

COMP4801 Final Year Project

Final Report

The Road to Castle 3D

Supervisor:

Dr. T. W. Chim

Group Members:

Leung Chiu Yuen, Rain (3035857294)

Ko King Nam, Vincent (3035864297)

Date of Submission:

26 April 2024

Abstract

The popularity of RPGs has spanned over decades, along with the development in gaming technology and growth of the game market. This project aims to develop and deliver a third-person 3D fantasy action RPG through a game rework. To achieve this goal, several gameplay systems and features will be implemented as objectives, by integrating Unity and other applications. Currently, many features have been implemented and the major ones are 3D World Editor, AI Music, Quest, RPG Basics, and Save and Load, have been tested and deployed successfully. These systems control the game mechanics in different scenarios and are the basic framework for future game development. In addition, the prologue of our story is created with a wonderful scene and interactable NPCs. Robust codebase and meticulous planning in the early stage has bore fruit. Adding content to the game now requires minimal effort with our new framework. During the development, difficulties on 3D modelling, collaboration, and time have been encountered and mitigated respectively.

Acknowledgment

We would like to express sincere gratitude to our supervisor, Dr. T. W. Chim, for his guidance and support throughout the development of this final year project. Dr. Chim has given us many valuable comments and suggestions for our game.

Abstract.....	I
Acknowledgment.....	II
List of Figures.....	V
List of Tables.....	VII
Abbreviations.....	VIII
1 Introduction.....	1
1.1 Background.....	1
1.2 Motivation.....	1
1.3 Objectives.....	2
1.4 Report Outline.....	2
2 Methodology.....	3
2.1 Development Tools.....	3
2.1.1 Game Engine – Unity.....	3
2.1.2 IDE – Visual Studio Code.....	4
2.1.3 3D Modelling and Animation – Blender.....	4
2.2 Codebase Design.....	5
2.3 Event-Driven Development.....	5
2.4 Singleton Pattern.....	5
2.5 Physics.....	6
2.6 Player Movement.....	6
2.7 Unity Techniques.....	7
2.8 World Editor.....	7
2.8.1 Basic Actions.....	7
2.8.2 Model-Quest Selection Panel.....	8
2.8.3 Scene Controls.....	9
2.9 Advanced Quest System.....	10
2.9.1 Core Quest System.....	11
2.9.2 Quest API.....	12
2.10 Advanced Data Persistence.....	12
2.11 Scene Switching.....	14
3 Results and Discussion.....	16

3.1	Original RPG Framework.....	16
3.2	Game Features	16
3.2.1	Inventory and Items	17
3.2.2	Combat.....	18
3.2.3	Quest and Conditions.....	19
3.2.4	Leveling	23
3.2.5	Save and Load.....	23
3.2.6	UI Panels.....	23
3.2.7	NPC Trading.....	25
3.2.8	Dungeon Editor.....	28
3.2.9	Game Maps	30
3.2.10	Models and Animations	32
3.2.11	Music and Sound Effects	32
3.3	Difficulties and Mitigations	33
4	Conclusion	34
4.1	Schedule.....	34
4.2	Limitations	35
	References.....	36
	Appendices.....	A
	Appendix A: Creating GameItem in Unity Asset Menu with Attributes	A
	Appendix B: Equation Derivations of 2D Projectile Motion.....	B
	Appendix C: Game Entities Configuration JSON File.....	C
	Appendix D: Creating A New Quest with the Developed Quest API.....	D
	Appendix E: All Subpanels in Game Panel	E

List of Figures

Figure 2.1 Conceptual “assembly line” of the project	4
Figure 2.2 3D models created in Blender (a) and imported to Unity (b).....	4
Figure 2.3 Player faces enemy such that a 2D plane is formed.	6
Figure 2.4 Player movement states and sub-states.....	7
Figure 2.5 Code snippet showing the restore function.....	8
Figure 2.6 Code snippet for DraggableModelButton.....	8
Figure 2.7 Code snippet for dragging a 3D model in the game.	9
Figure 2.8 Code snippet for creating a model in World Editor.....	9
Figure 2.9 Diagram visualizing the smooth rotation problem.	10
Figure 2.10 Code snippet for rotating a 3D model in the game.....	10
Figure 2.11 Storyline definition in code.	11
Figure 2.12 Flow of StoryTask documented in code.	11
Figure 2.13 Sub-dialogue code implementation.	12
Figure 2.14 Generic class definition of DataPersistence.	13
Figure 2.15 Code snippet of the abstract class.....	13
Figure 2.16 Part of the implementation for setting up a switched scene.	15
Figure 3.1 Opening Inventory panel by pressing “I” key.	17
Figure 3.2 Before (up) and after (down) using a Health Potion.	18
Figure 3.3 Damage indicators during battle.....	19
Figure 3.4 An overview of the Quest API.	20
Figure 3.5 Example for setting up a Storyline (a) and in-game interaction (b, c).	21
Figure 3.6 Player stats before (left) and after a level-up event (right).....	23
Figure 3.7 Game (a), Graphics (b), and Audio (c) tab pages in Setting panel.....	25
Figure 3.8 Interact with a tradable NPC (a) and open the Shop panel (b).	26
Figure 3.9 Tooltip on NPC-side Shop panel.....	27
Figure 3.10 Selling price discounted due to NPC Preferences.	28
Figure 3.11 World Editor panel in Dungeon Editor.....	28
Figure 3.12 Adding 3D models in Dungeon Editor.....	29
Figure 3.13 Editing quest conditions in Dungeon Editor.....	30
Figure 3.14 Overview of game maps.	30

Figure 3.15 Layout of the Village game map. 31
Figure 3.16 Layout of the Lost Woods. 32
Figure 3.17 Humanoid avatar (left) and its mappings (right) in Unity Editor 32

List of Tables

Table 4.1 Project schedule with current phase highlighted.....	34
---	----

Abbreviations

3D	Three-dimensional
AI	Artificial Intelligence
CG	Computer Graphics
ECS	Entity Component System
IDE	Integrated Development Environment
NPC	Non-player Character
RPG	Role-playing Game
UI	User Interface
XP	Experience Points

1 Introduction

This chapter introduces the context behind the project in 4 sections. In Section 1.1, a brief background of this project is introduced. In Section 1.2, project rationales are discussed. In Section 1.3, the purpose and objectives of this project are listed. In Section 1.4, an outline of subsequent chapters is provided.

1.1 Background

Role-playing is generally considered one of the most popular game genres, which originated from the publication of *Dungeon and Dragons (D&D)* in 1974 [1]. The term role-play often correlates with make-believe, an action of imaging and pretending something unreal. In RPGs, players can cast themselves and act as characters in fictional settings, based on “the range of imagination” [2] of game developers.

Over the past few years, there has been a rapid growth in the global gaming market, possibly due to pandemic lockdowns [3]. The game market has witnessed great success in classic RPG franchises, such as *Baldur’s Gate*, *Monster Hunter*, and *Dark Souls*. For a long time, the video game market has been dominated by large publishers and studios, with their huge investments in game development and research. But thanks to advances in gaming technology, indie game developers are now capable of creating games with good quality.

Various game development tools and software are now ubiquitous and often accessible to the public. There are free and yet powerful game engines, such as Unity and Unreal Engine 5, available on the market. Moreover, AI technologies are now widely employed in game development to lower production costs. For instance, content creators can utilize AI to write storylines, generate images, and create music for games [4].

1.2 Motivation

As game enthusiasts, the inspiration from RPG masterpieces and the great interest in their game design principles are the sources of motivation for this project. This project serves as a valuable opportunity for the team to gain hands-on experience in game development, and to develop communication and collaboration skills from teamwork.

1.3 Objectives

This project will be a rework and migration of *The Road to Castle* [5], which is a text-based fantasy RPG played on Linux terminal. The final deliverable will be a third-person 3D fantasy action RPG, with implemented gameplay systems and contents.

The main purpose of this game project is to present a fantasy game world with a captivating gaming experience to players. To achieve this, multiple game features and systems will be accomplished as the objectives:

- Develop a flexible, maintainable, and extendable framework from scratch
- Create a compelling storyline, integrated with tutorials
- Provide a user-friendly user interface (UI)
- Design maps with diverse terrains, eco-systems, and weathers
- Provide an inventory system
- Design and implement a balanced combat mechanism
- Allow players to save and load their progress locally and remotely
- Provide a level system for character development
- Allow players to design and share their own dungeons via an in-game editor

1.4 Report Outline

In Chapter 2, the methodologies of this project will be discussed, including development tools, concepts, and techniques to be used and their justifications. Following that, Chapter 3 introduces the results achieved in this project, as well as the problems encountered with mitigations. Finally, there will be a summary of current progress and future work plan in Chapter 4.

