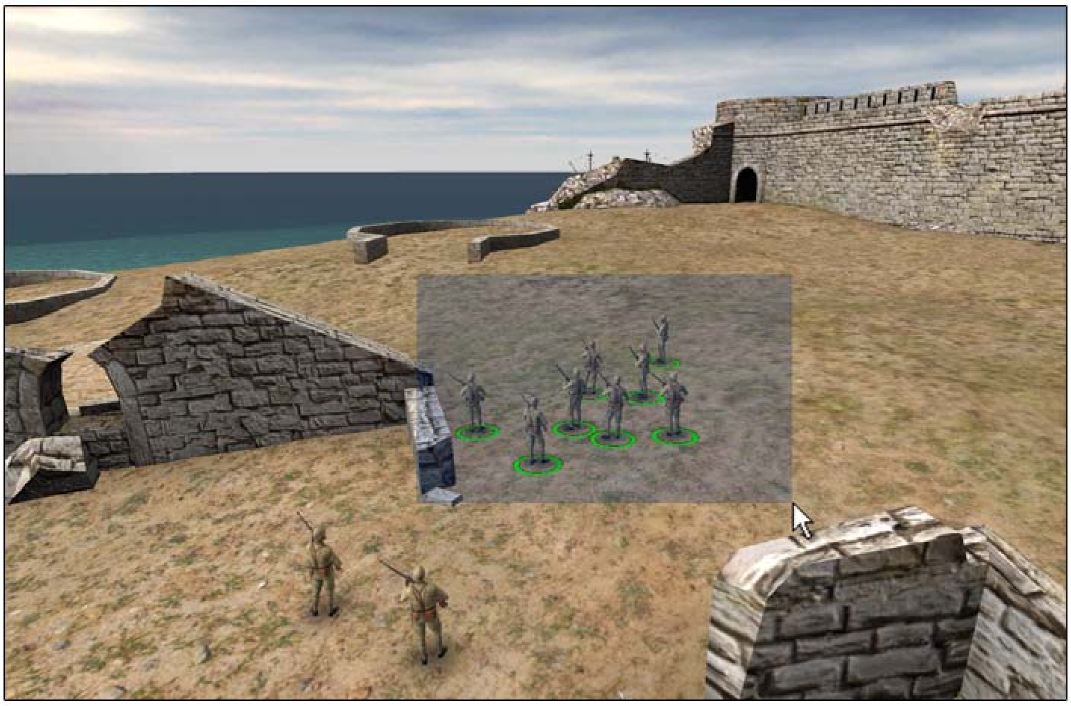


Figure 5.14.b. Selecting a group of soldiers by drawing a selection rectangle

Player can also control the camera to navigate through the scene as follows:

- Moving the camera
  - By moving the cursor to the edges of the screen
  - By pressing the arrow keys on the keyboard
- Rotating the camera
  - By clicking and holding the mouse wheel button and dragging the mouse
- Zooming the camera
  - By rotating the mouse wheel

After selecting one or more units, player can give a move order or an attack order to them by clicking on terrain or clicking on an enemy unit respectively. Player can also follow the selection procedures to make a new selection without giving any order to currently selected units.

As it can be seen in Figure 5.14.c. and Figure 5.14.d., Turkish soldiers were ordered to move in front of the castle and ordered to attack against enemy units in order to repulse the incoming attack.

