Installation

Prerequisites

- Git
- Unity
- An IDE or text editor

Cloning

The repo is located at: https://github.com/wc641/Team-Grey-SE-CW2
If you wish to clone just the 'Delivery' branch use:

```
git clone --single-branch --branch delivery
git@github.com:wc641/Team-Grey-SE-CW2.git
```

Otherwise use:

```
git clone git@github.com:wc641/Team-Grey-SE-CW2.git
```

Opening the project

The game was built with **Unity2020.3.22f1 (LTS).** Though it is likely the game will build with future updates, it is advised to install the listed version. Unity can be installed using the Unity Hub.

Once cloned, the project can be opened in the Unity Hub by selecting the Projects tab, pressing 'Add' and navigating to the directory where the repo is cloned. Selecting the project in the Unity Hub will automatically open it in Unity.

If you wish to edit the code, ensure you have set a default editor for Unity. This can be set by going to 'Preferences -> External Tools' and selecting an External Script Editor in Unity.

Debugging will differ depending on the editor chosen. Resort to Unity guides for further information.

Project Hierarchy

The following describes the hierarchy of the project. The sections below are all contained within the ‘Assets’ folder.

Scripts

Scripts in Unity allow the player to interact with the game in ways chosen by the developers. Furthermore, they allow for objects on the screen to interact with each other. All scripts follow a convention of using namespaces. The namespace of a class follows the hierarchy structure of the project. A namespace is a collection of classes that are referred to using the chosen prefix: *VS.CW2RTS.xxx* in the case of this project. The use of namespaces helps to avoid clashes between different classes with the same names. This problem becomes apparent as projects grow and more programmers get involved with it.

Buildings

BasicBuilding.cs

- Class **BasicBuilding**
    - The basic building class is an extension of **ScriptableObject**. The purpose of this class is to provide the user with quick access to the settings and the base stats of the existing buildings, specified in the **buildingType** enumerator. The building settings include the building **type**, **name**, **buildingPrefab**, **icon** and **spawnTime**. Base stats from the **BuildingStateTypes.Base** class can also be edited for each building’s **ScriptableObject**. Furthermore, new scriptable objects for buildings can be created via the Unity Editor, by selecting Assets > Create > New Building > Basic from the main menu.
    - Four building types available are defined in the **buildingType enum**: Barracks (used for testing, see Future Work), EnemyBasicBuilding, EnemyAttackTower and EnemyCore.

BuildingHandler.cs

- Class **BuildingHandler**
    - The building handler class is an extension of **MonoBehavior**. The purpose of the class is to match an instance of a building with the respective set of predefined stats.
    - **Awake()** - called when the parent object is instantiated, assigns the instance of the BuildingHandler to itself. This way, the instance of the building handler can be called from anywhere in the code via **BuildingHandler.instance**.
    - **GetBasicBuildingStats()** - returns the predefined base stats of an instance of a building, determined by the parameter **type**.

BuildingStatTypes.cs

- Class **BuildingStatTypes**
    - The building stat types class extends **ScriptableObject**. The purpose of the class is to initialise the types of stats that buildings have. These stats can be predefined for each building type and later on manipulated during gameplay.

Enemy

EnemyBuilding.cs

- Class **EnemyBuilding**
    - The enemy building class is an extension of **MonoBehavior**. The purpose of the class is to handle enemy buildings during gameplay. The script is attached to every instance of an enemy building object.
    - **Start()** - sets the base stats of the instance of the building of the respective building type, sets up the stat display above the building (health bar) and sets up the navigation mesh for the building, where the buildings act as obstacles.
    - **Update()** - called once per frame, handles the attack sequence for enemy attack towers.
    - **CheckForEnemyTargets()** - locates enemy targets in the given radius, defined by the aggro range stat.
    - **Attack()** - deals damage to the targeted unit and draws the projectile ray.

Player

PlayerBuilding.cs

- Class **PlayerBuilding**
    - The player building class is an extension of **MonoBehavior**. Similarly to the **EnemyBuilding** class, the purpose of this class is to handle player's buildings during gameplay. This class was used during development phases with the **Barracks** building type. See Future Work section for further details.

