

California State University, San Bernardino

CSUSB ScholarWorks

Theses Digitization Project

John M. Pfau Library

2011

Maerken: A multiplayer role playing game

Chanh Ho

Follow this and additional works at: <https://scholarworks.lib.csusb.edu/etd-project>



Part of the [Game Design Commons](#)

Recommended Citation

Ho, Chanh, "Maerken: A multiplayer role playing game" (2011). *Theses Digitization Project*. 3957.
<https://scholarworks.lib.csusb.edu/etd-project/3957>

This Project is brought to you for free and open access by the John M. Pfau Library at CSUSB ScholarWorks. It has been accepted for inclusion in Theses Digitization Project by an authorized administrator of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.

MAERKEN
A MULTIPLAYER ROLE PLAYING GAME

A Project
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Chanh Ho
December 2011

MAERKEN

A MULTIPLAYER ROLE PLAYING GAME

A Project
Presented to the
Faculty of
California State University,
San Bernardino

by

Chanh Ho

December 2011

Approved by:


Dr. David A. Turner, Advisor, School of
Computer Science and Engineering

11/30/11
Date


Dr. Tong Lai Yu


Dr. Arturo I. Concepcion

© 2011 Chanh Ho

ABSTRACT

Maerken is a three-dimensional (3D) role playing game (RPG) being developed at CSUSB. The game depicts the journey of a Gitan, a soldier who sacrificed humanity to fight for humanity in the Great War. Awakened after a thousand years of sleeping, the Gitan has to stop the Valkirs' plan to reopen the path between worlds to allow them to conquer the planet. Maerken gameplay resembles the gameplay of Vindictus [6], a multiplayer RPG developed by Nexon, and Fable [2], an action RPG developed by Lionhead Studios. Maerken was formerly named Mythic. Since January 2010, the Maerken development team has been using Unreal Development Kit (UDK) to develop the game. The purpose of this project is to develop a base implementation of Maerken using the UDK, a game framework that supports development of 3D games. In this report, we describe the game development process that was used to develop Maerken using UDK. The process includes game design, game implementation and game distribution. To describe the game design of Maerken, we explain the design for gameplay, user interface, level design, art, sound and music. To describe the implementation of Maerken, we explain how to use the game engine to implement the game logic, network, user interface and game serialization. The level design process and the game content management system are also included in this report. Finally, to describe the distribution process, we explain how to package and distribute the game to the users. This report also includes a discussion of the directions for future development of Maerken.

ACKNOWLEDGEMENTS

I would like to thank all the people with whom I have worked while pursuing my master's degree at California State University, San Bernardino (CSUSB). I wish I could list all their names but the list would be too long and I would still probably leave some people out. Studying in the School of Computer Science and Engineering at CSUSB has been a tremendous learning experience, both personally and professionally.

Thank you to the following faculty of the Computer Science and Engineering department for their invaluable guidance, advice, support, help, and patience during this project's long gestation: Dr. David Turner, Dr. Arturo Concepcion and Dr. Tong Yu.

Finally, thanks to my parents who encouraged me all along.

TABLE OF CONTENTS

<i>Abstract</i>	iii
<i>Acknowledgements</i>	iv
<i>List of Figures</i>	x
1. Introduction	1
1.1 Maerken Project	1
1.1.1 Overview	1
1.1.2 History	1
1.1.3 Current Status	2
1.2 Project Scope	3
1.3 Development Tools	4
1.3.1 Unreal Development Kit	4
1.3.2 Visual Studio 2010 and nFringe	4
1.3.3 Flash Professional	4
1.3.4 MotionBuilder 2011	5
1.3.5 World of Warcraft Client and WoW Model Viewer	5
1.4 Programming Languages	5
1.4.1 UnrealScript	5
1.4.2 ActionScript 2.0	6
1.4.3 C++	7

1.5	Definitions, Acronyms and Abbreviation	7
2.	<i>Game Design</i>	9
2.1	Story	9
2.2	Gameplay	9
2.2.1	Character Classes	10
2.2.2	Non-player Characters	12
2.2.3	Player Interactions	13
2.2.4	Game Mechanics	13
2.3	User Interface	15
2.3.1	General Game Controls	15
2.3.2	Main Menu	16
2.3.3	Choose Character Menu	17
2.3.4	Create Character Menu	18
2.3.5	Mid Game Menu	19
2.3.6	Inventory Menu	19
2.3.7	Head-Up Display (HUD)	20
2.4	Level Design	21
2.5	Art	22
2.6	Sound and Music	22
3.	<i>Architectural Overview</i>	24
3.1	Overview	24
3.2	Class Structure and Naming Conventions	25
3.2.1	Class Organization	25
3.2.2	Class Names	25
3.2.3	Variable Names	26
3.2.4	Function Names	26

3.2.5	State Names	26
3.3	Critical UnrealScript Concepts	27
3.3.1	States	27
3.3.2	Replication	28
3.3.3	Default Object Initialization	28
3.3.4	Editor Variables	29
3.3.5	Object Creation and Dynamic Loading	29
3.3.6	Unreal Engine Game Flow Concepts	30
3.4	Class Design	33
3.4.1	Overview	33
3.4.2	MyGame	33
3.4.3	MyPlayerController	36
3.4.4	MyPawn	39
3.4.5	MyPlayerPawn	42
3.4.6	MyNPCPawn	44
3.4.7	MyAI	45
3.4.8	MySkill	47
3.4.9	MyProjectile	50
3.4.10	MyUI	52
4.	<i>Implementation</i>	53
4.1	Multiplayer Implementation	53
4.1.1	Overview	53
4.1.2	Function Replication	53
4.1.3	Animation Replication	54
4.1.4	Variable Replication	55
4.1.5	Network Game Creation Process	55

4.1.6	Network Game Search and Join Process	56
4.2	AI Implementation	56
4.2.1	Overview	56
4.2.2	Unreal Engine AI Support	56
4.2.3	MyAI Class	57
4.2.4	AI Variables	57
4.2.5	AI States	57
4.3	User Interface Implementation	58
4.3.1	Overview	58
4.3.2	ScaleformGFX Components	58
4.3.3	Main Menu Implementation	60
4.3.4	MyUI_MainMenu ActionScript Functions	64
4.3.5	Mid Game Menu Implementation	65
4.3.6	MyUIMidGameMenu ActionScript Functions	66
4.3.7	Head-Up Display Implementation	67
4.3.8	MyUI_HUD ActionScript Functions	68
4.4	Game Serialization Implementation	69
4.4.1	Overview	69
4.4.2	SaveGameDLL Implementation	69
4.4.3	SaveGameDLL Usage	70
5.	<i>Game Content Creation, Management and Distribution</i>	72
5.1	Art Asset Management	72
5.1.1	Content Importing Process	72
5.1.2	Content Creation in Unreal Development Kit	74
5.2	Level Creation	77
5.3	Game Distribution	78

5.3.1	Configuring and Compiling	78
5.3.2	Cooking and Packaging	79
6.	<i>Conclusion and Future Direction</i>	80
6.1	Conclusion	80
6.2	Future Direction	81
	<i>References</i>	83

LIST OF FIGURES

1.1	Maerken Game	3
2.1	Warrior	10
2.2	Mage	11
2.3	Main Menu Layout	16
2.4	Choose Character Menu Layout	17
2.5	Create Character Menu Layout	18
2.6	Mid Game Menu Layout	19
2.7	Inventory Menu Layout	20
2.8	HUD Layout	21
3.1	Maerken Core Classes	24
3.2	UnrealScript Game Flow	31
3.3	MyGame Class	34
3.4	MyPlayerController Class	37
3.5	MyPawn Class	40
3.6	MyPlayerPawn Class	43
3.7	MyNPCPawn Class	44
3.8	MyAI Class	46
3.9	MySkill Class	48
3.10	MyProjectile Class	50
3.11	MyUI Class	52

4.1	Main Menu	61
4.2	Choose Character Menu	62
4.3	Create Character Menu	63
4.4	Mid Game Menu	66
4.5	Head-Up Display	68
5.1	Unreal Material Editor	75
5.2	Cascade Emitter Editor	76
5.3	Unreal AnimTree Editor	77
5.4	UnrealFrontend Interface	79

1. INTRODUCTION

1.1 Maerken Project

1.1.1 Overview

We want to develop a game that allows players to become a part of a magnificent world, exploring vast landscapes and deep dungeons, delving into the multiple cultures of the land and finding out the origins of the Mythic. Along the way, the players will meet, trade, and dispatch a variety of characters and enemies to meet their goals. Finally, players can join up with each other to undertake quests and discover the secrets of the world together.[4]

This is the vision for Maerken, formerly known as Mythic, developed by Mark Chapman, the former Maerken lead game designer. This master's project is an iteration of the Maerken project. Its purpose is to produce a base implementation of Maerken. In this iteration, we use UDK to implement the basic features of a multiplayer RPG. The game design for this iteration is simple. We focus on developing the RPG gameplay logic as well as a basic user interface system and AI system that can be extended to a larger system.

1.1.2 History

The original name of Maerken was Mythic. The Mythic project began in 2009. Since then, a few iterations of Mythic development have been done. The previous versions

of Maerken were implemented in Java, C++ and C#. This master's project uses the UnrealScript language to implement its version of Maerken. It also uses a small amount of C++ and ActionScript. The project has undergone some big changes in graphics engines and library dependencies. The previous versions of the game used Horde3D, OGRE [7] and an engine developed by William Herrera as their graphics engines. This master's project uses Unreal game engine to develop the game. The name Mythic is a copyright protected name of Mythic Entertainment so we changed the name to Maerken [3].

1.1.3 Current Status

Currently, the Maerken project is being developed by the Maerken team at CSUSB using the Unreal engine. The project has undergone some restructuring. With the participation of the Art Institute of California - Inland Empire, we have access to custom made game content so we can stop using content from other games. This will enable the team to implement a more complex combat system. Figure 1.1 shows the current combat scene of Maerken.



Fig. 1.1: Maerken Game

1.2 Project Scope

This project report describes the game development process of this iteration of Maerken. This includes the game design, code design, important implementation concepts, content pipeline and distribution of the game. The game design includes gameplay, user interface, conceptual level and art, sound and music. The code design includes our coding conventions and Unreal Script class design. Important implementation concepts include multiplayer, user interface, game state serialization and game AI. We do not intend to cover game content creation and level design in detail.

1.3 Development Tools

This section introduces the tools that we use in developing Maerken.

1.3.1 Unreal Development Kit

Unreal Development Kit (UDK) [5] is a game development framework that provides a variety of tools used to create 3D games on personal computer (PC) and iPhone operating system (iOS) platforms. UDK provides an editing environment that allow us to create gameplay levels, import art assets to be used in the game and manage and edit game contents. UDK also provides us with a high level scripting language to implement the game logic.

1.3.2 Visual Studio 2010 and nFringe

Visual Studio is an integrated development environment (IDE) developed by Microsoft. It supports many programming languages, including C, C++, C#. The Visual Studio add-on nFringe was developed by Pixel Mine to support UnrealScript. Visual Studio combined with nFringe allows us to work with UnrealScript in an integrated environment. This helps reduce the time and effort we spend learning and using Unreal Script. We also use Visual Studio in developing a dynamic linked library in the C++ language to support game serialization.

1.3.3 Flash Professional

Flash Professional is an application developed by Adobe Systems. Flash provides the environment to create flash movies and edit ActionScript code. We use Flash Professional to develop Scaleform flash movies that we import into the Unreal engine to implement the game user interface.

1.3.4 *MotionBuilder 2011*

Motion Builder is an application developed by Autodesk. It provides a real time 3D character animation environment. In this project, we use Motion Builder to edit the bone structure from FBX files exported from World of Warcraft Model Viewer to make them ready for importing into the Unreal engine.

1.3.5 *World of Warcraft Client and WoW Model Viewer*

World of Warcraft is a massive multiplayer online role playing game (MMORPG) developed by Blizzard Entertainment. WoW Model Viewer is an open-source application that uses the World of Warcraft MPQ files to export character models. In this project, we use these two programs to obtain 3D character models for prototype development of the game.

1.4 *Programming Languages*

This section introduces the programming languages we use in developing Maerken.

1.4.1 *UnrealScript*

UnrealScript is the scripting language designed specifically to work with the Unreal engine. It is an object oriented language designed based on the Java programming language. The following are some important characteristics of UnrealScript:

- UnrealScript is a case-insensitive language.
- In UnrealScript, only one class is defined in a script file. The class declaration ends with a semi-colon at the end of the statement.
- UnrealScript only has one package layer.

- In UnrealScript, there is no import statements; script references across all the packages available in the script folder.
- An UnrealScript class must have a parent, except the Object class, which is the highest in the class hierarchy.
- In UnrealScript, there is function overriding but there is no function overloading.
- The usage of static and dynamic arrays is different in UnrealScript.
- There are no global variables, constants or functions in UnrealScript.
- UnrealScript includes class state as a built-in language feature, which provides a convenient way to implement object behavior.
- UnrealScript has a built-in network architecture.
- UnrealScript supports live editing in the editor, which allows the developer to change the value of variables inside the editor without having to recompile the code.
- Object initialization in UnrealScript can be done in the default properties block.
- UnrealScript supports object serialization by using configuration files (ini files).

Unreal Script is the most important language in this project. It is used in implementing logical gameplay, game controls and visual representation of the game.

1.4.2 ActionScript 2.0

ActionScript is the programming language of the Adobe Flash Platform [1]. The current version of UDK supports user interface using Scaleform movies written in ActionScript 2.0 [5]. In this project, we use ActionScript 2.0 in Scaleform Flash movies to interact with the UnrealScript user interface classes. Since ActionScript

3.0 will be supported soon in UDK [5], we will use ActionScript 3.0 in the future iterations of Maerken.

1.4.3 C++

C++ is a very widely known programming language. C++ is also be used in the native engine code of UDK [5]. It is claimed to be executed 20 times faster than UnrealScript. The native C++ code accessibility is only available to Unreal Engine licensees. However, UDK allows the users to write C++ code to interact with the UnrealScript code using DLLBind feature. In this project, we use C++ to implement a dynamic linked library (DLL) used in game serialization.

1.5 Definitions, Acronyms and Abbreviation

- 3D: refers to three dimensional computer graphics, which models three dimensional objects.
- 2D: refers to two dimensional computer graphics. In this project, 2D graphics is used for presenting textures.
- LAN: local area network, a computer network that connects computers and devices in a limited geographical area.
- Multiplayer: a game mode that allows two or more players to play together in the same game environment.
- NPC: non-player character, a game character which is not a player controlled character.
- AI: artificial intelligence, a system controlling the behaviors of the NPC in Maerken.

- RPG: role playing game, a type of video game in which player play a role of a character in a story context.
- HUD: head-up display, a display in game that shows game information. HUD is usually displayed on top of everything else in the game.
- UI: user interface. In this project, it refers to the system that allow player to interact with the game.
- Spawn: a function to create new actors in UnrealScript with a starting location and rotation.
- Tick: a game tick, which is a cycle of the game loop.
- Map: refers to a level created by level designers in the game.
- Menu map: the map created to display the main menu. In this map, the player only interact with the game through menus.
- Gameplay map: the map in which the player can have control over a character model and interact with the game world through that character.
- Seamless Travel: a method to transition from map to map without disconnecting clients in multiplayer games.
- Replication: a method of communication between server and clients in Unreal engine.
- DLL: dynamic link library.
- DLL bind: an UnrealScript feature that allows us to write code in C and use it in UnrealScript.
- Vindictus: a mutiplayer RPG developed by Nexon [6].
- Fable: an action RPG developed by Lionhead Studios [2].