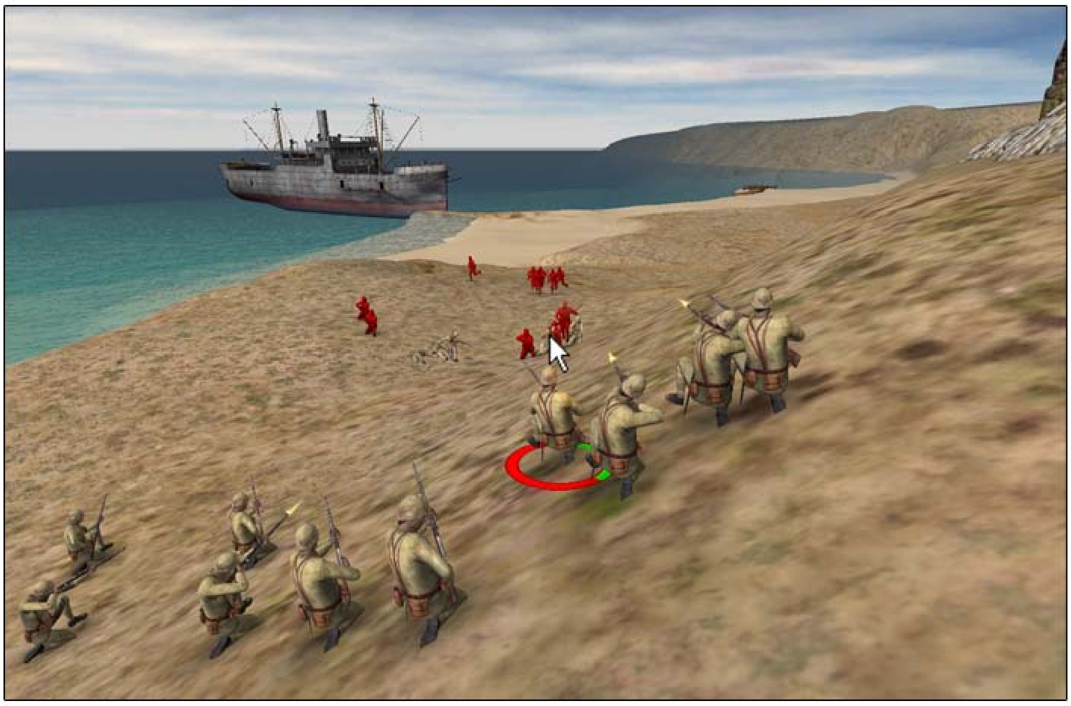


Figure 5.14.c. Turkish soldiers were deployed in front of the castle

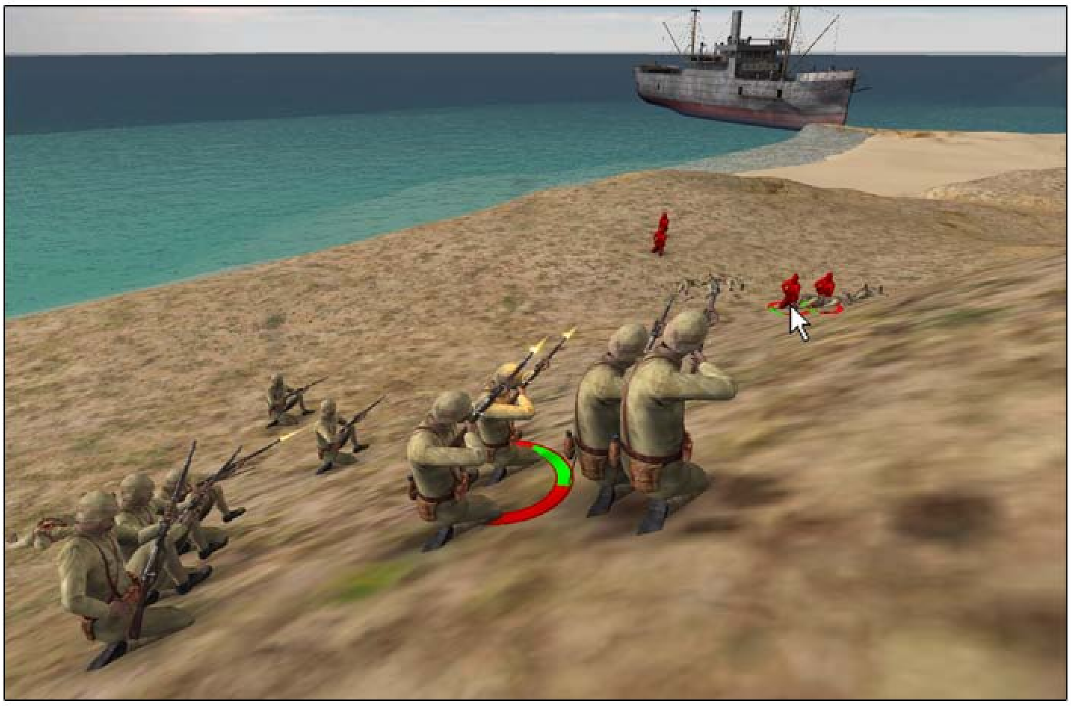


Figure 5.14.d. The attack has been repulsed with mild casualties

6. SUMMARY AND CONCLUSIONS

6.1. Summary

Game development is a special kind of software development where the creation and the use of art contents are heavily included. Specifically, the increasing emphasis on movie-like production values has required larger teams and greater specialization of roles played by team members. At the preproduction stages, game designers, writers and concept artists undertake an active role to form the game design. At the production stages, programming team works in parallel with graphics artists, sound artists, level designers, and game testers or a quality assurance (QA) department to bring to life the game design. Towards the end of the production, marketing and distribution teams undertake an important role to finalize the entire development process with a commercial success. As a matter of fact, there is no single way to develop computer games, and recent years have seen the rise of gaming on mobile phones, interactive TV services and browser based casual games. Equally, the creation of certain genres can force very different demands onto developers, but majority of the games share a similarly broad approach to the game development (Rollings and Morris, [2], Fullerton, Swain and Hoffman, [5], Bates, [6], Rollings and Adams, [8] and McCarthy et. al., [12]).

6.2. Conclusions

In this study, game development processes and methodologies were examined at a broad scope. Naturally, the broadness of the scope reduces the level of detail. All the design elements and different phases of the development process, and the technical aspects like ray-casting and ray-tracing algorithms, shading techniques and approaches to artificial intelligence could be described and analyzed in more detail. There are also many research areas up-to-date, like the possibility of the use of non-real-time lighting and rendering techniques, known as global illumination (or indirect illumination), which depends on the simulation of real-life photon behaviors, within a real-time game environment. Anyway, this study may provide a general understanding over the various aspects of game development, and may provide a wide range overview for the future studies that may focus in detail on specific aspects of the game development.

6.3. Extensions of the Study

As mentioned previously, “Gallipoli Wars” is thought as a full length strategy game. Therefore the systems (e.g. AI, scenario system, etc.) of the case study are designed in an extensible manner. The key features that should be in a real-time strategy game (e.g. camera navigation, unit management etc.) are already developed in this study. Also the British landing at V Beach (southwest tip of Gallipoli peninsula) is illustrated simply as a level of the game. However there is still a lot of work must be done to cover the entire war. Other landing areas, sea wars and land wars that occurred between 1914 and 1916, must be represented as individual levels. Also different kind of units (e.g. horsemen, machine gunners, and snipers), that may have various advantages and disadvantages over each other, can be added to enable more complicated strategies. All of these extensions require further design and development efforts, including quite time consuming artwork creation processes.

LIST OF REFERENCES:

[1] Crawford, C., Chris Crawford on Game Design, New Riders Publishing, 2003.

[2] Rollings, A. and Morris, D., Game Architecture and Design, A New Edition, Pearson Education, New Riders Publishing, 2004.

[3] Wolf, M. (Ed.), The Video Game Explosion: A History from Pong to Playstation and Beyond, Greenwood Press, 2008.

[4] Bergeron, B., Developing Serious Games, Charles River Media, 2006.

[5] Fullerton, T., Swain, C. and Hoffman, S., Game Design Workshop, A Playcentric Approach to Creating Innovative Games (2nd ed.), Morgan Kaufmann Publications, Elsevier Inc., 2008.

[6] Bates, B., Game Design (2nd ed.), Thomson Course Technology PTR., 2004.

[7] Salen, K. and Zimmerman, E., Rules of Play - Game Design Fundamentals, The MIT Press, 2004.

[8] Rollings, A. and Adams, E., Andrew Rollings and Ernest Adams on Game Design, New Riders Publishing, 2003.

[9] Rouse, R., Game design: theory & practice (2nd ed.), Wordware Publishing, Inc., 2005.

[10] Brathwaite, B. and Schreiber, I., Challenges for Game Designers, Charles River Media, 2009.