InputManager

InputHandler.cs

- Class InputHandler
    - The input handler class is an extension of MonoBehavior. It handles the interaction between the user's input and the main gameplay.
    - **Awake()** - called when the parent object is instantiated, assigns the instance of the InputHandler to itself. This way, the instance of the input handler can be called from anywhere in the code via **InputHandler.instance**.
    - **OnGUI()** - called for rendering and handling GUI events, checks if the LMB (Mouse Button 0 in Unity) and LShift are held down. If so, it draws a rectangle following the direction of the mouse, selecting all the player units within the rectangle's bounds.

- **HandleUnitMovement()** - main function for the user to interact with their units. Checks for the mouse buttons and keyboard keys pressed by the user, returning respective output. Holding down, dragging and releasing LMB draws a box and selects all player units within its bounds. Selecting units adds them to a **selectedUnits** list and enables a marker underneath them. Performing this action again, selects any units in the new box, deselecting the previously selected units if those are not within the new box's bounds. With units selected, multiple actions can be performed with RMB (Mouse Button 1 in Unity), depending on the target's layer. Clicking on another player unit will command the selected units to chase that unit. Clicking on an enemy unit will command the selected units to chase and attack that unit. Clicking on untraversable obstacles (rocks) has no effect and the selected units will carry on with their latest command. Clicking on the background will command the selected units to move to the target point.
- **DeselectUnits()** - clears the list of selected units or buildings, disabling the markers underneath them.
- **isWithinSelectionBounds()** - returns a boolean value depending whether a unit instance is within the selection box drawn.
- **HaveSelectedUnits()** - returns a boolean value depending whether there are currently any units selected.
- **addedUnit()** - adds a unit to the selected units list and deselects other units if multi selection is not being used. Enables the unit highlight and returns the added unit as a **Interactables.IUnit** object.
- **addedBuilding()** - serves similar function as above, except buildings are not multiselectable - selecting a building deselects everything else. More on player buildings in the Future Work section.

MultiSelect.cs

- Class **MultiSelect**
  - The purpose of the class is to create and draw the multi selection box when the user holds down the LMB.
  - **DrawScreenRect()** - draws a rectangle on the screen with the specified coordinates and of specified colour.
  - **DrawScreenRectBorder()** - draws the border for the multi selection box, utilises the above function to draw narrow rectangles on the edges of the selection rectangle.
  - **GetScreenRect()** - returns a 2D rectangle **Rect struct**, specified by the initial and final mouse positions whilst the LMB was held.
  - **GetVPBounds()** - returns an axis aligned bounding box **Bounds struct**. The bounding box is aligned with the camera axis. The return value is used to determine which units fall under the user drawn selection box.

Interactables

IBuilding.cs

- Class **IBuilding**
    - The interactable building class is an extension of the **Interactable** class. The main purpose of this class is to define the functionality of the buildings that the player can select and interact with.
    - The class was used for purposes of testing with player **Barracks** building, which can spawn player units. See Further Work for more info.
    - **OnInteractEnter()** - overrides the virtual function (also known as an abstract function) in the **Interactable** class. Defines the actions that take place when the user clicks on the interactable building. This includes activating the action frame (more about this in the HUD section), showing the spawn marker (the movement target for the spawned units) and highlighting the building with a marker underneath, to emphasise that it is selected.
    - **OnInteractExit()** - overrides the virtual function in the **Interactable** class. Defines the actions that take place when the user deselects the building. Hides the action frame, disables the spawn marker and the building highlight.
    - **SetSpawnMarkerLocation()** - sets the position of the spawn marker based on the mouse position. The function would be called when RMB is clicked on the map with the building selected.

Interactable.cs

- Class **Interactable**
    - The interactable class extends the **MonoBehaviour**. It also acts as the superclass for **IBuilding** and **IUnit** classes.
    - **Awake()** - called when the parent object is instantiated. Ensures that the highlight is deactivated.
    - **OnInteractEnter()** - virtual function, calls to activate the highlight and sets the boolean interacting variable to true.
    - **OnInteractExit()** - virtual function, calls to deactivate the highlight and sets the boolean interacting variable to false.
    - **ShowHighlight()** - virtual function, activates highlight.
    - **HideHighlight()** - virtual function, deactivates highlight.

