3D for Ancient Rome! (Knowledge Dissemination)

Demo video : https://youtu.be/xUnqanETWmA || Introcution video: https://youtu.be/-ndr5yL76Ug

Rui Shan
School of Computing Science
University of Newcastle
Newcastle Upon Tyne,UK
R.Shan3@ncl.ac.uk

ABSTRACT

Ancient Rome was a fascinating time, famous for its magnificent architecture and its many well-known historical figures,one of which is the Roman Fort Vindolanda, found in Northumbrian Countryside.This project aims to recreate the fortress of Vindelanda through the Unity engine, allowing players to experience the historical story of the ancient Roman period in the form of a game.

Players will talk to a variety of NPCs throughout the game, discover their stories and complete their quests.Some of the tasks are presented to the player in a format like solving riddles, requiring the player to pay more attention to the text.The game also provides players with knowledge of the Roman period in the form of quizzes and simple battles to experience the culture of ancient Rome.In addition, a number of famous historical figures make their appearance in the game to drive the plot forward.Once the player has completed all the tasks, they can win and end this fantastic journey through ancient Rome.

KEYWORDS

Roman Fort Vindolanda;Unity Engine;Quests;Quizzes;Battle;Famous historical figures

1 INTRODUCTION

1.1 Project Background

Ancient Rome, a civilisation that emerged from the early 9th century BC in the middle of the Italian peninsula, went through three successive phases: the Roman Kingship (753 - 509 BC), the Roman Republic (509 - 27 BC) and the Roman Empire (27 - 476/1453 BC).The 1st-2nd centuries AD were the most glorious period in Roman history[1],while the history of Vindolanda Fort began.

Vindolanda lies just to the south of the curtain wall of Hadrian's Wall and has a very different 'feel' to other sites along the Wall. It lies upon the first Roman frontier in the north – The Stanegate Road and in a stunning landscape which lets your imagination really connect with its past[2].

The earliest Roman forts at Vindolanda were built of wood and turf[3].But shortly after Hadrian's Wall was built, a stone fort was built at Vindolanda, possibly for the 2nd Cohort of Nervians[4].From 208 to 211 AD, there was a major rebellion against Rome in Britain, and the Emperor Septimius Severus led an army to Britain to cope with it personally. The old stone fort was demolished, and replaced by an unconventional set of army buildings on the west, and an unusual array of many round stone huts where the old fort had been. Some of these circular huts are visible by the north and the southwest walls of the final stone fort[5].

In the 1930s, the house at Chesterholm where the museum is now located was purchased by archaeologist Eric Birley, who was interested in excavating the site[6].Recent excavations have been accompanied by new archaeological methodologies. 3-D imaging has been used to investigate the use of an ox cranium in target practice[7].

1.2 Unity Engine

Unity is the platform for real-time 3D interactive content creation and operation. All creators, including game developers, artists, architects, automotive designers, film and television, use Unity to bring their ideas to life[8]. The Unity platform offers a complete suite of software solutions for creating, operating and realising any real-time interactive 2D and 3D content on mobile, tablet, PC, console, augmented reality and virtual reality devices[9].

The flexibility of the Unity engine allows developers to create and optimise content for over 20 platforms,including the PC platform to be used for this project.Unity can import a wide range of models and functional plugins from the unity store, the official unity assets shop, via the package manager.Developers can use these models to build 3D or 2D scenes and use these extended plugins to implement some interesting features such as quest and dialogue systems and scene interaction systems.

Unity can also provide developers with different design patterns to meet different needs, for example, the unity singleton pattern is commonly used in the design of the manager class to achieve uniqueness and persistence of objects in scene switching[10].This game design pattern is useful for creating a inventory system or switching between the main menu and game scenes.

1.3 Design idea

Interactive features are needed to allow players to experience the ancient Roman era and to be able to explore the Vindolanda region freely.These functions can be as simple as talking to the characters that appear in the game, either picking up or dropping items, and investigating hidden mechanisms.Regardless of the form of interaction, they all need to offer the player free

movement.There are also certain cues that we needed to give players when they encountered interactable objects.

Furthermore, as a way of guiding the player to a successful victory in the game, we needed a clear and linear main quest line - in other words, to make the game narrative.This seems to contradict what is said in the previous paragraph about giving the player freedom of movement, so we need to consider both situations where the player is in the main questline and situations where they are not, which means that multiple interaction events need to be designed for various circumstances.Quest guidance feature seems to solve this problem well, such as a compass system, or a quest journal system, plus having the camera traverse the entire quest scene.Nonetheless, notwithstanding the help of the quest guidance system, we still have to consider unusual situations, as when hidden mechanisms cannot be opened until the player has been given the appropriate hints, and so on.Anyway, we wish to guide the player through the main quest line, experiencing the local culture while travelling through ancient Rome, and eventually achieving victory.

To the above interactive and narrative sections, some new gameplay can be introduced.A simple question and answer system is used to enhance the educational aspect of the game, full of historical questions related to ancient Rome.Prior to that, gamers can lounge around playing a mini game of milking goats similar to ThunderFighter.For the highlight, there is a simple battle in which players have to beat the powerful gladiators in the arena using some fundamental attacks, defences or rolling and jumping moves.

Besides gameplay-related content, our hope is that the overall art style of the game will be as uniform as possible.It is well known that the ancient Roman period was marked by many magnificent buildings, such as the Colosseum and Constantine's triumphal arch.They are all distinctive and generally feature thick masonry walls, semi-circular arches, floor-by-floor projecting door frame decoration and cross-vaulted roof structures[11].It is therefore worthwhile to use a lowpoly style model to construct a portion of the scene, which will save a lot of rendering time and physics calculations while restoring the architectural style of the ancient Roman period.It is also a good idea to use some of the Unity engine's special modules to increase the graphical impact of the game, notably the particle system to create flames alongside the post processing system to adjust the saturation of the screen.

2 RELATED WORK AND DESIGN

2.1 Project management and materials import

The Unity engine, which is primarily used in this project, provides developers with various fundamental project templates, subdivided into core templates, learning templates and example templates.Each template has unique features, for example the example templates often have some popular application features imported into the project beforehand, but the most common and basic of these templates is the core template, closer to what is often referred to as an 'empty project'.This project will be developed with core template, partly to better learn the process of the Unity engine development and partly to improve the customisability of the project.The version 2020.3.30f1c1 of the Unity Editor has been chosen for this project to better implement some of its gameplay and features.

The Unity engine also provides developers with a extensive asset store which contains a wide range of character or building models, background music and sound effects, UI icons or even some feature integrations.We may purchase our desired materials from the Asset Store and import them into our project via the Package Manager component in the Unity engine.

Apart from the materials mentioned above, our project may also require the implementation of some character animations, which would be more complicated if we only used the animation system included with the Unity engine.To simplify our development, we are going to work with Mixamo.com to adapt character animations to our existing models.Mixamo.com is a free animation library from Adobe that provides developers with character models while also automatically rigging the character skeletal animations for the character models they upload[12].Mixamo.com also supports downloading characters and animations in multiple formats, ready to use in motion graphics, video games, film, or illustration.For example, it can export FBX type files for usage in the Unity engine.Whether we just need one animation or a hundred, export optimizations will keep our projects light and efficient.

Last but not least, Github is used to assist in the development of the project in order to allow for detailed sub-panel work as well as effective version control of the project.GitHub is a hosting platform for open source and private software projects, named GitHub because it only supports Git as the only repository format for hosting[13].Github provides support for Unity projects and makes it relatively easy to upload or cancel changes to online projects to achieve version control.

2.2 Scene management

The aim of this project is to simulate the ancient Roman period through a game, so a large number of building models and character models are needed to form the base units of the game world.With countless game objects expected to appear in our scenes, we need some methods of scene management that are efficient and do not take up too many in-game resources.

The most basic approach is to have common parent objects for game objects of the same type in the hierarchy window.This method is the most popular in unity game development, making it efficient to construct scenes and allowing us to find objects in the game more easily in the future.

Although the use of sub-parent objects is simple and works well, it still has some problems, such as the inability to effectively control the number of game objects in the scene, resulting in unnecessary rendering and wasteful physics calculations.Thus this project will take another approach to scene control, where different scenes are switched via scripts with Unity's own SceneManager class.

Prior to Unity version 5.3, developers typically used the

Application class to implement scene switching. However, as this class was redundant and did not have much functionality related to scene control, Unity introduced the SceneManager class after version 5.3, which is dedicated to controlling scene switching[14]. We can use the SceneManager class to achieve effective scene control, dividing the game's scene into different levels or even different areas, thus reducing the number of rendering and physics calculations performed by the project at the same time, increasing the efficiency of the project's operation and completing the optimisation of the project as well.

An important part of the scene is the skybox, which is a way of creating a background that makes the video game level appear larger than it actually is[15].In the Unity engine, a skybox is a special material that is rendered by a special shader, normally by sampling the texture in the form of a cubemap.It is worth noting that if there are separate scenes with varying skyboxes, we should prepare them in advance and generate their respective Light maps in the Rendering component of the Unity engine, otherwise they will not display the correct lighting effects when switching scenes

2.3 UI design

The user interface (UI), also known as the Human-MachineInterface, is a platform for communication between humans and machines, and is the medium through which information is input and output between users and machines. The graphical user interface has the characteristics of direct control, and it is an important part of various screen products[16].

The November 2014 release of Unity 4.6 introduced a new UI system, the predecessor to the UGUI we use today.The power of the new component-based UI framework and visualisation tools is that they allow you to easily build GUIs in games and applications[17].

On version 2020.3.30f1c1 of the Unity engine which is the one utilized for this project, the UI part is made up of a Canvas component and several functional controls.Under this premise, the Canvas component represents an abstract flat area where all UI elements are placed and rendered, and all UI elements must be a child objects of the game object that owns the canvas component.Other functional components fall into the main categories of interactable UI components, such as Button and Toggle, and non-interactable (or Visible) UI components, such as Text and Image.The main elements of this project regarding UI will be described separately in the following paragraphs:

2.3.1 Interactable UI components

UI's interactable components are the cornerstone of the entire UI section,which is one of the most vital aspects of UI design as it relates to the logic of the game and the player's overall play experience.The interactable UI components for this project are primarily the content in the main menu screen and the pause menu screen.

1) Main menu:

In the main menu scene, we will create a UI Canvas called the Main Menu screen, which will focus on guiding the player into and out of the game, as well as modifying some general game settings about the game quality, resolution and whether to enable full screen mode(It is worth mentioning that the content of Settings will affect the game' s main scene, hence we need to provide an interface to Settings once the player has entered the game as well).These functions will be implemented largely by the Button component of the UI, which is an essential element of the UGUI and is usually deployed to receive clicks from users and respond to them.The Settings screen also contains the Toggle component, a checkbox that allows the user to turn an option on or off to enable full screen mode; the Slider component, a UI control that allows the user to adjust the volume by dragging the mouse to select a value from a predefined range; and the Dropdown component, a UI control that displays the options control (when clicked, this control opens a list of options for selecting new ones. Once the new option is selected, the list closes again and the control will display the newly selected option. The list will also close if the user clicks on the control itself or any other location within the canvas) to enable the setting of the game's image quality and resolution.After constructing the framework for the main menu UI, we will connect these components to their corresponding functions, the implementation of which will be described later in the Project Approach section.