[11] Schell, J., The Art of Game Design, Morgan Kaufmann Publishers, Elsevier Inc., 2008.

[12] McCarthy, D., Curran, S. and Byron, S., The Complete Guide to Game Development, Art and Design, The Ilex Press Limited, 2005.

[13] Manninen, T., Kujanpää, T., Vallius, L., Korva, T. and Koskinen, P., Game Production Proces, A Preliminary Study (ELIAS project reports), LudoCraft, ELIAS-project (European Union Interreg III A Karjala), University of Oulu, Finland, 2006.

[14] Grassioulet, Y., A Cognitive Ergonomics Approach to the Process of Game Design and Development, TECFA, University of Geneva, Switzerland, 2004.

[15] White, W., Koch, C., Gehrke, J. and Demers A., Better Scripts, Better Games, Communications of the ACM, March 2009, Vol. 52, No. 3., 2009.

[16] Lengyel, E. (Ed.), Game Engine Gems, Volume One, Jones and Bartlett Publishers, 2011.

[17] Finney, K., 3D Game Programming All in One, Premier Press, 2004.

[18] Pipho, E., Focus On 3D Models, Premier Press, 2003.

[19] Ward, J., What is a Game Engine? http://www.gamecareerguide.com/features/529/what_is_a_game_.php, 2008.

[20] Zerbst, S. and Düvel, O., 3D Game Engine Programming, Thomson Course Technology PTR., 2004.

[21] Gregory, J., Game Engine Architecture, A. K. Peters Ltd., 2009.

[22] Eberly, D., 3D Game Engine Architecture, Morgan Kaufmann publications, Elsevier Inc., 2005.

[23] Eberly, D., Game Physics, Morgan Kaufmann publications, Elsevier Inc., 2004.

[24] Sánchez, D. and Dalmau, C., Core Techniques and Algorithms in Game Programming, New Riders Publishing, 2004.

[25] Luebke, D., Reddy, M., Cohen, D., Varshney, A., Watson, B. and Huebner, R., Level of Detail for 3D Graphics, Morgan Kaufmann Publishers, Elsevier Science, 2003.

[26] Harrison, L. H., Introduction to 3D Game Engine Design Using DirectX 9 and C#, Apress, 2003.

[27] De Sousa, B. M. T., Game Programming All in One, Premier Press, 2002.

[28] McShaffry, M., Game Coding Complete (3rd ed.), Course Technology PTR., 2009.

[29] Bourg D., Physics for Game Developers, O’Reilly and Associates Inc., 2002.

[30] Bourg, D. and Seeman, G., AI for Game Developers, O’Reilly Media, 2004.

[31] Bergen, G., Collision Detection in Interactive 3D Environments, Morgan Kaufmann publishers, Elsevier, 2004.

[32] Yılmaz, S., Mustafa Kemal ve Çanakkale Destanı, Toplumsal Dönüşüm Yayınları, 2010.

[33] Özakman, T., Diriliş, Çanakkale 1915, Bilgi Yayınevi, 2008.

[34] Fewster, K., Başarın, V. and Başarın, H. H., Gallipoli: The Turkish Story, Allen and Unwin, 2003.

[35] Haythornthwaite, P.J., Gallipoli 1915: Frontal Assault on Turkey, Osprey Publishing, 1991.

[36] Buckland, M., Programming Game AI by Example, Wordware Publishing, 2005.

[37] Topçu, O. and Oğuztüzün, H., Scenario Management Practices in HLA-Based Distributed Simulation, Journal of Naval Science and Engineering, Vol. 6, No.2, pp.1-33, 2010.

[38] Robson, W., Essential ZBrush, Worldware Publishing, Inc., 2008.

[39] Ahearn, L., 3D Game Textures, Focal Press. 2006.

[40] Feil, J. and Scattergood, M., Beginning Game Level Design, Thomson Course Technology PTR., 2005.