IUnit.cs

- Class **IUnit**
    - The interactable unit class is an extension of the **Interactable** class. The main purpose of this class is to define the functionality of the units that the player can select and interact with. Similarly to the **IBuildings** class, **IUnits** can be introduced to a HUD when interacted with by a player, with various abilities that the units could cast in the future iterations of the project. See Further Work for more details. The following functions are left for future use.
    - **OnInteractEnter()** - overrides the virtual function in the **Interactable** class. Calls for the **base** class' **OnInteractEnter()** function.
    - **OnInteractExit()** - overrides the virtual function in the **Interactable** class. Calls for the **base** class' **OnInteractExit()** function.

Player

PlayerManager.cs

- Class **PlayerManager**
    - The player manager class extends **MonoBehaviour**. The purpose of the class is to handle the gameplay during runtime.
    - **Awake()** - called when the parent object is instantiated. Calls to set the basic stats of the game objects.
    - **Start()** - sets the boolean value for the destruction of the enemy core to false.
    - **Update()** - called once per frame, calls for the **HandleUnitMovement** in the **InputHandler** class, checks for the win/lose conditions of the game.
    - **SetBasicStats()** - sets the basic stats for all the units and buildings in the scene.

UI

CameraController.cs

- Class **CameraController**
    - **Start()** - gets the current position of the camera and zoom level
    - **Update()** - called once per frame, triggers the movement inputs
    - **ClampZoom()** - Limits the zoom values between min and max
    - **HandleMouseInput()** - Handles mouse input from user. Determines where to position the in-game Camera based on where the user is clicking/dragging on the screen. Also calculates camera zoom based on scroll wheel movement.
    - **HandleMovementInput()** - Updates in-game camera position based on input of WASD keys. Also handles zoom via R and F keys. Interpolates values to ensure the movement is smooth

LevelEndScreen.cs

- Class **LevelEndScreen**
    - **ScreenToShow(bool hasWon)** - Sets an array accessor based on whether the player has won or lost.
    - **Setup(bool hasWon)** - Sets the corresponding Win or Lose page as active based on user result. Also sets PlayerPrefs to save level progress.
    - **GoToNextLevel()** - Disables the Win/Loss screen and moves to the next scene
    - **Restart()** - Disables the Win/Loss screen and reloads the current scene
    - **GoToMainMenu()** - Disables the Win/Loss screen and moves to the MainMenu scene.

LevelSelector.cs

- Class **LevelSelector**

- **Start()** - Reads player preferences and determines whether to make the level buttons accessible based on which level has been reached.
- **Update()** - method used for debugging purposes. Allows a developer to delete the preferences file when running a debug build.

SceneSwitcher.cs

- Class **SceneSwitcher**
  - **LoadScene(string sceneName)** - Uses unity's SceneManager object to switch to the scene passed to the method.
  - **quitGame()** - Quits the game.

TextBlink.cs

- Class **TextBlink**
  - **Start()** - Initializes _blinkText field with the text which will blink during execution of application. It also initializes _color field with the color of blink text.
  - **Update()** - called once per frame, performs the main logic of fadein and fadeout. This method increments _timechecker field by Time.deltaTime and this modified value is compared with various configurable fields like BlinkFadeInTime, BlinkStayTime and BlinkFadeOutTime to provide different color to the blinking text to achieve blink feature. Update method also checks for if the mouse is hovering over text, it changes its color to blue.
  - **OnPointerEnter()** - Sets _hover field to true.
  - **OnPointerExit()** - Sets _hover field to false.

PlayerIntroScreen.cs

- Class **PlayerIntroScreen**
  - **Play()** - This method deactivates the current gameObject and allows the game screen to be visible to the user.

PauseMenu.cs

- Class **PauseMenu**
  - **Start()** - Sets the pause menu object to inactive to ensure it doesn't show on load.
  - **Update()** - Polls to see if the ESC key has been pressed. If it has, and the bool - isPaused is false, pauses the game else resumes the game.
  - **PauseGame()** - Sets the pause menu to active, freezes in game time and sets the isPaused bool to true.
  - **ResumeGame()** - Sets the pause menu to inactive, resumes in game time sets the isPaused bool to false.
  - **MainMenuPressed()** - Loads the MapMenu scene, resumes in game time and sets isPaused bool to false.