2) Pause Menu:

Upon entering the main game scene, we hope that the player may pause the game at any time during the game and choose to resume the game, back to the main menu or restart the game from the pause menu. These features are similar to the enter or exit game features in the main menu , all of which will be implemented through the Button component.Also as stated above, we wish to provide an interface to the Settings screen in the Pause menu, whose content is consistent with that of the Settings screen in the Main Menu.On analysis, we learned that there are two major approaches to accomplish this requirement: one is to save the content of the Settings screen as a Prefab and place a Settings Prefab waiting to be enabled in each scene (the main menu scene and the game scene); the other is to place only one Settings screen in the main menu and preserve it by means of coding after switching scenes thereby allowing both scenes to share the same Settings screen.Whichever approach is taken, the Settings screen needs to be separated from the pause menu or the main menu, which means that the Settings screen should use a separate Canvas

2.3.2 Non-interactable UI components(Visible UI components)

Visible UI components, as the name suggests, are UI components that cannot be interacted with by the player and are typically used to guide the player in understanding the gameplay or to decorate the user interface.These components are simpler and more common than interactable UI components, for example the Panel control, whose main function is to act as a 'container'. A Panel control makes it easier to move and handle a group of controls as a whole, which is why a feature-rich UI interface often uses multiple Panel controls. This project focuses on the generation of dialog boxes for the dialog system, the inventory system's backpack interface and the journal interface for the quest system via the Panel control.Moreover, the Image component and Text

component will be used extensively in the UI part of the project, to guide the player or as an integrated part of other gameplay interfaces.Notably, the Image component of UGUI generally works with the Sprite2D format of the Unity system, which is why we will need to change it to Sprite2D format after importing your desired image resource.

2.3.3 Integrated application

The combination of interactable UI components and visible UI components in various UI interfaces is an essential part in the UI part of this project.This so-called integrated application can be as mentioned above, dividing the entire UI system into different panels, using a Panel component as the parent object of these panels and other functional components as sub-objects of this Panel component, enabling each panel to be independent of the other. This model solves most of the UI interface design issues and also applies on a large scale in this project.

What is worth mentioning is that some of the UI interfaces need to be updated dynamically because they are linked to other gameplay. For example, the Inventory system should update the UI interface in real time after the player has picked up an object, and the same goes for the Quest system.To make the UI interface more neat and beautiful, we can use the Grid Layout Group component in UGUI.GridLayoutGroup is a layout script encapsulated by UGUI, which is relatively simple to use and suitable for layouts with not a large number of child nodes and a single structure, such as the backpack field in the Inventory system and the journal field in the Quest system, the detailed implementation of which will be introduced in Project Approach.

Since this project supports the player to change the resolution in the game, to make sure that the layout of the grid does not have display issues at different resolutions, we can also take advantage of another component called Content Size Fitter.The Content Size Fitter is used as a layout controller to control the size of its own layout elements. The size is determined by the minimum or preferred size provided by the layout element component on the game object. Such layout elements can be "image" or "text" components, layout groups or "layout element" components.

2.4 Main scene construction

Until now, there are only a few game objects in the main scene about the UI part of the game, in which case we will start to construct our main scene.The first and most fundamental step is that the game needs a large enough and historically fitting environment, signalling that some models of ancient Roman-style buildings will be placed in the game space, for instance the Colosseum and the temples.A new problem arises, however, when we import the building scale models, which should not be placed in the air, but rather on a terrain object.Simplest terrain may be represented by a flat plane object or a cube object with as small a scale as possible in the y-direction, after placing them with a land-like material.However, the resulting terrain looks very unrealistic, as it is merely a flat object with no protrusions or depressions, while the surface material cannot be flexibly changed to represent multiple types of ground, such as roads or grass.The Unity engine fortunately provides developers with a unique type of object called Terrain, which allows them to customise the terrain to achieve a hightmap-like effect.

Terrain is a game object provided by Unity3D for drawing terrain. Apart from being used to create height maps, developers can draw mountains, rivers and seas, ponds, grass and trees, and much more on it.Terrain objects are made up of two major components, a Terrain Settings component for drawing terrain and a Terrain collider component for physically calculating and storing terrain settings.

Terrain component comprises six principal function menus: Raise/Lower Terrain, for raising or lowering the terrain in some type of shape to draw a heightmap of the terrain; Paint Height, for height flattening, which can be used to map platforms, basins and ponds on hills; Smooth Height, for height smoothing, which smooths out sharp parts of the terrain; Paint Picture, for painting ground and mountain mapping; Paint Trees, which provides a model interface for developers to import prefabs of trees, which can be painted on the ground in the same way as the terrain, with parameters such as tree size and height adjusted in advance; Paint Details, similar to Paint Trees, allows you to import detail maps of grass or flowers and paint them onto the terrain.Furthermore, Terrain component has a Terrain Settings function menu where developers can do more detailed customisation of the terrain, regarding for example the rendering details of objects in the terrain and lightmap details.

Terrain collider components present a special interface for saving details like heightmap, textures and trees from the Terrain component in Terrain Data format and accordingly endowing the terrain with the appropriate colliders for physics calculations.Developers can choose whether to turn on colliders for trees or to attach previously prepared physics materials to the terrain.

Now that the terrain has been created, we are free to place the building figures as we wish. A huge fortress-type construction will be imported into the scene to fit the historical context, and then we will also need to place some of the mechanisms and architecture that may be used during the game.

2.5 Game character design

Another essential part of the game, apart from the constructions and other objects in the scene, are the characters.The game characters in this project can be broadly divided into two categories: player characters and NPC characters, where the NPC characters can be subdivided into friendly NPCs, neutral NPCs and hostile NPCs.The design concepts and ideas for each type of character are outlined below:

2.5.1 Player

The player character is the main medium that gamers can use to interact with the game world , as well as being an important piece of all game characters.The player character has many functional components, such as behavioural components and dialogue-triggering components, which, in order to facilitate development work, as well as the maintenance and updating of the various

functions, will be divided into several sections, whose content will be briefly described below:

1) Player Movement Section:

Firstly, we would like the player character to be free to move, which requires a player movement module to perform the action.Unity Engine gives us a component called CharacterController, a component specifically designed to control the character (mostly movement related). Compared to using Transform or RigidBody directly, CharacterController works better, it has some of the important features of RigidBody but removes many of the physics effects so that situations such as die wear, sliding, being knocked away or displacing other objects may be avoided.A further advantage of using the CharacterController is that it is relatively straightforward to implement the movement up the stairs, which would become very problematic if we tried to do it in a purely physical way. It is actually possible for the developer to adjust the slope that the character can climb by modifying the Slope Limit parameter in the CharacterController component.Additionally, the CharacterController component holds the player's Collider component, a capsule with a customisable radius, height and centre, meaning that the character will no longer need to be coupled with additional physical components.After the character controller component has been added, the developer will be able to call the functions in the script to actually implement the various behaviours of the character, where normal movement and going up and down stairs will be relatively easy to realise, while jumping and rolling movements may require physical calculations to be done. The detailed application will be described in Project Approach.

2) First Person Camera Section

It is not enough for the player character to move, we still want a camera that moves with the player character.There are two main kinds of following cameras, third person and first person, but for this project we chose to adopt the first person camera to achieve the effect of camera following.We generally bind the main camera to the head of the character so that it matches the eye of the character model, and then add scripts to the camera that control the camera's pitch and yaw to follow the rotation of the mouse to achieve a first-person follow camera.Due to the fact that this project works with the Cinemachine component (a camera intergration provided by the Unity engine, which will be specifically described later), the main camera will act as the virtual camera brain, which means that we need an extra virtual camera dedicated as a first-person follow camera.Having done all the configuration, we now need to combine the finished character movement section with the first-person camera section in such a way that the character's movement is always referenced by the orientation of the camera.We also need to send a ray of length 2.0 from the centre of the camera to detect if the object in front of the player character is interactable.

3) Player Animation Section

This project is intended to allow the player character to animate movements such as walking, running, jumping, falling and tumbling, as well as simple attacks and defensive motions in certain situations.To achieve the effects described above, we take advantage of a component that the Unity engine includes for developers entitled Animator.Animator is an animation state machine. As its name suggests, developers can attach multiple animation states to the Animator, either as a single animation clip or as a BlendTree composed of multiple animation clips, with corresponding Transition components and transition conditions(the parameters available from the Animator component which are of various types, such as float, bool and Trigger)between these animation states.Furthermore, the Animator component entails an animation hierarchy called Layers, which means that we can mix animation effects by putting separate states on various animation layers, taking into account the weighting of these Layers, as well as their LayerMask (the component associated with human rigging animation that determines which riggers the animation clips of that animation layer are to be applied to), in other words, by performing animation clips from multiple animation states simultaneously.A typical example is where we wish the player character to be able to perform an attack while walking or running, then we need to create two separate animation layers, a Base Layer to perform the walking or running animation and an Attack Layer to perform the attack animation only on the character's hands, before implementing real-time changes to the weights of these two layers in the script. One last point of notice is that, as we mentioned in the Introduction section, we will make reference to some of the character animations from Mixamo.com, so the character model we use must have an Avatar component to represent the riggers of the character and import this Avatar component in the Animator component for application to the animation.

4) Player Input Section

The prime role of the player input section is to listen to the player's keyboard or mouse events and transform them into signals that can be used by the player movement section and the player animation section.This section will be implemented by a specific script that first receives a predefined input command, such as WASD on the keyboard for movement, and then converts it into a public variable of type bool or float as an output signal. These signals may be simply divided into Pressing signals, which are used to output continuously executed actions such as walking or running, and Trigger signals, that are designed to output actions such as jumping and tumbling which run only once in a period of time.These output signals will be received by other sections and subsequently used to perform particular actions or operations.It can be seen that the player input section can be thought of as the brain of the entire player character, responsible for coordinating the player character's behaviour and animations.

2.5.2 NPCs

NPC is an abbreviation for non-player character, a type of character in a game that refers to a game character in a video game that is not manipulated by a real player, a concept that first originated in single-player games and has since been applied to other areas of gaming[18].NPCs are widely used in all types of games , and are crucial ingredients in role-playing games (RPGs), and can be categorised broadly according to their function as

friendly, neutral or hostile.