2. GAME DESIGN

2.1 *Story*

The story for the game is being developed by the Maerken team at CSUSB. We do not attempt to implement the full story of Maerken in this iteration. Rather than that, we implement a short dungeon, which player has to go through at the start of the story. The story begins when a Gitan wakes up in an unknown ruins and has to fight his way out to the open. On his way out, he meet an adventurer who has lost his friend to an undead boss and needs assistance. The Gitan needs to kill the boss in order to get out of the ruins because the boss blocked the only way out.

2.2 *Gameplay*

This section explains how Maerken is played. The game starts with a main menu allowing the player to choose a character and start a game or join an existing game. When the player transitions to a gameplay level, he/she can control the character to move around the 3D world and interact with the game world objects and other player characters and NPCs in the game. The player can choose from two character classes to play the game. In the game, the player can obtain and equip various types of items to enhance the combat ability. The game can be played in single player mode or multiplayer of 2 to 4 players on LAN.

2.2.1 Character Classes

Warrior

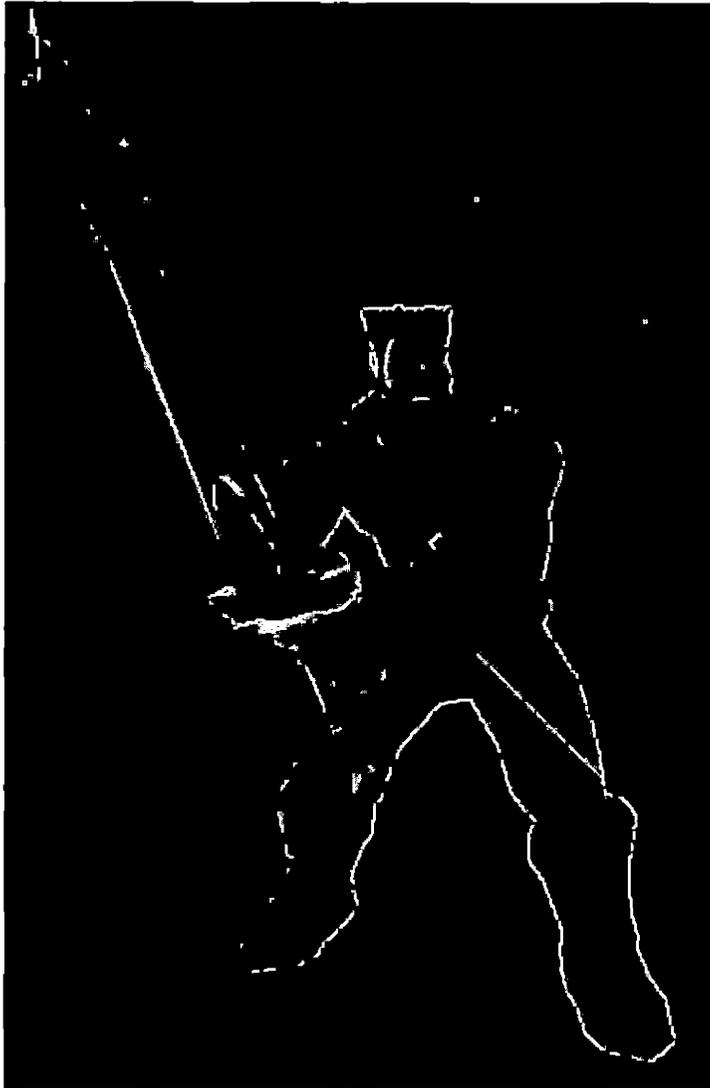


Fig. 2.1: Warrior

Warrior is a melee class. (See Figure 2.1.) A warrior uses two-handed blades to attack the enemies. He can use cleaving attack to hit multiple enemies in front of

him. He wears heavy armor to reduce the damage done to him. The warrior has a class skill called shockwave, which deals damage to enemies in front of the warrior them.

Mage



Fig. 2.2: Mage

Mage is a range class. (See Figure 2.2.) A mage uses staves to perform attacks and spells. She can do a great amount of damage from a long distance. She wears robes that enhance her magic ability. The mage has a class skill called fireball, which releases a fire ball that hits a target and any nearby enemies with a portion of damage dealt to the primary target.

2.2.2 *Non-player Characters*

NPCs in the game include humans, undeads and beasts. The NPCs in the dungeon are listed as follows:

- Wolf: melee fighter, has bite skill.
- Human helper: a range fighter, has heal skill.
- Undead: melee and range fighter, has fire arrow skill.
- Boss: melee fighter, has battle roar skill.

NPCs reaction toward player can be helpful, friendly, neutral or hostile. The reaction is determined by the following rules:

- A helpful NPC attacks the NPCs that attack the player.
- A friendly NPC does not interact with player.
- A neutral NPC does not attack the player automatically but will retaliate if the player provoke it.
- A hostile NPC will attack the player if he/she is in sight.
- A character belongs to a faction.
- The initial reaction of an NPC toward a player character is determined by their factions and the faction relationship table.
- When a player attacks a neutral NPC, the NPC will become hostile.

2.2.3 Player Interactions

In Maerken, a player character can interact with the game world in the following ways:

- A player character can move around the game world.
- A player character can attack neutral or hostile NPCs.
- A player character can get quests and rewards from friendly and helpful NPCs.
- A player character can obtain items by defeating NPCs or completing quests.
- A player character can equip items in the inventory to enhance the combat ability. A character can only have one weapon and one armor equipped at a time.

2.2.4 Game Mechanics

Character Stats

A character in Maerken has the following stats:

- Power: determines the amount of damage or healing that the character can do to its enemies.
- Speed: determines the speed of the character's weapon swings and spell casts.
- MoveSpeed: determines how fast the character can move.
- HitPoint: determines the durability of the character in battle.
- Armor: determines the amount of damage absorption of the character from enemy attacks.

Character Skill

Every character in Maerken can perform one character skill, which can cause damage to one or multiple enemies, or heal an amount of hitpoint lost for one or multiple allies. The damage or healing effect of a skill is based on the character's power. The following are the skills of the NPCs in the dungeon:

- Heal: heals a target for an amount of hitpoint lost.
- Bite: bite a target for a large amount of damage.
- Fire arrow: shoot a fire arrow at the target, deals a large amount of damage.
- Battle roar: increases the caster's and nearby allies an amount of power for a period of time.

Leveling a Character

The player gains experience on defeating enemies and completing quests. The experience gained can increase the player character's level. For each level, the player gains some additional stats points to make the character stronger in combat.

Item

In Maerken, the player can get various types of item to enhance their combat ability or to fulfill some NPC's request. The item types are listed as follows:

- Weapons: can increase the character's power and speed.
- Armors: can increase the character's armor, movement speed and hitpoint.
- Consumables: can restore hitpoint, or affect the stats momentarily.
- Quest items: can be used to finish quests.

Weapons and armors only affect the player character's stats if they are equipped. The character can only equip one weapon and one armor at a time. An item value is determined by its item level and quality. An item with a higher level has more stats modification. An item with a higher quality can affect more stats.

- Common items: only increase power or armor.
- Rare items: increase one additional stats.

Quest

In Maerken, the player will meet friendly characters who need help. The help requests can be rejected or accepted as quests. Completing quests rewards the player with experience or items. The following are some of the quest objective types:

- Kill a number of NPCs.
- Find a number of quest items.
- Go to a specific place in the world.

2.3 User Interface

2.3.1 General Game Controls

Movement

- A: move the character to the left.
- D: move the character to the right.
- W: move the character forward.
- S: move the character backward.

Action

- Left mouse click: perform attack.
- Right mouse click: perform the class skill.
- I: open the inventory menu.

2.3.2 Main Menu

The main menu is displayed when the player starts the game from the run shortcut. From here, the player can start a game, join an existing game, choose the character to play with and quit the game.

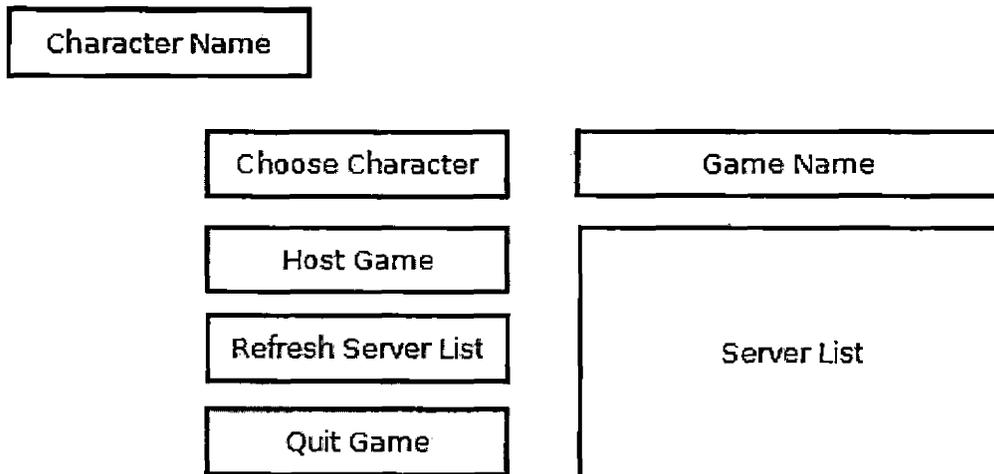


Fig. 2.3: Main Menu Layout

The main menu has the following items:

- A button to allow the player to choose character to start the game with.
- A text display to show the current player character.
- A button to create a LAN game (which is also used to start a single player game).

- A text input to allow the player to type in the game name.
- A list of hosts which are currently running on the local network. When the player clicks on an item on the list, it will attempt to join the corresponding game.
- A button to refresh the hosts.
- A button to exit the program.

2.3.3 Choose Character Menu

The choose character menu is displayed when the player clicks on the choose character button in the menu, or when the player finishes creating a new character. From here, the player can create a new character or choose one of the saved characters to play.

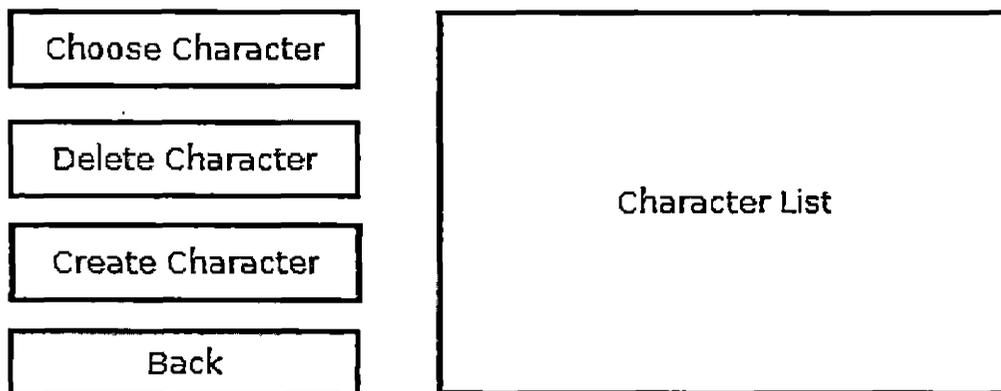


Fig. 2.4: Choose Character Menu Layout

The choose character menu has the following items:

- A button to allow the player to create a new character.
- A button to allow the player to go back to main menu with the selected character.

- A button allow the player to delete a character from the character list.
- A list to display saved characters on the computer. When the player clicks on an item on the list, it will switch the current character to the corresponding character.

2.3.4 Create Character Menu

The create character menu is displayed when the player clicks on the create character button. From here, the player can create a new character by choosing the character class and the character name.

The diagram illustrates the layout of the 'Create Character Menu'. It consists of several rectangular boxes arranged in a grid-like structure. On the left side, there are four stacked boxes: 'Character Name', 'Character Class', 'Create Character', and 'Back'. On the right side, there are three boxes: 'Name', a radio button selection box containing 'Warrior' (with a solid black circle) and 'Mage' (with an empty circle), and another empty radio button selection box.

Fig. 2.5: Create Character Menu Layout

The create character menu has the following items:

- A text input to allow the player to choose the new character name.
- A selection set to allow the player to choose the new character class.
- A button to create the new character based on the chosen name and class.

2.3.5 Mid Game Menu

The mid game menu is displayed when the player activates the menu while playing. The key for activating this menu is defined in the game controls section. From here, the player can save or exit the game.

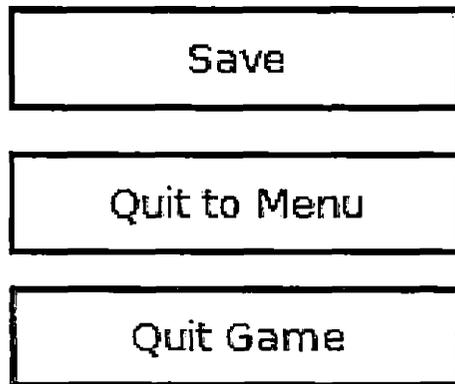


Fig. 2.6: Mid Game Menu Layout

The mid game menu has the following items:

- A button to save the character information and the game progress.
- A button to save the game and quit to main menu.
- A button to exit the game.

2.3.6 Inventory Menu

The inventory menu is displayed when the player activates the menu while playing. The key for activating this menu is defined in the game controls section. From here, the player can manage the equipments and items in the character inventory and see the character stats change when the equipment changes.

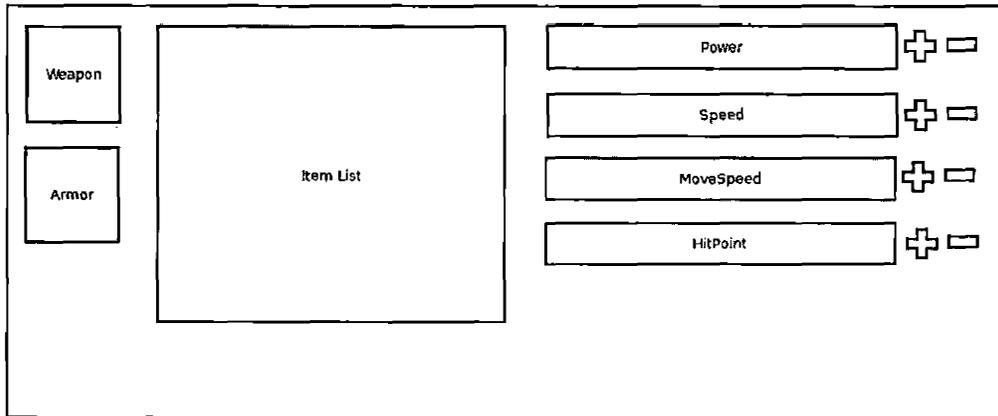


Fig. 2.7: Inventory Menu Layout

The inventory menu has the following items:

- An equipment set to display the player character's equipped items.
- A list of items that player character has in the inventory. When player click on an item, it will attempt to equip the item if it is a weapon or an armor, or use it if it is a consumable item.
- 4 text display contain 4 character stats including plus and minus button to modify stats when the player levels up.

