UXG2165 State Pattern and AI

Assignment 2

Purpose

Complete a mini-game with a basic AI state machine, using design patterns like state and strategy patterns. Do simple scene management by adding and removing scenes. Implement a basic menu using Unity UI.

Topics Covered

- Design patterns: State pattern, Strategy pattern
- Scene management
- Unity UI

Introduction

- Start a new Unity **2D** project in **Unity 2021.3.8f1**.
- Import the **UXG2165A2 Unity package** into the project folders. You can do this by dragging the Unity package from your computer folder to the Assets folder in the **Project window** in Unity.
- Load **all scenes** in folder **Assignment2** (A2Scene, A2StartScreen, A2Level1, A2Level2) and **add** the scenes in Build Settings (File -> Build Settings -> Add Open Scenes). Make sure you see **all** the above scenes within the list of "**Scenes In Build**".
- You can now **close all other scenes except A2Scene**. Set the **screen resolution** of the Game window to **1920x1080**. Press Play to test the scene. Output should be displayed in the Game window. Press Play again to stop the program.

Requirements

- Unless otherwise stated, only edit lines from **`//Task <number> START`** until **`//Task <number> END`**.
- Follow the instructions in the **Submission Instructions** section. Export **only the required scripts and scenes** in a **Unity package**. Any **extra scripts, prefabs or scenes** submitted or **lines changed outside of the marked sections** will not be considered as part of the answer and will be **penalized**.
- Name your Unity package **uxg2165A2_<login_name>** (eg. uxg2165A2_jingying_goh). Replace all '.' in your login name with '_'.
- Submit the Unity package to moodle at the **Assignment 2** submission link on the moodle page.
- **Deadline for submission is 8th June 2023 2pm (1 hour into the Week 6 Thursday lesson). Late submissions will have their grades capped**.

Tasks

Starting from A2Scene, the game shows a start screen with 2 levels to select from. The prototype will not work correctly at the start of the assignment.

Remember to set screen resolution to 1920x1080.

1. Scene Management

Complete the **LoadScene** and **RemoveScene** functions in **GameController** to move between the start menu and game levels by adding and removing scenes from the game.

1a. Initialize Scene Controller

In **GameController.cs**, **LoadScene** function, the scene of the name given by aScene has been loaded using **LoadSceneAsync** function. Upon completion, **Scene Controllers** need to be **initialized** for the scene to function correctly.

Run Initialize function on the GameSceneController within the newly-loaded scene. To do this, get the newly-loaded scene using **GetSceneByName** function in **SceneManager**, then get an array of **root GameObjects** within the scene using the **GetRootGameObjects** function. **Loop** through the root game objects and set **currentSceneController** to any **GameSceneController** found. You can use **GetComponentInChild** function for this. The scene should contain **only 1 GameSceneController**. Run **Initialize** function on the GameSceneController found. Use the **this** keyword to pass the current GameController as input to the Initialize function.

