UXG2165 Controls and Movement

Assignment 1

Purpose

Complete a mini-game with keyboard input for movement controls. Detect interactions with objects and update game logic. Create and destroy objects in the scene with or without object pooling.

Topics Covered

- Unity basics
- Input handling and player movement
- Colliders and triggers
- Object creation and destruction, object pooling

Introduction

- Start a new Unity **2D** project in **Unity 2020.3.17f1**.
- Import the **UXG2165A1 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 **A1Scene** (in folder **Assignment1**) and 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.

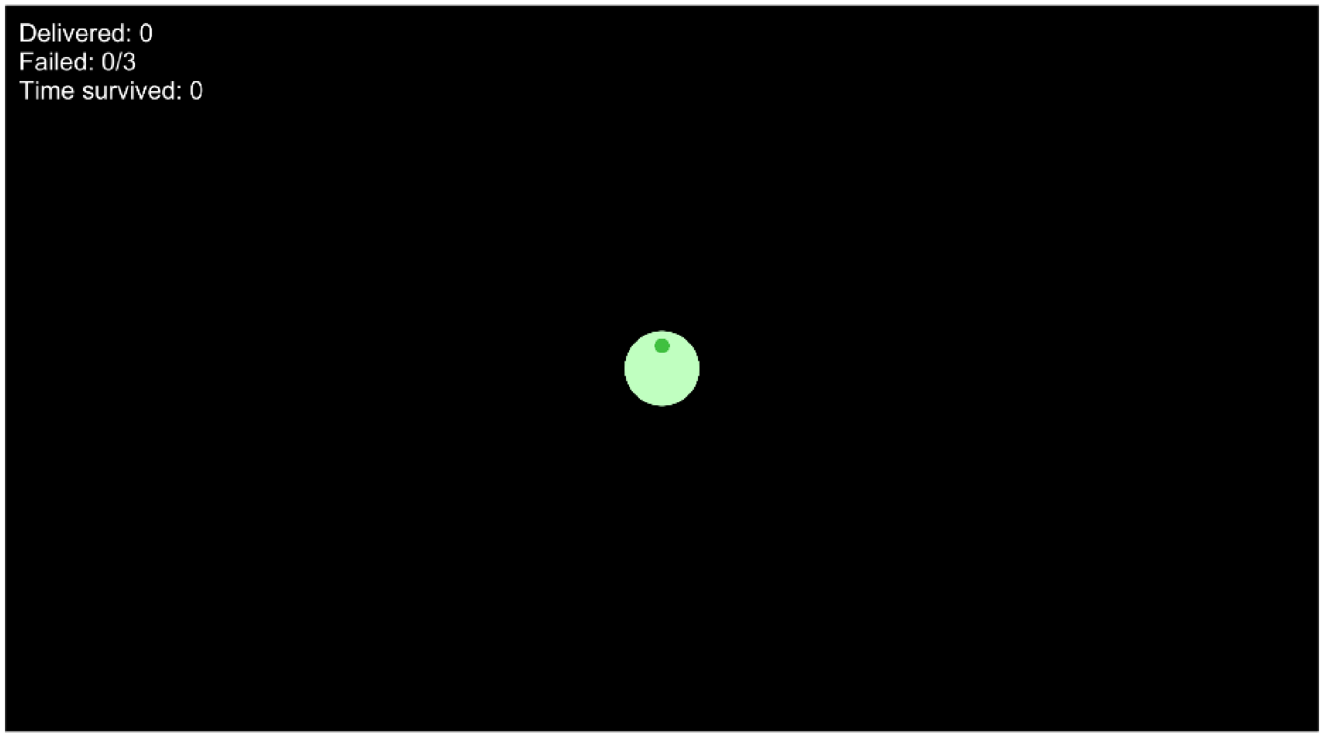
Requirements

- Only edit lines from **`//Task <number> START`** until **`//Task <number> END`**.
- Follow the instructions in the **Submission Instructions** section.
- Export **only the required scripts** 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 **uxg2165A1_<login_name>** (eg. uxg2165A1_jingying_goh).
- Replace all '.' in your login name with '_'.
- Submit the Unity package to moodle at the **Assignment 1** submission link on the moodle page.
- **Deadline for submission is 18th May 2023 2pm (1 hour into the Week 3 Thursday lesson). Late submissions will have their grades capped**.