2 Methodology

In this chapter, software, concepts, and other methods used in this project and their justifications are discussed in 8 sections. In Section 2.1, the choices of game engine, IDE, and CG software are discussed. In Section 2.2, the project structure and codebase design are explained. In Section 2.3, the event-driven design of the game is introduced. In Section 2.4, the singleton pattern is discussed. In Section 2.5, the physics and projectile motion is elaborated. In Section 2.6, player movements such as walking and climbing are discussed. In Section 2.7, different useful techniques in Unity are presented. In Section 2.8, functionalities of the world editor are presented. In Section 2.9, the updated quest system and its API are mentioned. In Section 2.10, an advanced data class for serialization named data persistence is discussed. Finally, in Section 2.11, scene-to- scene switching and its idea is explained.

2.1 Development Tools

2.1.1 Game Engine – Unity

The game will be made with Unity, a free and user-friendly game engine, which supports cross-platform 3D game development with C# scripting. It acts as the platform that loads and integrates all gameplay systems and local game contents into a game.

Unity also serves as the “backbone” for connecting all contents from different software. Figure 2.1 below illustrates the concept of such integration between these applications. In the figure, empty entities are created in Unity as containers to store scripts, 3D models, and animations from IDE and 3D modeler. These entities are later composed into unique game objects for different game scenes.

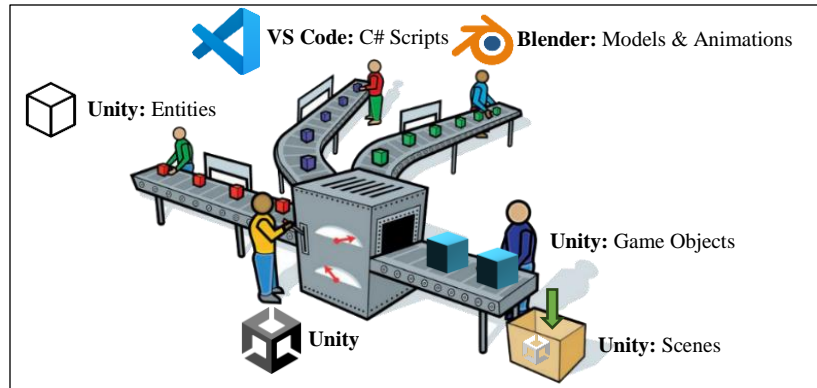


Figure 2.1 Conceptual “assembly line” of the project

Compared with other alternatives like Unreal Engine 5, Unity is more user-friendly in terms of its interface and functions, and it is easier to run on small-scale projects. Also, it provides some useful built-in systems that could facilitate the development process.

2.1.2 IDE – Visual Studio Code

An integrated development environment (IDE) is an application that provides developer tools for software development. Visual Studio Code is selected as the IDE, as it provides high extensibility with custom extensions to fulfill different requirements. For this project, extensions can be installed to support Unity development and debugging on C# scripts.

2.1.3 3D Modelling and Animation – Blender

Blender is a free 3D creation software that supports most CG functionalities, such as modelling, animation, and rendering. In the project, architectural models, character models, and animation can be created in Blender (see Figure 2.2a), and then imported directly to Unity (see Figure 2.2b).

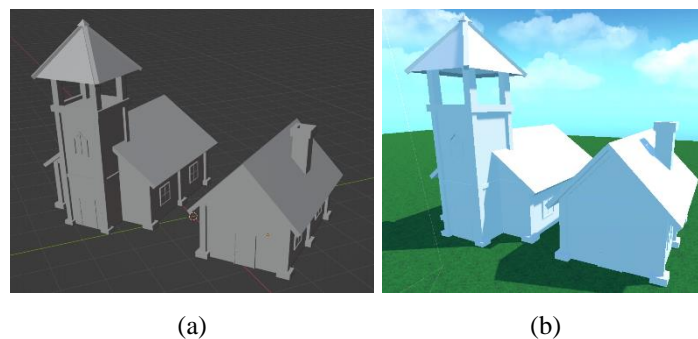


Figure 2.2 3D models created in Blender (a) and imported to Unity (b)

2.2 Codebase Design

To develop a flexible and extendable gaming framework, a well-organized file structure is required. In this project, game contents are modularized and grouped under the standard convention [6]. For example, a “Scripts” directory is used to contain all scripts of the game, with sub-directories separated by content types. This improves division of work, as team members can develop different features simultaneously without interruption.

In addition, Unity attributes and data container classes have been applied to facilitate the development process on game assets. For instance, with “[CreateAssetMenu]” attribute and objects inherited from ScriptableObject class in Unity, self-defined assets can then be created by simply clicking on the asset menu, instead of repetitive programming, as shown in Appendix A.

For code reuse, inheritance is extensively used in this project. For example, a GameItem class has been developed as the superclass for different game items, such as game equipment and consumables, to inherit common behaviors.

2.3 Event-Driven Development

For games, event-driven design is an architecture that implements the game flow based on events, instead of a predetermined series of actions. Events in games are generally the inputs from users, such as mouse click and keypress. These events are captured as the “triggers” for follow-up actions or other events designed by the developers.

Such design allows the game to provide prompt and dynamic responses to players depending on their behaviors, as well as for system-to-system communications. For example, the combat system can notify the quest system whenever the player has made progress in quests. Also, different events are independent and are developed separately for specific tasks, which facilitate the development and testing process.

2.4 Singleton Pattern

A singleton is a globally accessible class that has only one instance at a time. A singleton object will not be re-assigned after instantiation. This helps protect variables and values

from unexpected behaviors. One example is to create a singleton for user settings to prevent them from changing when switching to a new game scene.

As Unity allows configuration on objects in the editor before the game runs, constant values can be assigned to singletons. This prevents repetitive coding and simplifies the development process of complicated systems, such as the UI systems.

2.5 Physics

To make the game more immersive, simulations on physics such as gravity and collision are added to the game world. For example, projectiles with weights like arrows are shot with parabola-like locus instead of a straight line. Such curve on a 2D plane follows the equations derived in Appendix B. In a 3D game world, this is done by first forming a 2D plane between the faces of two entities, such as a player and an enemy (see Figure 2.3). An ideal angle can then be calculated and assigned to the projectile.

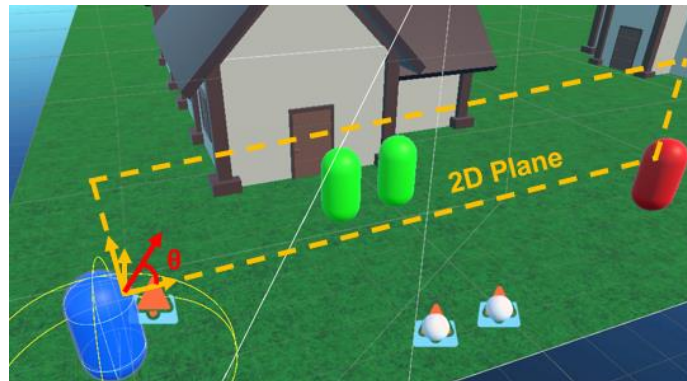


Figure 2.3 Player faces enemy such that a 2D plane is formed.

2.6 Player Movement

To support player movements on the map, two movement states for moving and standing are required. These two states are combined to provide a smooth movement behavior of the player, such as walking, jumping, and climbing. Also, they allow later integrations of character animations. Figure 2.4 below shows the two states and their respective sub-states.

```
public enum MovementStates{
    RUN, WALK, IDLE, JUMP, CLIMB, AIR
}
public enum GroundStates{
    NORMAL, SLOPED
}
```

Figure 2.4 Player movement states and sub-states.

2.7 Unity Techniques

In this project, Unity APIs are utilized to reduce development time. For example, Physics API provides some useful functions for ray casting and creating colliding spheres on the fly. Rays are cast onto the floor to detect whether the player is on the ground, while collider spheres are created to detect objects near the player. These can be applied in player-object interactions, where spheres are used to detect and check whether colliding objects are interactable.

It is also remarkable that a wise use of Unity can sometimes lead to no-code solutions. For example, by using built-in packages like Cinemachine, the camera controls can be simplified.