1) Friendly NPCs and Neutral NPCs:

Friendly NPCs are commonly referred to as game characters who do not, in general, become hostile or perform aggressive acts towards players.Friendly NPCs will be put on a large scale in the main quest line as the key quest giver during this project, which will include some famous historical figures from the ancient Roman period as well.Neutral NPCs are similar to Friendly NPCs in that they remain friendly to the player when the player does not act aggressively towards them. Neutral NPCs will be placed outside the main quest line to guide the player through quests.Whether they are friendly or neutral, or even player characters, they all share a common property that is a key one in the dialogue system of this project, which will determine whether a character or object is capable of being interacted with by the player and triggering some events, and which we have named the Actor property.The Actor property is a list of important properties in the dialogue system, including a bool type variable for whether it is the player character, the character name, the character portrait and the distance to trigger the event.The player character will be given a unique player type Actor, other NPCs, if there is a need for conversation, must be given a non-player type Actor property so that the player can trigger some dialogue events by entering the range of triggered events (an invisible sphere with a settable radius centred on the character) and clicking on a pre-set interaction key (the default for this project is the E key on the keyboard) (more on Events will follow in the Gameplay section).We may add the Actor property to Friendly NPCs enabling them to post conversations with players and issue key quests, or we may also do the same for Neutral NPCs so that they will give players some quest hints.

2) Hostile NPCs:

A hostile NPC is a character who remains hostile to the player and will actively attack them, which will not be frequently seen and only appears in a specific environment, as the project is not dominated by combat.Hostile NPCs are quite distinct from the other two types of NPCs in that they require a more detailed AI construction and they do not require the Actor property to engage in dialogue with the player.This project will include the Blaze AI System, a hostile AI continuous integration based on the Unity engine[19], to perform hostile NPC attacks.The following is a brief description of what this continuous integration entails:

Blaze AI is a fast, highly-customizable and a powerhouse universal enemy AI engine. If we want enemy AIs in our project no matter the genre, Blaze will build any gameobject of our choice and make it smart, realistic and challenging for game.It is been inspired by several AI systems of many modern games making Blaze a powerhouse in it’s features, functionality and workflow. What’s more,it supports many sub-systems such as integrations with audio, animations, distractions system, attack, patrolling in different states, visions, emotions, local avoidance, reactions and a whole lot more.

Firstly, an essential part of what makes up the Blaze AI System is pathfinding, which is a unique way of pathfinding based on the Unity engine's Navmesh system, so when we attach Blaze AI-related components to an enemy, the Navmesh Agent component is automatically applied to it.Blaze AI System provides developers with interfaces to customise pathfinding details, for example we can increase or decrease the pathfinding frequency of the AI system by modifying the value of the Path Recalculation Rate.Obstacles are a notable element regarding pathfinding, and Blaze AI System allows developers to customise the tag or layer of obstacles to allow enemies automatically to avoid objects with such properties in their field of view when they are finding their way.As well as calculating a series of wayfinding points for the character using its own pathfinding algorithm, Blaze AI System also has an interface for developers to customise the waypoints, meaning they can have the character move along a designated route.

Concerning the state machine, Blaze AI consists of three major states: the Patrol state, the Alert state and the Attack state.The default Patrol state, the basic state when the NPC hasn’t seen any hostile or alert tag objec. When no conditions are triggered, the AI character will change from other states to Patrol state after a certain period of time; Alert state, mainly a state that is entered when an object containing a hostile tag or layer appears in the NPC's vision range, by further determining if there are players around, and then switching to another state.When a player is spotted, it changes to Attack status, and when no player is spotted for a certain amount of time, it changes to Patrol status; Attack state is only available when the player character is targeted by a hostile NPC. The hostile NPC will chase the player as the end of the pathfinding and will randomly play an imported attack animation when the distance to the player reaches a value set in advance by the developer. After finishing a round of attack action, the enemy will wait for a while before proceeding to the next round of pursuit and attack.

The switch between these states is closely related to a property called Vision, which is a Blaze AI System property that represents the enemy's field of view and consists of a number of hidden sectors, spheres and planes that can be customised in size or position to change the width of the enemy's vision, alert range, pursuit range, and so on.Additionally, the Vision property contains the tag and layer used by the AI to identify hostile creatures, which allows enemies to perform hostile acts only on game objects with the corresponding properties.

The main features of the Blaze AI System are described above, and the detailed implementation will be presented in Project Approach.

2.6 Gameplay

Gameplay is the core nature of a game, which defines the style of play and the exact way in which the game is played[20].To make this project more educational as well as interactive and narrative as described in the Introduction section, Gameplay section will focus on player interaction with the environment and characters, for instance picking up props, talking to NPCs and simple mini-games.

2.6.1 Inventory System

The Inventory system is a fairly common feature of the role-playing genre and is used to store the items collected by the player - in other words, an Inventory is in fact a collection of many Items.The Inventory system contains three main parts: the UI part for display, the structure part for storing Items, and the part for user interaction.

In the UI design section, the Inventory system's UI screen has been briefly mentioned, which consists of a Panel and a number of Buttons. The article also gives a general description of the Grid layout groups and Content size fitter components, which are used in the UI section of the Inventory system to create item fields.Each item bar is a special Button that the player can click on to view the item description and to display the discard button.

The Inventory system's principal function is to store items, which makes a suitable storage structure particularly essential.In this project a special class called ScriptableObjects will work to produce the Inventory system's backend storage structure.According to Unity API documentation,"ScriptableObject is a class you can derive from if you want to create objects that don't need to be attached to game objects". In other words, unlike Monobehaviour for objects, ScriptableObjects, although they are a type of object, can only contain data, so they do not participate in the actual game[21].The advantage of using ScriptableObjects to create items and Inventory systems is that we can customise their properties and take advantages of them more easily from other scripts.

In the player interaction part, we firstly associated the Inventory system with the Player Input System, enabling the player to open or close the Inventory screen by means of key presses, followed by the possibility for the player to perform the action of collecting or discarding items, which will be displayed on the Inventory screen in real-time.

2.6.2 Quest and Dialogue System

The Dialogue and Quest System is a significant part of the project, allowing players to interact with a variety of NPCs and objects to learn more about the game's backstory and to learn interesting historical knowledge as they take on and complete quests.The Dialogue and Quest system for this project is based on a continuous integration of Unity Engine called "Dialogue and Quest" which will be further customised and interlinked with other modules[22].

The dialogue part of the system is made up of three main components: a master control logic, actors who perform the dialogue and dialogue events.Master logic is a game object that has a special script attached to it. This special script determines which dialogue event should be carried out based on factors such as the conditions of the dialogue event and the distance between the actors.In contrast to the sole master logic, actor and dialogue events can be of many kinds. In Game Character Design section, the article has already introduced actor properties, whereas dialogue events are similar and a special property, the difference being that since dialogue events must depend on the game to exist, actor properties will be implemented with ScriptableObjects, while dialogue attributes are going to be attached to game objects in the form of scripts.For a single dialogue event, the trigger form, trigger conditions, who triggers it and whether or not the game is paused when it is triggered are all customisable.Besides this, it is possible to continue adding specific content under dialogue events, such as dialogue message and dialogue choice, or even some custom effects which can then be used to enable quests to be posted and completed.

The Quest section of the system shares the same master logic with the Dialogue section which is used to control a special property called Quest.Quest properties, like Actor properties, take a data structure of the ScriptableObjects type, which contains tangible features including the name and description of the quest.There are four states of the Quest property, not started, active, failed and completed, which serve as references to the quest categories and are used by the master logic to display the quest categories in the quest journal panel.The change of quest status is then achieved by the effect of dialogue events, which connects the dialogue system to the quest system.

2.6.3 the Compass System

To make it easier for the player to know where they are and to guide them successfully through the quest, many 3D games, the Assassin's Creed series for example, have a compass bar on the UI to show directions and mark the location of significant quest targets.This is an effective and viable solution for this project and will allow players to better explore Vindolanda Fort.

First of all, from a macro perspective, the compass system requires a set of definite four-way orientations; in other words, one needs to determine the north, south, east and west.A reference object can be prepared which contains sub-objects in all four directions. This reference object is then placed at the first level in the hierarchy, meaning that the reference object does not have any parent objects, and the Transform component of the reference object is also required to be reset to the centre of the world, so that we only obtain the direction of the player relative to the reference object to identify where the player is.

Secondly, when viewed from the player's perspective, the first person camera rotates with the mouse movement and this offset should also be represented on the compass bar. A simple replacement of the player entity with a first person camera could be used to capture information about the player's position. Besides, since the location of the quest target sometimes needs to be displayed on the compass bar, in case the player's facing is completely off the target, or put differently, when the first person camera's facing is more than 90 degrees offset from the target's position, it needs to be displayed on the corresponding side of the screens, not on the compass bar.

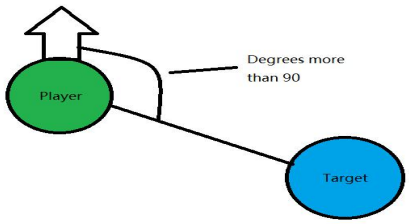

Figure 1: Angle between the player and the target

The last part of the compass system concerns the UI. We first need to create a compass bar, which consists of an outer frame, four orientation icons and a ticker. The orientation icons and ticker need to be displayed in the compass bar through a script that controls its presentation which, as mentioned above, needs to show the offset between the camera and the target. Additionally, a marker icon should be imported into the control script to allow the marker to be added to the quest target later.

2.6.4 Mini games-Quiz and Goat-milking game

This project will incorporate two UI based 2D puzzle games, a quiz on Roman knowledge and a goat milking game similar to Air Duel, created to enhance the educational and entertaining content of the game.These two games will both be scheduled in the main quest and players must achieve the necessary conditions in the mini-games to complete the quest.

Quiz game contains three main parts: UI, question library and answering logic. The UI part is similar to the Inventory system, with a Panel as the parent class, a few Button components to switch up and down the questions, and some Toggle components to represent the options.The question library section is a text file storing questions in a fixed format, where each line is a separate question, from left to right, with the question number, the question, the option and the correct option, all separated by a colon, in order to be read later in the answer logic.The answer logic part is done by a C# script which uses the Unity engine's file reading API to read the questions from the question library and make the question content, options, question number to be displayed in the corresponding UI component. Then the answer logic script will read the correct options from the question library and determine if the answer is correct based on the Toggle selected by the player. The player's correct percentage will be displayed on the UI, and the correct answer can also be displayed by clicking on the Tips button.To win, players will have to answer ten questions about the history of Rome with at least 50 percent accuracy. Players can answer these questions an unlimited number of times.

Milk the Goat is also a 2D mini-game based on UGUI, which is similar to Air Duel (a vertically scrolling shooter game released by Irem in 1990[23]), where the player controls an empty milk bucket to catch milk and dodge penalties. Both milk and penalties are randomly generated in the UI, with one point added for receiving milk and one point subtracted for receiving penalties, giving the player more than twenty points in thirty seconds to win.