2.3.7 Head-Up Display (HUD)

The HUD is displayed when the player starts playing the game (hosts or joins a game). It displays the player information such as the player's hitpoint and the cooldown of skills.

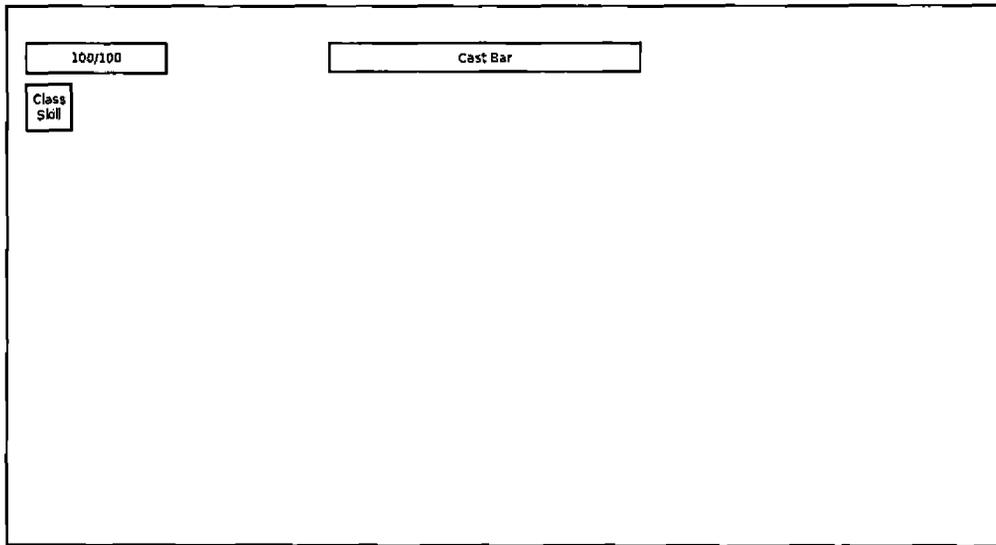


Fig. 2.8: HUD Layout

The game HUD has the following elements:

- A progress bar to display the player's hitpoint.
- An icon to display the class skill icon and the cooldown remaining of the skill.
- A progress bar to display the spell casting.

2.4 Level Design

We have 2 maps in the game, one is the entry map for displaying main menu, the other one is the dungeon to let the player actually play the game. The entry map is a plain room with a ruins texture. It has a dark gate to indicate that the player can go through it to move to the dungeon map. The dungeon map has 5 rooms listed as follows:

- Starting room: the player starts here after entering the game. The room has a door that leads to the second room.

- Second room: the player encounters an enemy in here.
- Third room: the player encounter 2 enemies in here.
- Fourth room: the player encounters 3 enemies in here. This time, a help NPC appears and helps the player fight the enemies. He also asks the player to help him kill the boss in the next room to avenge his friend.
- Boss room: the player and the helper fight the boss.

2.5 Art

We use character models and animations from the World of Warcraft client. The models we use are listed as follows:

- We use a human male model with a plate armor set as the warrior class model.
- We use a female human model with a cloth armor set as the mage class model.
- We use a direwolf model as a wolf model.
- We use a undead male model with mail armor set as the hostile NPC.
- We use a human male model with cloth armor set as the help NPC.
- We use a Arthas model as the boss model.

2.6 Sound and Music

Josh Richardson is responsible for sound and music in the game. The following are the sound and music presented in the game:

- Background music for menu map
- Background music for dungeon map

- Encounter music
- Boss fight music
- Attack sounds for the following characters: warrior, mage, help NPC, wolf, undead enemy and boss
- Casting sounds for the following skills: fireball, shockwave, bite, heal, fire arrow and battle roar
- Interface sounds: clicking on a button, choosing a list item and starting a game
- Sounds for equipping weapons and armors

3. ARCHITECTURAL OVERVIEW

3.1 Overview

In this section, we introduce the coding conventions we follow in the project and explain the purpose and structure of the important UnrealScript classes in Maerken package. We also explain the important concepts of UnrealScript coding and how they affect the project. Figure 3.1 shows the core classes of Maerken and their engine parent classes.

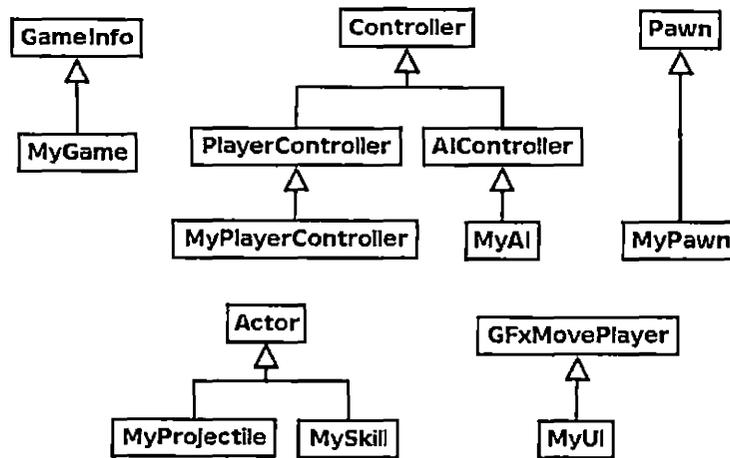


Fig. 3.1: Maerken Core Classes

3.2 *Class Structure and Naming Conventions*

This section explains the structure of the UnrealScript classes that we implement in the project and the naming conventions that we follow.

3.2.1 *Class Organization*

A Maerken class is written in the following order:

- Class comment block
- Class declaration
- Variables grouped by functionality
- Initialization functions
- Other functions grouped by functionality
- States
- DefaultProperties block

3.2.2 *Class Names*

We use nouns to name our classes.

We always start our class names with “My”, followed by the general name for the class. If a class inherits an engine class, the engine class name should be put after “My” with the exception of children of Actor and Object classes.

Beyond these rules, we allow flexibility in naming our classes. However, we still try to follow some patterns, which are listed as follows.

- A custom class in the project should have a descriptive name.

- If a class has 3 or more children, the child names should use the parent class name as a prefix. For example, the children of MyUI class are MyUI.MainMenu, MyUI.MidGameMenu, MyUI.InventoryMenu and MyUI.HUD.
- A class should not have more than one prefix. For example, we use MyPlayer-Pawn.Mage instead of MyPawn.Player.Mage.

3.2.3 *Variable Names*

We separate our variables into groups based on the functionality of the variables. Some common group names that we use are animation, logic and combat.

We use descriptive nouns to name our variables.

We use upper case letters for the start of every word in the variable names, for example CharacterName and MovementSpeed. For local variables declared in functions, we allow the use of short names, for example i, Loc and Rot to make them different from the class variables that have similar meaning.

3.2.4 *Function Names*

We use descriptive verbs to name our functions.

We use upper case letters for the start of every word in the function names, for example ApplyAttack, PerformClassSkill.

3.2.5 *State Names*

We use descriptive adjectives to name our classes.

We use upper case letters for the start of every word in the state names, for example PerformingClassSkill, Attacking, Dead.

3.3 Critical UnrealScript Concepts

3.3.1 States

UnrealScript supports states at the programming language level. States provides a simple way of managing complex object behavior. In UnrealScript, a object is always in one state. The object's state determine which actions it can perform. For example, a pawn can only attack or perform a skill if it is in "Neutral" state. UnrealScript states provide these benefits:

- States allow us to write state-specific functions, which are implemented in states. This allow us to use different versions of a functions based on the object's state.
- In UnrealScript, states also utilize latent functions, which are functions that execute and return after a certain amount of game time has passed. Such functions allows us to write game code exactly the way we want it to happens. For example, in AI patrolling state, we want the AI to choose a random destination, move to the destination, then wait 1 second before deciding where to go next, this state code can be written:

```
state Patrolling {
Begin:
    while (true) {
        MoveTo(GetNextRandomDestination());
        Sleep(1);
    }
}
```

In Maerken, we use states to implement the behavior of the pawns and the AI controllers.

3.3.2 Replication

UnrealScript uses a replication mechanism to communicate between the server and clients. There are 3 types of replication operations:

- Actor replication: a client maintains a set of actors that are duplicated from those on the server. The server has the complete authority on those actors. However, the client can have an approximate version of those actors. The client can change the properties of those actors, but eventually, they will be synchronized to have the exact values of the server actors.
- Variable replication: in an actor, only variables defined in the replication block are replicated to the clients. When it is the time for network update, if the actor's replicated variables change their values, the server will send the updated values to the clients. If a replicated variable is also declared 'renotify', the function `ReplicatedEvent()` is also called on client to let the client code perform update operations.
- Function replication: this is the only way a client can send information to the server. A function can be routed to be called on the server or a remote client. In Maerken, we only use function replication to let a client request actions on server.

3.3.3 Default Object Initialization

UnrealScript has a unique way of initializing an object. The object variables can have their default values defined in the default properties block. These default values can be accessed later, even when the variables change their values.

3.3.4 Editor Variables

UnrealScript supports variable editing in the editor. When a variable is declared editor editable, the editor user can edit its value without having to recompile the code. This is convenient for game designers to test the gameplay variables using the editor so they can see the effect of changing them right inside the editor. This also allows the designers to create many objects of a class with different initializations using archetypes.

3.3.5 Object Creation and Dynamic Loading

There are 2 ways of creating a new object in UnrealScript.

- Creating a new object from the class definition: this method is the standard way to create an object in many languages. In order to create a new object of a class, we call the 'new' operator if the class is not an actor, or call the `Spawn()` function if the class is an actor.
- Creating a new object using a template: this method allows us to create many templates of a class and save them in content packages, then use them to create new in-game objects when needed. For example, we can create many versions of `MyNPCPawn_Wolf` class to make different color wolves, then we spawn them in the game. In order to do that, we have to use the `DynamicLoadObject()` function to load the template from a content package, then use it as a parameter in 'new' or `Spawn()`. We should not load the template and use them directly because that will modify the game package and cause an engine error.

3.3.6 Unreal Engine Game Flow Concepts

Game Initialization

According to UDK documentation [5], the general flow of events when we start an Unreal game is:

1. The engine is initiated.
2. The engine loads a map.
3. The game type is set.
4. Gameplay is initialized.
5. `GameInfo::InitGame()` is called to initialize the gametype.
6. `Actor::PreBeginPlay()` is called to initialize all Actors.
7. `Actor::PostBeginPlay()` is called to initialize all Actors.

Additionally, in a multiplayer game, when a player connects to a server, these functions will be called:

1. `GameInfo::PreLogin()` is called on the gametype to accept or reject the client.
2. `GameInfo::Login()` is called on the gametype to handle spawning the player.
3. `GameInfo::PostLogin()` is called on the gametype to handle the new player.

These `Login()` and `PostLogin()` functions are also called on the server if it is not a dedicated server. Figure 3.2 shows the gameflow graph of a typical UDK game.

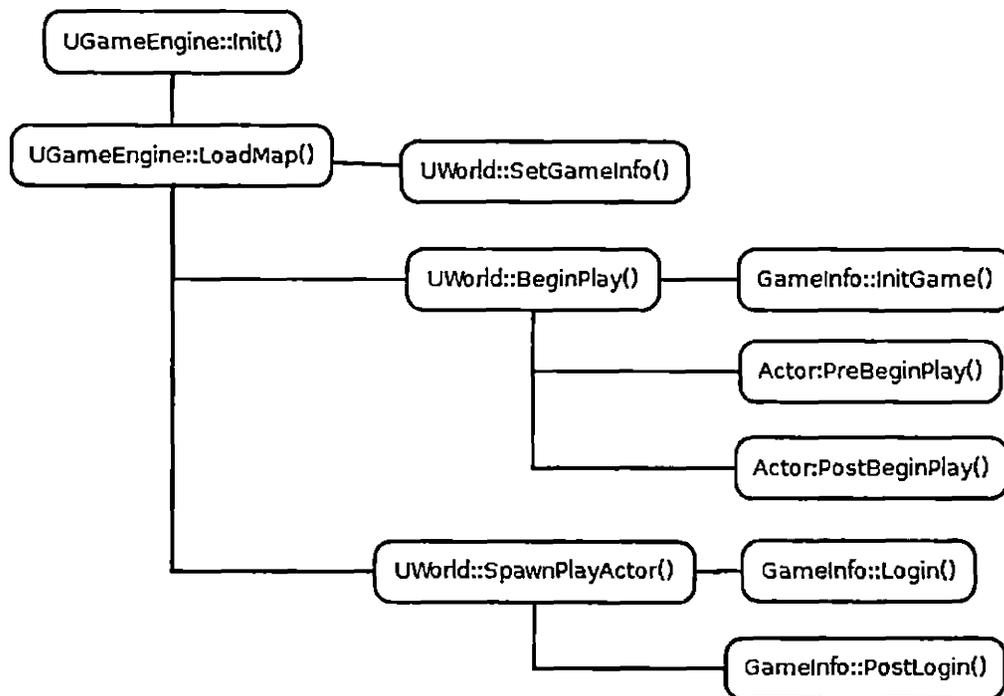


Fig. 3.2: UnrealScript Game Flow

After spawning all the player controllers, the GameInfo object calls the function StartMatch() to spawn the player's pawn for player to play. In UnrealScript, we can not alter the order of functions that the engine calls to start the game. However, we can override these function to initialize the game data at the start of the game. We can also change the way the pawn is spawned to create a unique game.

Controller and Pawn

In order to represent a character in the game, UDK uses a controller and a pawn object. The pawn object is responsible for displaying the model and playing the animations. The controller object is responsible for controlling the pawn's behavior. The creation and initialization processes of the controller and pawn objects for player characters and AI characters are different.

- Player character creation: the `PlayerController` is created by the `Login()` function on the `GameInfo` object, as described above. After that, the `PlayerPawn` is created using `SpawnDefaultPawnFor()` in `GameInfo`. The code below is called in the `GameInfo` class to spawn the pawn object for the player controller.

```
function RestartPlayer(Controller NewPlayer) {
...
    if (NewPlayer.Pawn == None) {
        NewPlayer.Pawn =
            SpawnDefaultPawnFor(NewPlayer, StartSpot);
    }
...
}
```

- AI character creation: AI characters are created during gameplay. For an AI character, the pawn is created first using `Spawn()` function, then the controller is created using `SpawnDefaultController()`. The code below is called in the `Pawn` class to spawn the default controller for the pawns placed in the level after the level is loaded.

```
event PostBeginPlay() {
...
    if ( WorldInfo.bStartup && (Health > 0) &&
        !bDontPossess ) {
        SpawnDefaultController();
    }
...
}
```

After being created, the controller calls `Possess()` function to attach itself to the pawn, then the pawn calls its `PossessedBy()` function to attach itself to the controller. We can override these functions to perform initialization on pawns and controllers.