APPENDIX A - CameraMovement Class

```
using UnityEngine;
using System.Collections;

public class CameraMovement : MonoBehaviour
{
    private Vector3 mousePosition;

    //Screen Variables & Movement Variables
    private int screenHeight;
    private int screenWidth;
    private float screenHeightEdgeZoneValue;
    private float screenWidthEdgeZoneValue;
    public float screenEdgeZonePercentege;
    public float cameraMovementSpeed;
    private Vector3 ScreenMidPoint;

    //Rotation Variables
    private Vector3 mousePosionOnPressCameraFunction;
    private bool isCameraRotating;
    public float maxRotationAngle;
    public float minRotationAngle;
    private float rotationFactor;

    //Camera Game Object Variables
    private GameObject CameraContainer;
    private GameObject CameraContainerPivot;
    private GameObject cam;

    //Shifting (For Avoidig Collision With Terrain) Variables
    float TotalShiftDistance = 0.0f;
    public float MinDistanceFromTerrain;

    //Zoom Variables
    public float minZoomSize;
    public float maxZoomSize;
    public float maxzoomStepSize;
    public float zoomPercentagePerStep;
    private float mouseWheelValue;
    private Vector3 cameraNextZoomPosition;
    float zoomDistance;
    private bool isZoom = false;

    public void Start()
    {
      // Initializing variables about current system

        screenHeight = Screen.height;
        screenHeightEdgeZoneValue = (screenHeight * (0.01f)) *
                                     screenEdgeZonePercentege;

        screenWidth = Screen.width;
        screenWidthEdgeZoneValue = (screenWidth * (0.01f)) *
                                    screenEdgeZonePercentege;

        ScreenMidPoint = new Vector3(float.Parse((screenWidth *
                        (0.5f)).ToString()), float.Parse((screenHeight *
                        (0.5f)).ToString()), 0f);
        rotationFactor = 540f / screenWidth;

        cam = GameObject.FindWithTag("MainCamera");
        CameraContainer = GameObject.FindWithTag("CameraContainer");
        CameraContainerPivot = GameObject.FindWithTag(
                                "CameraContainerPivot");
        mouseWheelValue = 0.0f;
        cameraNextZoomPosition = cam.transform.localPosition;
    }

    public void Update()
    {
        mousePosition = Input.mousePosition;
        zoomDistance = -1 * cam.transform.localPosition.z;

        // ZOOM
        mouseWheelValue = GetMouseWheelValue();
        if (mouseWheelValue != 0.0f)
        {
            this.ZoomCamera(mouseWheelValue);
        }
        if (isZoom)
            this.Continue2Lerp();


        // MOVE
        if (isCursorOnBorders() && !isCameraRotating)
        {
            MoveCameraWithMouse();
        }
        else if (Input.GetButton("MoveForward") ||
                 Input.GetButton("MoveBackward") ||
                 Input.GetButton("MoveLeft") ||
                 Input.GetButton("MoveRight"))
        {
            if (Input.GetButton("MoveForward"))
            {
                MoveCameraWithKeys(Vector3.forward);
            }
            else if (Input.GetButton("MoveBackward"))
            {
                MoveCameraWithKeys(Vector3.back);
            }
            if (Input.GetButton("MoveLeft"))
            {
                MoveCameraWithKeys(Vector3.left);
            }
            else if (Input.GetButton("MoveRight"))
            {
                MoveCameraWithKeys(Vector3.right);
            }

            LayerMask terrainLayerMask = 1 << 8;//Terrain
            RaycastHit hit = Util.VerticalRaycast(
                                CameraContainer.transform.position,
                                terrainLayerMask);
            CameraContainer.transform.position = hit.point;
        }
        // SHIFT (AVOIDING GOLLISION WITH TERRAIN)

        ShiftCamera();

        // ROTATION

        if (Input.GetButtonDown("CameraFunction"))
        {
            mousePosionOnPressCameraFunction = Input.mousePosition;
            isCameraRotating = true;
        }
        if (Input.GetButtonUp("CameraFunction"))
        {
            isCameraRotating = false;
        }
        if (isCameraRotating)
            RotateCamera();
    }

    // ZOOM

    #region Zoom

    /// <summary>
    /// Calculates the rotation of mouse scroll wheel
    /// </summary>

    private float GetMouseWheelValue()
    {
        return Input.GetAxis("MouseScrollWheel");
    }

    /// <summary>
    /// Adjusts the zoom level of the camera
    /// </summary>
    /// <param name="zoomValue"></param>
    private void ZoomCamera(float zoomValue)
    {
        bool isZoomIn = true;
        if (zoomValue < 0)
            isZoomIn = false;

        float absoluteZoomValue = Mathf.Abs(zoomValue);
        CalculateCameraNextPosition(absoluteZoomValue, isZoomIn);
        isZoom = true;
    }

    /// <summary>
    /// Calculates the change in position of the camera when zooming
    /// </summary>
    /// <param name="zoomValue"></param>
    /// <param name="isZoomIn"></param>
    private void CalculateCameraNextPosition(float zoomValue,
                                             bool isZoomIn)
    {
        float nextZValue = cameraNextZoomPosition.z;
        for (int i = 0; i < zoomValue; i++)
        {
            float tempAddingValue = ((-1) * cameraNextZoomPosition.z *
                                        zoomPercentagePerStep);
            if (tempAddingValue > maxzoomStepSize)
                tempAddingValue = maxzoomStepSize;

            if (isZoomIn)
            {
                nextZValue += tempAddingValue;

                if (nextZValue > minZoomSize)
                    nextZValue = minZoomSize;
            }
            else