2.6.5 Combat System

As previously stated, there are certain occasions in the game that need the player to engage in simple fights with hostile NPCs, which is why a fully fledged combat system is necessary.The combat system contains the AI of the hostile NPCs, the combat actions of the characters and some combat-related properties, such as the character's health bar and the damage dealt by attacks.Of these, the AI in question has been briefly described in the Character Design section, which is a hostile NPC state machine based on the Blaze AI.The two parts that remain will be described as follows:

Firstly, there is the content related to the combat action, which covers the animation of the player character's attacks and defences as well as the design of the weapon.The player's attack and defence animations, as previously mentioned, will be created with Mixamo.com and imported into the project as suitable character animations for use. It is worth noting that the attack and defence animations will be placed on a separate animation layer (the concept of which is described in the Character Design section), so that the character can properly animate the movement of the Base Layer while attacking or defending, and to do this we also need to attach a hands-only AvatorMask to the attack and defence animation layers and dynamically adjust the weights of the two animation layers in the script.Moreover, the weapon worn by the character needs to be bound to the character's corresponding hand. Before this, we should attach a suitable collider to the weapon to detect collisions with opposing characters and thus damage them.

Secondly, to give the battle a start and a finish, we need some properties concerning the combat, such as the character's health bar and the amount of damage the weapon can deal.We will have separate visual health bar HUD for the player character and the hostile NPC, where the player character has 1000 health and the hostile NPC has 1500 health.For weapon damage, a player can deal 100 points of damage per attack that hits a hostile NPC, while a hostile NPC's attack action will deal 200 points of damage if it hits a player who is not in a defensive state.When the player's character's health level reaches 0, a death animation will be played automatically and the player will be prompted to restart the game; when the blood level of an enemy NPC reaches 0, the corresponding death animation will also be played automatically and the player will be prompted to claim victory and may leave the battlefield.

2.7 Artwork

Unity engine has many features for developers to enhance graphical performance as well as convenient and practical continuous integration. Some of these will be exploited in this project to achieve the desired art effects.

2.7.1 Cinemachine and Timeline

Cinemachine, as previously mentioned, is a suite of tools for dynamic, smart, codeless cameras that let the best shots emerge based on scene composition and interaction, allowing developers to tune, iterate, experiment and create camera behaviors in real-time[24].Cinemachine can be easily installed into a project via the Unity Engine's Package Manager component. It allows for smooth camera switching by enabling or disabling the virtual camera, while developers can adjust the properties of the virtual camera to achieve the effect of the camera following or focusing on an object.Furthermore, Cinemachine contains various types of virtual cameras to achieve specific effects, including a unique Dolly track camera that works with the TimeLine(a feature that allows developers to visually edit music, play animations, show/hide objects, control particles, etc. along the time axis[25]) component of the Unity engine to achieve cinematic camera movements.

As a summary, in this project we will use Cinemachine to implement some simple camera switching and work with

TimeLine to achieve a surround view effect.

2.7.2 Universal Render Pipeline(URP) and Post Processing

Universal Render Pipeline (URP) is a premade scripted rendering pipeline, which is produced by Unity. URP provides a user-friendly workflow that allows developers to quickly and easily create optimised graphics across a range of platforms[26].Due to the fact that the materials for some models in certain scenes of this project are rendered with URP type shaders, URP needs to be installed into the project and switched to a specific URP Asset for rendering in the Graphics interface in Project Settings.However, as we are now using the URP for rendering, the original Post Processing functionality is no longer available and the URP-specific Post Processing component needs to be installed into the project with the Package Manager.Post Processing will be applied to enhance the game's graphical presentation in this project, for example using the Bloom effect to improve lighting realism and the Color Adjustments effect to rebalance the saturation of the screen.

2.7.3 Particle System

Unity Engine supports developers with a powerful and comprehensive particle system, which essentially works with a particle emitter that continuously shoots particles to achieve a variety of effects.Developer can change the operation of the particle emitter by adjusting various properties, and can also customize properties such as the shape and size of the particles.The project will use the Particle System to produce some simple flame and sparkle effects to decorate the environment and illuminate the scene at night.

2.7.4 Day and Night System

The Day and Night system is one of the most common weather systems in games which is often associated closely with skyboxes and lighting effects. As mentioned earlier, in the Unity engine, a skybox is a material that is rendered with a special shader while developers can also generate a Lightmap by placing a custom light source in the Lighting window.The Day and Night system is a way to append a time controller to the Unity engine's skybox shader so that parameters such as lightness or darkness and colour of the skybox change with the flow of time. Two directional light objects are also needed to represent the sun and the moon, and a script to control the movement of the directional light and parameters such as light intensity, colour , or shadow.A simple cloud layer is also included in the Day and Night system, which, similar to the skybox, is a plane object composed of a material rendered by a special shader, whose textures can also move with the flow of time.

3 PROJECT APPROACH-CODE AND IMPLEMENTATION

3.1 Project Preparation

Select the 3D template from the core templates in UnityHub to create the project template, then change the External Script Editor in the External Tool in Preferences to Visual Studio 2019 for subsequent scripting, and lastly log in to the Unity account and make sure the internet connection is available for downloading and importing materials from Package Manager.

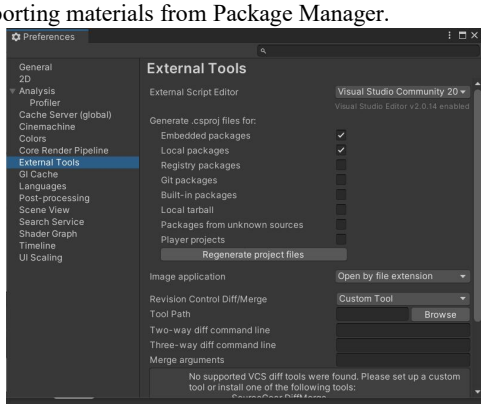

Figure 2: Preferences

Due to the fact that the SceneManager class will be used in the project, all Scene files will be previously added to the Open Scenes list in Build Settings.We can also customise some of the settings regarding Build and Run in this screen for later game releases.

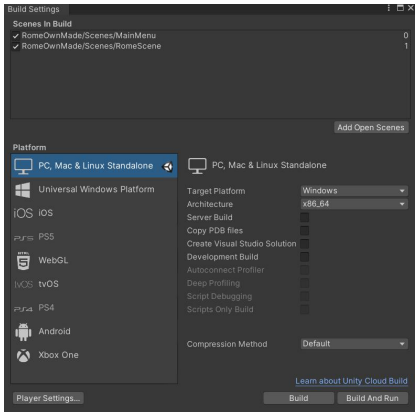

Figure 3: Build Settings

3.2 UI Implementation

3.2.1 UI of the Main Scene

As shown in Figure 5, the UI of the Main Scene contains four parts: a Title made by a Image component and three function buttons made by Button components. Clicking on the Quit button will exit the game which is achieved through the Application.Quit function.

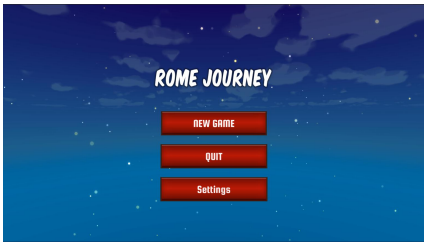

Figure 4: UI of the Main Scene

The New Game button will first display the Loading screen, which consists of a Text component and a Slider component, where the Text component will dynamically display different text depending on the Slider's value, while the Slider component's value will grow from 0 to 1 within 3 seconds.The fuction executed in the New Game button will call two IEnumerators for loading the scene and modifying the text in the text box accordingly, where the loading of the scene is done by the SceneManager.LoadSceneAsync function and temporarily turns off the activability of the new scene after it has been loaded, waiting for the Slider value to reach 1 before turning it on again. The reason for this is to prevent new scenes from loading too quickly and causing problems such as a splash screen on the Loading page.

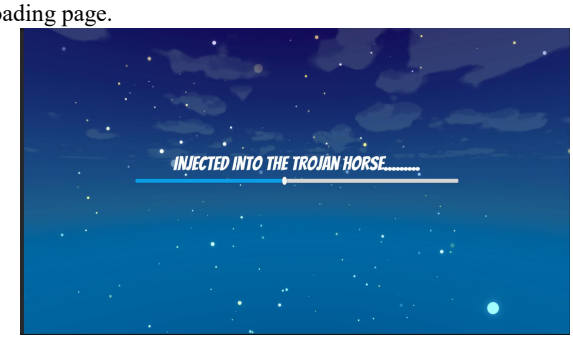

Figure 5: Loading Screen

Clicking on the Settings button will activate a new Canvas with a Quality option made with the Dropdown component, a Resolution option also made with the Dropdown component and a Full Screen option made with the Toggle component.

The content of the Quality Dropdown is obtained via the QualitySettings.names.Tolist fucntion; The content of the Resolution Dropdown is obtained via the Screen.resolutions fucntion; The value of Full Screen Toggle will affect the change of the Screen.fullScreen parameter.

Changes to the values within Dropdown or Toggle do not take effect immediately and it is necessary to click the Apply button in the bottom left corner to complete the application while saving the settings with PlayerPrefs (a local storage structure available in Unity engine). Clicking the Close button in the bottom right corner will close the Settings Canvas.

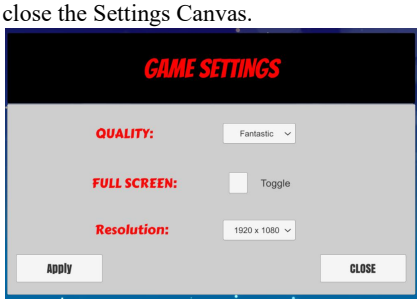

Figure 6:Game Settings in Main Scene

3.2.2 UI of the Game Scene

The Game Scene begins with a foreword that gives a general introduction to the background and instructions of the game. This section consists of a Text component for showing the text and a Button component for turning the page and actually entering the game. A script is attached to the Text component to control the output of the text in a typewriter style (dynamically controlling the maximum number of characters displayed in the text box), facilitating careful reading.

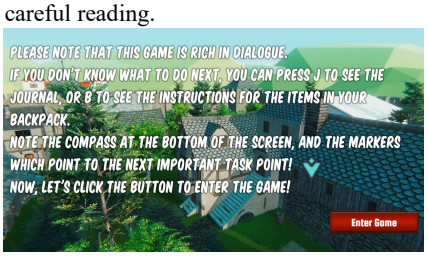

Figure 7:Forewards UI

Clicking the Enter Game button will first implement a cinematic camera switch by Cinemachine, which will switch to the first-person view camera after about two seconds. As shown in Figure 8, the UI in first-person view is roughly divided into three sections: the operation introduction, the compass and gameplay-related icons.

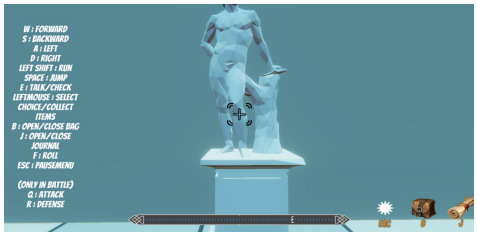

Figure 8:UI in first-person view

The operation introduction is a Text that cannot be selected by Raycast which introduces the basic operations in the game;The Compass implementation principle is described in the Related Work section where the UI part is mainly a combination of a few Image components, controlled by a script (as shown below);The icon section, similar to the operation introduction, is also unselectable by Raycast, the reason for which is to prevent them from conflicting with the UI-related gameplay.