3.4 Class Design

3.4.1 Overview

This section describes the purpose, important functions and important variables of classes in the Maerken package. For every class, there is a short description of the native parent class if it inherits directly from the native class to help understanding the class better. The important functions and variables of the parent class also have a short description for each. For more information on the native classes, refer to the comments in the code.

3.4.2 MyGame

MyGame Overview

This is the start point of the game execution. It inherits the `GameInfo` class, which is an native engine class. The class defines how the game is played: game rules, what actors to be used in the game, who may enter the game. The `GameInfo` object is instantiated when the level is initialized for gameplay. The class of this `GameInfo` object is determined by the game shortcut (`URL?game=<GameInfoClass>`) or the `DefaultGame` entry in the game's configuration file (`Engine.Engine` section). In the case of a network game, the `DefaultServerGame` is used. The `MyGame` class is defined to specify the `PlayerController` class and the `PlayerPawn` class to be used. It also modifies some of the parent's functions to perform initialization when creating `PlayerController` and `PlayerPawn`.

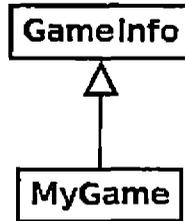


Fig. 3.3: MyGame Class

MyGame Functions

The following are the functions in the GameInfo class that are overridden in MyGame:

- `GenericPlayerInitialization()`: performs initialization when spawning `PlayerController`. The function is called in the `PostLogin()` function, after the player request to join is accepted and in the `HandleSeamlessTravelPlayer()` function, which happens after doing seamless travel. This function is chosen to be overridden instead of `SpawnPlayerController()` because it is recommended to perform initialization in `PostLogin()` rather than in the `Login()` function.
- `SpawnDefaultPawnFor()`: spawns the `PlayerPawn` class based on the character class defined in `PlayerController`. This function is called from the `RestartPlayer()` function.

The following are functions in the GameInfo class that can be overridden to implement more complex game logic:

- `InitGame()`: starting point of the script code, can be overridden to implement general game information initialization. This function also performs the creation of the helper classes: `AccessControl` (to help with log in logic) and `BroadcastHandler` (to help with game communication).
- `PreLogin()`: called when a client attempts to connect. It accepts or rejects the player.

- `Login()`: called when a client is accepted to spawn the default `PlayerController`.
- `PostLogin()`: finishes initialization for `PlayerController`, call `RestartPlayer()`.
- `RestartPlayer()`: this function is called from `PostLogin()` and `StartHumans()` to spawn the default Pawn for the corresponding `PlayerController`. It is also called in `StartBots()` function to restart the bots at the start of the game, but in our case, we use a different method of spawning NPC so we do not need to look into that.
- `StartHumans()`: attempts to do a `RestartPlayer()` on all `PlayerController` class.
- `HandleSeamlessTravelPlayer()`: reinitializes `PlayerController` through a seamless travel.
- `PostSeamlessTravel()`: called after a seamless level transition has been completed on the new `GameInfo` to reinitialize players already in the game as they won't have `Login()` called on them.
- `SpawnPlayerController()`: called in `HandleSeamlessTravelPlayer()` and `Login()` to spawn the new `PlayerController` for the player. It can be overridden to perform custom initialization. Maerken use `GenericPlayerInitialization()` to initialize `PlayerController`.

MyGame Initialization

The `MyGame` class is initialized as follows:

- `bDelayedStart=false`: the game starts spawning `PlayerController` right after running instead of waiting for other player to log in.
- `DefaultPawnClass=class'MyPlayerPawn'`: use `MyPlayerPawn` class when spawning the new pawn in `SpawnDefaultPawnFor()`

- `PlayerControllerClass=class'MyPlayerController'`: use `MyPlayerController` class when spawning the `PlayerController` in `SpawnPlayerController()`
- `bUseSeamlessTravel=true`: use seamless travel type when transitioning to new map.

3.4.3 *MyPlayerController*

Overview

`MyPlayerController` inherits the `PlayerController` class, which is a subclass of the `Controller` class. The `Controller` and `Pawn` classes are tightly paired together. Each pawn has a controller and each controller has a pawn. The controller takes control of the pawn through the `Possess()` function and the pawn then links with the controller by the `PossessedBy()` function. The `PlayerController` class handles player inputs from keystrokes, mouse input and console controller buttons through the `PlayerInput` helper class. The `PlayerController` class also uses the `Camera` class as a helper class to define the player view point.

In `MyPlayerController`, we override functions from `PlayerController` and create new functions to achieve the following goals:

- Process input directly from keyboard instead of using inputs defined in the configuration file. This helps organizing the code better for small projects. It also helps avoiding to implement a subclass of `PlayerInput`.
- Initiate actions based on input from keyboard, mouse and user interface system.
- Send instructions to the pawn so that the pawn can perform actions that the player wants.
- Manage all Scaleform UI screens, including the Scaleform HUD.

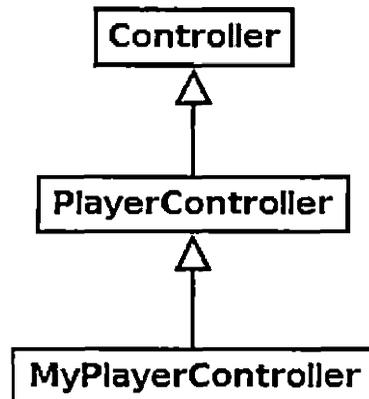


Fig. 3.4: MyPlayerController Class

MyPlayerController Variables

The following are the variables used in the MyPlayerController class:

- MainMenu, InventoryMenu, MidGameMenu, HUD: hold the reference to the Scaleform movies.
- PlayerPawn: holds reference to the Pawn, this variable helps reduce the type casting in the code when calling new functions in MyPlayerPawn class.
- CharacterName, CharacterClass, Experience: hold the character information.
- Quests: holds the quest array.
- EquipmentSet: manages the equipment.
- InventoryBag: manages the inventory.
- SaveGame: manages game serialization.

MyPlayerController Functions

The following are the functions in the PlayerController class that are overridden in MyPlayerController:

- `Possess()`: initializes the controller when it takes control over the pawn. The overridden version of this function assigns the `PlayerPawn` variable with the possessed pawn. This helps reduce the number of conversions from `Pawn` to `MyPlayerPawn` in the whole class.
- `Unpossess()`: clears the pawn-related variables when the pawn is detached from the controller. The overridden version of this function removes the reference from the `PlayerPawn` variable.
- `InitInputSystem()`: initializes the input system. The overridden version of this function adds a delegate to intercept the native inputs.
- `PlayerTick()`: updates the status of `PlayerController` every tick. This function is used in the `PlayerController` class to process the movement. It is only called on `PlayerController` objects that has a `PlayerInput` object. Therefore, it is not called on server for non-local `PlayerController`. The overridden version of this function also updates the information display on the HUD and other UI screens.

The following are the new functions implemented in `MyPlayerController`:

- `GenericPlayerInitialization()`: called from `MyGame` class. This function is created to let the `PlayerController` do the self initialization. It also shows the main menu if the player is in the menu map or shows the HUD if player is in the gameplay map.
- `ProcessNativeInputKey()`: a delegate to process the input directly from the source.
- `Open<MenuName>()`: functions to open menu screens. They are called in `ProcessNativeInputKey()` functions based on the input keys.
- `Attack()`, `PerformClassSkill()`: action functions to perform game logic. We precede these functions with 'reliable server' so that they are only called on the

server.

- `ReceiveQuest()`: adds a quest to the quest array.
- `UpdateQuest()`: updates a quest based on the input game event.

MyPlayerController States

The `PlayerController` class has many states to support different phases of the game. In `MyPlayerController`, we only use the `PlayerWalking` state, which handle the pawn's movement on land. Movement logic is implemented here by overriding the `PlayerMove()` and `ProcessMove()` functions.

3.4.4 MyPawn

Overview

`MyPawn` inherits the `Pawn` class. A pawn is the physical representation of a player or an NPC in the level. The `Pawn` class is responsible for physical interactions with the world such as animations, collisions and physics. The `MyPawn` class also implements combat system, allowing the pawn to interact with other pawns base on combat stats and the relationship between the pawns in the level.

MyPawn Variables

The following are the variables used in `MyPawn`:

- `AttackingAnimationName`: determines the animation sequence to be played in the `Attacking` state.
- `DeadAnimationName`: determines the animation sequence to be played in the `Dead` state.
- `FullBodySlot`: an `AnimSlot` variable used to play the custom animations.

- PawnState: holds the state of the pawn, helps with animation replication.
- HitPoint, MaxHitPoint, Power, Speed, MoveSpeed: hold the character stats.
- Faction: an enumeration variable, which determines the relationship between two pawns.
- ActiveSkill: holds the reference to the skill being played in the PerformSkill state.

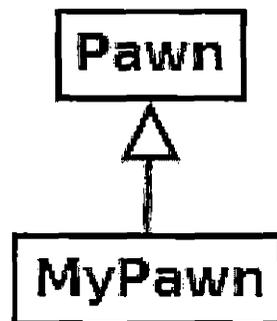


Fig. 3.5: MyPawn Class

MyPawn Functions

The following are the functions in the Pawn class that are overridden in MyPawn:

- ReplicatedEvent(): sets the new state for the client pawn to update the animation.
- PostInitAnimTree(): initializes the FullBodySlot variable to play custom animations.
- Died(): overridden to remove the default death conditions such as getting crushed by other pawns.
- BeginState(): changes the PawnState variable and replicate it immediately so the clients can play the correct animation.

The following are the new function implemented in MyPawn:

- `IsEnemy()`: checks if the target is attackable. In this iteration, the return value of this function only depends on the factions of the pawn and the target pawn. This function can be implemented with more complex logic for faction interaction.
- `SelectTarget()`: returns a target pawn based on the range, the width of selection cone and whether or not the returned target should be an enemy.
- `GetHit()` and `ApplyDamage()`: paired together, `ApplyDamage()` calls `GetHit()` on the target pawn to reduce target's hitpoint based on the pawn's power. `GetHit()` recalculates the hitpoint reduction based on armor. These functions can be implemented with more complex mechanics such as applying debuffs and reflecting attacks.
- `Attack()`: can only be called if the pawn is in Neutral state. This function transitions the pawn to attacking state and lets the pawn perform attacking logic in the middle of the attacking animation.
- `ApplyAttack()`: called in the middle of attacking state. This function is called by the timer set by `Attack()` function. It attempts to get a target based on the pawn's location and rotation then apply damage to it.

MyPawn States

A state of MyPawn class is responsible for playing the animations of that state. The following are the states implemented in MyPawn:

- Neutral state is responsible for playing the animations define in the pawn's AnimTree template. Neutral state also has the functions allowing the pawn to perform physical actions such as attack and perform skill.

- Dead state plays the dead animation, it also contains codes to make the pawn disappear after a few seconds.
- Attacking state plays the attacking animation.
- PerformingSkill state plays the skill animation.

In states other than Neutral, the state code calls the Sleep() function to let the pawn stay in the state while the animation is being played, then returns to Neutral state.

MyPawn Initialization

We initialize the MyPawn class as follows:

- Add a LightEnvironmentComponent to the component array in order to see the light and texture on the pawn's skeletal mesh.
- Add a SkeletalMesh to the component array to have a physical representation of the pawn.
- Add a CollisionCylinder to the component array to allow the pawn to walk on rigid surfaces.
- Set the Physics property to PHYS_Walking to allow movement on land.
- Set the controller class to be used by the AI system.
- Set default combat and animation properties.

3.4.5 MyPlayerPawn

Overview

MyPlayerPawn inherits MyPawn class. It represents a player character in the game. We implement this class as a template class and have its children implement the actual logic for attacking and playing the class skill. However, we implement camera

function and HUD updating function because they do the same work for every player.

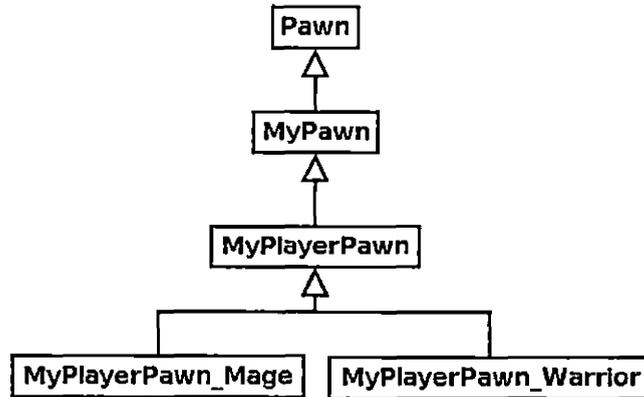


Fig. 3.6: MyPlayerPawn Class

MyPlayerPawn Variables

The following are the variables use in MyPlayerPawn:

- **PlayerController**: holds the reference to the player controller. We create this variable reduce the number of type casting from PlayerController to MyPlayerController.
- **ClassSkill**: holds the reference to the class skill. This variable is initiated in the child classes to determine which skill the pawn has.

MyPlayerPawn Functions

The following are the functions in the Pawn class that are overridden in MyPlayerPawn:

- **PossessedBy()**: overridden to initiate the PlayerContorller variable.
- **UnPossessed()**: overridden to remove the reference to the PlayerController variable.

- CalcCamera(): calculates the camera position and rotation for the player character. In Maerken, we implement third person camera style.
- Tick(): updates the cooldown of the class skill to assist the HUD in displaying cooldown information.

The following are the new function implemented in MyPlayerPawn:

- UpdateHUD(): updates the class skill cooldown and the cast bar in the HUD.
- PerformClassSkill(): implemented in child classes to perform the class skill.

MyPlayerPawn States

The player pawn's dead state plays the dead animation and respawns the player when the player press the left mouse button.

3.4.6 *MyNPCPawn*

Overview

MyNPCPawn inherits MyPawn class. This is the template class for MyNPCPawn and its child classes. We implement attacking and performing skill logic in the child classes.

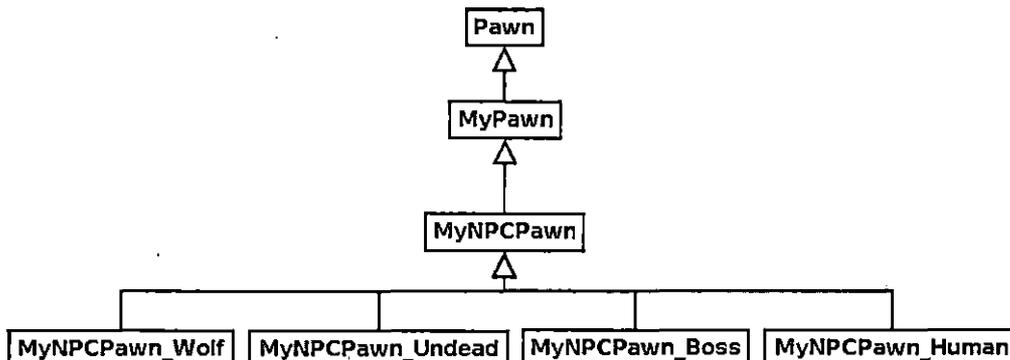


Fig. 3.7: MyNPCPawn Class

MyNPCPawn Variables

The following are the variables used in MyNPCPawn:

- **AIController:** holds the reference to the AI controller. We create this variable to reduce the number of type casting from Controller to MyAI.
- **UniqueSkill:** holds the reference to the skill of an NPC.