2.8 World Editor

World Editor is also known as Dungeon Editor in the game. And it is mainly separated into 3 parts: Basic Actions, Model-Quest Selection Panel and Scene Controls.

2.8.1 Basic Actions

Five major actions are implemented: Restore, Remove, Save, Load and Leave. As a world editor, we want basic functions such as add and remove. However, adding is also implemented in the Model-Quest Selection Panel. The remove functionality is defined to remove the currently selected model.

Save and Load is simply saving and loading the scene and quests back and forth from a file with our own Data Persistence API.

Just in case the user modified the original dungeon by accident. There is a function to restore the built-in dungeon. To do so, we have decided to replace the current json save with the original built-in json.


```

EditorControlsPanel.instance.restoreButton.onClick.AddListener(()=>{
    string ogFile = Path.GetFullPath(Path.Combine("Saves/original", saveFilename));
    string currentFile = Path.GetFullPath(Path.Combine("Saves", saveFilename));
    File.Delete(currentFile);
    File.Copy(ogFile, currentFile);
});

```

Figure 2.5 Code snippet showing the restore function.

Since world editor is designed to be used by users normally in the game, we want them to leave the world editor without breaking anything. That is the reason for the “leave” function. Loading the main world back in the game client is the concept behind this function.

2.8.2 Model-Quest Selection Panel

This panel allows users to drag and add 3D models into the scene. In addition to adding basic 3D models into the scene, users can also customize the quest they want in the current dungeon. Currently, not all quests can be saved. But major ones like puzzle and kill quests can be brought into a dungeon. Saving and loading of quests will be discussed in detail later in the section Advanced Serialization.

The mechanism of dragging a model from a UI into a 3D scene requires some tricks. To start with, we want the UI button to detect mouse drags. This can be achieved by implementing DragHandlers to the class as shown in the following code.

```

public class DraggableModelButton: MonoBehaviour, IBeginDragHandler, IDragHandler, IEndDragHandler{
    private Image image;

    private Vector3 anchorPos;
    private bool isDragging = false;
    // private Transform ogTransform;

    public bool horizontalDragOnly = false;
    public float threshold = 1;

    public Action<PointerEventData> onBeforeBeginDragListener;
    public Action<ModelInfo> onBeginDragListener;
    public Action<ModelInfo> onDragListener;
    public Action<ModelInfo> onEndDragListener;
}

```

Figure 2.6 Code snippet for DraggableModelButton.

The code above handles the dragging of a button. But it does not account for creating and dragging of a 3D model. For dragging, a 2D to 3D conversion process is needed. Since the mouse is dragging in a 2D space (screen), we want to convert it into 3D space (game scene).

Here, a Unity built-in API is used to shoot a ray from camera to the world, hitting the floor to obtain the world position.

```
public void moveModelToMouseLocation(){
    if(modelSelected == null) return;
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    if(Physics.Raycast(ray, out RaycastHit rayHitInfo, 100, whatIsGround)){
        modelSelected.transform.position = rayHitInfo.point + anchorOffset;
        selectedIndicator.transform.position = modelSelected.transform.position;
    }
}
```

Figure 2.7 Code snippet for dragging a 3D model in the game.

To create it, we simply use the built-in function from Unity to create and load a 3D model (fbx) into the scene at mouse location.

```
/// <returns>A model with UniqueModel component</returns>
public GameObject createAndAddModel(ModelInfo info, WorldEditorModelData modelConfig = null, string id = null){
    GameObject prefab = (GameObject) Resources.Load(info.getResourcePath());
    if(prefab == null){
        throw new InvalidOperationException("Invalid resource path: " + info.getResourcePath());
    }
    GameObject newModel = null;
    if(modelConfig != null)
        newModel = Instantiate(prefab, modelConfig.pos, modelConfig.rot, ...);
    else{
        newModel = Instantiate(prefab, ...);
    }
}
```

Figure 2.8 Code snippet for creating a model in World Editor.

2.8.3 Scene Controls

With the conversion process being implemented, positioning and rotation of the 3D model can now be handled next. Positioning is simple, it is shown in the code above. But rotation is tricky. We want rotation to be smooth, meaning that rotation angle will not start from 0 the moment we start rotating the model. Instead, we want the rotation to start from the current rotation. To solve this problem, it is best to draw out a diagram as follows.

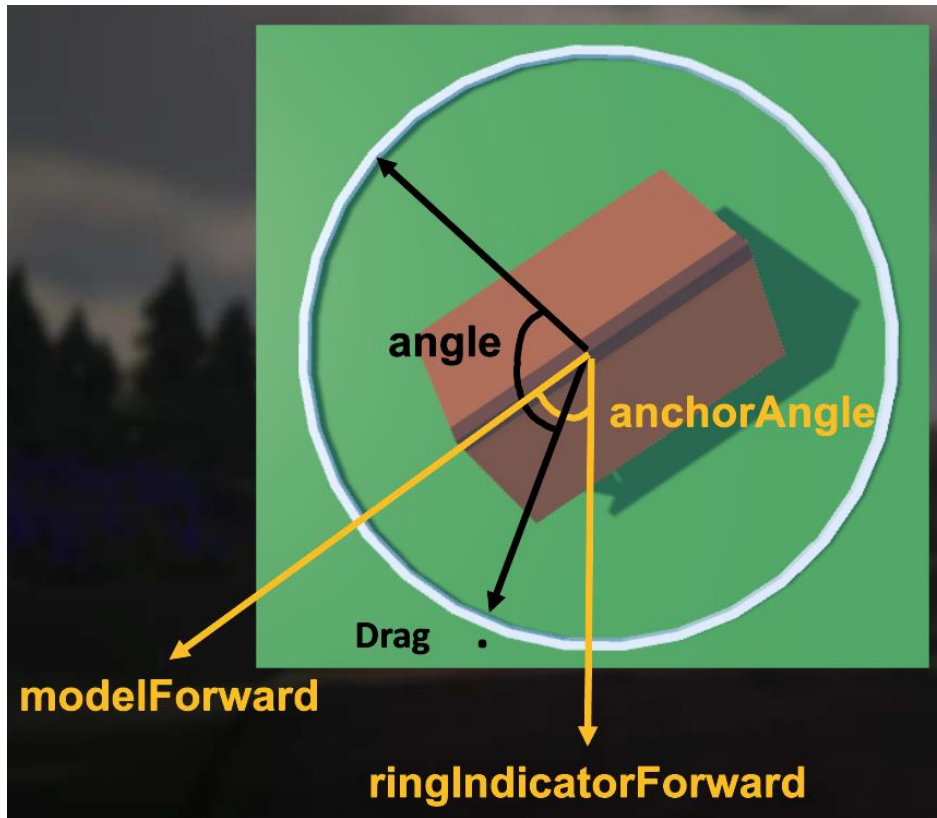


Figure 2.9 Diagram visualizing the smooth rotation problem.

Using the information above, the solution is clear that the new drag angle once the mouse starts dragging the rotation ring indicator is $\text{angle} + \text{anchorAngle}$.

```
private void rotateModelFromIndicator(){
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    if(Physics.Raycast(ray, out RaycastHit hitInfo, 100, whatIsGround)){
        Vector3 modelToMouseVector = hitInfo.point - selectedIndicator.transform.position;
        modelToMouseVector.y = 0;
        float angle = VectorUtils.vectAngle(selectedIndicator.transform.up, anchorVector, modelToMouseVector);
        modelSelected.transform.rotation = Quaternion.AngleAxis(angle + anchorAngle, modelSelected.transform.up);
    }
}
```

Figure 2.10 Code snippet for rotating a 3D model in the game.

2.9 Advanced Quest System

Story Quests and World Quests are a huge part of an RPG game. Hence, the team have spent a lot of time designing the system to make Quest API simple and extendable. There are 2 parts to our quest system. The first one is the core of the system. This includes dialogue and narration, core logic and flow of quests and data persistence of quests. The second part is the design of Quest API such that content creators can use the API at ease.

2.9.1 Core Quest System

A storyline contains multiple tasks. Tasks must go in a linear fashion due to the nature of its implementation as shown below.

```
[System.Serializable]
public class Storyline{
    public string name;
    public int currentIndex = 0;
    public bool repeatable = false;
    public bool hideFromQuestPanel = false;

    // Currently, storyline goes in a straight line. No branches.
    public List<StoryTask> tasks = new List<StoryTask>();
    [NonSerialized] public Action<Storyline> onCompleted;
```

Figure 2.11 Storyline definition in code.