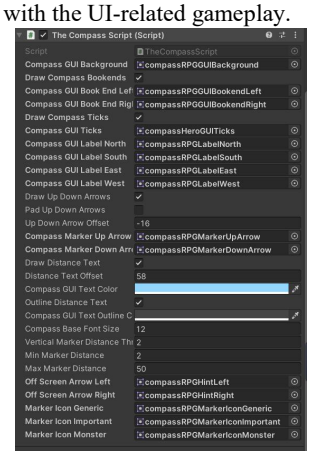

Figure 9:Compass Script

When the player presses the ESC key on the keyboard, the pause

menu is called up (as shown below) and the timeScale value is set to 0 to achieve the pause effect. The pause menu consists of a Panel component and four Button components, with the Panel component acting as the background for the pause menu.

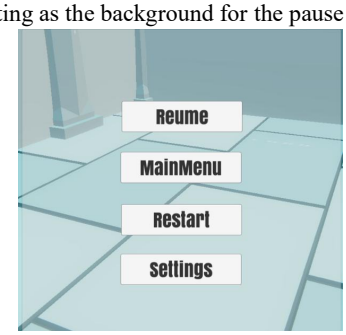

Figure 10:Pause Menu

Clicking the RESUME button will return to the game and unpause it;Clicking on the RESTART button will restart the game by reloading the game scene through the function of the SceneManager class;Clicking on the SETTINGS button will invoke the same Game Settings Canvas as in the main menu scene, with the exception that the Game Settings in the Game Scene contains an additional Slider component for adjusting the volume of the background music;Clicking on the MAINMENU button will return to the Main Menu scene. It works in the same way as the RESTART button, which loads the scene with the SceneManager class.

3.3 Scene Construction

First, create a new Terrain type object (the concept of Terrain has already been introduced in Related Work), import the corresponding model or texture in the Textures, Trees and Details windows, then select Rise or Lower Terrain to draw the heightmap. Once all this has been done, take care to turn on the tree collider in Terrain Collider to prevent the character from going through the trees.

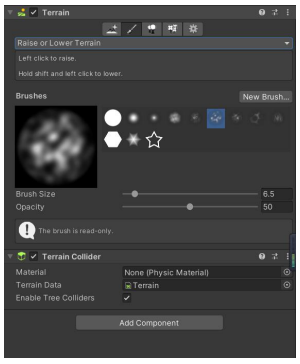

Figure 11:Terrain and Terrain Collider

Other construction models, whether in Prefab or FBX format, can be dragged directly into the scene and Unity will automatically place them on the ground. It is worth noting that the corresponding Collider must be appended to these construction models in order for the physics to work correctly.

3.4 Game Character Implementation

3.4.1 Player

1) Movement module

The first section of the player character is the movement module, which consists of a character controller and a script to control the movement of the character.As shown below, the Character Controller is a special component provided by Unity to control the movement of the character by adjusting the relevant parameters to determine the slope the player can climb, the size of the collider, and so forth.

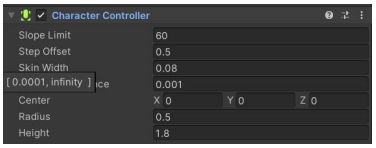

Figure 12:Character Controller

PlayerMovement script is used to control the player's movement and the state switching of animations. The player character's planar movement based on the x and z axis is implemented by the CharacterController.Move() function, which implements basic movement; GetAxis("Horizontal") and Input.GetAxis("Vertical") functions return the player's keyboard input in AD and WS respectively, and multiply them with the character's own forward and right vectors to get the actual direction of movement; The player's movement speed is divided into two types, running and walking, both in float form, and can be customized in advance.

```
float x = Input.GetAxis("Horizontal");
float z = Input.GetAxis("Vertical");

if (!isJumping)
    move = transform.right * x + transform.forward * z;
 //GroundMove
 controller.Move(move * speed * Time.deltaTime);
 if (move != Vector3.zero)
 {
     if (!walkAudio.isPlaying && !isJumping)
     {
         walkAudio.Play();
     }
 }
 else
 {
     walkAudio.Stop();
 }
```

Figure 13:Player Ground Move

Movement based on the y-axis requires a physics formula to calculate the velocity of movement in the y-direction, which can then be applied to the character using the CharacterController.Move() function; When the character is in free-fall, the velocity in the y-direction is the original velocity plus gravity multiplied by Time.deltaTime; When the character is jumping, the velocity in the y-direction is calculated via the formula v=sqrt(2gh) (g for gravity, h for jump height).It is also worth paying attention to the fact that since the character cannot double jump, a parameter that detects whether the character is on the ground is necessary to control the jumping action, and CharacterController.isGround is a good choice for this.

```
//JumpMove
velocity.y += gravity * Time.deltaTime;
controller.Move(velocity * Time.deltaTime);
if (isGround && velocity.y < 0)
{
    velocity.y = -2f;
}
if (Input.GetButtonDown("Jump") && isGround)
{
    velocity.y = Mathf.Sqrt(-2 * jumpHeight * gravity); //v=sqrt (2gh)
    walkAudio.Stop();
}
```

Figure 14:Player Jump Move

2) Input module

The second section of the player character is the Input module, which is implemented by a script. It is mainly used to capture the player's keyboard input and convert it into a signal in the form of a bool or float, thus controlling the state switching of the animation.There are two main types of signal: a pressing signal, the former can be obtained by listening to the corresponding keyboard event via the Input.GetKey() function, while the latter needs to return to the initial state after each corresponding keyboard event is transmitted to achieve a Trigger-like effect.

Figure 15:Player Input Script

3) Animator module

Animation state machines are described extensively in Related Work. One thing to note is that we will add scripts to call functions in StateMachineBehaviour such as OnStateEnter() and OnStateExit() to achieve special effects in some particular animation states, for example we will clear the Trigger type parameters for jumps and rolls in the initial state of each layer to avoid the problem of command residue caused by players pressing keys continuously within a short period of time.

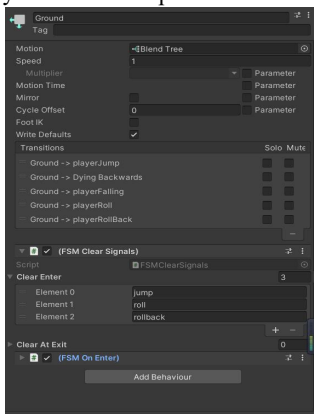

Figure 16:Script in animation state

Furthermore, in the combat system, in order to prevent the OnTriggerEnter() function from being called multiple times when a weapon comes into contact with an enemy and thus causes repeated damage, functions will be called at specific frames in the animation clip to control the switching on and off of the weapon's collider.

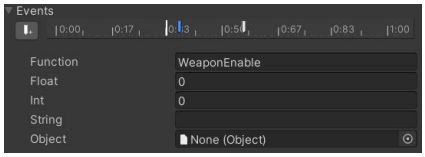

Figure 17:Event functions in animation clips

If we wish to adjust the animation in Unity, apart from modifying the Transform value in AnimatonClip, we can also use the IK system to make changes to the character's animation by calling the OnAnimatorIK() function, where we get the corresponding animation, and calling the SetBoneLocalRotation() function to tweak the specified bone animation.

```
private void OnAnimatorIK()
{
    if(anim.GetBool("defense"))
    {
        Transform leftArm = anim.GetBoneTransform(HumanBodyBones.LeftLowerArm);
        leftArm.localEulerAngles += a;
        anim.SetBoneLocalRotation(HumanBodyBones.LeftLowerArm, Quaternion.Euler(leftArm.localEulerAngles));
    }
}
```

Figure 18:IK functions

3.4.2 NPC

1) Friendly and Neutral NPCs

Friendly and Neutral NPCs require an appropriate capsule collider to be attached for detection by the player character's Raycast. An Actor script will also be included for these NPCs to store the relevant Actor properties (a ScriptableObjects type file, the concept of which has been described in Related Work) and the distance from which the character will trigger dialogue events.

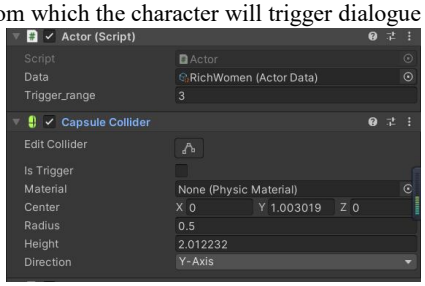

Figure 19:Actor Script and Capsule Collider

Some special NPCs will also mount an NPCController script, whose main role is to keep the NPC facing the player character when a dialogue is triggered with the transform.LookAt() function, as well as being marked as a quest target and presented on the compass when required.

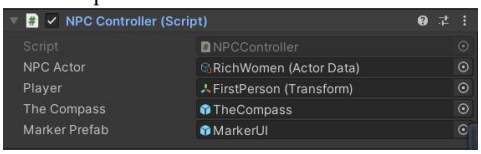

Figure 20:NPC Controller Script

2) Hostile NPC

The main component of the hostile NPCs is the Blaze AI script. As mentioned earlier, when the Blaze AI script is attached to a character, it will automatically assign a Capsule Collider, a CharacterController and a NavMesh Agent to the character by means of the RequireComponent function, which will be used for physics detection, character movement and pathfinding respectively.

```
[RequireComponent(typeof(Animator))]
[RequireComponent(typeof(NavMeshAgent))]
[RequireComponent(typeof(CharacterController))]
[RequireComponent(typeof(CapsuleCollider))]
[RequireComponent(typeof(NavMeshObstacle))]
```

Figure 21:RequireComponent in Blaze AI

Blaze AI is very powerful in its own right, yet its pathfinding and state machine functions are the main features which will be put into use in this project.As you can see below, under the General section there are a number of settings for pathfinding, including how often to update the waypoint and the starting point for pathfinding. The developer can also customise some of the settings of the obstacles for the AI character to avoid, including the physical layer where the obstacle is located and how far away it is from the obstacle to start avoiding it.

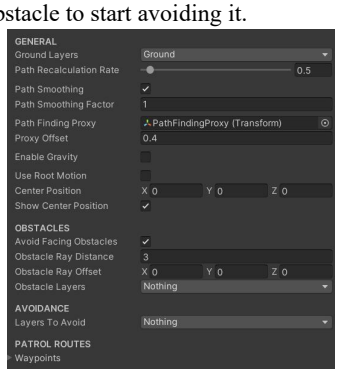

Figure 22:General panel in Blaze AI

The General section also contains a number of properties regarding the character's field of view, where the developer can customise the Tag and physical layer with the AI character's antagonist, as well as define the size and range of the AI's vision in its three states, and so on.