MyNPCPawn Functions

The following are the functions in the Pawn class that are overridden in MyPlayerPawn:

- **PossessedBy():** overridden to initiate the AIController variable.
- **UnPossessed():** overridden to remove the reference to AIController.

3.4.7 MyAI

Overview

MyAI inherits the AIController class. This is the template class for MyAI and its child classes. These classes control the NPC behaviors based on the state of the game. The logic for the NPC behaviors is implemented in the child classes.

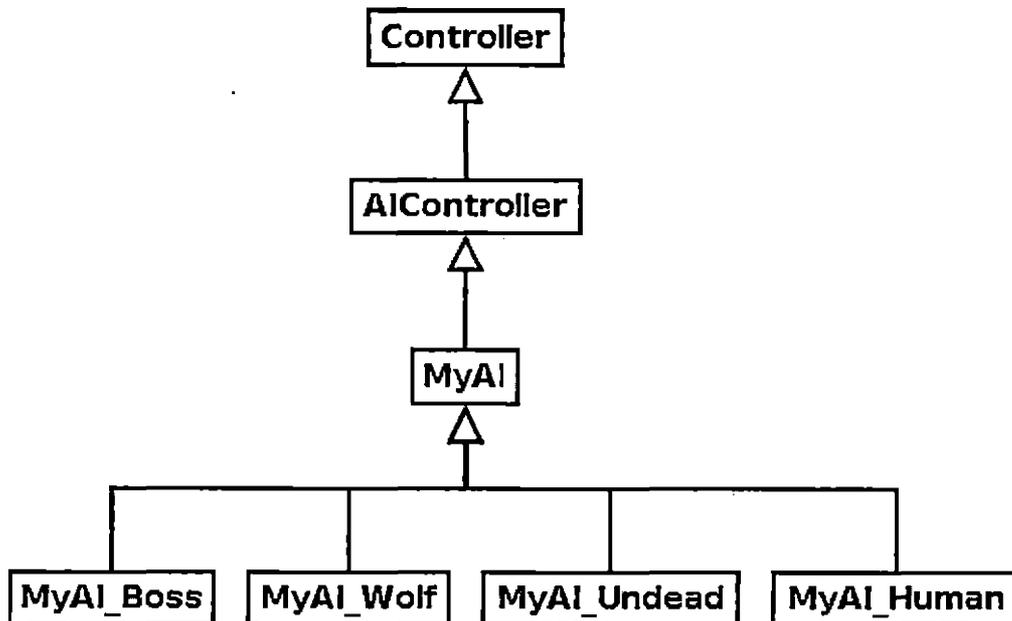


Fig. 3.8: MyAI Class

MyAI Variables

In **MyAI**, we only create one variable called **NPCPawn** to hold the reference to the **NPCPawn**. We create this variable reduce the number of type casting from **Pawn** to **MyNPCPawn**.

MyAI Functions

The following are the functions in the **Controller** class that are overridden in **MyAI**:

- **Possess()**: this function is overridden to initiate the **NPCPawn** variable.
- **UnPossess()**: this function is overridden to remove the reference to **NPCPawn**.

The following are the functions in the **Controller** class that can be overridden in the child classes of **MyAI**:

- **CanSee()**: returns if a pawn is in sight.

- SeePlayer(): activated when the pawn sees a player pawn.
- SeeMonster(): activated when the pawn sees an NPC pawn.
- MoveTo(): directs the pawn to move to a location.
- MoveToward(): directs the pawn to move to an actor.

MyAI States

The following are some possible states of an NPC that can be implemented in the child classes:

- Wandering: initial state. It lets the NPC wander in a set radius from its spawn point. This state has events that transition the NPC to other states.
- Attacking: when the pawn see an enemy, it changes to this state. This state allows the NPC to attack the enemy and move to attacking position if the attack is not possible.
- Running: when the pawn's hitpoint gets low, some NPC will run away.

3.4.8 MySkill

Overview

This class is the template class for MySkill and its child classes. A skill class is responsible for playing a skill on the character including the visual effect for the skill. Although the animation is played by the pawn, the skill class determines which animation sequence to be played. In order to make a pawn perform a skill, we need to follow the following steps:

- Create a new skill object in the pawn.
- In the pawn, set the active skill to the skill object.

- Set the timer to activate Perform() function on the skill object.
- Set the cooldown timer for the skill.
- Go to the pawn's 'PerformingSkill' state.

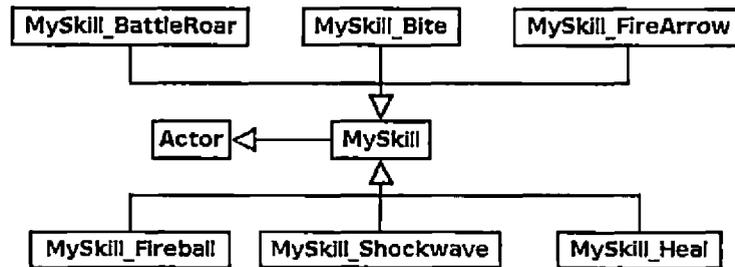


Fig. 3.9: MySkill Class

MySkill Variables

The following are the variables used in MySkill:

- **Pawn**: holds the reference to the owning pawn. This variable allows the skill to get the combat stats from the pawn.
- **AnimationName**: determines which animation sequence the pawn should play while performing this skill.
- **CastTime**: determines the cast time of the skill.
- **Cooldown**: holds the cooldown for the skill, which determines how long the pawn should hold in order to activate the skill again.
- **blsInstant**: determines if the HUD should display the cast bar while playing this skill. Although every skill has a cast time, some skills has a longer cast time and should be tracked on the HUD. Usually, a skill with 1 second cast time is set to be instant cast.

- `CurrentCooldown`: holds the current cooldown value. This variable is used to update the HUD.

MySkill Functions

Every skill class has a function called `Perform()`. This function perform the logic for performing the skill. For example, in `MySkill_Fireball`, it spawns a fireball projectile at the selected enemy.

```
function Perform()
{
    local MyPawn Target;
    local MyProjectile_Fireball Fireball;

    Target = Pawn.SelectTarget(1000, 0.7);

    if (Target != none) {
        Fireball = Spawn(class'MyProjectile_Fireball',,,
Pawn.Location + Vector(Pawn.Rotation) * 50, Pawn.Rotation);
        Fireball.Pawn = Pawn;
        Fireball.Target = Target;
        Fireball.Power = FireballDamage;
    }
}
```

MySkill Initialization

In the default properties block, we set `RemoteRole = ROLE_SimulatedProxy` to replicate the skill to the clients. For a specific skill, we initialize the skill by setting the

values for AnimationName, CastTime, Cooldown and bIsInstant variables. The Pawn variable is initialize in the pawn class when we instance the skill object.

3.4.9 MyProjectile

This is the template class for MyProjectile and its child classes. A projectile is spawn whenever a range attack or skill is initiated. The projectile then moves to the target location and explode to deal damage to the target and destroy itself.

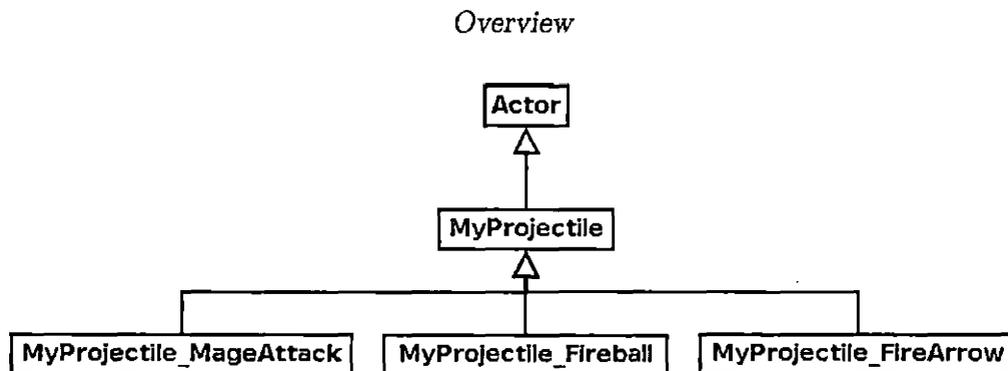


Fig. 3.10: MyProjectile Class

MyProjectile Variables

The following are the variables used in MyProjectile:

- Pawn: holds the reference to the owning pawn.
- Target: holds the reference to the target that it has to move to.
- Speed: determines how fast the projectile should move.
- Power: determines how much damage the projectile should do to the target.

MyProjectile Functions

We override the Tick() function from the Actor class to update the location, rotation and speed based on the Pawn and Target variables. For example, the Tick() function of the attack projectile is implemented as follows:

```
event Tick(float DeltaTime)
{
    super.Tick(DeltaTime);

    SetRotation(Rotator(Location - Target.Location));
    Velocity = Speed * Normal(Target.Location - Location);
    if (VSize(Location - Target.Location) < 20)
    {
        Pawn.ApplyDamage(Target, Power);
        Destroy();
    }
}
```

MyProjectile Initialization

The MyProjectile class is initialized as follows:

- Initialize a particle system component for its visual effect.
- Set Physics = PHYS_Projectile to allow the projectile to move.
- Set RemoteRole = ROLE_SimulatedProxy to replicate the projectile to the clients.
- Set bUpdateSimulatedPosition = true to let the clients see its movement.

3.4.10 MyUI

MyUI inherits the GfxMoviePlayer class. This is the template class for MyUI and its child classes, including HUD and menus. These classes interact with the ActionScript code through the wrapper functions and receive calls from the ActionScript code to perform game logic. The implementation of the child classes are explained in user interface implementation section.

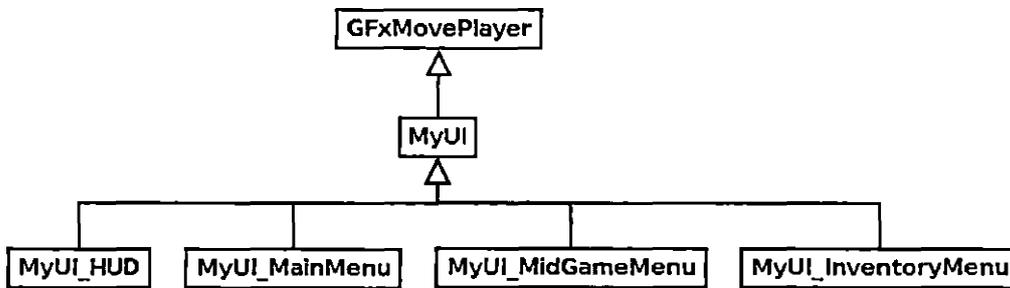


Fig. 3.11: MyUI Class

In MyUI class, we create a variable call MyActivationKey to determine which key activates and closes this movie. We override the FilterButtonInput() function to intercept the escape button and activation keypress and and close the menu. In the default properties block of the MyUI class, we set the MovieInfo to a flash movie imported to an Unreal content package.

4. IMPLEMENTATION

4.1 *Multiplayer Implementation*

4.1.1 *Overview*

This chapter explains how the multiplayer aspect of Maerken is implemented. In Maerken, we use a replication mechanism to handle data communication. In Unreal engine, we have one or more clients connect to a server. Clients only send requests to the server through function calls. All the logic calculations are done by the server. A portion of the game information will be sent to clients by using variable replication. The information sent to clients will be used for displaying animation and effect on client machines. The following sections explain how we replicate different information.

4.1.2 *Function Replication*

In Maerken, we only replicate functions from client to server. To do that, we implement action functions in `MyPlayerController` class with the “reliable server” modifier. When a server function is called on the server, it executes immediately. When a server function is called on a client, it will send the request to the server to execute that version of the function on the server. For example, the attack request made by a client has the function call implemented as follows:

```
reliable server function Attack() {  
    PlayerPawn.Attack();  
}
```

This invokes the `Attack()` function on the `PlayerPawn` object on the server. The `PlayerPawn` object on the client does not execute its `Attack()` function.

4.1.3 Animation Replication

Since we only allow clients to send action requests to the server, animations started by these actions are only played on the server. In order to let these animations be played on clients, we implement an animation replication system as follows.

Movement animations are done through `AnimTree`. They are replicated by the engine code. We only have to replicate the custom animation in script code. All the custom animations are done in the state code, each state performs one sequence of animation. The pawn's `Neutral` state performs animations from the `AnimTree`. All states are declared "simulated" to allow the state code to be run on the client. We create a variable called `PawnState` to store the state of `MyPawn` object and notify the client `MyPawn` object whenever the server `MyPawn` object change its state.

We implement `BeginState()` event to change the `PawnState` variable base on the state's name.

```
event BeginState(Name PreviousStateName) {
    PawnState = GetStateName();
}
```

`PawnState` variable is replicated whenever it changes.

```
replication {
    if (bNetDirty && Role == ROLE_Authority)
        PawnState;
}
```

When the client receives the change, it changes its state to perform the animation.

```
simulated event ReplicatedEvent(name VarName) {
```

```

if (VarName == 'PawnState') {
    if (!IsInState(PawnState)) {
        GotoState(PawnState, , true);
    }
}
super.ReplicatedEvent (VarName);
}

```

4.1.4 Variable Replication

Variables that need to be displayed on the client (player information in HUD and menus) are calculated on the server, then replicated to the clients. The following is a list of classes and variables that are replicated:

- Pawn class: HitPoint, MaxHitPoint, AttackValue, TargetPawn.
- Skill class: Cooldown, CurrentCooldown, Pawn, Target.
- Projectile class: Power, Speed, Pawn, Target.

4.1.5 Network Game Creation Process

In order to host a multiplayer game, we need to follow the following procedure:

1. Get OnlineSubSystem: class'GameEngine'.static.GetOnlineSubsystem().
2. Get OnlineGameInterface: OnlineSub.GameInterface.
3. Add the delegate to be activated when the online game creation completes:
GameInterface.AddCreateOnlineGameCompleteDelegate(OnGameCreated).
4. Call CreateOnlineGame() function from GameInterface:
GameInterface.CreateOnlineGame().
5. Implement the delegate: get the game options and do a travel to play map.

4.1.6 Network Game Search and Join Process

In order to search for multiplayer games, we need to follow the following procedure:

1. Get OnlineSubSystem: `class'GameEngine'.static.GetOnlineSubsystem()`.
2. Get OnlineGameInterface: `OnlineSub.GameInterface`.
3. Add the delegate to be activated when the search completes:
`GameInterface.AddFindOnlineGameCompleteDelegate(OnFindOnlineGameCompleted)`.
4. Call `FindOnlineGame()` function from `GameInterface`:
`GameInterface.FindOnlineGame()`.
5. Implement the delegate: populate the results.

After we have the results populated, we can join the game we want by calling `JoinOnlineGame()` on the `OnlineGameInterface` object.

4.2 AI Implementation

4.2.1 Overview

This section explains how Unreal engine supports AI and how we use Unreal engine to implement our AI system.