            {
                nextZValue += tempAddingValue * (-1);

                if (nextZValue < maxZoomSize)
                    nextZValue = maxZoomSize;
            }

            cameraNextZoomPosition = new Vector3(0.0f, 0.0f, nextZValue);
        }
    }

    /// <summary>
    /// Smooth zooming with linear interpolation
    /// </summary>
    private void Continue2Lerp()
    {
        cam.transform.localPosition = Vector3.Lerp(
                                      cam.transform.localPosition,
                                      cameraNextZoomPosition,
                                      Time.deltaTime * 3);
        if (cam.transform.localPosition.z == cameraNextZoomPosition.z)
            isZoom = false;
    }

    #endregion

    #region Move

    private void MoveCameraWithMouse()
    {
        // Moves the camera container
        Vector2 dragDifference = mousePosition - ScreenMidPoint;
        Vector3 moveDirection = new Vector3(dragDifference.x, 0f,
                                            dragDifference.y);
        CameraContainer.transform.Translate(moveDirection.normalized *
                                           zoomDistance *
                                           cameraMovementSpeed *
                                           Time.deltaTime);

        // Sets camera container to the terrain surface
        LayerMask terrainLayerMask = 1 << 8; // Terrain Layer
        RaycastHit hit = Util.VerticalRaycast(
                         CameraContainer.transform.position,
                         terrainLayerMask);
        CameraContainer.transform.position = hit.point;
    }
```

```
private void MoveCameraWithKeys(Vector3 moveDirection)
{
    CameraContainer.transform.Translate(moveDirection * zoomDistance
      * cameraMovementSpeed * 2f * Time.deltaTime);
}

private bool isCursorOnBorders()
{
    if (mousePosition.x < screenWidthEdgeZoneValue ||
        mousePosition.x > screenWidth – screenWidthEdgeZoneValue ||
        mousePosition.y < screenHeightEdgeZoneValue ||
        mousePosition.y > screenHeight - screenHeightEdgeZoneValue)
        return true;

    return false;
}

#endregion

#region Shifting

private void ShiftCamera()
{
    if (TotalShiftDistance > 0)
    {
        this.ReduceShiftingDistance();
    }

    this.LiftCamera();
}
/// <summary>
/// Checks Collision with terrain by casting a vertical ray
/// </summary>

private float CalculateShiftDistance()
{
    float difference = 0.0f;
    LayerMask terrainLayerMask = 1 << 8;// Terrain Layer
    RaycastHit hit = Util.VerticalRaycast(cam.transform.position,
                                          terrainLayerMask);
    float resultDistance = 0.0f;
    difference = cam.transform.position.y - (hit.point.y +
                                           MinDistanceFromTerrain);
    if (difference <= 0.0f)
    {
        resultDistance = (-1) * difference;
    }
    return resultDistance;
}

private void ReduceShiftingDistance()
{
    Vector3 newCameraPosition = new Vector3(cam.transform.position.x,
                                 cam.transform.position.y -
                                 TotalShiftDistance,
                                 cam.transform.position.z);
    cam.transform.position = newCameraPosition;
    cam.transform.LookAt(CameraContainer.transform);
    TotalShiftDistance = 0.0f;
}
```

```
    private void LiftCamera()
    {
        float shiftDistance = this.CalculateShiftDistance();
        Vector3 newCamPosition = new Vector3(cam.transform.position.x,
                             cam.transform.position.y + shiftDistance,
                             cam.transform.position.z);
        cam.transform.position = newCamPosition;
        TotalShiftDistance = shiftDistance;
        cam.transform.LookAt(CameraContainer.transform);
    }
    #endregion
    #region Rotate
    /// <summary>
    /// Rotate camera according to the cursor movements
    /// </summary>
    private void RotateCamera()
    {
        Vector3 mouseCurrentPos = Input.mousePosition;
        float rotateDifferanceOn_X = mousePosionOnPressCameraFunction.x -
                                     mouseCurrentPos.x;
        float rotateDifferanceOn_Y = mousePosionOnPressCameraFunction.y -
                                     mouseCurrentPos.y;
        transform.Rotate(0f, -1f * (rotateDifferanceOn_X *
                                   (rotationFactor)), 0f, Space.World);
        float resultRotationValue = this.CalculateFinalRotation(
                                    rotateDifferanceOn_Y);
        if (TotalShiftDistance > 0.0f && resultRotationValue < 0.0f)
            resultRotationValue = 0.0f;
        CameraContainerPivot.transform.Rotate(resultRotationValue, 0f,
                                              0f, Space.Self);

        mousePosionOnPressCameraFunction = Input.mousePosition;
    }

    private float CalculateFinalRotation(float rotationValue)
    {
        rotationValue = rotationValue * (0.1f);
        float currentPivot_X = CameraContainerPivot.transform.
                               localRotation.eulerAngles.x;
        float finalRotationValue = currentPivot_X + rotationValue;
        if (finalRotationValue > maxRotationAngle)
        {
            return maxRotationAngle - currentPivot_X;
        }
        else if (finalRotationValue < maxRotationAngle &&
                 finalRotationValue > minRotationAngle)
        {
            return rotationValue;
        }
        else if (maxRotationAngle == currentPivot_X ||
                 minRotationAngle == currentPivot_X)
        {
            return 0;
        }
        else
        {
            return minRotationAngle - currentPivot_X;
        }
    }
    #endregion
}
```