For content creators, they can also use event listeners like `onCompleted` to implement quest-specific logic.

A `StoryTask` is like a quest. Each task can contain dialogues and conditions. Players must complete all conditions of a task in order to pass a `StoryTask`. Most of the time, a combination of dialogues and conditions are used. Moreover, dialogue now supports responses. This makes conversation between NPCs a two-way communication, instead of the original boring one-way route. Example flow of a `StoryTask` is shown as follows:

```
/**
 * <summary>
 * Example Flows:
 * 1. Dialogue -> Dialogue -> Dialogue -> (optional) Close -> (optional) Conditions -> Finish
 * 2. Conditions -> Finish
 *
 *         based on response
 * 3. Dialogue -> Branch Dialogue -> Dialogue -- -> (optional) Conditions -> Finish
 *
 *         \_ -> Dialogue _/
 *
 * It is impossible to do:
 * Kill monsters -> Dialogue -> Dialogue -> Finish
 * </summary>
 */
```

Figure 2.12 Flow of `StoryTask` documented in code.

Implementing dialogue with responses is hard, but the basic concept of it is to replace the current flow of dialogue with a branch.

```
Dialogue dialogue;  
if(subdialogues == null)  
    dialogue = dialogues[currentDialogueIndex++];  
else dialogue = subdialogues[currentSubdialogueIndex++];  
DialoguePanel.instance.showDialogue(dialogue);
```

Figure 2.13 Sub-dialogue code implementation.

Dialogues can now repeat from a previous point. So that, when players forget the quest requirement, they can still talk to the NPC to get task relevant information to complete a quest.

2.9.2 Quest API

Designing a flexible API takes time. We also aim to make it as simple to use as possible. It is also required that the API can do a wide variety of things. For example, the quest API allows puzzles and world quests to be implemented. The API is very simple, and diving deep into the implementation is not needed. Only by looking at code examples suffices content creators need. Application of Quest API will be discussed in detail in the latter part of this report.

2.10 Advanced Data Persistence

Serialization is a process of converting data structures into certain file format for storage and transmission. Objects are saved into specific file formats via serialization and are later loaded back to the game. There is also a deserialization process where a file is converted back into an OOP object. For simplicity, we have generalized the whole system into Data Persistence.

To allow players to save their game progress and resume afterwards, a “savable” object class and a Save and Load system is desired. One possible solution is to loop through all game objects once to check whether they are savable or not. For example, a GameData class is designed as the container for the game data in objects. The GameData objects will be converted and saved as JSON files using a famous c# library called Newtonsoft Json.

To further improve code quality and robustness, a generic `DataPersistence` class is created to make this process applicable to all types of data objects. For example, world editor data, game data and entities configuration objects can all go through the same generic `DataPersistence` class to convert themselves into a json file. Below shows a snippet of the generic class. Newtonsoft Json will handle the serialization and deserialization of common type values.

```
public class DataPersistence<T> where T: new(){
    private T data = new T();

    public string dataFolder = "Saves";
    public string dataFile;
```

Figure 2.14 Generic class definition of `DataPersistence`.

In addition to a generic data persistence class, we have also implemented a way to convert abstract classes back and forth from a file. This process is widely used in our advanced quest system to simplify the creation of new quest tasks. The concept behind the implementation is to store the type of a class. Then, using the type information, the corresponding type of the class will be created. Below shows the abstract class used to implement player-defined quests in the Dungeon Editor.

```
/// <summary>
/// Interface for all TaskConditions that can be added in a WorldEditor.
/// </summary>
public abstract class TaskConditionEditorBlueprint{
    public string id;
    [JsonProperty] protected string blueprintType;

    public TaskConditionEditorBlueprint(){
        blueprintType = GetType().ToString();
        id = Guid.NewGuid().ToString();
    }

    public abstract void init();
```

Figure 2.15 Code snippet of the abstract class.

The whole in-house data persistence solution is packaged into a name called DataPersistence API.

2.11 Scene Switching

Games often require scene switching. Scene switching is often used in games that have levels, for example, Angry Birds. A common application of scene switching in all games is switching between the main menu and the game world. We also have implemented scene switching for the main menu and the game world. However, in our case, additional preparations are needed.

Because the singleton pattern is widely used in our game, there are times when the singleton is no longer available in a scene. For example, UI singleton in the main world and the game world is fundamentally different because the UI of the two scenes is different.

The additional preparation is to re-assign or re-create the singletons needed for a scene. The whole process is packed into a single utility class called SceneUtils. Below shows part of the class, where the camera in the new scene is re-assigned back into the singletons.

```

public class SceneUtils: MonoBehaviour{
    private static void setupScene(Scene scene, GameObject cameraCollection, GameObject player){
        cameraCollection.name = "CameraCollection";
        player.name = "Player";

        ThirdPersonCam thirdPersonCam = cameraCollection.transform.Find("Main Camera").GetComponent<ThirdPersonCam>();
        thirdPersonCam.orientation = player.transform.Find("Orientation");
        thirdPersonCam.player = player.transform;
        thirdPersonCam.playerObj = player.transform.Find("CharacterModel");
        thirdPersonCam.playerRb = player.GetComponent<Rigidbody>();
        thirdPersonCam.combatLookAt = thirdPersonCam.orientation.transform.Find("CombatLookAt");

        thirdPersonCam.characterCamera = cameraCollection.transform.Find("3rdPersonCam").gameObject;
        thirdPersonCam.freemoveCamera = cameraCollection.transform.Find("FreemoveCam").gameObject;
        thirdPersonCam.dollyTrackCamera = cameraCollection.transform.Find("TrackCam").gameObject;

        CinemachineFreeLook characterCamera = thirdPersonCam.characterCamera.GetComponent<CinemachineFreeLook>();
        characterCamera.Follow = player.transform;
        characterCamera.LookAt = thirdPersonCam.orientation;

        CinemachineVirtualCamera freemoveCamera = thirdPersonCam.freemoveCamera.GetComponent<CinemachineVirtualCamera>();
        freemoveCamera.Follow = player.transform.Find("FreemoveCamPoint");
        freemoveCamera.LookAt = freemoveCamera.Follow.Find("FreeCamOrientation");

        GlobalScriptObject.instance.currentScene = scene;
        GlobalScriptObject.instance.virtualWorld = GameObjectUtils.findGameObjectByNameFromArr(scene.GetRootGameObjects(), "VirtualWorld");
        GlobalScriptObject.instance.player = player.GetComponent<PlayerObjectLogic>();
        InventoryPanel.instance.player = GlobalScriptObject.instance.player;

        if(GameObjectUtils.findGameObjectByNameFromArr(scene.GetRootGameObjects(), "WorldEditor") == null){
            GlobalScriptObject.instance.worldEditorGO = scene.CreateGameObject("WorldEditor");
        }
    }
}

```

Figure 2.16 Part of the implementation for setting up a switched scene.

3 Results and Discussion

This chapter provides an overview of the project results, organized into two sections. In addition, technical issues encountered during the development are addressed. In Section 3.1, an RPG framework developed for the game is introduced. In Section 3.2, twelve implemented game features are explained. In Section 3.3, the problems faced in the project and their mitigations are discussed.

3.1 Original RPG Framework

To facilitate the development process and improve maintainability, an original RPG framework is first created from scratch before introducing game content to the game. It mainly integrates with Unity Editor and works as a foundation or “skeleton” for game feature implementations.

This RPG framework offers powerful functionalities to develop game logic across systems easily in Unity. It includes a Main Event System that allows system-to-system communications between different gameplay systems. For instance, the Trading system (see Section 3.2.7) relies on the inventories of both the player and NPCs, while the Inventory system itself does not have access to NPC inventories. Hence, the transactions are made by calling the Main Event System to transfer the game item from one side to another.

The framework also provides ways to manipulate game configurations. Developers can create, configure, save, and load between game configurations and game objects in JSON easily with the DataPersistence class (see Section 2.10).

3.2 Game Features

Eight major gameplay systems are introduced as game features, namely Inventory and Items, Combat, Quest and Dialogue, Leveling, Save and Load, UI Panels, Trading, and Dungeon Editor. In addition, there are also four minor content-related game features, including Storyline and Dialogue, Game Maps, Models and Animations, as well as Music and Sound Effects.