Tasks

You are prototyping a simple game mechanic based on delivering packages. Packages are represented with small squares and their destinations are the larger squares. The prototype uses standard keyboard input (WASD or arrow keys) for movement controls and hovering the mouse over a package or destination highlights both the package and the destination. A package is picked up automatically when the player moves over it and is delivered when the player reaches its destination (highlighted in yellow) while holding the package. Only one package can be held at a time. The deliveries spawn at an increasing rate as the game proceeds, and ends when the player has failed a certain number of deliveries.

The prototype will not work correctly at the start of the assignment. The following screenshot shows how the game should look at the start:


Delivered: 0
Failed: 0/3
Time survived: 0

Remember to set screen resolution to 1920x1080 if your game screen looks different. The timer on the HUD should be working but the player circle would not move and no packages spawn.

1. Keyboard Input and Player Movement

This task is to detect and handle player keyboard input. Currently the player object (represented by the white circle) does not move or react to player input.

1a. Get movement direction

In **InputHandler.cs**, **FixedUpdate** function, there is a **Vector2 moveDir** set to Vector2.zero. Using Unity Input Manager, get the values of the **Horizontal** and **Vertical** axes and **set them to moveDir as x and y values**. You can get the axis values using **Input.GetAxis**.

You can check default input from Unity Input Manager in Edit -> Project Settings -> Input Manager. For this submission, no default values need to be changed. By default, Horizontal and Vertical axes use WASD or arrow keys.

1b. Move Player Object

In **PlayerMovement.cs**, **PlayerMovement** function, Vector2 **moveDir** is the direction the player will be moving in, and has been received from InputHandler and normalized (magnitude set to 1). Use the moveDir value to move the player object in the input direction with reference to **moveSpeed**. You are encouraged to use the **MovePosition** function in Rigidbody2D for this.

If your calculation results in unusually slow or fast movement, you may specify a new moveSpeed in a clear comment within the task area in the script (eg. //recommended moveSpeed = 10), and set your moveSpeed in the Player object in scene. If no new moveSpeed is specified, default speed of 10 will be used.

Take Note:

- Get the Rigidbody2D component in an object using the **GetComponent** function.
- **MovePosition** function **moves** the object to the given **world position**. Vector2 moveDir is the **direction** moved within this frame, **not the end position** of the object.
- **DoMoveDir** is called upon **FixedUpdate** (by InputHandler) so you can include **Time.fixedDeltaTime** in the movement calculation to scale the movement speed.
- **moveSpeed** needs to be included in the calculation as will.

1c. Move Player Object

In **PlayerMovement.cs**, **PlayerMovement** function, Vector2 **moveDir** is the direction the player will be moving in. Rotate the player object such that the dot on the object points in the direction the player is moving. This direction should only be set when the player is moving.

Take Note:

- When the player is not moving, **moveDir** will be **(0, 0)**, and have a **magnitude of 0**.
- The dot on the player object is positioned at the **top** of the circle.
- The **up** vector on a transform can be set to **rotate** the object such that the **top** of the object points in the **direction** up vector is set to.

1d. Get Input Keys

For this task, input detection will be done be detecting the keys directly instead of Input Manager. In **InputHandler.cs**, **Update** function, without using Unity Input Manager, **detect keyboard inpu**t for Yes and No responses and call the **DoYesAction** and **DoNoAction** functions respectively in **activeReceiver**. The input should be detected when the key is **pressed down**.