APPENDIX B – PlayerCommands Class

```
using UnityEngine;
using System.Collections;

public class PlayerCommands : MonoBehaviour
{
    #region MouseMembers
    private Vector2 mouse1DownPos;
    private Vector2 mouse1UpPos;
    public float mouseMoveBuffer;
    private bool isDrawRectangle = false;
    private float lastMouse1UpTime = 0.0f;
    #endregion

    #region RaycastMembers
    public float rayLength;
    private LayerMask terrainMask = 1 << 8;
    #endregion

    public Texture SelectionTexture;
    public float timeBuffer4Run;

    public void OnGUI()
    {
      /// SELECTION RECTANGLE
        if (isDrawRectangle)
        {
            float width = mouse1UpPos.x - mouse1DownPos.x;
            float height = (Screen.height - mouse1UpPos.y) -
                           (Screen.height - mouse1DownPos.y);
            Rect rect = new Rect(mouse1DownPos.x, Screen.height -
                                 mouse1DownPos.y, width, height);
            GUI.DrawTexture(rect, SelectionTexture,
                            ScaleMode.StretchToFill, true);
        }
    }

    public void Update()
    {
        if (Input.GetButtonDown("Fire1"))
            Mouse1Down(Input.mousePosition);
        if (Input.GetButtonUp("Fire1"))
            Mouse1Up(Input.mousePosition);
        if (Input.GetButton("Fire1"))
            MouseDrag(Input.mousePosition);
    }

    /// <summary>
    /// PICKING
    /// </summary>
    /// <param name="mousePosition"></param>
    private void Mouse1Down(Vector2 mousePosition)
    {
        mouse1DownPos = Input.mousePosition;
        mouse1UpPos = Input.mousePosition;
        RaycastHit hit = Util.RayFromCamera2ScreenPoint(mouse1UpPos,
                                  rayLength);
    if (hit.collider == null)
    {
        UnitManager.GetInstance().ClearSelectedUnitList("Ally");
    }
    else if (hit.collider.tag == "Ally")
    {
        UnitManager.GetInstance().ClearSelectedUnitList("Ally");
    }
}

/// <summary>
/// Move or attack commands according to the picking
/// </summary>
/// <param name="mousePosition"></param>

private void Mouse1Up(Vector2 mousePosition)
{
    mouse1UpPos = Input.mousePosition;
    isDrawRectangle = false;
    if (isJustClick(mouse1DownPos, mouse1UpPos))
    {
        if (UnitManager.GetInstance().GetAllySelectedUnitsCount() ==
            0)
        {
            RaycastHit hit =
              Util.RayFromCamera2ScreenPoint(mouse1DownPos,
              rayLength, ~terrainMask);
            if (hit.collider != null && hit.collider.tag == "Ally")
            {
              UnitManager.GetInstance().
              AddSelectedUnit(hit.collider.gameObject, "Ally");
            }
        }
        else
        {
            RaycastHit hitForAttack = Util.RayFromCamera2ScreenPoint(
                                      mouse1DownPos, rayLength,
                                      ~terrainMask);

            if (hitForAttack.collider != null &&
                hitForAttack.collider.tag == "Enemy" &&
                (hitForAttack.collider.GetComponent("StateManager")
                as StateManager).currentState.GetType() !=
                typeof(DeadState))
            {
                UnitManager.GetInstance().Attack2Target(
                hitForAttack.collider.gameObject, "Ally");
            }

            else
            {
                RaycastHit hit = Util.RayFromCamera2ScreenPoint(
                                      mouse1UpPos, rayLength,
                                      terrainMask);
                //Run command (move with double click)
                if (Time.time - lastMouse1UpTime < timeBuffer4Run)
                {
                    UnitManager.GetInstance().
                    MoveSelectedUnitsToPoint(hit.point,
                    UnitMove.MoveCommand.Run, "Ally");
                }
```

```
                    else

                        UnitManager.GetInstance().
                        MoveSelectedUnitsToPoint(hit.point,
                        UnitMove.MoveCommand.Walk, "Ally");
                }

                lastMouse1UpTime = Time.time;
            }
        }
    }
    /// <summary>
    /// Finding the units that are within the selection area
    /// </summary>
    /// <param name="mouseCurrentPos"></param>
    private void MouseDrag(Vector2 mouseCurrentPos)
    {
        if (!this.isJustClick(mouse1DownPos, mouseCurrentPos))
        {
            Vector3[] recPointsHit = new Vector3[4];
            isDrawRectangle = true;
            mouse1UpPos = mouseCurrentPos;

            Vector2[] recPoints = this.FindSelectionRectanglePoints();
            for (int i = 0; i < recPoints.Length; i++)
            {
                RaycastHit hit = Util.RayFromCamera2ScreenPoint(
                                  recPoints[i], rayLength, terrainMask);
                recPointsHit[i] = hit.point;
            }
            UnitManager.GetInstance().ClearSelectedUnitList("Ally");
            UnitManager.GetInstance().SelectUnitsInArea(recPointsHit);
        }
    }
    /// <summary>
    /// Projection of the selection rectangle
    /// </summary>
    private Vector2[] FindSelectionRectanglePoints()
    {
        float differanceOn_X = mouse1UpPos.x - mouse1DownPos.x;
        float differanceOn_Y = mouse1UpPos.y - mouse1DownPos.y;

        Vector2[] recPoints = new Vector2[4];
        recPoints[0] = mouse1DownPos;
        recPoints[1] = new Vector2(mouse1DownPos.x + differanceOn_X,
                        mouse1DownPos.y);
        recPoints[2] = mouse1UpPos;
        recPoints[3] = new Vector2(mouse1DownPos.x, mouse1DownPos.y +
                        differanceOn_Y);
        return recPoints;
    }

    private bool isJustClick(Vector2 v1, Vector2 v2)
    {
        float dist = Vector2.Distance(v1, v2);
        if (dist < mouseMoveBuffer)
            return true;
        return false;
    }
}
```