4.2.2 Unreal Engine AI Support

Unreal engine supports AI implementation by providing us with a navigation system and a controller class to take control of the AI pawns. In Maerken, we use the waypoints to implement the navigation system. This requires the level designers to put various waypoints and pathnodes around the map to help the AI with navigation. The waypoints are then built into the map. Everytime a controller use `MoveTo()` or

MoveToward() function to instruct a pawn to a specific location, the engine use the waypoints to find the closest way to the destination.

4.2.3 MyAI Class

In Maerken, the AI is not simply just navigation. In MyAI class, we take advantage of the game events generated by Unreal engine such as SeePlayer() and SeeMonster() to make more complex AI. We also create our own game events such as GetHit() so the NPC can react to the environment more intelligently.

4.2.4 AI Variables

The following are the variables that the AI can use to determine what to do next:

- TargetPawn: used to get combat information on the target.
- NPCPawn: used to get combat information on the AI pawn.
- AnchorPoint: determines the point an NPC should wander about, and the point to return after an unsuccessful chase.
- MaxWanderingDistance: determines the distance from anchor point to patrol.
- SightRadius: determines the distance to notice another pawn.

4.2.5 AI States

The following are states that an AI can be in:

- Wandering: the pawn wanders about the anchor point.
- Attacking: the pawn sees a target, tries to get in attacking range and attack the target.
- Running: some NPCs will run when the pawn hitpoint gets low.

4.3 User Interface Implementation

4.3.1 Overview

This section explains how we use the Scaleform technology to implement our user interface system. In order to have a menu running in the game, we follow these steps:

1. Create a flash menu with external calls to UnrealScript for engine operations. For example, when we click on the exit button, we call `ExternalInterface.call("ConsoleCommand", "Exit")`.
2. Put the flash swf file in `UDKGame\Flash\MythicUI\<MenuName>`.
3. Import the swf file using Unreal Editor.
4. Create a new class in UnrealScript that extends `MyUI` and set `MovieInfo` in the default properties block to be that imported movie.
5. Create a function to open the movie. This function creates a new object of the menu class, then calls `Start()` on the new object.

4.3.2 ScaleformGFX Components

UDK provides us with a list of flash CLIK components that we can use to develop our flash menu. These components have their flash version stored in the `Development\Flash\CLIK\components` folder. The following list will explain briefly what they are.

- **Button:** implements a button that can be clicked. This is also used to implement other components.
- **Button Bar:** creates a group of buttons. This can be used to create button instances on the fly.
- **Check Box:** this is a button that is set to toggle the selected states when clicked.

- Dialog: displays a dialog view.
- Dropdown Menu: this component contains a list of elements. It displays its selected component in idle state and displays the whole list when clicked.
- List Item Renderer: this component is used to display the elements in dropdown menus, scrolling lists, and tile lists.
- Label: displays a label text. This is a very common used component.
- Numeric Stepper: displays a number in its assigned range. The number can be increased or decreased using the forward and backward buttons.
- Option Stepper: display an element in its element list. The active element can be changed by the forward and backward buttons.
- Progress Bar: displays a percentage of a bar. We use this component to display the health bar and cast bar in Maerken HUD.
- Radio Button: used in a set of buttons to display and change a single value.
- Scroll Bar: displays and controls the scroll position of another component, such as scrolling list and text area. It has a draggable thumb button and two stepper buttons.
- Scrolling List: display a list of components that can be scrolled.
- Slider: displays a numerical value in a range. It has a thumb button to display and change the value.
- Status Indicator: displays a percentage of a bar using small bars.
- Text Area: displays an editable text. This component supports multiple line text and scroll bar.
- Text Input: display an editable text.

- **Tile List:** displays and tiles a list of element. This component also supports scrolling using a scroll bar.
- **UI Loader:** displays an external image from Unreal package.

These components are ready to be used. However, we can also use them to create our unique components that support the game better. The following sections explain how we implement the menus. For UnrealScript functions, we only explain their purpose.

4.3.3 Main Menu Implementation

To implement the main menu, choose character menu and create character menu, we create a flash movie called `MyULMainMenu` and an UnrealScript class called `MyULMainMenu` extending `MyUI`.

MyULMainMenu Flash Elements

In the flash movie, we use 3 keyframes representing the main menu, choose character menu and create character menu. Because these menus are only displayed in the menu map and are not displayed in the gameplay maps, this implementation reduces the number of UnrealScript calls when we change the menus. The `MainMenu` keyframe includes the following:

- `CharacterName` label: displays the selected character name.
- `ChooseCharacter` button: transitions to choose character menu.
- `HostGame` button: calls the `hostGame()` function.
- `GameName` text input: allows player to enter the game name.
- `RefreshHosts` button: calls the `refreshHosts()` function.
- `Quit` button: calls the `quitGame()` function.

- ServerList scrolling list: displays the list of server, when an item is clicked, the `joinGame()` function is called.
- A scrolling bar associated with the scrolling list.

Figure 4.1 shows the main menu implemented in Maerken.

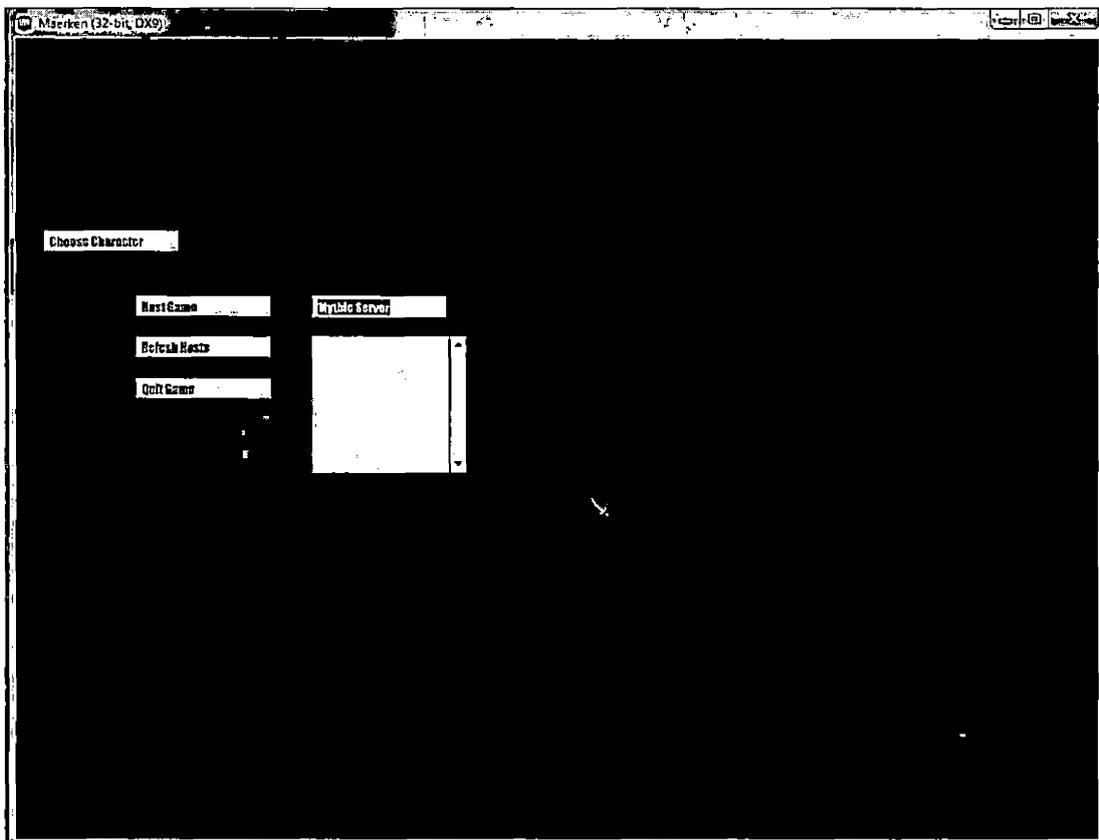


Fig. 4.1: Main Menu

The ChooseCharacter keyframe includes the following:

- CreateCharacter button: transitions to create character menu.
- SelectCharacter button: transitions back to main menu with the selected character in the list.

- DeleteCharacter button: delete the selected character in the list.
- CharacterList scrolling list: display the list of character, when an item is clicked, set the selected character.
- Back button: when it is clicked, the main menu is displayed.

Figure 4.2 shows the choose character menu implemented in Maerken.

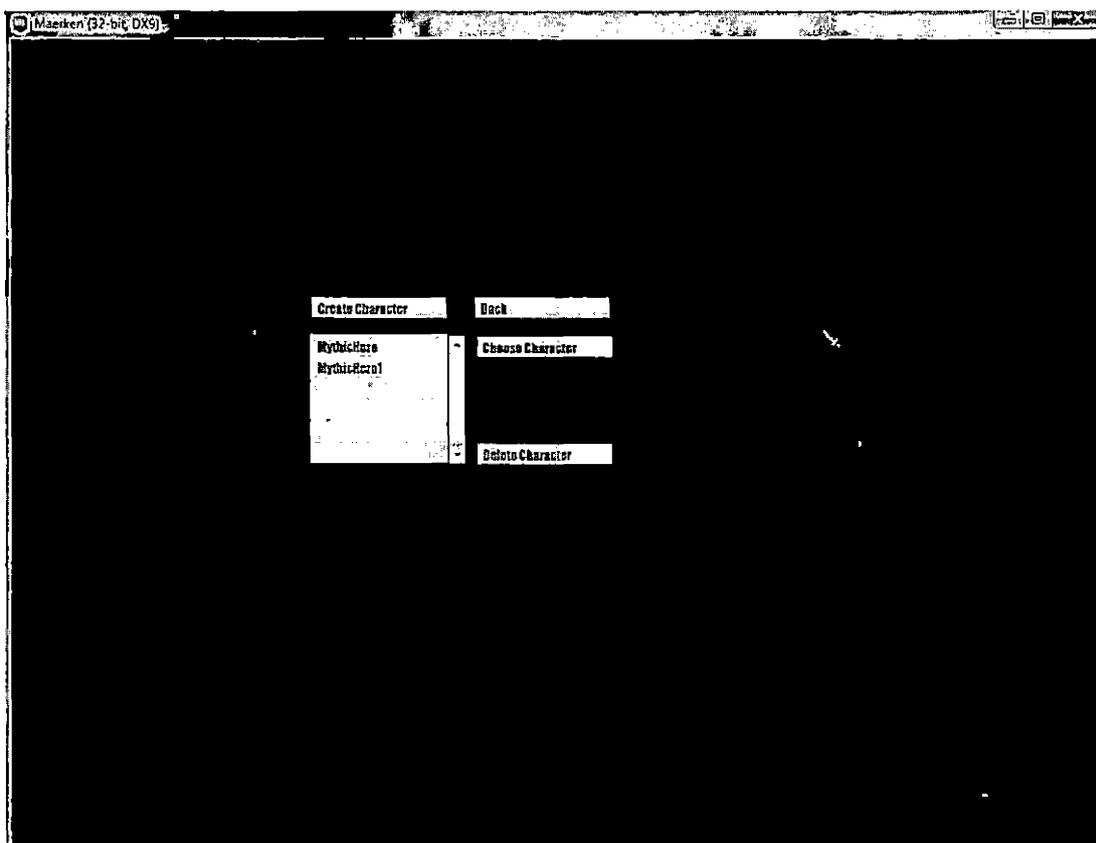


Fig. 4.2: Choose Character Menu

The CreateCharacter keyframe includes the following:

- A label displays "Character Name".

- A label displays "Character Class".
- CharacterName text input: allows player to enter the character name.
- Warrior and Mage radio buttons: allow player to choose the class.
- Create button: calls the createCharacter() function.
- Back button: transitions back to choose character menu.

Figure 4.3 shows the create character menu implemented in Maerken.

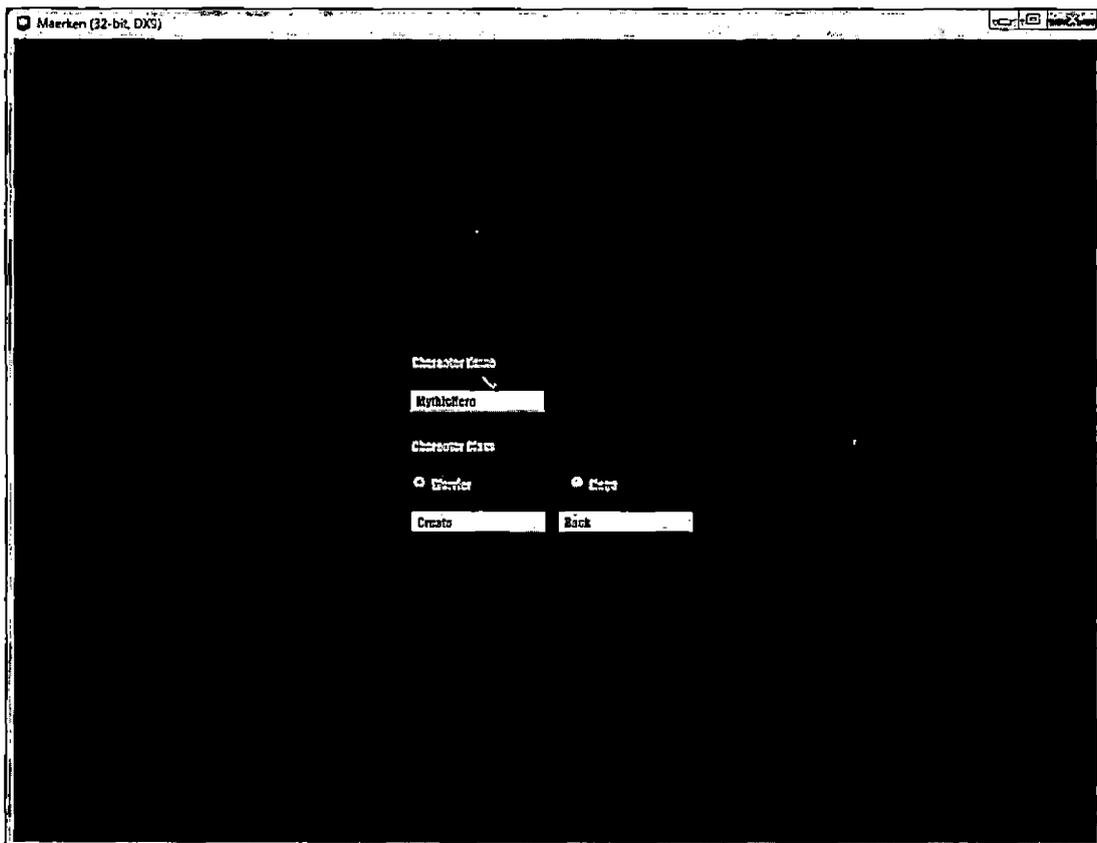


Fig. 4.3: Create Character Menu

4.3.4 *MyUIMainMenu ActionScript Functions*

We create the following functions in the flash movie:

- `quitGame()`: calls `ConsoleCommand("exit")` in `UnrealScript` to close the application.
- `hostGame()`: calls the `HostGame()` function in `UnrealScript`.
- `refreshHosts()`: calls the `RefreshHosts()` function in `UnrealScript`.
- `joinGame()`: calls the `JoinGame()` function in `UnrealScript`.
- `loadCharacterList()`: calls the `LoadCharacterList()` function in `UnrealScript` to initialize the character list. This is called every time the menu is opened.
- `deleteCharacter()`: calls the `DeleteCharacter()` function in `UnrealScript`.
- `createCharacter()`: calls the `CreateCharacter()` function in `UnrealScript`.