3.2.1 Inventory and Items

Inventory management is generally considered an important feature of an RPG. In the game, an Inventory system is developed for the player to collect and store items in the inventory, with a UI panel to view, use, and equip game items. By pressing the “I” hotkey, an Inventory panel pops up on the left side of the screen (see Figure 3.1).



Figure 3.1 Opening Inventory panel by pressing “I” key.

The player can use a “usable” game item, such as a consumable or an equipment, by clicking the icon of the corresponding item in the Inventory panel. For instance, when the player gets a Health Potion in his inventory and clicks on it, he will consume it and recover his health points (see Figure 3.2).



Figure 3.2 Before (up) and after (down) using a Health Potion.

These game items can be easily created in Unity Editor

3.2.2 Combat

A Combat system is developed to handle all the logic in battles, such as calculations of attack damages and statistics of game items. For instance, equipping a helmet provides specific player stats like armor, which lowers the damage received by the player. Also, the player can choose to use consumable game items for combat, such as using health potions

for healing or strength potions for attack damage. During the battle, damage indicators are created to show the damage received from or dealt to an enemy (see Figure 3.3).



Figure 3.3 Damage indicators during battle.

Once the player defeats an enemy, loots such as game items and equipment are dropped according to game configuration. Such configuration lists all items to be dropped by specific enemies and their corresponding drop rates. An example is shown in Appendix D.

To make the Combat system and the gameplay more balanced, enemies will get stronger along with the level of the player. Additionally, an enemy AI agent is developed to make the game more immersive. When the player moves close enough to an enemy, it will start chasing and try to attack the player.

3.2.3 Quest and Conditions

A Quest system is created to allow the player to progress through the storylines with different challenges, also known as story tasks in the game. The player will be rewarded with experience points, game items, and gold once they complete a task. This system is particularly designed to be maintainable and extendible due to its size of contents.

A Quest API, as mentioned, is developed to serve such purposes. In the design, the game world will have a unique collection of storylines, known as the StoryBank. A StoryBank

is composed of multiple parallel Storyline objects, each containing a series of StoryTask (see Figure 3.4).

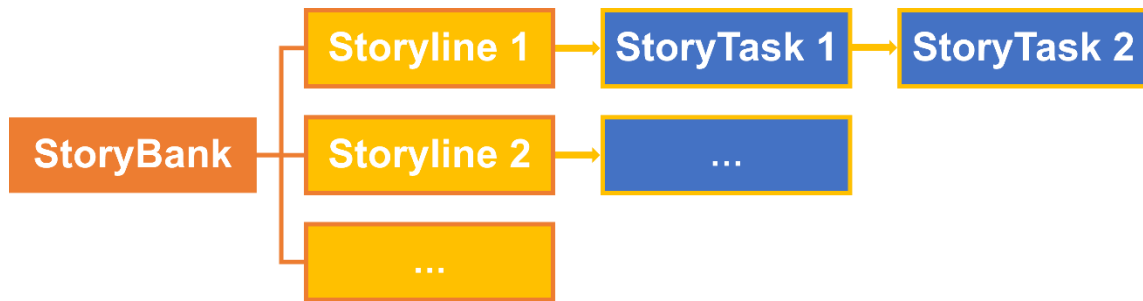


Figure 3.4 An overview of the Quest API.

Such API allows developers to implement Storylines and StoryTasks with dialogue and rewards. By creating a Storyline object with a name, adding multiple StoryTasks with single or branching dialogue (dialogue that provides a few options for response), and setting the rewards in terms of experience points and gold for each of them (see Figure 3.5).

```
// Storyline - John the Blacksmith
Storyline blacksmithStory = new Storyline("John the Blacksmith");

// Task 0 - Talk to Bob
StoryTask talkToJohn = new StoryTask("Talk to John");

talkToJohn.addDialogue(new BranchingDialogue(
    bob, "Hey, how are you?",
    new List<string>(){ "I'm fine.", "How are you?" },
    new Dictionary<int, List<Dialogue>>(){
        {0, new List<Dialogue>(){
            new Dialogue(bob, "That's good to hear. *laugh*")
        }},
        {1, new List<Dialogue>(){
            new Dialogue(bob, "Why are you asking me the same question?"),
            new Dialogue(bob, "Anyways, good to hear that.")
        }}
    }
));

talkToJohn.addDialogue(new BranchingDialogue(
    bob, "I heard that John is looking for you. Maybe you can pay him a visit.",
    new List<string>(){ "I'll go to him.", "I'll go to him later." },
    new Dictionary<int, List<Dialogue>>(){
        {0, new List<Dialogue>(){
            new Dialogue(bob, "Good luck!")
        }},
        {1, new List<Dialogue>(){
            new Dialogue(bob, "Alright, take your time.")
        }}
    }
));

talkToJohn.addDialogue(new Dialogue(bob, "He should be in the smithy nearby."), true);

talkToJohn.rewards.xp = 10;
blacksmithStory.addTask(talkToJohn);
```

(a)



(b)



(c)

Figure 3.5 Example for setting up a Storyline (a) and in-game interaction (b, c).

The API also includes in total seven types of quest conditions for StoryTasks, namely Item condition, Kill condition, Puzzle condition, Area condition, Defender condition, Open-door condition, and Dolly Track condition. These conditions allow developers to create different types of tasks and mini games, such as investigation quest that requires players to

move to a certain location or kill quest that asks players to defeat certain number of target enemies.

The first quest condition is Item condition, it requires the player to deliver specific items to an NPC. By utilizing the Quest API, the required game items for a quest can be retrieved in the form of `InGameLogicalObjects` (a special class for storing game item data) and added to the task, with their required amounts.

Another quest condition is Kill condition, which requires the player to slay specific enemies in certain location like typical RPGs. The API provides a way to spawn such enemies from prefabs to game objects on specific coordinates and counts the number of slayed enemies to keep track of the task status.

The third condition is Puzzle condition. This condition sets up different types of puzzles designed by the team to certain location for players to solve. Currently, two types of puzzles, Basic and Easy, have been developed. These puzzles require the player to activate all the interactable towers in specific sequence and turn them into red color to complete the condition.

The fourth condition is Area condition, which is by-design completed when the player walks within a certain area in the game world. This condition acts like a trigger to proceed with the Storyline or to start a new one.

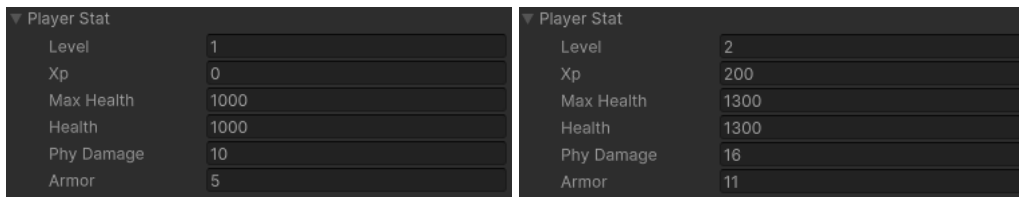
The fifth condition is Defender condition. This condition first spawns a base named Nexus and waves of enemies marching to it. The player is required to defend the Nexus and slay all enemies to complete the condition.

The sixth condition is Open-door condition, which simply removes doors or borders that are blocking the player from certain areas. This is used to open a new game map for players when certain conditions are fulfilled.

The last quest condition is Dolly Track condition. This is a special condition that controls the main camera and moves it in predefined tracks created by the developers. It is useful in cases like creating cut scenes during the quest, or during scene switching.

3.2.4 Leveling

A Leveling system is built to manage the experience points (XP), levels, and their effects on player attributes and abilities. The threshold for each level, or the XP required to level up, is designed to grow quadratically to balance the difficulty of the game. When a player accumulates enough XP and reaches a threshold, the system will trigger a level-up event, which increases the level and corresponding player stats according to the level (see Figure 3.6).



Player Stat	Level 1	Level 2
Level	1	2
Xp	0	200
Max Health	1000	1300
Health	1000	1300
Phy Damage	10	16
Armor	5	11

Figure 3.6 Player stats before (left) and after a level-up event (right).

3.2.5 Save and Load

A Save and Load system is developed based on serialization method as mentioned. It allows the player to save and load not only his progress during gameplay, but also user-customized dungeons made with the Dungeon Editor (see Section 3.2.8). Also, game configurations in different game scenes can also be saved and loaded. One example is the game entities configuration for looting on item drops and drop rates in Appendix D.