1b. Remove Scene

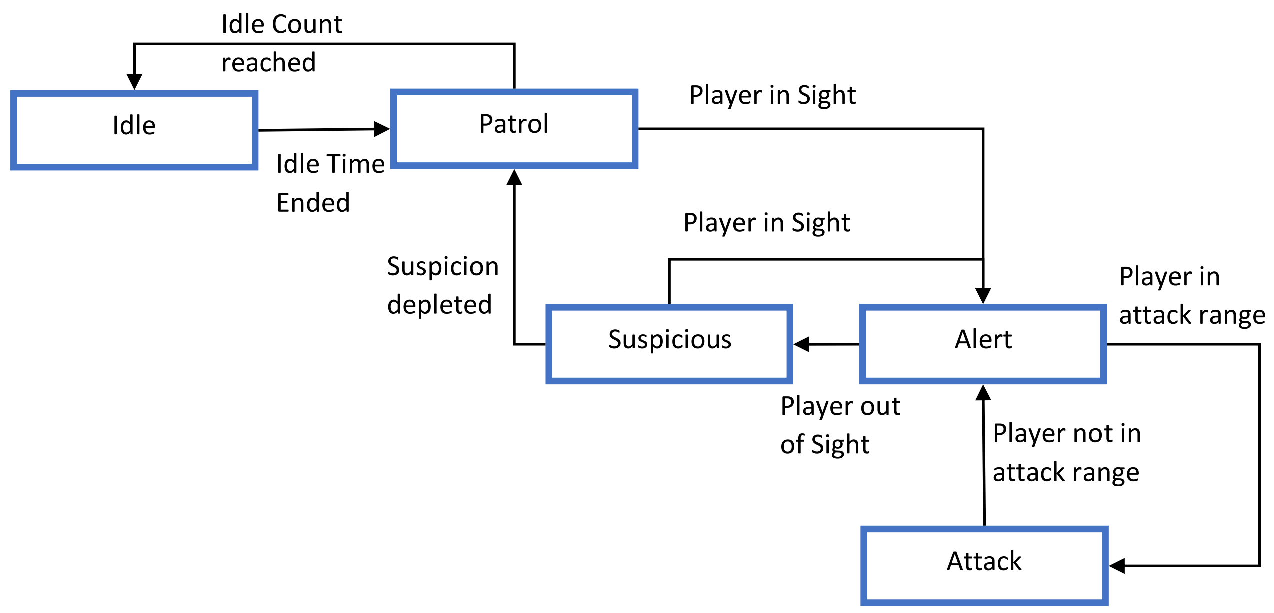
In **GameController.cs**, **RemoveScene** function, **unload** the scene of the name given by aScene. Use the **UnloadSceneAsync** function in **SceneManager**. There is no need to do any other actions before or after unloading the scene.

2. Set States and Do State Actions

At this point, it should be possible to start a game level. Currently only level 1 shows a level with enemies and collectibles. Level 1 shows a map in a top-down view, with a Player avatar controlled with WASD keys. To complete the level, the Player must collect all collectibles (rotating hexagons) within the level, to reveal the end point (rotating square). The level ends upon touching the end point. An additional rule is that if the player leaves the camera boundary for more than a specified number of seconds, it is game over.

The level contains Guards that will patrol the map by moving from waypoint to waypoint. If the Player within their line of sight, the Guard will start chasing the Player until the player is within attack range or they lose sight of the player. If the player is within attack range, the Guard will perform an attack, depending on what attack is assigned to the Guard. If they lose the player, the Guard will go into a Suspicious state where they would rotate on the spot looking for the player. If they find the player again, they will chase the player. If the player is not found, the Guard returns to its Patrol state.

The target State Machine for the enemy behaviour (after all tasks are completed) should be as below:



Initialize and do actions within implemented Enemy action states.

2a. Set Current State

In **EnemyScript.cs**, **SetCurrentState** function sets the current state of the enemy to the input **nextState**. Complete the function to set **currentState** to **nextState**.

2b. Set Initial State

In **EnemyScript.cs**, **Initialize** function, **initialize** the current state by setting it to the default state of all Guards, which is the **Patrol** state. (This function is called when a level is started or restarted.) You can set this by setting currentState to a new **EnemyStatePatrol** state. You will need to pass in the current EnemyScript as an input parameter to the constructor of EnemyStatePatrol.

2c. Update Guard Action

In **EnemyScript.cs**, **FixedUpdate** function, update the position, action and state of the Guard using the **DoActionUpdate** function of the **current state**. As this is done in FixedUpdate, the delta time to be passed as input parameter will be **Time.fixedDeltaTime**.

3. Make EnemyStateIdle State

Create an **EnemyStateIdle** state for the Guards in **EnemyState.cs**. You may refer to other existing states to see what can be done.

Idle state is triggered when **idleCount** (stored in EnemyScript) reaches or exceeds **idleInterval**. **idleCount** is increased every time a Guard **reaches a waypoint** when in **Patrol** state. In Idle state, the Guard stays on the spot and is unable to detect the player's proximity or line of sight. The Guard when stays in idle state for an amount of time defined by **idleTime** in **enemyScript**, before returning to **Patrol** state.

When Idle state is entered, the Guard's eye colour should be set to **black** and idleCount is **reset to 0** (using **SetIdleCount** function in EnemyScript).

During action updates, a **timer** should be used to keep track of how long the Guard is idle. Once the timer **reaches or exceeds idleTime**, change the **current state** of the enemy to **Patrol** state. This can be done using **SetCurrentState** function in EnemyScript.