APPENDIX C – UnitManager Class

```
using UnityEngine;
using System.Collections;

public class UnitManager : MonoBehaviour
{
    #region members

    private UnitCollection allyUnitCollection = new UnitCollection();

    private UnitCollection enemyUnitCollection = new UnitCollection();

    private UnitCollection allySelectedUnitCollection = new
                                                  UnitCollection();

    private UnitCollection enemySelectedUnitCollection = new
                                                  UnitCollection();
    public float selectionAngleTolerance;
    public float averagePointRadius;

    #endregion

    #region SingletonStructure

    private static UnitManager instance;

    public static UnitManager GetInstance()
    {
        if (instance == null)
        {
            instance=(UnitManager)FindObjectOfType(typeof(UnitManager));
        }
        return instance;
    }
    #endregion

    /// <summary>
    /// Returns the count of selected units
    /// </summary>

    public int GetAllySelectedUnitsCount()
    {
        return allySelectedUnitCollection.GetList().Count;
    }

    public int GetEnemySelectedUnitsCount()
    {
        return enemySelectedUnitCollection.GetList().Count;
    }
```

```
/// <summary>
/// Adding and Removing a unit to/from its collection
/// </summary>
/// <param name="gameObject">gameObject to be added</param>
public void AddUnit(GameObject gameObject)
{
    string unitTag = gameObject.transform.tag;
    if (unitTag == "Enemy")
        enemyUnitCollection.Add(gameObject);
    else
        allyUnitCollection.Add(gameObject);
}
public void RemoveUnit(GameObject gameObject)
{
    string unitTag = gameObject.transform.tag;
    if (unitTag == "Enemy")
        enemyUnitCollection.Remove(gameObject);
    else
        allyUnitCollection.Remove(gameObject);
}
public void RemoveUnitFromSelectedUnitList(GameObject gameObject)
{
    if (gameObject.tag == "Ally" &&
        allySelectedUnitCollection.GetList().Contains(gameObject))
    {
        allySelectedUnitCollection.Remove(gameObject);
    }
}

/// <summary>
/// Adding aunit to the selected  units collection
/// </summary>
/// <param name="gameObject"></param>
public void AddSelectedUnit(GameObject gameObject, string unitTag)
{
    if (unitTag == "Enemy")
        enemySelectedUnitCollection.Add(gameObject);
    else
    {
        allySelectedUnitCollection.Add(gameObject);
        gameObject.SendMessage("SetUnitSelected", true);
    }
}

/// <summary>
/// Clear selected units collection
/// </summary>
public void ClearSelectedUnitList(string unitTag)
{
    if (unitTag == "Ally")
    {
        IList allySelectedUnitList =
              allySelectedUnitCollection.GetList();
        for (int i = 0; i < allySelectedUnitList.Count; i++)
        {
            (allySelectedUnitList[i] as
              GameObject).SendMessage("SetUnitSelected", false);
        }
        allySelectedUnitCollection.Clear();
    }
    else
    {
        IList enemySelectedUnitList =
              enemySelectedUnitCollection.GetList();
        for (int i = 0; i < enemySelectedUnitList.Count; i++)
        {
            (enemySelectedUnitList[i] as
              GameObject).SendMessage("SetUnitSelected", false);
        }
        enemySelectedUnitCollection.Clear();
    }
}

public UnitCollection GetUnitCollection(string unitTag,
                                        bool forAttack)
{
    if (forAttack)
    {
        if (unitTag == "Ally")
            unitTag = "Enemy";
        else
            unitTag = "Ally";
    }


    if (unitTag == "Ally")
        return allyUnitCollection;
    else
        return enemyUnitCollection;
}

/// <summary>
/// Find units which are within the area of projeced selection
/// rectangle
/// </summary>
/// <param name="points"></param>
public void SelectUnitsInArea(Vector3[] points)
{
    Vector3[] vectorsFromGameObject2Points = new Vector3[4];
    IList allyUnitList = allyUnitCollection.GetList();
    for (int i = 0; i < allyUnitList.Count; i++)
    {
        float totalAngle = 0.0f;
        GameObject gameObject = allyUnitList[i] as GameObject;
        for (int j = 0; j < points.Length; j++)
        {
            vectorsFromGameObject2Points[j] = points[j] -
                         gameObject.transform.position;
        }
        for (int k = 0; k < vectorsFromGameObject2Points.Length; k++)
        {
            if (k == 3)
                totalAngle += Vector3.Angle(
                               vectorsFromGameObject2Points[k],
                               vectorsFromGameObject2Points[0]);
            else
                totalAngle += Vector3.Angle(
                               vectorsFromGameObject2Points[k],
                               vectorsFromGameObject2Points[k + 1]);
        }
        if (360f - totalAngle < selectionAngleTolerance)
        {
            allySelectedUnitCollection.Add(gameObject);
            gameObject.SendMessage("SetUnitSelected", true);
        }
    }

}

private void OnApplicationQuit()
{
    instance = null;
}

#region MOVE

/// <summary>
/// Movement as a formation
/// </summary>
/// <param name="destinationPoint"> </param>
public void MoveSelectedUnitsToPoint(Vector3 destinationPoint,
                    UnitMove.MoveCommand command, string unitTag)
{
    Vector3 averagePoint =
              this.CalculateAveragePoint4SelectedUnits(unitTag);
    Hashtable ht = this.FindRelativeDestinationPosition(
                    averagePoint, destinationPoint, unitTag);
    foreach (GameObject go in ht.Keys)
    {
        MoveState ms = new MoveState((Vector3)(ht[go]), go, command);
        go.SendMessage("ChangeState", ms);
    }
}
/// <summary>
/// Average position of the selected units
/// </summary>
/// <returns></returns>
private Vector3 CalculateAveragePoint4SelectedUnits(string unitTag)
{
    float sumXPositions = 0.0f;
    float sumYPositions = 0.0f;
    float sumZPositions = 0.0f;
    IList selectedUnitList;

    if (unitTag == "Ally")
        selectedUnitList = allySelectedUnitCollection.GetList();
    else
        selectedUnitList = enemySelectedUnitCollection.GetList();

    for (int i = 0; i < selectedUnitList.Count; i++)
    {
        Vector3 position = (selectedUnitList[i] as
                              GameObject).transform.position;
        sumXPositions += position.x;
        sumYPositions += position.y;
        sumZPositions += position.z;
    }
    return new Vector3((sumXPositions / selectedUnitList.Count),
                        (sumYPositions / selectedUnitList.Count),
                        (sumZPositions / selectedUnitList.Count));
}
```

```
/// <summary>
/// Finding destination points of units in formation
/// </summary>
/// <param name="averagePoint"></param>
/// <param name="destinatinPoint"></param>
/// <returns></returns>
private Hashtable FindRelativeDestinationPosition(Vector3
                   averagePoint, Vector3 destinatinPoint, string
                   unitTag)
{
    IList selectedUnitList;
    Hashtable ht = new Hashtable();

    if (unitTag == "Ally")
        selectedUnitList = allySelectedUnitCollection.GetList();
    else
        selectedUnitList = enemySelectedUnitCollection.GetList();

    for (int i = 0; i < selectedUnitList.Count; i++)
    {
        Vector3 position = (selectedUnitList[i] as
                           GameObject).transform.position;
        Vector3 relativePosition = averagePoint - position;

        if (this.IsTransformOutOfRange(position, averagePoint,
             unitTag))
        {
            relativePosition = relativePosition.normalized;
        }

        Vector3 finalDestinationPoint = destinatinPoint -
                                        relativePosition;
        ht.Add(selectedUnitList[i] as GameObject,
               finalDestinationPoint);
    }
    return ht;
}

/// <summary>
/// Check unit position for considering or not within the formation
/// </summary>
/// <param name="objectPosition"></param>
/// <param name="averagePoint"></param>
private bool IsTransformOutOfRange(Vector3 objectPosition,
                                   Vector3 averagePoint, string unitTag)
{
    bool result = false;
    float distance = Vector3.Distance(averagePoint, objectPosition);
    UnitCollection selectedUnitCollection;
    if (unitTag == "Ally")
        selectedUnitCollection = allySelectedUnitCollection;
    else
        selectedUnitCollection = enemySelectedUnitCollection;

    if (distance > averagePointRadius * selectedUnitCollection.Count)
    {
        result = true;
    }
    return result;
}
#endregion
```

```
    #region Attack

    public void Attack2Target(GameObject target, string unitTag)
    {
        UnitCollection selectedUnitCollection;

        if (unitTag == "Ally")
            selectedUnitCollection = allySelectedUnitCollection;
        else
            selectedUnitCollection = enemySelectedUnitCollection;

        foreach (GameObject go in selectedUnitCollection)
        {
            AttackState attackState = new AttackState(target, go);
            go.SendMessage("ChangeState", attackState);
        }
    }

    #endregion
}

public class UnitCollection : CollectionBase
{

    public void Add(GameObject gameObject)
    {
        List.Add(gameObject);
    }

    public void Remove(GameObject gameObject)
    {
        List.Remove(gameObject);
    }

    public IList GetList()
    {
        return List;
    }
}
```