3.2.6 UI Panels

A UI system is built to serve most of the gameplay systems. It consists of three main panels, which are the Game panel, Setting panel, and World Editor panel. These UI panels act as the medium for the player to interact with the game. For example, the Inventory panel mentioned above (see Section 3.2.1) is a part of the Game panel.

The Game panel includes all UI components or subpanels for normal gameplay, including the Minimap panel, Player Stat panel, Inventory panel, Quest panel, Interaction Info panel, Shop panel, and Dialogue panel (see Appendix E).

The Setting panel comprises of three tab pages, namely Game, Graphics, and Audio (see Figure 3.7). The Game page provides ways for players to exit back to the main menu or

exit the game. The Graphics tab page includes drop-downs for resolution and quality level for the game, as well as a toggle for full screen. It allows the player to configure the graphics based on their preferences and game performance. The audio page allows the player to adjust the volumes of music and sound effects with sliders, as well as toggles to mute them.



(a)



(b)



(c)

Figure 3.7 Game (a), Graphics (b), and Audio (c) tab pages in Setting panel.

3.2.7 NPC Trading

A Trading system is implemented as an extra feature that allows the player to exchange game items. When the player interacts with a tradable NPC, a “Shop” option will be shown on the Interaction Info panel. By pressing “F” to interact, a Shop panel is opened, showing the inventories of the player and the NPC, with labels of gold amounts (see Figure 3.9).



(a)



(b)

Figure 3.8 Interact with a tradable NPC (a) and open the Shop panel (b).

In the Shop panel, the player can hover his cursor on an item on either side, and a tooltip with info about that item will be shown on it (see Figure 3.9). Similar to the tooltip in the Inventory panel, the tooltip contains the name and description of the item. In addition, it lists the price (buying price in NPC-side panel, selling price in player-side panel) of that item.



Figure 3.9 Tooltip on NPC-side Shop panel.

Intuitively, by left clicking on the item in the NPC inventory, the player can buy the item from the NPC. And vice versa, the player can sell his item by left clicking on the item in his own inventory.

To make the game more immersive and realistic, NPCs are set with different preferences on buying and selling different types of items, known as the `BuyItemPreferences` and `SellItemPreferences`. Such preferences affect the buying price and selling price of an NPC. For instance, from the perspective of an NPC, a Health Potion costs 50 Gold when selling to the player but is priced as 40 Gold when buying it back from the player (see Figure 3.10).



Figure 3.10 Selling price discounted due to NPC Preferences.

3.2.8 Dungeon Editor

A Dungeon Editor is developed based on the World Editor mentioned (see Figure 3.11). It serves to provide players with a user-friendly editor for customizing their own unique dungeon maps, with 3D models and a series of challenges using quest conditions and share the maps with their friends.



Figure 3.11 World Editor panel in Dungeon Editor.

For control, players can move the camera freely in four directions with W, A, S, D keys. The camera can also be raised or lowered with space and control keys.

To add 3D models and quest conditions to the dungeon, the player can make use of the Content panel on the right, with “Models” and “Condition” buttons. After clicking “Models”, a list of available 3D models will be shown (see Figure 3.12). By drag-and-drop, players can add these models to the dungeon easily.

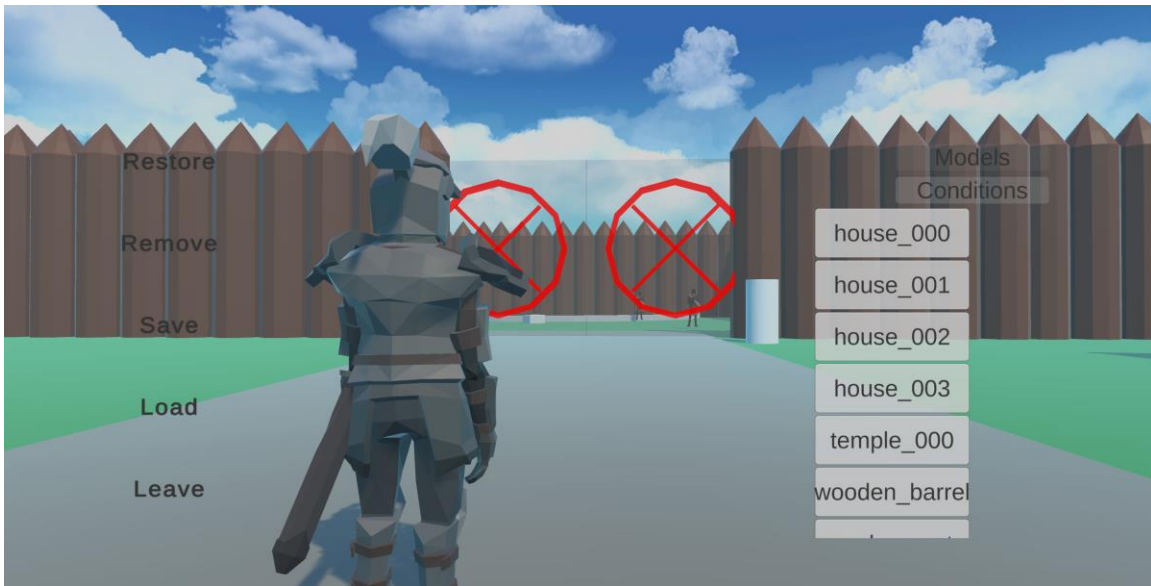


Figure 3.12 Adding 3D models in Dungeon Editor.

In addition, players can also add quest conditions introduced in Section 3.2.3 as a series of tasks, using the “Conditions” button and a panel (see Figure 3.13).



Figure 3.13 Editing quest conditions in Dungeon Editor.

3.2.9 Game Maps

Currently, two game maps have been added to the game, including a village and a forest named Lost Woods (see Figure 3.14).



Figure 3.14 Overview of game maps.

In the village, architectural models such as houses and wooden pavements, as well as a few interactable NPCs, have been created and added to the game world (see Figure 3.15). This village serves as the starting point for the player and the focus of the first Storyline.



Figure 3.15 Layout of the Village game map.

In Lost Woods, trees and a bandit camp have been added to the map (see Figure 3.16). This game map serves as a location for the player to complete certain quests in the Storylines.

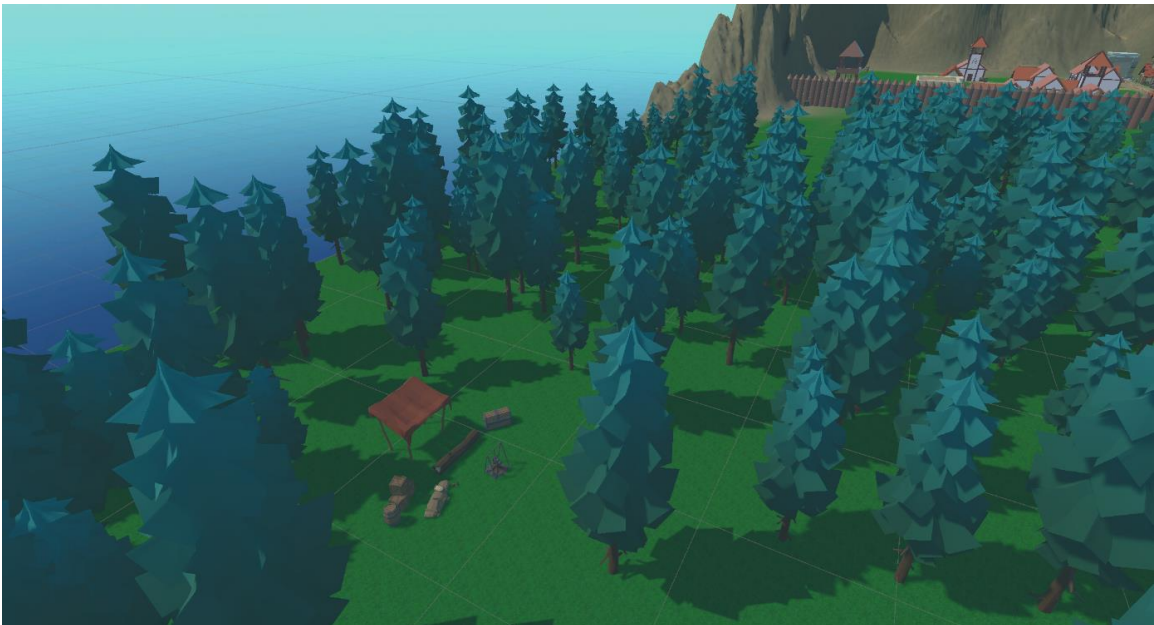


Figure 3.16 Layout of the Lost Woods.