- **ShowControls()** - Sets the pauseScreen object to inactive and sets the controlsScreen object to active.
- **BackPressed()** - Sets the pauseScreen object to active and sets the controlsScreen object to inactive.

PauseMenu.cs

- Class **PauseMenu**
  - **Start()** - Sets the passed in game object to inactive
  - **Show()** - Sets the passed in game object to active

SoundManager.cs ***unused***

- Class **SoundManager**
  - **Start()** - sets volume to a default level in PlayerPrefs, or loads the last used volume level
  - **ChangeVolume()** - Sets a new volume level and saves to prefs
  - **Load()** - Loads the last used volume level from PlayerPrefs
  - **Save()** - Saves the last used volume level to the PlayerPrefs

DontDestoryAudio.cs ***unused***

- Class **DontDestoryAudio**
  - **Awake()** - Called on instantiation, keeps the audio object alive throughout the game's lifetime.

HUD

The HUD scripts were used during the testing phase of development and are not used in the current iteration of the build. See Further Work for more details.

Action.cs

- Class **Action**
  - The action class is an extension of the **MonoBehaviour** class. The purpose of the class is to manage the actions made by the player, such as spawning units from buildings or using unit abilities.
  - **OnClick()** - when an action is clicked on and used, the function sets a timer for the cooldown before this action can be used again.

ActionFrame.cs

- Class **ActionFrame**
  - The action frame class is an extension of the **MonoBehaviour** class. The purpose of the class is to set up a HUD frame of available actions with the selected interactable and allow the player to perform the actions.
  - **Awake()** - called when the parent object is instantiated, assigns the instance of the **ActionFrame** to itself.

- **SetActionButtons()** - instantiates the HUD with the available action buttons to the player, if the selected interactable has any actions.
- **ClearActions()** - clears and removes the HUD for an interactable that has actions available, called when this interactable is deselected.
- **StartSpawnTimer()** - called when the player presses a button on the HUD to spawn a new object. The function checks the type of object and whether it can be spawned. If it can, the object is ordered in the spawn queue. Initiates the spawn timer - amount of time it takes before the object is spawned.
- **IsUnit()** - called if the player presses a button to spawn a unit. Returns the unit as a **BasicUnit** object.
- **IsBuilding()** - called if the player presses a button to spawn a building. Returns the building as a **BasicBuilding** object.
- **SpawnUnit()** - called when the object is ready to be spawned. Instantiates the first object in the spawn queue, assigns the object to the correct place in the hierarchy and sets the movement destination of the object to the spawn marker. Lastly, removes the object from the spawn queue.

ActionTimer.cs

- Class **ActionTimer**
  - The action timer class is an extension of the **MonoBehaviour** class. Its purpose is to time until an action can be used again.
  - **Awake()** - called when the parent object is instantiated, assigns the instance of the **ActionTimer** to itself.
  - **SpawnQueueTimer()** - applies the IEnumerator interface to handle the real time taken before an object is spawned. Uses Coroutine to handle a queue of multiple objects to be spawned.

PlayerActions.cs

- Class PlayerActions
  - The action timer class is an extension of the **ScriptableObject** class. The purpose of the class is to initialise the types of objects that can be spawned by the user. For instance, one building can be used to spawn Knights, another building can be used to spawn Healers.Furthermore, new scriptable objects for player actions can be created via the Unity Editor, by selecting Assets > Create > PlayerActions from the main menu.

Units

BasicUnit.cs

- Class **BasicUnit**
  - The action timer class is an extension of the **ScriptableObject** class. The purpose of this class is to provide the user with quick access to the settings and the base stats of the existing buildings, specified in the **unitType** enumerator. The building settings include the building **type**, **unitName**, **playerPrefab**, **enemyPrefab**, **icon** and **spawnTime**. Base stats from the **UnitStateTypes.Base**

class can be edited for each **ScriptableObjec**t. Furthermore, new scriptable objects for buildings can be created via the Unity Editor, by selecting Assets > Create > New Building > Basic from the main menu.
    - The four unit types currently available are defined in the **unitType enum**: Knight, Warrior, Archer and Healer.