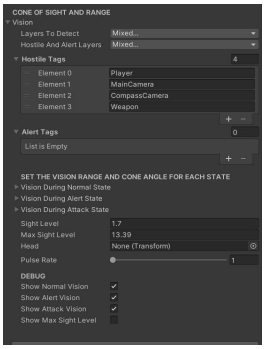

Figure 23:Sight Section in Blaze AI

The State panel contains the specific settings for the three states of the AI state machine, which are Normal State, Alert State and Attack State.Normal State is similar to Alert State in that the developer can set the speed at which the AI character moves in that state, the name and duration of the idle and movement animations for the clip. The difference is that in Normal State, the developer needs to set the interval when the AI character patrols, while in Alert State, the developer will have to set the conditions and time for the AI character to switch from Alert State back to Normal State.

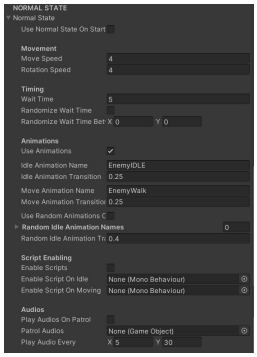

Figure 24:Normal State in Blaze AI

In addition to setting the same properties as the previous two states, the Attack state also allows for multiple attack animations to be imported and the duration of each to be set so that Blaze AI will randomly select one of those imported animations to play. Developers will also be asked to set parameters such as how far away from the pursuing target the AI character will perform an attack action, the time between attacks, and whether it needs to back off when it gets too close.

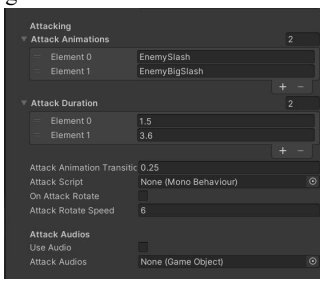

Figure 25:Attack State in Blaze AI

3.5 Gameplay Implementation

3.5.1 Inventory System

Inventory system, as detailed in Related Work, comprises two important ScriptableObjects type files, one for Inventory and the other for concrete items.Inventory is essentially a list of item types that holds the items the player has collected in the scene. Items contain a number of parameters that will later be presented in the Inventory UI.

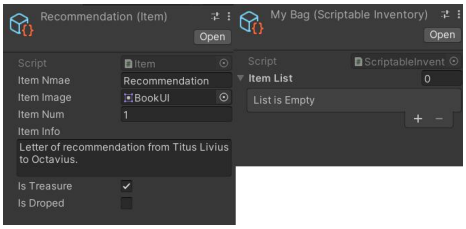

Figure 26:Inventory and Item ScriptableObjects

We need the player character to pick up some specific objects in the scene, which are those that have the Item attribute mounted on them and have the Tag "Item".A reliable implementation would be to emit a ray of length 2 and type float from the centre of the screen in first person camera for Raycast, and if the collided object meets all the requirements to be collectable, prompt the player on the UI that the item is ready to be collected and place the item in the MyBag's item list after the player clicks the left mouse button.

```
Ray ray = followCamera.ScreenPointToRay(new Vector3(Screen.width / 2, Screen.height / 2, 0));
RaycastHit hit;
if (Physics.Raycast(ray, out hit, RayDistance) && !PauseMenu.GameIsEnd)
{
    if (hit.collider.gameObject.tag == "Items")
    {
        ItemNotice.SetActive(true);
        NPCNotice.SetActive(false);
        TriggerNotice.SetActive(false);
        if (Input.GetMouseButtonDown(0))
        {
            CatchAndSave(hit);
        }
    }
```

Figure 27:Raycast and CatchItems

Apart from a Text-type title and a Button-type close button, the most important part of the Inventory system's UI is the item grid. As we mentioned before, the item grid will sort the sub-objects through a component called Grid layout Group, and a Content Size Fitter component will change the size of the item grid at various resolutions.In fact, each item slot is a Button type Prefab and the control script passes the relevant properties of the item to the item slot as the player collects it. The player can drop objects by clicking on an individual item slot to reveal the Drop button and then clicking on that.

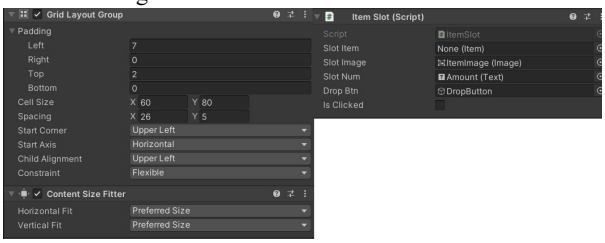

Figure 28:Item Grid and Item Slot

3.5.2 Dialogue and Quest System

1) Quest

Quest system is divided into two main parts, one is the quest file of type ScriptableObjects and the other is the UI of the quest panel.A quest file is similar to an Item in the Inventory system containing properties such as a quest name, quest description and an icon. The only difference is that the Quest file has a decisive property called Quest Status, which cannot be changed directly in ScriptableObjects, but only through the Custom Effect of Narrative Event.There are four states - not started, active, failed and completed. There are logical sequences between these states, for example a quest that has not been started can be changed to any of the other states, while a failed quest cannot be started again, and these states also determine how the quest appears in the quest panel.

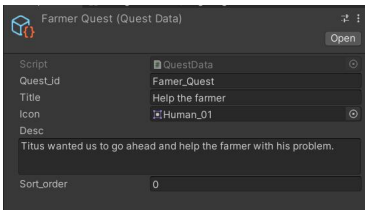

Figure 29:Quest ScriptableObjects

In Dialogue and Quest, all quest and dialogue related UI is on a separate Canvas, and the quest related UI is divided into two sections, QuestNotice, which is used to inform the player of the quest status, and the Quest Journal panel, which the player can turn on and off manually.The master script for both sections inherits the base class of the Dialogue and Quest system UI script, which contains functions related to displaying and closing.QuestNotice section will be displayed at the top of the screen for four seconds when the quest status changes and if the player clicks on this section of the UI it will jump to the Quest Journal panel.

```
base.Start();

NarrativeManager.Get().onQuestStart += (QuestData quest) => { ShowBox(quest, "New Quest"); };
NarrativeManager.Get().onQuestComplete += (QuestData quest) => { ShowBox(quest, "Quest Completed!"); };
NarrativeManager.Get().onQuestFail += (QuestData quest) => { ShowBox(quest, "Quest Failed!"); };
```

Figure 30:QuestNotice Functions

Quest Journal section is similar to the Inventory system UI in that a Grid Layout Group will be used to sort the presentation of quests. A combined UI component called ScrollView will also be used to implement a vertical scrolling display.Quest Journal section also contains three Button components for filtering the displayed quests with a reference to the quest status.

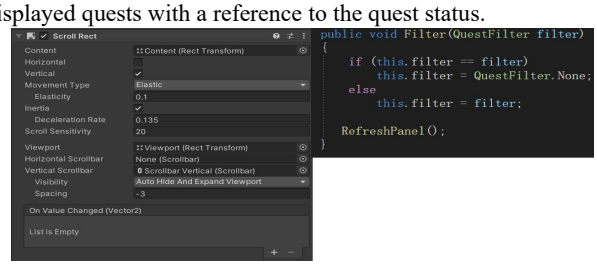

Figure 31:ScrollView and Fillter Function

2) Dialogue

As mentioned in the Related Work part, the most critical component of the dialogue system is the Narrative Event, which is implemented by a C# script and contains a series of properties about the trigger.Trigger_type determines how the event will be triggered, either by talking to a specific character or by entering a given area (a unique rectangular Prefab); Trigger_target determines which object can trigger the event, if empty it defaults

to the player; Trigger_actor determines the event's sponsor; Trigger_limit determines how many times the event can be triggered.

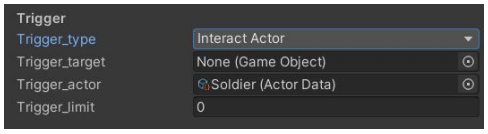

Figure 32:Trigger panel in Narrative Event

If the event is a Dialogue event, then there are some Dialogue related properties that need to be set:Dialogue_type is divided into two types, In Game and Dialogue Panel, which determines what type of UI the dialogue will be displayed in; Pause_gameplay determines whether the game will be paused when the event is triggered, and it is worth noting that this pause does not simply mean setting the game's timeScale to 0, but is a UnityAction, where the developer can add the desired effect to OnPause(); Skipable determines whether the event can be skipped by pressing the specified keyboard key.

Figure 33:Dialogue panel in Narrative Event

Scripts mounted on the sub-objects of a Narrative Event will be executed sequentially after the event has been triggered.Dialogue and Quest system provides a number of scripts to either present the dialogue or carry out some custom effects:Dialogue Message is used to store the content of the dialogue and the speaker's Actor file. The Duration and Pause properties will only work with In Game type dialogues, controlling the time the dialogue is displayed and the time between two short dialogues; Dialogue Choice provides the player with different options, containing a text to be displayed and an event that will be jumped to after the selection;Narrative Effects serve to perform some custom effects, either as functions in existing scripts, or as changes to quest states, or so on.

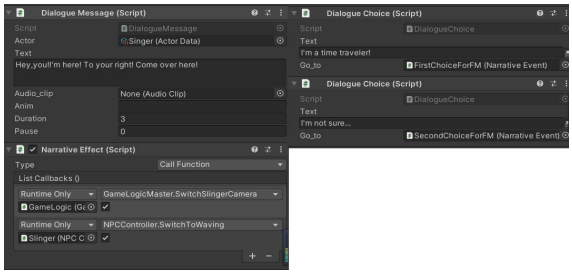

Figure 34:Custom Effect under Narrative Event

Furthermore, players can customise the conditions when events are triggered, which are special ScriptableObjects that return True or False in specific cases by overriding the parent class' IsMet() function.

```
[CreateAssetMenu(fileName = "CheckItemCondition", menuName = "CustomCondition/CheckItemCondition")]
public class CheckItemCondition : CustomCondition
{
    public ScriptableInventory myBag;
    public Item QuestItem;
    public override bool IsMet(Actor player)
    {
        if(myBag.itemList.Find(item => item.itemNmae == QuestItem.itemNmae))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

Figure 35:Custom Condition to Check if the specified item is available in the Inventory

3.5.3 Quiz and Goat-milking Game

1) Quiz

The Quiz system is implemented by a control script and a Canvas. All the functions of the Quiz System are in the control script, including reading the questions from the file, determining whether the player has answered correctly and adding the corresponding functions to the various components of the Canvas.

First read the specified text file with Resources.Load() function, then split each line of the text file in String format with the Split() function, and finally split each line into an array of some strings with the reference ";" again by the Split() function.These strings arrays are stored in another two-dimensional array where the information such as question number, title and answer will be passed to the appropriate UI component.