No action needs to be done in **ReachTarget** function.

4. States and Transitions

Currently only the state transition from **Attack** state to **Alert** state is completed. Complete the enemy AI state machine by changing between the remaining enemy states. You may refer to **EnemyStateAttack** class in **EnemyState.cs** for an example of how to do state transitions.

4a. Patrol State Transitions

In **EnemyState.cs**, **EnemyStatePatrol** class, **DoActionUpdate** function, check if the player is within the enemy's sight, using **CheckPlayerWithinSight** function in EnemyScript. If player is **within range**, set **current state** to **EnemyStateAlert**. If the player is **NOT within range**, check if the guard is due for **idle state**, using **CheckIsIdle** function in EnemyScript and change to **EnemyStateIdle** state if true. When changing states, you can pass **enemyScript** as an input parameter.

4b. Alert State Transitions

In **EnemyState.cs**, **EnemyStateAlert** class, **DoActionUpdate** function, check if the player is within attack range using **CheckPlayerWithinAttackRange** function in EnemyScript, and change current state to **EnemyStateAttack** if true.

Note that **EnemyStateAlert** State also transitions to **EnemyStateSuspicious** when **ReachTarget** function is called. ReachTarget function is called in Suspicious state when the Guard reaches the **last seen position** of the player.

4c. Suspicious State Transitions

In **EnemyState.cs**, **EnemyStateSuspicious** class, **DoActionUpdate** function, some code is included where the enemy randomly selects a direction to rotate on the spot while a suspicion gauge reduces, but no state changes are currently included.

Modify this function to check if the player is **within sight or attack range**, using **CheckPlayerWithinSight** and **CheckPlayerWithinAttackRange** in EnemyScript. If true, go to **Alert** state. Otherwise, the enemy should rotate in a random direction on the spot **until suspicion reaches or goes below 0**, at which point it returns to **Patrol** state. Current suspicion value can be obtained using **GetSuspicion** function in EnemyScript.

5. Enemy Attack

In Attack state, Guards perform an attack according to the attack strategy assigned to the respective prefab, through an **AttackStrategy** script added to the game object as a component. In the **GuardShoot** prefab (which is a **prefab variant** of **Guard**), the **AttackShoot** script is added as a component.

5a. Do Attack

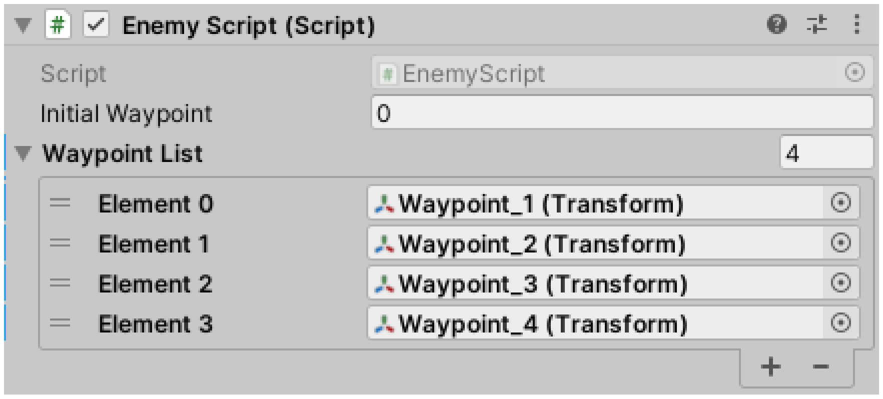
In **EnemyScript.cs**, **DoAttack** function is called in **EnemyStateAttack** when the enemy is able to attack. Perform the attack assigned to the enemy by calling **DoAttack** function in the **AttackStrategy** attached to the Guard game object. You should first **check** if an AttackStrategy has been attached to this gameObject.

5b. Bonus Attack Type (BONUS)

Complete **AttackBonus.cs** to create a new attack pattern. The class should inherit the **AttackStrategy** base class and **override** the **DoAttack** function for the attack to perform successfully. The attack should have a **delay** (so players have time to react), before whatever action occurs, as well as a maximum **distance**, so the attack has a range. The variables delay and distance are both public variables in **AttackStrategy**.