DrawProjectile.cs

- Class **DrawProjectile**
    - The draw projectile class is an extension of the **MonoBehaviour** class. The purpose of the class is to render a line over the ray casted in the following script.
    - **Awake()** - called when the parent object is instantiated, assigns the **LineRenderer** (projectile graphic) to a variable.
    - **SetupProjectile()** - called to set the start and end positions of the line graphic. Begins the **Coroutine** for timing the projectile.
    - **timeProjectile()** - enables the projectile line (makes it visible) for the set time duration, emphasising the long ranged attack.

ProjectileRaycast.cs

- Class **ProjectileRaycast**
    - The purpose of the class is to cast a ray between a long ranged unit (Archer, enemy attack tower) and its target. This ray is then used to place the visible projectile line over it, to make the visible graphic of a shot being made.
    - **Shoot()** - casts an invisible ray from the position of the attacker to the position of its target.

UnitHandler.cs

- Class **UnitHandler**
    - The unit handler class is an extension of the **MonoBehaviour** class. The purpose of the class is to assign unit instances their respective base stats.
    - **Awake()** - called when the parent object is instantiated, assigns the instance of the unit handler to itself.
    - **GetBasicUnitStats()** - returns the base stats of the required unit in form of the UnitStatTypes.Base object.

UnitStatDisplay.cs

- Class **UnitStatDisplay**
    - The unit stat display class is an extension of the **MonoBehaviour** class. Its purpose is to set up and dynamically vary the display of objects' stats, which is currently an object's health bar, representing their current health amount. The class also handles objects' health.
    - **SetStatDisplayBasicUnit()** - sets up the display with the current health amount (maximum health). Determines whether the unit is a player or an enemy unit. Assigns armour and maximum health values to variables for further processing.

- **SetStatDisplayBasicBuilding()** - similarly to the above, sets up the display with the current health amount (maximum health). Determines whether the building is a player or an enemy building. Assigns armour and maximum health values to variables for further processing.
- **Update()** - called once per frame, calls to handle the health of the parent unit.
- **TakeDamage()** - called when the parent unit gets attacked, reduces the current health by the attacker's attack strength minus the parent's armour amount.
- **Heal()** - called when the parent unit gets healed, increases the current health by the heal amount, up to the set maximum health value.
- **HandleHealth()** - called every frame to update the graphic of the health bar, reducing the fill amount of the line as the parent's health amount is decreased. Also updates the direction of the stat display to always face towards the player's camera. Calls for the **Die()** function if the health drops to or below zero.
- **Die()** - destroys the object that has run to zero health. If the object was a selected player unit, the unit gets removed from the list of the currently selected units. If the object was the enemy core, the boolean for its destruction is set to true, which completes the level.

UnitStatTypes

- Class **UnitStatTypes**
  - The building stat types class extends **ScriptableObject**. The purpose of the class is to initialise the types of stats that units have. These stats can be predefined for each unit type and later on manipulated during gameplay.

Enemy

EnemyUnit

- Class **EnemyUnit**
  - The enemy unit class is an extension of the **MonoBehaviour** class. It's purpose is to handle enemy units during gameplay.
  - **Start()** - sets the parent unit's base stats and navigation agent to local variables. Calls to set up the stat display.
  - **Update()** - handles the cooldown of the attacks. If the parent unit does not currently have aggro, calls to check for ally targets if the unit is a Healer, and for player targets if any other unit. If it does have aggro, it calls to chase and attack the aggro target.
  - **CheckForEnemyTargets()** - searches for player targets within the predefined aggro range. Creates a list of all the targets in the range and the aggro target is set to the first unit instance in the range. The units in the list are ordered alphabetically and therefore, normally, Archers would be prioritized. For a better player experience, the search script is enhanced to prioritise them last.
  - **CheckForAllyTargets()** - similarly to the search for player targets, enemy healers perform a search for the enemy units on their team, setting them as the aggro targets.

- **Attack()** - checks that the attack target is within the attack range and the attack is off its cooldown. Ensures that there is not attempt made to attack a destroyed unit, to mitigate game crashes. Satisfying the above conditions, if the parent unit is a Healer, it heals the target, otherwise, the target takes damage. If the parent is an archer, it shoots a projectile as well.
- **MoveToAggroTarget()** - checks that the aggro target still exists, if not, the unit halts and restarts search for new targets. Otherwise, as long as the target is within the aggro range, the navigation agent is used to dynamically chase the target, as the target's position changes when it moves.