MyUIMainMenu UnrealScript Functions

We create the following functions in the `MyUIMainmenu` class:

- `HostGame()`: follows the create network game creation process in multiplayer implementation section to create a network game.
- `RefreshHosts()`: follows the search network game process in multiplayer implementation section to populate the game list.
- `JoinGame()`: follows the join network game process in multiplayer implementation section to join the selected game.
- `LoadCharacterList()`: gets the character list and populates it on the menu.
- `DeleteCharacter()`: deletes the selected character.

4.3.5 Mid Game Menu Implementation

To implement the mid game menu, we create a flash movie called `MyUIMidGameMenu` and an UnrealScript class called `MyUIMidGameMenu` extending `MyUI`.

MyUIMidGameMenu Flash Elements

We create the following elements in the flash movie:

- Save button: calls the `saveGame()` function.
- QuitToMenu button: calls the `quitToMenu()` function.
- Quit button: calls the `quitGame()` function.

Figure 4.4 shows the mid game menu implemented in Maerken.

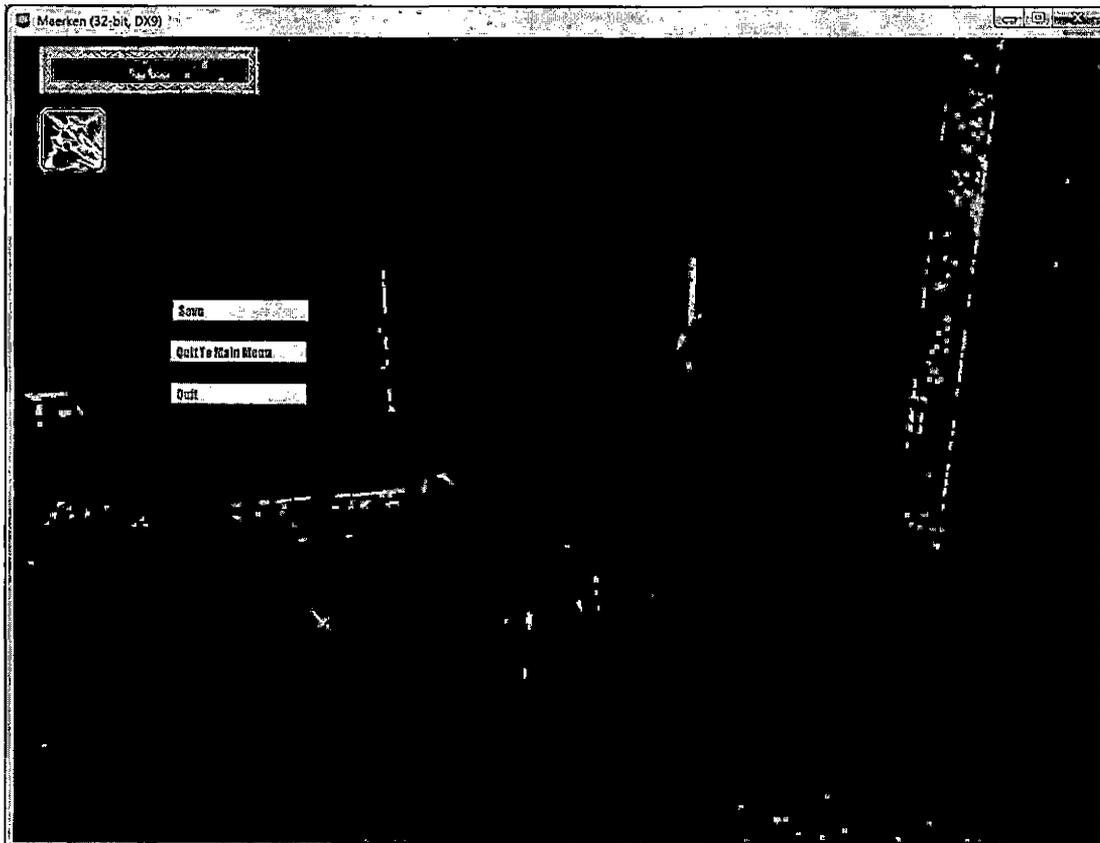


Fig. 4.4: Mid Game Menu

4.3.6 *MyUIMidGameMenu* ActionScript Functions

We create the following functions in the flash movie:

- `saveGame()`: calls the `SaveGame()` function in UnrealScript.
- `quitToMenu()`: calls `ConsoleCommand("open StartMap")` in UnrealScript to open the menu map and display the menu.
- `quitGame()`: calls `ConsoleCommand("exit")` in UnrealScript to close the application.

MyUI.MidGameMenu UnrealScript Functions

In the `MyUI.MidGameMenu` class, we create a function called `SaveGame()` to save the game information.

4.3.7 Head-Up Display Implementation

To implement the mid game menu, we create a flash movie called `MyULHUD` and an UnrealScript class called `MyULHUD` extending `MyUI`. The `MyULHUD` class has the `bCaptureInput` variable set to `false` to make it non-interactive.

MyULHUD Flash Elements

In the flash movie, we create the following elements:

- `HeathBar` progress bar: displays the health bar.
- `CastBar` progress bar: displays the cast bar if the player is casting a non-instant skill.
- `ClassSkillIcon` progress bar: displays the class skill icon and the cooldown of the class skill.

Figure 4.5 shows Maerken gameplay scene with the HUD implemented.

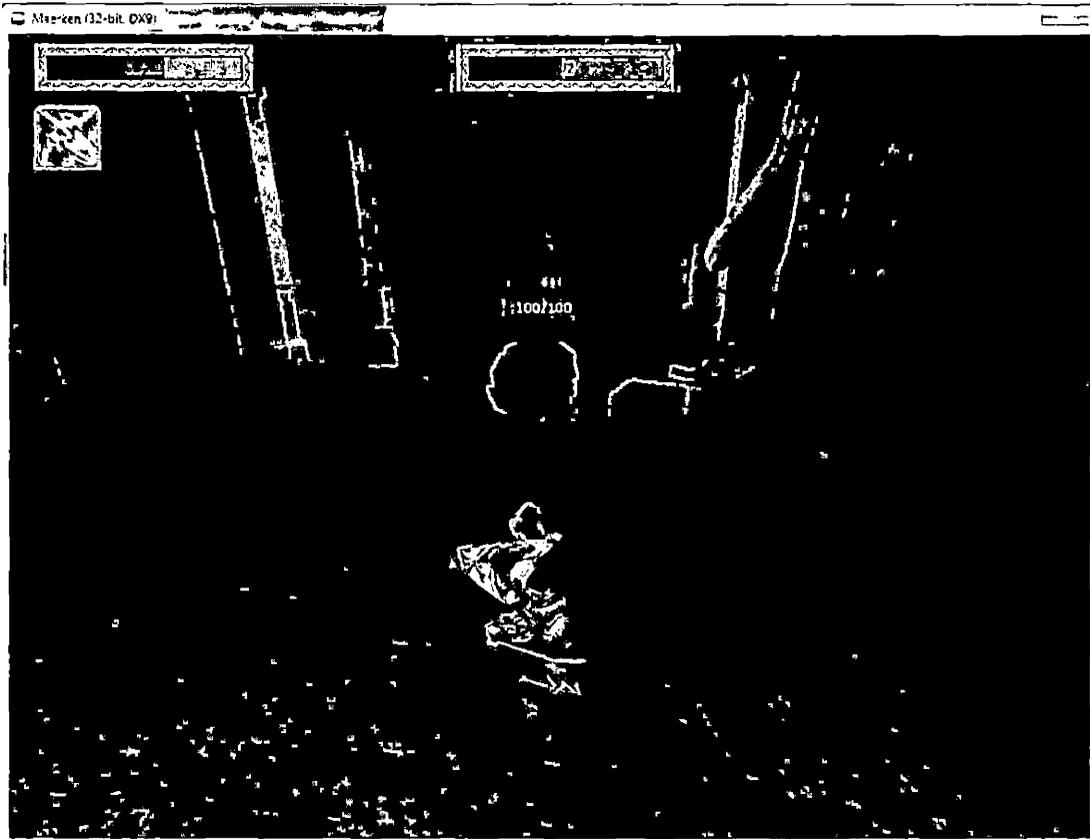


Fig. 4.5: Head-Up Display

4.3.8 MyUIHUD ActionScript Functions

In the flash movie, we create the following functions to let UnrealScript set the information displayed on the movie:

- `setHealthBar()`: sets the progress on the HealthBar progress bar.
- `setCastBar()`: sets the progress on the HealthBar progress bar.
- `setClassSkillIcon()`: sets the progress on the ClassSkillIcon progress bar.

MyULHUD UnrealScript Functions

These functions are called in the PlayerController's Tick() function to update the information on the HUD.

- SetHealthBar(): calls the setHealthBar() function in ActionScript.
- SetCastBar(): calls the setCastBar() function in ActionScript.
- SetClassSkillIcon(): calls the setClassSkillIcon() function in ActionScript.

4.4 Game Serialization Implementation

4.4.1 Overview

This section explains how we use the DLL binding feature in UnrealScript to implement our game serialization. UnrealScript supports two methods of saving the game state. The first method is to use the configuration file system. The second method is to use DLL bind feature to interact directly with the file system. We choose the DLL bind method because it allows more access to the files.

4.4.2 SaveGameDLL Implementation

We create a DLL call SaveGameDLL to allow UnrealScript to read and write files. We only allow one file to be opened at a time. The operation of file reading and writing should be perform as follow: For writing to a file, the game script should follow this pattern:

```
MyOpenFileForWriting (Filename);  
MyWriteString (CharacterName);  
MyWriteFloat (HitPoint);  
MyCloseFile ();
```

For reading a file, the script should first check for the file existence and do the reading as follows:

```
if (!MyFileExists(Filename)) {  
    MyOpenFileForWriting(filename);  
    MyCloseFile();  
}  
MyOpenFileForReading(Filename)  
CharacterName = MyReadString();  
HitPoint = MyReadFloat(HitPoint);  
MyCloseFile();
```

The program will halt with an error message under the following conditions:

1. If you open a new file before closing an existing file.
2. If you read from a file opened for writing.
3. If you write to a file opened for reading.
4. If you read from a file that doesn't exist.
5. If you read a bad value.

4.4.3 *SaveGameDLL Usage*

We create a class called MySaveGame and add all exported functions from the DLL to the class. We create functions that perform game serialization in this class:

- GetCharacterList()
- CreateCharacter()
- DeleteCharacter()
- SaveCharacter()

An object of this class is created in `MyPlayerController` to allow interaction between player and the system.

5. GAME CONTENT CREATION, MANAGEMENT AND DISTRIBUTION

5.1 Art Asset Management

5.1.1 Content Importing Process

Obtaining External Content

UDK allows the developer to import and display various types of content, which are described as follows:

- **Static Mesh:** used to display a 3D static object in the game. Static meshes can be converted to dynamic objects to simulate physics in the game. They can be created using 3D modeling applications such as 3DS Max and Maya.
- **Skeletal Mesh:** used to display a 3D object with animations. Skeletal meshes are static meshes with a bone structure.
- **Texture:** used create material to decorate 3D objects or particle systems. The textures can be created using 2D graphics applications such as Gimp or Photoshop.
- **Animation:** used to play animations on skeletal meshes. Animations can be created by the 3D modeling applications mentioned above or applications designed specifically for animation such as Motion Builder.
- **Sound:** used to create sounds in the game.

- SpeedTree: used to create procedural trees in the game. The SpeedTree templates can be created using the SpeedTree Modeler application included in the UDK.
- Bink Movie: used to display movie in the game.
- Scaleform Gfx Movie: used to display menus and HUD in the game. The Scaleform Gfx movies can be created using Flash Professional.

In this iteration, we use the 3D models imported from World of Warcraft client program to display character models. In order to obtain the skeletal meshes and animations for these models, we use Wow Model Viewer to export the models along with the animations as FBX files. For level design, we use static models and textures created by Ken Trowbridge, a member of Maerken team. The game sound files are created by Josh Richardson, a student at CSUSB.

Importing Content

After having the files ready, we import the files to UDK using Unreal Editor. The imported content has to be organized into packages and groups. An Unreal package is stored in a package file and put in the content folder. The extension for the packages is upk for normal packages and udk for maps. Every package can contain one or more groups and a group can also contain sub groups. We can put the assets directly in the package or a group inside a package. In Maerken, we organize the asset according to the following rules:

- Player character assets are put in a package.
- NPC character assets are put in a package.
- User interface assets are put in a package.
- Inventory assets are put in a package.

- Common level assets are put in a package.
- Specific level assets are put in the level package.
- In a package, assets with different types are put in different groups except for MythicUI package.
- In MythicUI package, every movie is put in a group along with its assets.

Editing Imported Content

UDK provides tools for editing game assets after importing. The following tools are used in Maerken:

- Material Editor: used to edit the materials.
- AnimSet Editor: used to edit the animation sequences.
- Sound Cue Editor: used to edit sound cues that can be put in the levels.

5.1.2 Content Creation in Unreal Development Kit

In addition to importing art assets from external programs, UDK provides various tools for creating game content.

Material

Unreal Material Editor is used to create materials to be used in the game from imported textures. In Unreal Material Editor, we can perform a lot of operations on 2D images to obtain the result material to be applied on 3D objects. We can even create animated and dynamic lighting materials using Unreal Material Editor. Figure 5.1 shows an example of material creation using Unreal Material Editor.

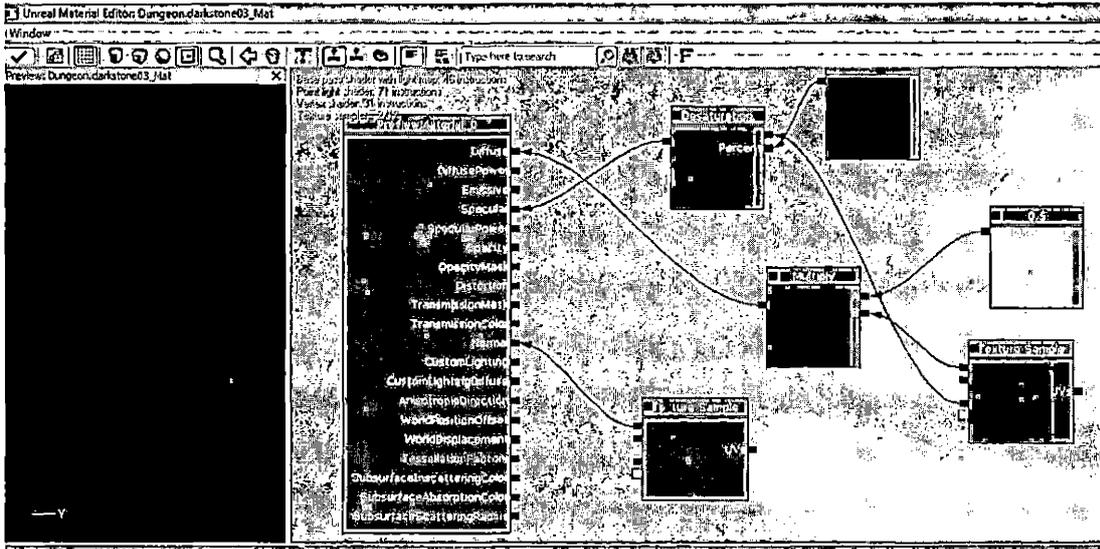


Fig. 5.1: Unreal Material Editor

Particle System

UDK also supports particle systems creation by using Cascade Emitter Editor. This tool can create various particle emitters with different parameters to produce complex particle systems. In the game, the particle systems are used to display the projectiles and the skill effects. Figure 5.2 shows an example of particle system creation using Cascade Emitter Editor.