To facilitate this additional attack pattern, you may add up to **1 prefab** and **1 script** (similar to **Bullet** prefab and **BulletScript.cs**). Any prefab spawned by AttackBonus **MUST contain a class that inherits DamagerScript** (found in **AttackStrategy.cs**) for the level controller to work correctly. **DamagerScript** allows a **collider** on the prefab to be detected when it **touches the player collider to deal 1 damage**. DamagerScript also **automatically destroys** the prefab if it exists at the start of the level, and contains a virtual function **DoOnHit** that is called when the prefab is detected to hit the player.

After completing this new attack pattern, create a **prefab variant** of **Guard** named **GuardBonus**, and **add** the new **AttackBonus** script as a component. **Replace** the **GuardShoot_1** game object in **A2Level1** scene with your new GuardBonus, naming it **GuardBonus_1**. In the Inspector for GuardBonus_1 **EnemyScript**, add **Waypoint_1 to _4** to the **Waypoint List**, and set **Initial Waypoint** to **0**. Your EnemyScript should look like this:


Enemy Script (Script)

Script	EnemyScript
Initial Waypoint	0
Waypoint List	4
Element 0	Waypoint_1 (Transform)
Element 1	Waypoint_2 (Transform)
Element 2	Waypoint_3 (Transform)
Element 3	Waypoint_4 (Transform)

Please refer to **GuardShoot prefab, AttackShoot.cs, BulletScript.cs and Bullet prefab** for an example for how an attack pattern can be created.

A suggested attack pattern is an explosion with a circular radius around the Guard. There is no penalty or bonus for implemented the suggested explosion pattern or a different pattern altogether. Avoid implementing something too similar to the AttackShoot pattern. Use your own discretion. If your code is mostly copied from my existing code or your prefab is almost exactly the same as the Bullet prefab, your implementation might be too similar.

6. Pause Menu

Currently, the game cannot be **paused with the Escape key**. Complete the function and the menu such that the game can be paused and show a Pause menu with functioning buttons.

6a. Set Paused State

In **GameController.cs**, **SetPause** function is called whenever the game is **paused or unpaused**. Set the boolean **isPaused** according to the input aPause. Set **Time.timeScale** to 0 if paused, and 1f if unpaused. Set **pauseScreen gameObject** to active if paused and inactive if unpaused.