Player

PlayerUnit

- Class **PlayerUnit**
  - The player unit class is an extension of the **MonoBehaviour** class. Similarly to the **EnemyUnit** class, it's purpose is to handle player units during gameplay.
  - **OnEnable()** - called when the parent object becomes enabled and active. Assigns the navigation agent to a local variable.
  - **Start()** - sets the parent unit's base stats to a local variable. Calls to set up the unit's stat display. Also assigns the **Animator** object to a variable, which was used during testing phases for animated units, see Future Work for more details.
  - **Update()** - handles the cooldown of the attacks. Similarly to the enemy units, if the parent unit does not currently have aggro, calls to check for ally targets if the unit is a Healer, and for enemy targets if any other unit. If it does have aggro, it calls to chase and attack the aggro target. Unlike the enemy units, player units give preference for player's commands, which can interrupt any other command that the unit is performing. If an animator is available, units perform animations depending on their current action (moving, attacking).
  - **MoveUnit()** - function that navigates the units to their target destination using the navigation agent.
  - **CheckForEnemyTargets()** - similarly to the enemy units, searches for enemy targets within the predefined aggro range. Creates a list of all the targets in the range and the aggro target is set to the first unit instance in the range. The function can be enhanced to set the closest unit in the list as the target. Additionally, if no enemy units are found, the search is made for enemy buildings.
  - **CheckForAllyTargets()** - similarly to the search for enemy targets, player healers perform a search for other player units, setting them as the aggro targets.
  - **Attack()** - mimics the functionality of that of the enemy units, suggesting that the two functions require merging to avoid repeating code.
  - **MoveToAggroTarget()** - similarly to the enemy units, this function also takes the player command in consideration, as the selected unit can be given command to chase an enemy target, even if it is outside of the unit's aggro range.

Fonts

- Contains the title font .ttf file

Materials

Materials are used in this project to assign colours to the objects.

- Green shades are used for Player Characters
- Red shades are used for Enemy Characters
- Pink is used for the ground.
- White is used for obstacles.
- Yellow is used for the unit highlights.

Music

All of the music files used in the game are stored in the music assets folder. Various music is played throughout the levels and game menus.

Prefabs

Prefabs are used as a template to store complete GameObjects, with their components, properties and child GameObjects. They're reusable assets for creating instances of the objects on the scenes.

Prefabs used in the current iteration of the project include:

- **playerKnight** - the player knight character.
    - Utilises the PlayerUnit script.
    - Utilises the IUnit script.
    - Utilises the Navigation Mesh Agent.
- **playerArcher** - the player archer character.
    - Utilises the PlayerUnit script.
    - Utilises the IUnit script.
    - Utilises the Navigation Mesh Agent.
- **playerWarrior** - the player warrior character.
    - Utilises the PlayerUnit script.
    - Utilises the IUnit script.
    - Utilises the Navigation Mesh Agent.
- **playerHealer** - the player healer character.
    - Utilises the PlayerUnit script.
    - Utilises the IUnit script.
    - Utilises the Navigation Mesh Agent.
- **enemyKnight** - the enemy knight NPC.
    - Utilises the EnemyUnit script.

    - Utilises the Navigation Mesh Agent.
- **enemyArcher** - the enemy archer NPC.
    - Utilises the EnemyUnit script.
    - Utilises the Navigation Mesh Agent.
- **enemyWarrior** - the enemy warrior NPC.
    - Utilises the EnemyUnit script.
    - Utilises the Navigation Mesh Agent.
- **enemyHealer** - the enemy healer NPC.
    - Utilises the EnemyUnit script.
    - Utilises the Navigation Mesh Agent.
- **enemyAttackTower** - the static enemy that can attack players.
    - Utilises the EnemyBuilding script.
    - Utilises the Navigation Mesh Obstacle.
- **enemyBasicBuilding** - a static enemy that blocks players. Can be destroyed.
    - Utilises the EnemyBuilding script.
    - Utilises the Navigation Mesh Obstacle.
- **enemyCore** - the main building required to be destroyed to win the game.
    - Utilises the EnemyBuilding script.
    - Utilises the Navigation Mesh Obstacle.
- **Projectile** - Line Renderer component, used for displaying the projectiles of long ranged units.
    - Utilises the DrawProjectile script.
    - Utilised by the enemyAttackTower, enemyArcher and playerArcher.
- **Rocks** - a basic obstacle that cannot be destroyed or moved.
    - Utilises the Navigation Mesh Obstacle.