3.2.10 Models and Animations

The 3D architectural models used in this project, such as the houses and pavements in the village, are mostly free or licensed assets downloaded from the Unity Asset Store, while a small proportion of them are created by the team using Blender.

For character models, the game also includes a lot of animations on the player control and NPCs. In a game, an animation can be thought of as a sequence of transformations of the “joints” or the “skeleton” on a model. In Unity, character models are usually rigged and skinned in the form of humanoid (see Figure 3.17), which can be easily mapped with animation clips. In other words, the same clips can work on different types of humanoid models.

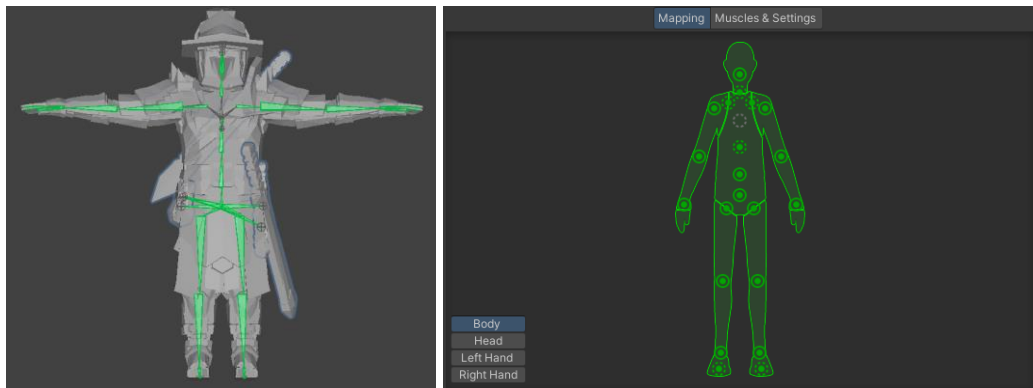


Figure 3.17 Humanoid avatar (left) and its mappings (right) in Unity Editor

In our game, these animations are achieved by first setting up scripts to control the transitions between different states with Booleans, as well as some finite state machine controllers for characters such as the player and NPCs. Then, the states in these controllers and the models will be mapped to specific animation clips, such that they can be played during the state. For example, when the player is in “Running” state, an animation of running will be played with the player model.

3.2.11 Music and Sound Effects

All music in this game is generated by a popular and free composition AI, Suno AI. For our game, six high-quality background instruments are composed in total, with different

styles and instruments. Also, some free-to-use sound effects are downloaded from online forums and mapped along with the animations.

3.3 Difficulties and Mitigations

Time constraint is always the major problem across the whole project. Given a period of 6 months and small team size of two, it is challenging to develop a good 3D game with various gameplay systems, features, and contents. To accelerate the development process, software engineering techniques like code-reuse, and pair programming have been applied. For example, regular meetings have been hosted for the team to share their progress. In the meeting, one member, known as the “driver”, focuses on developing new game features, while another one, known as the “navigator”, offers suggestions and focuses on debugging.

Another problem is the steep learning curve of game development. In this project, each teammate has taken up several roles, not only as a game developer, but also game designer, 3D modeler, and animator. There is no shortcut for acquiring such knowledge, but with hours of practice and reading, the team has eventually overcome most of the challenges encountered during development.

4 Conclusion

To conclude, this project aims to deliver a 3D fantasy action RPG world through a rework on a previous game project. All proposed gameplay systems and features, such as Inventory, Combat, and Dungeon Editor, have been tested and successfully implemented. In addition, game contents, including two game maps and a list of 3D architectural models, have been created and added. Extra features like Trading have also been added to the game.

4.1 Schedule

As planned, all the proposed objectives, including multiple gameplay systems and features, have been completed (see Table 4.1). In the remaining time, the focus will be on the testing and preparing for the coming project exhibition.

Table 4.1 Project schedule with current phase highlighted.

Milestones	Objectives	Deadline
Phase 1 (Inception)	Detailed project plan Project webpage	1 Oct 2023
Minimum Viable Project Development	Inventory system Combat system Leveling system Basic 3D models Basic map design	31 Dec 2023
First Presentation		8-12 Jan 2024
Phase 2 (Elaboration)	Preliminary implementation Detailed interim report	21 Jan 2024
Final Project Development	Dungeon Level Editor Character models Animations	15 Mar 2024
Game Testing and Adjustments		25 Mar 2024
Final Presentation		15-19 Apr 2024
Phase 3 (Construction)	Finalized implementation Final report	23 Apr 2024

Project Exhibition		26 April 2024
--------------------	--	---------------

4.2 Limitations

Game development involves a wide variety of soft and hard skills, which are generally specified by roles in real production. For instance, game developers specialize in coding and programming, while game designers create game content. In this project, all the teammates have taken on several roles that are unfamiliar to them.

At the beginning of the project, the project team is facing steep learning curves, and the development process is slower than expected, especially for 3D modelling. With extremely much time and effort, the team has managed to complete all the objectives for the project, as well as to include a few extra features to improve game experience.

References

[1] P. Mason, “In Search of the Self”, *Beyond Role and Play: tools, toys and theory for harnessing the imagination*. Helsinki: Ropecon ry, 2004, pp. 1-14.

[2] M. Hitchens and A. Drachen, “The Many Faces of Role-Playing Games,” *International Journal of Role-Playing*, no. 1, pp. 3–21, Dec. 2008, doi: <https://doi.org/10.33063/ijrp.vi1.185>.

[3] S. Read, “Gaming Is Booming and Is Expected to Keep growing. This Chart Tells You All You Need to Know,” *World Economic Forum*, Jul. 28, 2022. <https://www.weforum.org/agenda/2022/07/gaming-pandemic-lockdowns-pwc-growth/>

[4] N. Anantrasirichai and D. Bull, “Artificial Intelligence in the Creative industries: A Review,” *Artificial Intelligence Review*, vol. 55, no. 1, Jul. 2021, doi: <https://doi.org/10.1007/s10462-021-10039-7>.

[5] “The Road to Castle”, *GitHub*. https://github.com/pystander/ENGG1340_Gp106_RTC

[6] “Best practices for organizing your Unity project | Unity,” *Unity*. <https://unity.com/how-to/organizing-your-project>

Appendices

Appendix A: Creating GameItem in Unity Asset Menu with Attributes

```
[CreateAssetMenu(fileName = "New Equipment", menuName = "Road2Castle/Equipment")]
public class GameEquipment : GameItem{
    [Header("Equipment info")]
    public EquipmentSlotName slot;
    public WeaponType weaponType = WeaponType.NONE;
    public EquipmentEffect effect;

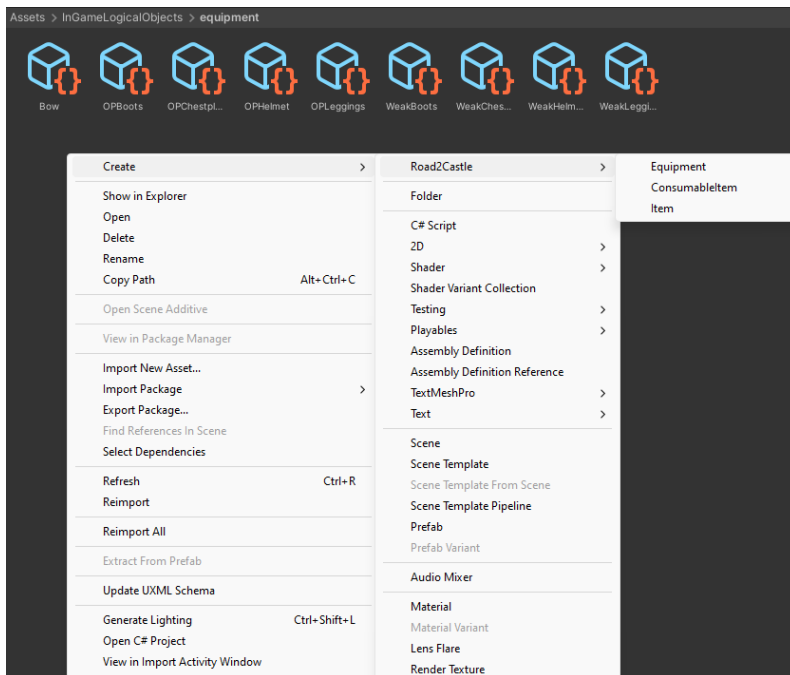
    public override bool use(GameEntity actor){
        return base.use(actor);
    }
}

public enum EquipmentSlotName{
    HEAD, CHEST, LEGS, FEET,

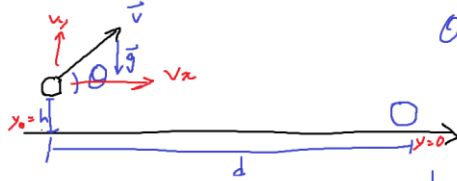
    WEAPON,
    SHIELD
}

public enum WeaponType{
    NONE, SWORD, BOW, STAFF
}

[System.Serializable]
public class EquipmentEffect{
    public float armorModifier = 1;
    public float damageModifier = 1;
}
```