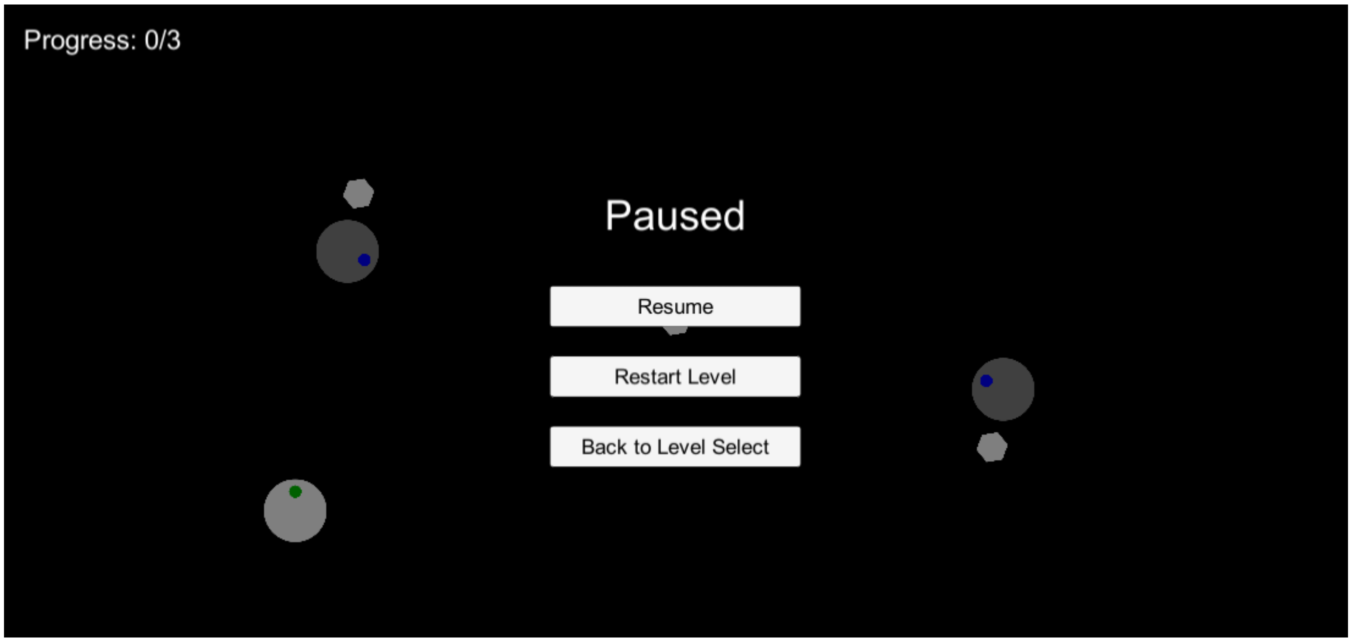
6b. Create Pause Menu

Create a simple **Pause Menu** in **A2Scene**, under the **GameHUDCanvas -> PauseScreen** object. The PauseScreen object should contain 4 elements:

1. A Text element showing the text "**Paused**".
2. A Button labeled "**Resume**", that calls the **SetPause** function in GameController to **unpause** the game.
3. A Button labeled "**Restart Level**", that calls the **RestartLevel** function in GameController to restart the level.
4. A Button labeled "**Back to Level Select**", that calls the **GoToLevelSelect** function in GameController to return to the level selection screen.

The 4 screen elements should be arranged in a **neat layout** in the **middle of the screen**. You are encouraged to use the **Vertical** or **Horizontal Layout Group** components in Unity to arrange the layout of the menu. You may refer to the **GameOverScreen** object for an example of how the menu can be arranged.

Here is a sample of how your Pause Menu should look.



7. Level 2 (BONUS)

Currently only Level 1 of the game is playable. **Create a Level 2** using the given prefabs and any new attack patterns created (if applicable). The level should be **completable** and all level elements should be **visible** within the screen at a screen resolution of 1920x1080.

Modify A2Level2 to create a functioning level. Levels must contain:

- A LevelController GameObject containing a LevelController component. Set the Scene Name to A2Level2 and Out Of Bounds Duration to your preferred duration (3 seconds by default). All other objects within the scene should be parented to this object.
- 1 Player prefab with Initial Position set to a StartPoint Transform.
- 1 StartPoint, an empty GameObject with no script attached.
- 1 EndPoint prefab.
- At least 1 Collectible prefab.
- A number of Guards. Recommend 2 to 4. Use GuardShoot prefab and GuardBonus prefab (if implemented). Remember to set their Waypoint List and Initial Waypoint (the index of the waypoint the Guard starts from).
- A number of Waypoints for Guards to patrol along. Use as many as needed (preferably not exceeding 10). Waypoints can be empty GameObjects with no scripts attached.

Please refer to A2Level1 to see how a functioning level can be constructed.

If your game objects are not appearing on the scene correctly, check their positions and make sure the Z depth of the objects are 0.

Submission Instructions

1. In Unity Editor, select **Assets -> Export Package…**
2. Select **only** the following scripts/objects/scenes:
    - GameController.cs
    - EnemyScript.cs
    - EnemyState.cs
    - A2Scene scene
    - AttackBonus.cs (ONLY if task 5b is attempted.)
    - Any additional prefab associated with the bonus attack strategy, if applicable (ONLY if task 5b is attempted.)
    - Any additional script associated with the bonus attack strategy, if applicable (ONLY if task 5b is attempted.)
    - GuardBonus prefab (ONLY if task 5b is attempted.)
    - A2Level1 scene (ONLY if task 5b is attempted and GuardShoot_1 has been replaced.)
    - A2Level2 scene (ONLY if task 7 is attempted.)

    During this step, if any scene or prefab is missing from the list, close the menu, save the project then reopen the Export Package menu.
3. Click Export…
4. Select your preferred output folder and name the Unity package.
5. Click Save.
6. Submit the exported Unity package to moodle.