UI prefab elements:

- **LostBackground** - the page that shows when a player loses a level
- **WinBackground** - the page that shows when a player wins a level
- **PauseMenuBundle** - a pause menu prefab. Contains both the UI controls tab and also the main pause menu buttons
- **UnitStatDisplay** - is used to display the destroyable objects' health bars to the user.
    - Utilises the UnitStatDisplay script.
    - Utilised by the enemy buildings, enemy units and player units.

UI/HUD prefab elements were used during the testing phases of the game for creating player interactable buildings that can spawn more units. See Further Work for more details. Action contains buttons for player interaction, which can be pressed to spawn more units, it utilises the Action script. ActionGraphics contains the icon used for spawning more units.

Scenes

Scenes are Unity's working environment. They are assets that each contain a part of the game. The following scenes can be found within Square Wars:

- **IntroVideo** - the scene that shows the introductory video and has a button to skip
- **Menu** - the main menu. Features the Start button and quit button.
- **MapMenu** - The scene that allows user to choose which level they wish to play.
- **Level (X)** - The game levels.
- **Hard Level** - the final level of the game.
- **CanvasTimeline** - an object that allows videos to be played.
- **NavMesh** - navigation mesh, defines traversable and non traversable areas of the map for the units. Provides units' pathfinding from point A to point B.

Scriptable Objects

ScriptableObjects are used to save large amounts of data independent of class instances. The use of ScriptableObjects reduces the memory usage of the game by avoiding copies of values. For instance, there is only one copy of base stats data for all Knights in the game. The following ScriptableObjects are found within Square Wars, they contain information such as the prefabs used for the units and buildings, base stats, and the type of the unit or building.

- Barracks*
- EnemyAttackTower
- EnemyBasicBuilding
- EnemyCore
- BarracksPlayerActions*
- Archer
- Healer
- Knight
- Warrior

* - used during the testing phases of the game for creating player interactable buildings that can spawn more units. See Further Work for more details.

Textures

The textures asset folder contains all the pictures used throughout the project, these textures can be laid over the UI or onto prefabs. The textures include:

- **PlayerIntro** - textures used to help introduce the player to various elements of the game.
- **UI** - textures used to create the user interface, such as logos, backgrounds, text, buttons.
- **axe, bow_and_arrow, healer_sign, sword** - used on the character prefabs to differentiate between the different characters during gameplay.
- **Square WarsIcon1** - game icon of the game's executable file (Windows) / application bundle (macOS).

TextMesh Pro

TextMeshPro package provides increased customisation variety to text editing. It is used in Win Lose screens for richer text visuals.

Video

The video directory contains the game introductory video, displayed to the player upon launching the game.

Future work

Here are the proposed recommendations for the future development of the game. As discussed earlier in the document, interactable player buildings were within the design considerations. As such, as part of the further development it is suggested that player buildings are introduced. These can be used to spawn additional units at the cost of a resource, which could be gathered on the map or as a reward for completing levels. Multiple player buildings could be made to accelerate the production of the army. To balance this, the enemy's army would also be growing, introducing more strategic thinking into the game. Following this, the addition of spawning abilities can lead to other actions available to the player, such as unit actions - various abilities that player's units can use to gain advantage over the enemy (faster movement and attack speed or increased heath/armour/damage/range).

Another future development to consider is unit selection prior to starting a level. The player gets to view the enemy's army setup and can buy a selection of units that they want to tackle the battle with. The cost would depend on the individual unit's strength and abilities. The maximum value of the units would be capped for each level, so that the player has to decide whether to adopt for strength in numbers or fewer but stronger units.

The package ToonyTinyPeople (Blacksmith, 2019) can be found along with other main folders in the project's hierarchy. The development considered animated characters for the different units. However, due to the time constraints, this feature was omitted in preference to the functionality of the game.

References

Blacksmith, P., 2019. *Toony Tiny RTS Demo* [Online]. Unity Asset Store. Available from: https://assetstore.unity.com/packages/3d/characters/toony-tiny-rts-demo-141705 [Accessed 20th December 2021]