APPENDIX D – StateManager Class

```
using UnityEngine;
using System.Collections;

public class StateManager : MonoBehaviour
{
    public State currentState;

    public void Start()
    {
        this.currentState = new IdleState(this.transform.gameObject);
        this.currentState.OnEnter();
    }

    public void Update()
    {
        if (currentState.isActive)
            currentState.Execute();

        else
        {
            if (!(currentState is DeadState))
                this.ChangeState(new
                IdleState(this.transform.gameObject));
        }
    }

    public void ChangeState(State nextState)
    {
        currentState.OnExit();
        currentState = nextState;
        currentState.OnEnter();
    }
}
```

APPENDIX E – IdleState Class

```
using UnityEngine;
using System.Collections;

public class IdleState : State
{
    private UnitIdle ui;
    private UnitAttack ua;

    public IdleState(GameObject self)
    {
        this.go = self;
        this.ua = go.GetComponent("UnitAttack") as UnitAttack;
        this.ui = go.GetComponent("UnitIdle") as UnitIdle;
    }

    public override void OnEnter()
    {
        this.SetWeapon();
        ui.Search();
    }

    public override void Execute()
    {
        if (!ui.isSearching && !ua.keepAttacking)
        {
            ua.StopAttack();
            ui.Search();
        }
        else if (ui.target != null && ui.IsTargetAlive() &&
                 !ui.IsTargetGetOutOfWeaponRange(ua.weapon.weaponRange)
            && !ua.keepAttacking)
        {
            ua.Attack(ui.target);
        }
        else if (ui.target == null)
        {
            ua.StopAttack();
        }
    }

    public override void OnExit()
    {
        ui.StopSearch();
        ua.StopAttack();
    }

    private void SetWeapon()
    {
        if (ua.weapon == null)
            ua.ChangeWeapon(ua.weaponType);
    }
}
```

APPENDIX F – MoveState Class

```
using UnityEngine;
using System.Collections;

public class MoveState : State
{
    private Vector3 movePoint;

    private UnitMove.MoveCommand moveCommand;

    private UnitMove um;

    public MoveState(Vector3 move2Point, GameObject self,
                    UnitMove.MoveCommand pMoveCommand)
    {
        this.movePoint = move2Point;
        this.go = self;
        this.um = go.transform.GetComponent("UnitMove") as UnitMove;
        this.moveCommand = pMoveCommand;
    }

    public override void OnEnter()
    {
        um.enabled = true;
        um.MoveToPoint(this.movePoint, this.moveCommand);
    }

    public override void Execute()
    {

        if (um.tempCommand == UnitMove.MoveCommand.Stop)
            this.isActive = false;

    }

    public override void OnExit()
    {
        um.StopMove();
        um.enabled = false;
    }
}
```

APPENDIX G – AttackState Class

```
using UnityEngine;
using System.Collections;

public class AttackState : State
{
    private GameObject target;
    private UnitAttack ua;
    private UnitMove um;
    private float enemyGroupRange;
    private float targetMoveBuffer = 15.0f;
    private float weaponRangeBuffer = 5.0f;
    private Vector3 targetLastPosition;
    private bool stillRun2Target;

    public AttackState(GameObject target, GameObject self)
    {
        this.target = target;
        this.targetLastPosition = target.transform.position;
        this.go = self;
        this.ua = go.GetComponent("UnitAttack") as UnitAttack;
        this.um = go.GetComponent("UnitMove") as UnitMove;
        this.enemyGroupRange = ua.enemyGroupRange;
    }

    public override void OnEnter()
    {
        this.SetWeapon();
        this.Attack();
        this.targetLastPosition = target.transform.position;
    }

    public override void Execute()
    {
        if (target != null)
        {
            this.Attack();
        }
        else
        {
            ua.StopAttack();
            this.target = null;
            this.target = this.FindNextTarget();
            if (this.target != null)
                this.Attack();
            else
                this.isActive = false;
        }
    }

    public override void OnExit()
    {
        ua.StopAttack();
        um.enabled = true;
    }
```

```
private void Attack()
{
    if (this.FindTargetMoveDistance() > this.targetMoveBuffer)
    {
        um.enabled = true;
        this.targetLastPosition = target.transform.position;
        ua.StopAttack();
        um.MoveToPoint(target.transform.position,
                       UnitMove.MoveCommand.Run);
    }
    else
    {
        if (IsOpenFire())
        {
            this.stillRun2Target = false;
            this.targetLastPosition = target.transform.position;
            if (!ua.keepAttacking)
            {
                ua.Attack(target);
            }
            um.StopMove();
        }
        else
        {
            if (!this.stillRun2Target)
            {
                this.stillRun2Target = true;
                um.enabled = true;
                this.targetLastPosition = target.transform.position;
                ua.StopAttack();
                um.MoveToPoint(target.transform.position,
                               UnitMove.MoveCommand.Run);
            }
        }
    }
}

private bool IsOpenFire()
{
    bool result = false;

    float distance = ua.FindTargetDistance(this.target);
    if (distance < ua.weapon.weaponRange - weaponRangeBuffer &&
       ua.weapon.IsTargetClear(go, target))
    {
        result = true;
    }

    if (result == true)
    {
        this.weaponRangeBuffer = 0.0f;
    }
    else
    {
        this.weaponRangeBuffer = 5.0f;
    }

    return result;
}
```

```
    private GameObject FindNextTarget()
    {
        string opponentTag = string.Empty;
        Collider[] colliders = Physics.OverlapSphere(targetLastPosition,
                                                      enemyGroupRange);

        foreach (Collider col in colliders)
        {
            if (go.tag == "Enemy")
                opponentTag = "Ally";
            else
                opponentTag = "Enemy";
            if (col.transform.tag == opponentTag)
            {
                if (ua.weapon.IsTargetClear(go, col.gameObject))
                    return col.gameObject;
            }
        }
        return null;
    }

    private void SetWeapon()
    {
        if (ua.weapon == null)
            ua.ChangeWeapon(ua.weaponType);
    }

    private float FindTargetMoveDistance()
    {
        return ua.FindTargetMoveDistance(target, targetLastPosition);
    }
}
```

APPENDIX H – DeadState Class

```
using UnityEngine;
using System.Collections;


public class DeadState : State
{
    public DeadState(GameObject self)
    {
        this.go = self;
    }

    public override void OnEnter()
    {
        AIUtil aiu = this.go.GetComponent("AIUtil") as AIUtil;
        if (aiu != null)
            aiu.RemoveFromTeam();

        UnitManager.GetInstance().
            RemoveUnitFromSelectedUnitList(this.go);

        UnitManager.GetInstance().RemoveUnit(this.go);
    }

    public override void Execute()
    {
    }

    public override void OnExit()
    {
    }
}
```