You may retrieve the keycodes assigned for Yes and No input from Dictionary inputList. By default, Yes input is Y key and No input is N, not case-sensitive.

Take Note:

- You can detect a key pressed within the frame using **Input.GetKeyDown**.
- **inputList** is a **Dictionary<InputType, KeyCode>**. InputType is an enum with YES and NO values, defined within InputHandle.cs. KeyCode is a Unity Engine enum containing values representing all detectable input.
- In this Dictionary, the key is an InputType and value is a KeyCode. KeyCode value for InputType.YES can be retrieved from the Dictionary with **inputList[InputType.YES]**.

1e. Yes and No Actions (Open-ended question)

Answer the following questions as a comment within the task area in InputHandler.cs script: **activeReceiver** is an **InputReceiver** interface object. Apart from PlayerMovement, what other class is an **InputReceiver**? What do **DoYesAction** and **DoNoAction** do in that class?

2. Object Creation and Destruction

Currently no delivery objects are being spawned in the game. This task is to create and destroy the prefabs as needed. For this section, Tasks 2b and 2d are Bonus tasks. You can attempt them, but if you are unable to complete them, you are encouraged to revert to the answers from Tasks 2a and 2c, as non-working code for Tasks 2b and 2d are likely to affect the subsequent tasks.

2a. Create GameObject from Prefab

In **GameController.cs**, **GetPrefab** function receives an **Object prefab** as input. Create a new GameObject from the given prefab object and return this new prefab.

2b. Get GameObject from Object Pool (Bonus)

Modify object creation code to use object pooling. The object pool is provided as a **Dictionary<string, List<GameObject>> objectPool**. The **key** of the dictionary should be a **string** representing the **name of the prefab**. The **value** is a **List<GameObject>** containing **all GameObjects of that prefab**.

If objectPool contains a key **matching the name of the prefab**, and the value paired with the key contains **at least 1 GameObject in it**, **remove** the first GameObject from the list and **return** it. If **no** pool or object is found, create a new GameObject.

You may modify the code for Task 2a to complete Task 2b.

2c. Destroy Prefab

In **GameController.cs**, **DestroyPrefab** function receives a **GameObject aObj** to be destroyed. Destroy the given GameObject.

2d. Add GameObject to Object Pool (Bonus)

Modify object destruction code to use object pooling. Check if **objectPool** contains a key matching the **name of the prefab** the input GameObject belongs to. Note that the name of the GameObject is set to **<prefabName>_<deliveryIndex>** (i.e. Package_1 or Dropoff_3, etc.). If the key is not found, **add** a new key-value pair to the Dictionary with a key that matches the **prefab name** (i.e. Package or Dropoff), where value is a **new GameObject list**.

Next, check if the list of GameObjects corresponding to the prefab name in the objectPool has **fewer** items than integer constant **POOL_MAX**. POOL_MAX is intended to be the **maximum size** of each object pool, to prevent the lists from expanding infinitely. If **fewer**, **add** the GameObject to the pool, otherwise **destroy** it.

You may modify the code for Task 2c to complete Task 2d.

Take Note:

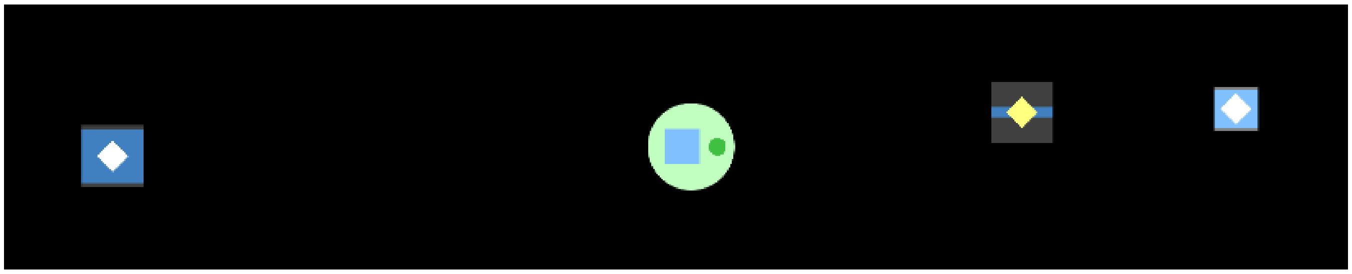
- You can split a string into an array of strings around a character using the Split function. (Eg. Splitting a string “AB_CD” by character ‘_’ results in an array containing “AB” and “CD”.)