Appendix B: Equation Derivations of 2D Projectile Motion



$$v_y = v \sin \theta$$

$$v_x = v \cos \theta$$

Type of Formula	Linear	Angular
Velocity	$v = \frac{\Delta s}{\Delta t}$	$\omega = \frac{\Delta \theta}{\Delta t}$
Acceleration	$a = \frac{\Delta v}{\Delta t}$	$\alpha = \frac{\Delta \omega}{\Delta t}$
Displacement	$s = v_i t + \frac{1}{2} a t^2$	$\theta = \omega_i t + \frac{1}{2} \alpha t^2$
Motion with time canceled out	$v_f^2 - v_i^2 = 2as$	$\omega_f^2 - \omega_i^2 = 2\alpha\theta$

$$s_y = y_0 + v_{y0} t + \frac{1}{2} a_y t^2$$

$$0 = h + v_y t - \frac{1}{2} a_y t^2$$

$$= h + v \sin \theta \left(\frac{d}{v \cos \theta} \right) - \frac{1}{2} a_y \left(\frac{d}{v \cos \theta} \right)^2$$

$$= h + \frac{d \sin \theta}{\cos \theta} - \frac{1}{2} a_y \left(\frac{d}{v \cos \theta} \right)^2$$

$$= h \cos^2 \theta + d \sin \theta \cos \theta - \frac{1}{2} a_y \frac{d^2}{v^2}$$

$$= h (t \sin^2 \theta) + d \sin \theta \cos \theta - a$$

$$= h - h \sin^2 \theta + d \sin \theta \cos \theta - a$$

$$= -h \sin^2 \theta + d \sin \theta \cos \theta + (h - a)$$

$$= -h \sin^2 \theta + d \frac{1}{2} \sin 2\theta + (h - a)$$

$$= \frac{1}{2} \cos 2\theta + \frac{d}{2} \sin 2\theta + \left(\frac{h}{2} - a \right)$$

$$= h \cos 2\theta + d \sin 2\theta + (h - 2a)$$

$$= \sqrt{h^2 + d^2} \cos(2\theta - \tan^{-1}(\frac{d}{h})) + (h - 2a)$$

$$x = v_{0x} t + \frac{1}{2} a_x t^2$$

$$t = \frac{d}{v \cos \theta}$$

$$A \sin(\alpha) + B \cos(\alpha) = R \cos(\alpha - \phi)$$

$$\phi = \tan^{-1}(\frac{A}{B})$$

$$R = \sqrt{A^2 + B^2}$$

$$\frac{2a - h}{\sqrt{h^2 + d^2}} = \cos(2\theta - \tan^{-1}(\frac{d}{h}))$$

$$\cos^{-1}(\frac{2a - h}{\sqrt{h^2 + d^2}}) = 2\theta - \tan^{-1}(\frac{d}{h})$$

$$a = \frac{1}{2} a_y \frac{d^2}{v^2}$$

$$\cos^{-1}(\frac{a_y \frac{d^2}{2v^2} - h}{\sqrt{h^2 + d^2}}) - \tan^{-1}(\frac{d}{h}) = 2\theta$$

Appendix C: Game Entities Configuration JSON File

```
{
  "assetPaths": {
    "Bow": "InGameLogicalObjects/equipment/Bow",
    "Pickaxe": "InGameLogicalObjects/equipment/Pickaxe",

    "OPHelmet": "InGameLogicalObjects/equipment/OPHelmet",
    "OPChestplate": "InGameLogicalObjects/equipment/OPChestplate",
    "OPLeggings": "InGameLogicalObjects/equipment/OPLeggings",
    "OPBoots": "InGameLogicalObjects/equipment/OPBoots",
    "WeakHelmet": "InGameLogicalObjects/equipment/WeakHelmet",
    "WeakChestplate": "InGameLogicalObjects/equipment/WeakChestplate",
    "WeakLeggings": "InGameLogicalObjects/equipment/WeakLeggings",
    "WeakBoots": "InGameLogicalObjects/equipment/WeakBoots",

    "OPItem": "InGameLogicalObjects/items/OPItem",
    "HealthPotion": "InGameLogicalObjects/items/HealthPotion",
    "StrengthPotion": "InGameLogicalObjects/items/StrengthPotion"
  },
  "entities": {
    "Enemy": {
      "xpReward": {
        "xp": 200
      },
      "rewardStat": {
        "gold": 300
      },
      "drops": [
        {"id": "WeakHelmet", "type": "e", "prob": 0.1},
        {"id": "WeakChestplate", "type": "e", "prob": 0.1},
        {"id": "WeakLeggings", "type": "e", "prob": 0.1},
        {"id": "WeakBoots", "type": "e", "prob": 0.1},
        {"id": "Bow", "type": "e", "prob": 0.05}
      ],
      "equipment": ["WeakHelmet"]
    },
    "WorldTreasureChest": {
      "xpReward": {
        "xp": 100
      },
      "rewardStat": {
        "gold": 100
      },
      "drops": [
        {"id": "OPHelmet", "type": "e", "prob": 0.01},
        {"id": "OPChestplate", "type": "e", "prob": 0.01},
        {"id": "OPLeggings", "type": "e", "prob": 0.01},
        {"id": "OPBoots", "type": "e", "prob": 0.01},
        {"id": "OPItem", "type": "c", "prob": 0.01},
        {"id": "Bow", "type": "e", "prob": 0.05},
        {"id": "HealthPotion", "type": "c", "prob": 0.2},
        {"id": "StrengthPotion", "type": "c", "prob": 0.1}
      ]
    },
    "Bob": {
      "shopItems": [
        {"id": "HealthPotion", "type": "c"},
        {"id": "StrengthPotion", "type": "c"}
      ]
    },
    "John": {
      "shopItems": [
        {"id": "Pickaxe", "type": "e"}
      ]
    }
  }
}
```


Appendix D: Creating A New Quest with the Developed Quest API

```
// SampleStory: AreaCondition -> Welcome msg -> Why me? -> KillCondition -> Done
Storyline sampleStory = new Storyline("Sample Story");
StoryTask task0 = new StoryTask("Walk to Location");
task0.rewards.xp = 200;
task0.rewards.items.addItem(weakHelmet);
task0.rewards.currencyStat.money = 9999;
task0.conditions.Add(new AreaCondition(new Vector3(6.63f,0.13f,3.65f), 2));

StoryTask task1 = new StoryTask("Welcome msg");
task1.addDialogue(new Dialogue(npc1, "Welcome to The Road to Castle 3D.));
task1.addDialogue(new Dialogue(npc1, "What you are looking at is a sample dialogue.));
task1.addDialogue(new Dialogue(npc1, "The dialogue will subject to change.));
task1.addDialogue(new Dialogue(npc1, "Please find Npc2 for more information.));
task1.addDialogue(new Dialogue(npc2, "Hey, I'm here!!!!"), true);

StoryTask task2 = new StoryTask("Why me?");
task2.addDialogue(new Dialogue(npc2, "Welcome to The Road to Castle 3D again.));
//...
StoryTask task3 = new StoryTask("Revenge");
task3.conditions.Add(new KillCondition(1));
task3.addDialogue(new Dialogue(npc2, "I want you to beat one of those monsters up."), true);

StoryTask task4 = new StoryTask("Thank you");
task4.addDialogue(new Dialogue(npc2, "Thank you for testing out the quest system."), true);

sampleStory.addTask(task0);
//...
stories.Add(sampleStory);
```

Appendix E: All Subpanels in Game Panel