```
//Read csv binary files
TextAsset binAsset = Resources.Load("YW", typeof(TextAsset)) as TextAsset;
//Read the contents of each line
lineArray = binAsset.text.Split('\r');
//Creating two-dimensional arrays
ArrayX = new string[lineArray.Length][];
//Storing data from csv in a two-dimensional array
for (int i = 0; i < lineArray.Length; i++)
{
    ArrayX[i] = lineArray[i].Split(';');
}
//Set questions' status
topicMax = lineArray.Length;
for (int x = 0; x < topicMax + 1; x++)
{
    isAnserList.Add(false);
}
```

```
    {
        toggleList[i].isOn = false;
    }
    for (int i = 0; i < toggleList.Count; i++)
    {
        toggleList[i].interactable = true;
    }

    tipsbtn.SetActive(false);
    tipsText.text = "";

    indexText.text = "QUESTION " + (topicIndex + 1) + ":";//CURRENT QUESTION
    TM_Text.text = ArrayX[topicIndex][1];//QUESTION CONTENT
    int idx = ArrayX[topicIndex].Length - 3;//OPTIONS NUM
    for (int x = 0; x < idx; x++)
    {
        DA_TextList[x].text = ArrayX[topicIndex][x + 2];//OPTIONS
    }
```

Figure 36:ReadText() and LoadAnswer() functions

2) Goat-milking Game

Goat-Milking Game is implemented in the same way as Quiz, with a control script and a Canvas. This game uses the InvokeRepeating() function to generate milk and penalty objects (two Image components with Trigger type collider) at random x-axis locations on the same y-axis of the screen at regular intervals.The player can move an empty bucket by pressing the left and right key on the keyboard. The empty bucket is the same as the milk, but with a special tag that when the empty bucket collides with the milk it will clear the milk object and add a point, while subtracting a point when it hits a penalty object.

```
#region milkpart
public void CreateMilk()
{
    float x = Random.Range(-Screen.width*2 / 3, Screen.width*2 / 3);
    float y = Screen.height / 3;
    GameObject Milk = Instantiate(MilkImage, MilkPanel.transform);
    Milk.transform.localPosition = new Vector3(x, y, 0);
}
```

```
public void RepeatCreating(bool isReapting)
{
    if(isReapting)
    {
        InvokeRepeating("CreateMilk", 0.1f, 0.6f);
        InvokeRepeating("CreatePunish", 0.1f, 1.5f);
    }
    else
    {
        CancelInvoke("CreateMilk");
        CancelInvoke("CreatePunish");
    }
}
#endregion
```

Figure 37:CreateMilk() and RepeatCreating() functions

3.5.4 Combat System

Two important parts of the battle system, the enemy AI and the implementation of the attack animations have already been

described in detail earlier. Moreover, the combat system contains a Health system which manages the death of characters and the victory or defeat of battles.

The UI of the Health system is implemented by an empty health slot, an actual health bar and a Text component showing the value of health. It works by changing the size of the displayed health bar according to the percentage of health left in relation to the total value of health.It is remarkable that the Health Bar UI, unlike other Canvas, will exist in world space as a sub-object of the character - in other words, be implemented as a HUD. The Health Bar HUD will also have a script mounted on it which keeps it player-oriented with the transform.LookAt() function.

Figure 38:Health Bar HUD

Health system health control will be achieved by a script that will be added to the appropriate character. The script contains parameters such as the character's total health, current health and whether they are alive or not, as well as interfaces to functions used to create damage that can be called when a weapon hits a target.

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Player" && !PlayerMovement.isDefensing)
    {
        if (other.TryGetComponent<HealthSystemForDummies>(out HealthSystemForDummies healSystem))
        {
            healSystem.AddToCurrentHealth(-200);
        }
    }
}
```

Figure 39:Do Damage Function

When the character's current health reaches 0, the Health system's isAlive parameter will be set to false, and some existing functions can be called, such as playing the death animation and showing the battle results.

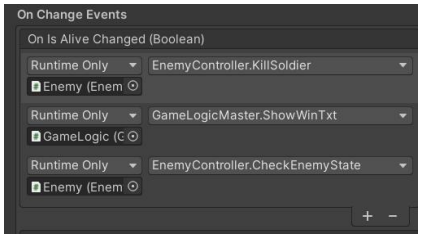

Figure 40:Functions executed when isAlive changed

3.6 Artwork Implementation

3.6.1 Camera Related

Cinemachine is utilised in this project to implement a variety of camera effects.After importing the Cinemachine from the PackageManager, the virtual camera can be created in the project and a new CinemachineBrain component will be automatically added under the original main camera to control the camera related parameters, at which point the main camera will become the brain of all the virtual cameras and it will be possible to see in the Live Camera parameters which virtual camera is currently active.

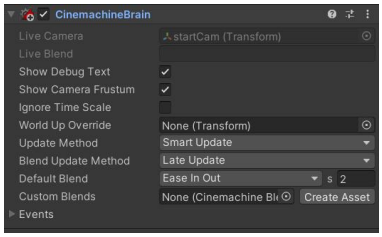

Figure 41:CinemachineBrain

As the original main camera can no longer be used as a first-person camera, a new virtual camera will be tied to the player character's head position to replace the main camera.A MouseLook script will be mounted under this virtual camera, which serves to change the overall orientation of the character as the mouse moves along the x-axis and to change the Euler angle of the first person camera as the mouse moves along the y-axis.

```
void Update()
{
    float MouseX = Input.GetAxis("Mouse X") * MouseSensitivity * Time.deltaTime;
    float MouseY = Input.GetAxis("Mouse Y") * MouseSensitivity * Time.deltaTime;

    XRotation -= MouseY;
    XRotation = Mathf.Clamp(XRotation, -90f, 90f);

    transform.localRotation = Quaternion.Euler(XRotation, 0f, 0f);

    PlayerBody.Rotate(Vector3.up * MouseX);
}
```

Figure 42:Camera Rotation Function

Cinemachine also has a Dolly Virtual Camera that can be moved along a set track, which works in conjunction with Timeline for cinematic camera movement. First set the camera's path in the appropriate Dolly Track, then create an AnimationTrack in the Timeline and use the Path Position as the key to create a keyframe to gradually move the camera to that position. Additionally, a CinemachineTrack can be created in the Timeline to switch between different virtual cameras over time.

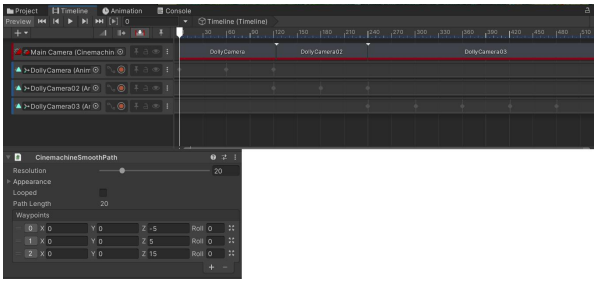

Figure 43:Timeline and Dolly Track

3.6.2 Day and Night System

The Day and Night system consists of two parts, a script for controlling the time and position of light sources, the other being the shader for changing the skybox representation.The control script requires two directional lights to represent the sun and the moon, as well as a central point and a rotation axis to change the position of the directional light.We calculate the current angle of sunlight by subtracting 6.0 from the current time (in twenty-four hours) and multiplying by 15.0[27].Each update then rotates the centre point along the rotation axis by the offset of the sun light

angle, while keeping the sun or moon on the forward vector of the centre point and facing the centre point at all times, which completes the movement of the sun and moon.

```
void setTime()
{
    if (Time >= 24.0f)
    {
        Time = 0.0f;
    }
    SunAngleH1 = (Time - 6.0f) * 15.0f;
    Time += 0.001f * TimeSetSpeed;
    if (Time >= 0.0f && Time < 12.0f)
    {
        SkyboxMaterial.SetFloat("_TimeMapping", (Time - 6.0f) / 6.0f);
    }
    else if (Time >= 12.0f && Time < 24.0f)
    {
        SkyboxMaterial.SetFloat("_TimeMapping", (18.0f - Time) / 6.0f);
    }
}
void doRotation()
{
    CenterPoint.transform.rotation *= Quaternion.AngleAxis(SunAngleH1 - SunAngleH2, RotationAxis);
    SunAngleH2 = SunAngleH1;
    SunPosition = CenterPoint.transform.position + Vector3.Normalize(-CenterPoint.transform.forward) * Distance;
    MoonPosition = CenterPoint.transform.position + Vector3.Normalize(CenterPoint.transform.forward) * Distance;
    Sun.transform.position = SunPosition;
    Moon.transform.position = MoonPosition;
    Sun.transform.LookAt(CenterPoint.transform);
```

Figure 44:SetTime and doRotation function

The script also calculates the change in light intensity, light colour and shadow strength parameters of the directional light over time. The sun is shown from 6 o'clock to 18 o'clock each day and the moon for the rest of the day.Moreover, the script needs to pass the time to the shader of the skybox in real time.

```
void doLightSwitch()
{
    if (Time >= 6.0f && Time < 18.0f)
    {
        Sun.SetActive(true);
        Moon.SetActive(false);

        Sun.transform.GetComponent<Light>().intensity = 0.01f + SunLightIntensity * smoothstep(0.0f, 0.4f, System.Math.Max(Vector3.D
        Sun.transform.GetComponent<Light>().color = Color.Lerp(new Color(1.0f, 0.75f, 0.5f), new Color(1.0f, 0.95f, 0.9f), smoothste
        Sun.transform.GetComponent<Light>().shadowStrength = Sun.transform.GetComponent<Light>().intensity / SunLightIntensity / 1.5
        //DirectionalLight.transform.GetComponent<Light>().intensity = SunLightIntensity * smoothstep(0.0f, 0.4f, System.Math.Max(Ve
        //DirectionalLight.transform.GetComponent<Light>().color = Color.Lerp(new Color(1.0f, 0.75f, 0.5f), new Color(1.0f, 0.95f, 0
        //DirectionalLight.transform.GetComponent<Light>().shadowStrength = DirectionalLight.transform.GetComponent<Light>().intensi
    }
    else
    {
        Moon.SetActive(true);
        Sun.SetActive(false);

        Moon.transform.GetComponent<Light>().intensity = 0.01f + MoonLightIntensity * smoothstep(0.0f, 0.3f, System.Math.Max(Vector3
        Moon.transform.GetComponent<Light>().color = Color.Lerp(new Color(0.5f, 0.5f, 0.5f), new Color(0.6f, 0.8f, 1.0f), smoothstep
        Moon.transform.GetComponent<Light>().shadowStrength = Moon.transform.GetComponent<Light>().intensity / MoonLightIntensity /
```

Figure 45:doLightSwitch function

The sky box shader is mainly used for sky effects that change over time with day and night, as well as sun, moon and star conditions. We can choose the ideal day and night colours and the fragment shader will dynamically adjust the output sky lightness and darkness according to the time we pass in from the script.The shader also contains parameters for the position of the horizon and the gradient colour, which can be adjusted to suit any preference.