3. Mouse Hover Delivery Hints

The game is intended to have a feature where the player can hover their mouse pointer over a package or dropoff point to show an icon on both the package and its dropoff point, to indicate which points are paired. This icon shows as a pair of white icons when showing this hint. The white icons should disappear when the mouse is not on either delivery point.

The same icon is also used to highlight the dropoff point of the current package held by the player, which shows in yellow, as long as the player is holding a package. The dropoff indicator yellow icon takes precedence over the white icon, meaning hovering the mouse over a dropoff point with the yellow indicator does not change the icon to white.

The screenshot below shows the player holding a package with the yellow icon showing its dropoff point, while the white icons show up on a package and dropoff pair due to the player having their mouse over either delivery point.



3a. Mouse Input Detection

GameController class has **SetDeliveryHint** and **ClearDeliveryHint** functions. In **DeliveryScript.cs**, write function(s) within the task area to detect mouse input and call the 2 functions as appropriate.

SetDeliveryHint should pass **deliveryIndex** to **gameController** when the player moves the mouse over a package or dropoff object. **ClearDeliveryHint** should pass **deliveryIndex** to **gameController** when the player stops hovering the mouse over the object.

DeliveryScript is inherited by **PackageScript** and **DropoffScript**, which are added to the **package** and **dropoff objects** respectively.

The answer should contain **2 MonoBehaviour functions** to detect mouse input, each calling one of the specified GameController functions.

3b. Update Icon

In **DeliveryScript.cs**, there is an **UpdateTargetIcon** function that is called by an **event** in GameController whenever either the current delivery index or hint index are changed. UpdateTargetIcon receives the indices of the **current delivery** and **current hint** as integers **currentDelivery** and **currentHint** respectively. The indicator icon on the delivery object is referenced within DeliveryScript as the **GameObject targetIcon** and the delivery index of the package the script is attached on is stored in **int deliveryIndex**.

Complete the function such that **targetIcon** is **set active** and set to a **yellow** colour when the current object has deliveryIndex equal to the current delivery. If the current object is **not** indicated as the delivery dropoff point, check if the object has deliveryIndex equal to the current hint. If yes, **show** the target icon and set its colour to **white**, otherwise **hide** the icon.

The colour of targetIcon can be set on the **SpriteRenderer** attached to the icon object, by setting the **color** property. Use **Color(1f, 1f, 0.5f)** for yellow and **Color(1f, 1f, 1f)** or **Color.white** for white.

4. Ending the Delivery

At this point, the packages and dropoff points should be appearing and disappearing when picked up, though dropping off the packages correctly does not yet update the score.

In **GameController.cs**, there is a **FinishDelivery** function which ends the player's current delivery as a success or a failure, updates the player's delivery status and checks the game over condition. You may need to look at this function to see what it does and how to use it.

4a. Finish Failed Delivery

In **DropoffScript.cs**, complete the **Update** function such that the delivery is completed and recorded as a **failure**.

4b. Finish Successful Delivery

In **DropoffScript.cs**, complete the **Update** function such that the delivery is completed and recorded as a **success**.

Submission Instructions

1. In Unity Editor, select **Assets -> Export Package…**
2. Select **only** the following scripts/objects:
   - InputHandler.cs
   - PlayerMovement.cs
   - DeliveryScript.cs
   - GameController.cs
   - DropoffScript.cs

   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.