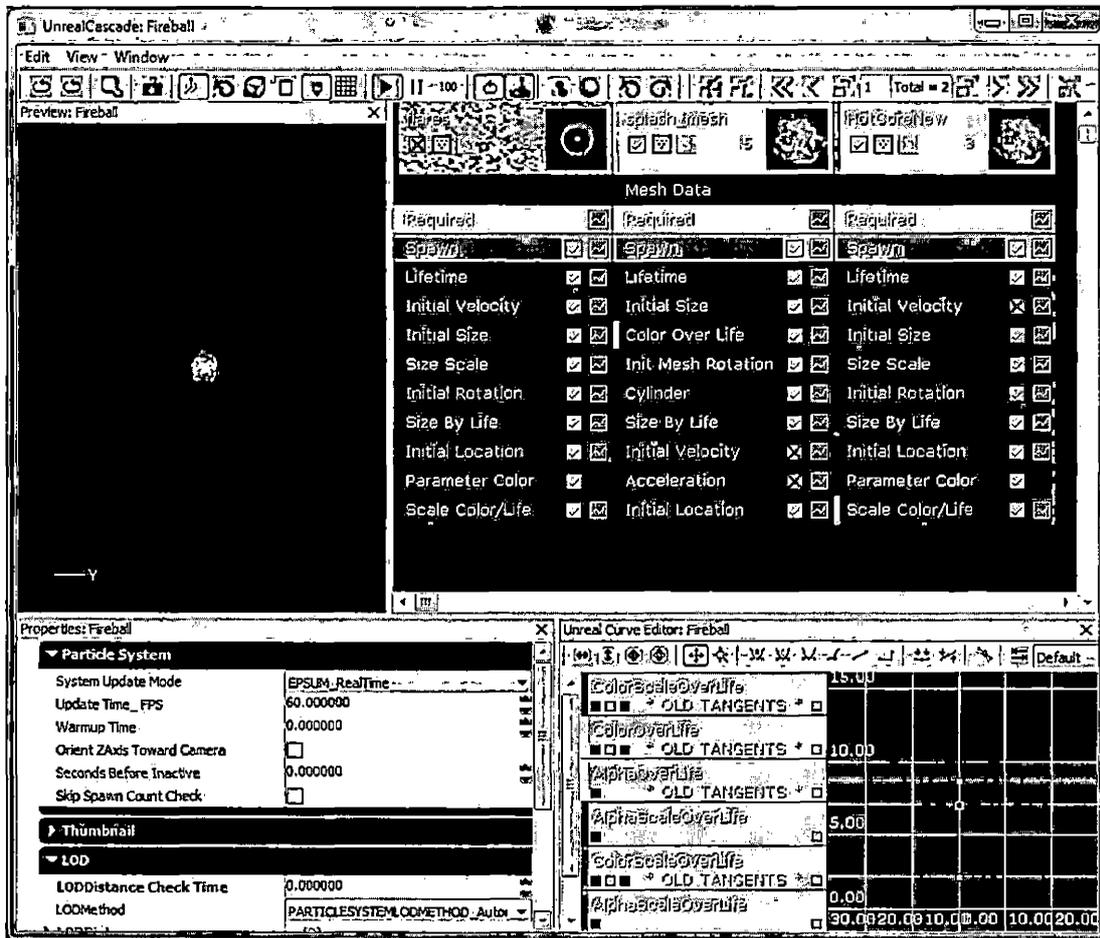


Fig. 5.2: Cascade Emitter Editor

Animation Tree

In order to see the character animation in the game, we need to create an animation tree template for the model. The tree defines rules to play animations. For example, we can use the tree to instruct the model to play animations based on speed, direction or status. Below is the animation tree used to control the animations of the warrior character (figure 5.3).

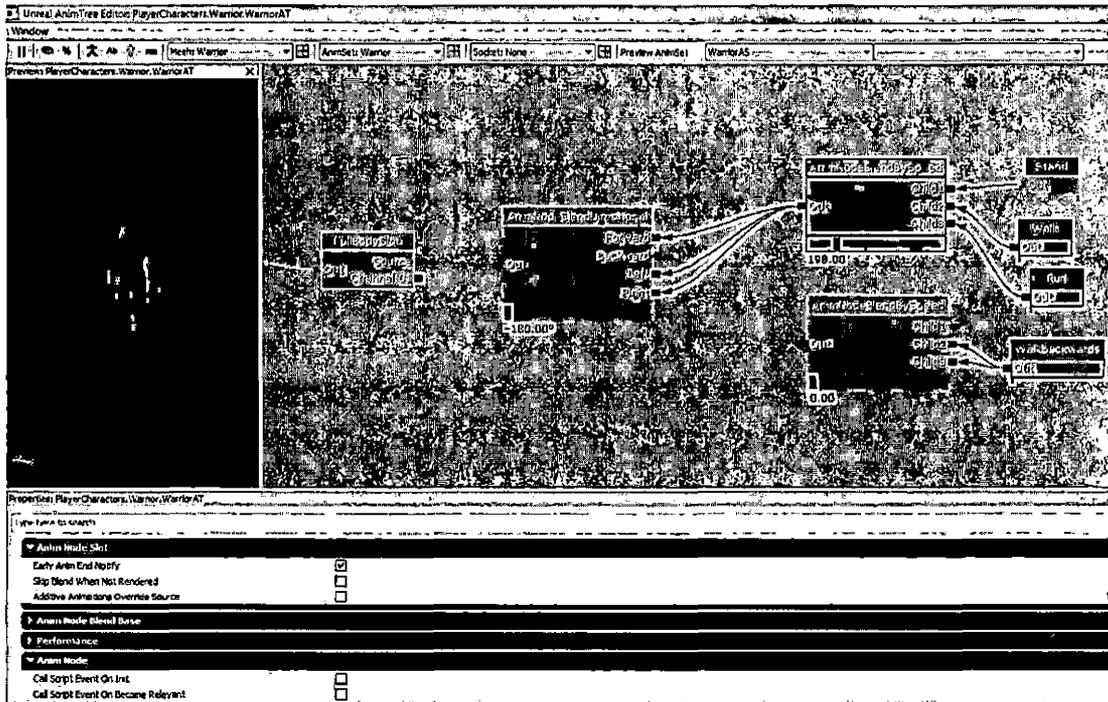


Fig. 5.3: Unreal AnimTree Editor

5.2 Level Creation

Level creation is an important step of game development. It involves a lot of knowledge in different fields, especially in art. It creates the environment where the player can play the game. In this report, we do not explain the process of making a level because the levels used in this iteration are created by Ken Trowbridge, a member of the Maerken team.

5.3 Game Distribution

5.3.1 Configuring and Compiling

In order to run the game, we need to configure the engine to recognize the Maerken package in UnrealScript and then compile it. First, open the file `DefaultEngineUDK.ini` in `UDKGame\Config` folder. Then add the line `+EditPackages=Maerken` right after `+EditPackages=UTEditor` line in `[UnrealEd.EditorEngine]` section. That will direct the compiler to include Maerken package in the compiling process. There are three ways to compile the code:

- Compile from command line: run “UDK make” from the command line in the binaries directory.
- Compile using Visual Studio: in the Visual Studio project, set the compiler to `UDK.exe` in the binaries directory and set the build configuration to release, then choose Build-Build Solution.
- Compile using UnrealFrontend tool: run `UnrealFrontend` from the binaries directory and click on choose Script-Compile scripts.

In order to make the game play Maerken game as default instead of play Unreal Tournament game, we need to modify the `DefaultGameUDK.ini`. The `[Engine.GameInfo]` section in the file should look like

```
[Engine.GameInfo]
DefaultGame=Maerken.MyGame
DefaultServerGame=Maerken.MyGame
PlayerControllerClassName=Maerken.MyPlayerController
DefaultGameType="Maerken.MyGame"
```

5.3.2 Cooking and Packaging

In order to package a game for distribution, we need to cook the content. In Unreal Frontend, we add the menu map and the dungeon map to the list of maps to be cooked then choose Cook. Below is the interface of the Unreal Frontend tool with Maerken configuration (figure 5.4).

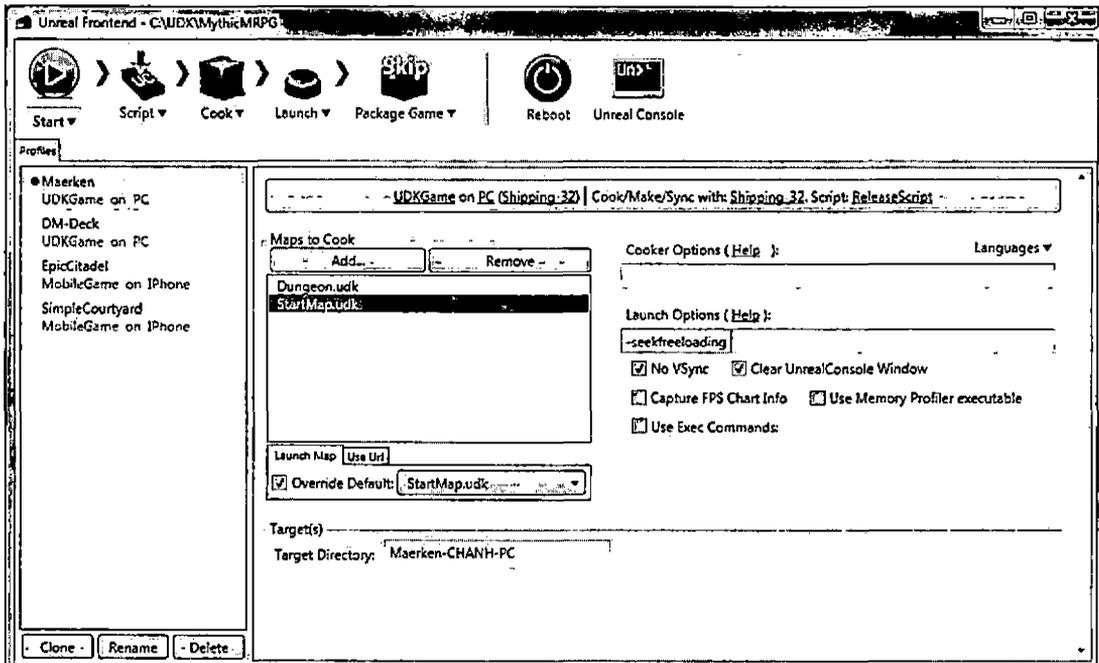


Fig. 5.4: UnrealFrontend Interface

The final step to distribute the game is to package the game and create an installer. This is done by clicking on the Package Game button on the Unreal Frontend.

6. CONCLUSION AND FUTURE DIRECTION

6.1 Conclusion

This iteration is a solid step toward the complete implementation of Maerken. We have achieved a stable design that can be extended to implement a more complex game. In this iteration, we have finished designing the base classes needed to implement the complete game. We have also finished implementing the following features of the game:

1. Game controls: Maerken allows players to use the keyboard to move the character around the gameplay maps. When the player presses the left mouse button, the character attacks the enemy in front of it. When the player presses the right mouse button, the character perform the class skill. There are two class skills implemented: shockwave and fireball.
2. Character stats: in Maerken, a character has four combat stats, namely power, speed, movespeed and armor. Power affects the amount of damage a character can do. Speed affects the attacking and casting speed of a character. Movespeed affects the movement speed of a character. Armor affects the amount of damage a character can absorb from hostile attacks. For player characters, these four stats can be increased every time they level up.
3. Quest system: in Maerken, the player can receive a quest given by a friendly NPC before the boss. After defeating the boss in the boss room, the player receives an item reward from the NPC.

4. AI system: NPCs in Maerken can perform various type of actions. All of the NPCs can move around the map randomly in wandering mode. The hostile NPCs can attack the player when he is in range of them. The friendly NPCs can help the player fight other hostile NPCs.
5. Inventory system: in Maerken, the player can obtain items by defeating hostile NPCs and completing quests. The items are stored in an inventory bag attached to the player character and can be equipped to enhance the player's combat ability.
6. User interface: we implemented 6 menu screens and the HUD for Maerken. The player can use the main menu to start a game or join an existing game. He can use the choose character menu to choose the character to play with and the create character menu to create a new character to be played in the game. While playing the game, the player can open the mid game menu to save or quit the game and open the inventory menu to manage the items he obtained. The HUD can display the health of the player character, the class skill icon along with its cooldown and a cast bar while the player is performing a skill.
7. Game serialization: we implemented the DLL to perform the game serialization. The player can now create and delete character save files. The player can also save the information to the files while playing.
8. Multiplayer system: Maerken allows up to 4 players to play together in a local area network. The game information is synchronized on the server and the clients.

6.2 Future Direction

Although the basic implementation is finished, a great amount of work needs to be done in order to finish the game. We need a complete story to back the game because

the game is an RPG. We also need a more interesting game design especially in combat to make the game more appealing. After that, we need to implement these specifications. However, while we are working on a complex game design, we can improve the implementation in the following aspects:

- **Game content:** we can create and improve the game content artistically. For example, we can decorate the menus with better textures or create better particle systems to make the game look good. In addition, as the story is being developed, we can continue designing new maps. The level designers can start making conceptual maps by designing the base terrain and putting placeholder object in the maps.
- **AI:** we can define more behaviors for the AI system. This improvement can be done along side with the game design to allow the game designers to be more flexible in designing. Some additional AI behaviors that we can implement are: patrolling along a defined path, attacking everything along a defined path, following a player character and moving to a specific destination in the map. Furthermore, we can define new game events to support flexibility in AI reaction. For example, we can notify friendly characters when the player is attacked or notify all hostile characters in a region when a hostile character see the player.

REFERENCES

- [1] ActionScript Documentation. <http://www.adobe.com/devnet/actionscript.html>.
- [2] Fable. <http://lionhead.com/Fable/Fable/Default.aspx>.
- [3] Mythic Entertainment. <http://www.mythicentertainment.com/>.
- [4] Mythic Project. <http://cse.csusb.edu/mythic/overview.php>.
- [5] Unreal Development Kit Documentation. <http://udk.com/documentation>.
- [6] Vindictus. <http://vindictus.nexon.net/>.
- [7] Chris Ballinger. Mythic game project addition of artificial intelligence and quest system components. *CSUSB Masters Project*, June 2010.