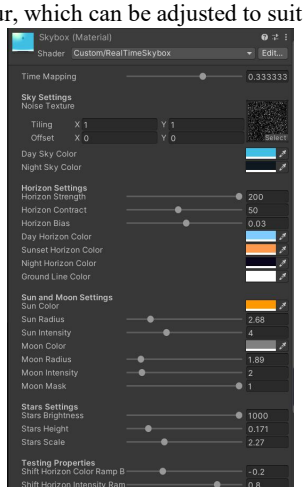

Figure 46:Skybox shader

4 RESULTS AND EVALUATION

4.1 Gameplay Demonstration

Figure 47 shows the situation when the player triggers a dialogue event, at which point the player's movement actions are locked and the UI for other gameplay cannot be opened.

Figure 47:Dialogue UI

Figure 48 shows the situation when the player is in answering the question. After finishing the quiz, the player can submit the answer by clicking Submit in the dialogue window.

Figure 48:Dialogue UI

Figure 49 shows the scene when the player is taking part in the goat milking game, with relevant hint messages on the UI where the player can press T to return to the main game screen at the end of the game.

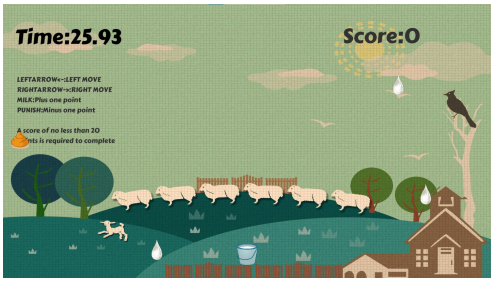

Figure 49:Goat-milking UI

Figure 50 shows part of a battle scene where the player is fighting in a gladiatorial arena in which the player needs to defeat his opponent in order to leave the field. If the player dies, they have the option to restart from the starting point.

Figure 50:Battle Scene

Figure 51 shows the UI of the Inventory system and Quest system, where players can click on items or quests to view their descriptions.

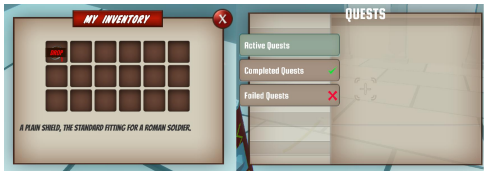

Figure 51:Inventory and Quest UI

4.2 Project Evaluation

4.2.1 Technical Evaluation

Unity project has been shared on Github and it works fine with UnityEditor version 2020.3.30f1c1, which has also been shared online via MEGA and OneDrive. A playable version which is published by Build and Run has also been shared - see the README word file for links.

All materials used in the project, including audio, are from Unity Asset Store, Mixamo.com and Github, which are all open source sites with no copyright issues.

This project implements a narrative role-playing game where the player can enter the game through the main menu and win after completing a series of quests.

The project implements many RPG game classic gameplay features such as dialogue, quests and items, with techniques including but not limited to Monobehaviour scripts, ScriptableObjects scripts, StateMachineBehaviour scripts, Universal Render Pipeline, Particle System, etc.

The logic of the game is relatively clear and there have been no multiple gameplay conflicts in the tests. The structure of the project is fairly straightforward, with a moderate degree of coupling between various parts, making it easy to monitor as well as maintain and upgrade the project.

4.2.2 Educational Significance Evaluation

The project takes players back in time to the ancient Roman era to experience the culture and enjoy the unique landscapes of the time. Players will be guided as they explore the world freely without feeling lost through quests and item descriptions.

In the first quest, the player needs to follow the directions to find a noble lady who needs help. Completing this quest requires the player to look carefully at the environment and pay attention to the text messages. Players can also turn to NPCs for hints when they are at a loss, practising their logical thinking skills while gaining rewards.

In the second quest, players need to get help from a famous historical figure, but before that, they need to complete his commission and experience milking a goat.After successfully completing the commission, players will also need to prove their wisdom by answering a number of questions related to ancient Roman history. These questions may not be complicated, but they can educate the player about the history of the ancient Roman period. There is no limit to how many times the player can answer the questions, so the player can try again and again to discover the correct option.

In the last quest, players will meet a famous emperor and, in order to get his help, they will have to prove their bravery by defeating an enemy in a gladiatorial arena.The battle may not be easy, but players can try unlimited times, training patience and perseverance while experiencing the Roman gladiatorial culture, where, as Quintus Horatius Flaccus says, 'Adversity reveals genius; fortune conceals it'.

5 CONCLUSION

5.1 Project Results and Challenges

This is the first relatively complete Unity3D project developed by the author. The game architecture is comparatively simple, completing common functional modules in games such as player input and output module (OI), character animation module, camera module and guidance module.Additionally, special gameplay modules have been extended in this project, such as an event-based dialogue and quest system, an Inventory system that implements Item fetching, discarding and information reading, a pseudo-2D mini-game based on UI and file streaming, plus a combat system that includes an AI state machine and a character property module.Some art-related modules are also used in this project, such as Post Processing, Particle System and skybox shader sections. The process of learning these modules serves as an introduction to working with game effects.

During development, the author learned more about the usage of the Unity engine features, the game architecture and possible ways to implement common gameplay, while also encountering some difficulties and trying to solve them by searching for information.

For instance, the first problem met in development is that the UI behaves abnormally at different resolutions. The solution to this problem is to adjust the anchor properties of the UI components to the right position, whilst some special UI components are unable to change the anchor properties for which alternative solutions are needed, for example, Grid Layout Group in Inventory system requires a Content Size Fitter to adapt the UI to various resolutions.

Another troublesome issue we faced was the first person camera going through the model during player movement (a so-called 'clipping bug'). After reviewing materials, we found that currently the most common solution to clipping bug is to show only the

player's hand model in the first person camera (all shadows are rendered normally).Unfortunately, the player model used in this project was not detailed enough to classify individual body parts in such a way that it would be extremely difficult to use this method, therefore another possible method was used in the project, where a motion camera is also placed on the character's head to prevent clipping while the character is moving. When the player performs actions such as jumping and rolling, it will automatically switch to the motion camera, and the camera's blend mode will also switch from Ease In Out to Cut.This does not solve the problem of players still seeing their bodies when they look down, but it does effectively tackle clipping bugs that occur during movement.

Some logical inconsistencies in the game have also been addressed one by one. In summary, many details of the game are continuously being refined after the overall development of the project has been completed.

5.2 Future Work

The project lacks many more features than RPGs commonly found today. It is possible to try to add account login to the game in the future, as we did in CSC8501; An attempt can be made to have network online functionality added to the game, which requires an in-depth study of the network module of Unity3D.

Besides, a load and save function may be integrated into the project, achieved with Unity's JSON API. But before we can do that, we have to specify which objects and which properties need to be saved and loaded, something that is difficult to achieve in the current project due to the numerous events. A possible solution is to give events a special logical property, like the state property of a quest, so that events in the same state can be stored together.

REFERENCES

[1]. Baidu.2022.Baidu Bike.[Online] Available: https://baike.baidu.com/item/%E5%8F%A4%E7%BD%97%E9%A9%AC/888289?fr=aladdin

[2]. Vindolanda Charitable Trust.2022.Roman Vindolanda Fort & Museum.[Online] Available: https://www.vindolanda.com/roman-vindolanda-fort-museum?gclid=CjwKCAjwt7SWBhAnEiwAx8ZLamH5SQ_U_YSUtiI3ja3dK4AM0pnHTxsdfZevcqKJJNG5d5n3iWl7iRoCG7QQAvD_BwE

[3]. C.Michael Hogan.2007.Vindolanda Roman Fort, The Megalithic Portal, ed. A.Burnham

[4]. Birley, Vindolanda Guide, 2012, page 35.

[5]. Birley, Vindolanda Guide, 2012, page 36.

[6]. Birley, Barbara.2016."Keeping up Appearances on the Romano-British Frontier".Internet Archaeology(42).doi:10.11141/ia.42.6.6.

[7]. Williams, Rhys; Thompson, Tim; Orr, Caroline; Birley, Andrew; Taylor, Gillian.21 June 2019."3D Imaging as a Public Engagement Tool: Investigating an Ox Cranium Used in Target Practice at Vindolanda".Theoretical Roman Archaeology Journal.2(1): 2.doi:10.16995/traj.364.ISSN2515-2289.

[8]. Wang Shengwei.10 October 2021."An analysis of Unity3D-based game development." Digital Design. [Online] Available:http://qikan.cqvip.com/Qikan/Article/Detail?id=7104521010

[9]. Engine Appreciation Group.7 July 2020."About Unity (detailed version)" .[Online] Available:https://developer.unity.cn/projects/5f02da69edbc2a001f442a7b

[10]. Deng Wei.2020.Design and application of the singleton pattern in Unity 3D [J]. Journal of Kashgar University

[11]. Wang Yu.2013.An introduction to the art of architecture in the ancient Roman period[J]. China-ASEAN Expo.

[12]. Mixamo.2022.Mixamo Get animated .[Online] Available : https://www.mixamo.com/#/

[13]. Baidu.2022.Baidu Bike .[Online] Available : https://baike.baidu.com/item/Github/10145341?fr=aladdin

[14]. Liang Chao,Wang Xing.2021.A method for managing Unity scenes and interfaces:, CN112685027A [P]

[15]. Valve Developer Community.22 August 2015 ."Skybox Basics"

[16]. Geng Mingming.2010.The art of communication in web UI design [D]. Harbin University of Science and Technology Master's thesis

[17]. Kirill Muzykov.29 September 2014.Unity 4.6 New GUl Tutorial. [Online] Available:http://kirillmuzykov. com/unity-4-6-new-gui-tutorial/.

[18]. Bin Xiaohua.2013. Player-NPC interaction method in online games:, CN103116685A [P]. [Online] Available:https://baike.baidu.com/item/NPC/53782?fr=aladdin

[19]. Pathiral.2022.Blaze AI Engine .[Online] Available : https://assetstore.unity.com/packages/tools/ai/blaze-ai-engine-194525

[20]. Geoff Howland.1999.The Focus Of Gameplay

[21]. Tank V .2022. Create Scriptable Objects with Unity.

[22]. Indie Marc.2022.Dialogue and Quest.[Online]. Available: https://assetstore.unity.com/packages/tools/ai/dialogue-and-quests-183780#description.

[23]. WIKIPEDIA.2022.AirDuel .[Online] Available : https://en.wikipedia.org/wiki/Air_Duel

[24]. Unity.2022.Cinemachine documentation .[Online] Available : https://unity.com/unity/features/editor/art-and-design/cinemachine

[25]. STYLY .2 September 2020. “Understanding the Basics of Timelines [Unity]” .[Online] Available : https://styly.cc/tips/timeline_unity_kaki/#:~:text=The%20Unity%20Timeline%20is%20a,track%20in%20video%20editing%20software.

[26]. liquanyi007.2020.Universal Render Pipeline Outline .[Online] Available : https://blog.csdn.net/liquanyi007/article/details/109749420

[27]. XenonSetsuna.10 Mar 2022. EnvironmentSystemForURP.[Online] Available : https://github.com/XenonSetsuna/EnvironmentSystemForURP