Elinging on Time Documentation

HRK

May 11, 2020

Game Architecture:

1 Game Features **3**
- 1.1 Basic Control . . . 4
- 1.2 Game Elements . . . 4
- 1.3 Scene Flow . . . 5

2 Contents **7**
- 2.1 Event Handling Pattern . . . 7
- 2.2 Configuration Data Utilities & Game Initialisation . . . 13
- 2.3 Menu & Scene Management . . . 16
- 2.4 The Player . . . 21
- 2.5 Interactive Game Elements & Spawning . . . 29
- 2.6 Background Environment . . . 34
- 2.7 Utility Classes . . . 37
- 2.8 Indices and tables . . . 39

Elinging On Time was the birthday gift I have prepared for my girlfriend Elina Liu, and the name Elinging is a nickname I created for her avatar in the game. The game is based on an occasion where Elinging has an important seminar in the morning. However she got up late, thus needed to run like hell in order to catch up with the meeting. Since Elina and I were both Imperial College students, the game has been taken place in London as you will soon spot that there are many iconic elements in the game such as the double-decker bus and red telephone booth.

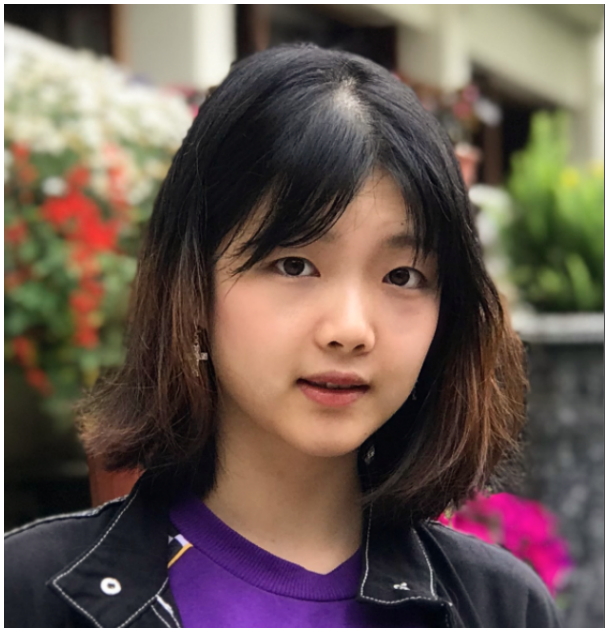


Fig. 1: The lovely girlfriend, Elina Liu

CHAPTER 1

Game Features

- Please see the Sample Play on YouTube to see the basic game features.

Fig. 1: Elingling On Time follows a pixel art style

1.1 Basic Control

WebGL The WebGL version of the game follows the most basic simple control of using *w* and *s* to control the avatar to move up and down.

Phone The Phone version of the game utilises the accelerometer of the phone, when the phone has been tilting over a certain degree, the player will go towards the up direction and vice versa.

1.2 Game Elements

Fig. 2: All game elements

Vehicle Two kinds of vehicles are on the road in the game, the beetles and the double-decker bus. When the player avatar crash with the vehicle, one health point will be deducted.

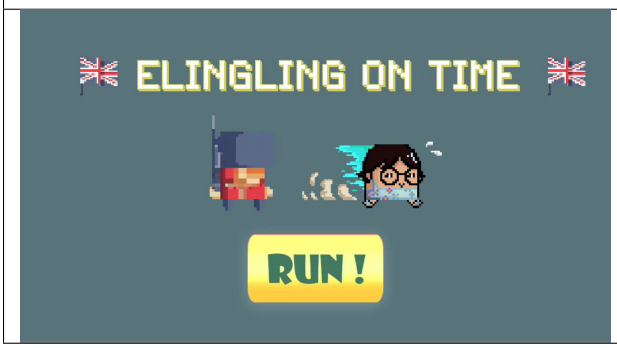
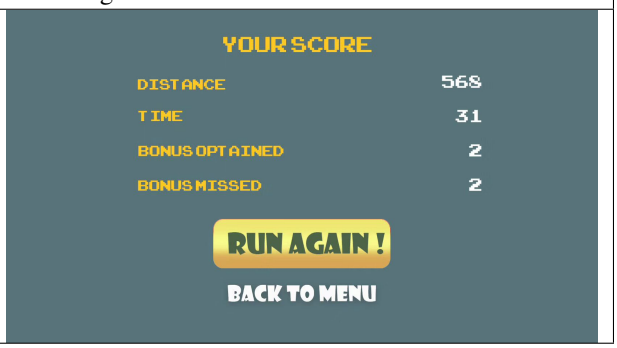
Soldier Due that Elinging violates the traffic rules, soldiers and on the road will start chasing down Elinging during the running, the speed of soldier has been set to slightly faster than the running speed of Elingling thus Elingling need to ride a bicycle in order to escape away from the soldiers.

Bicycle The Santander bicycle grants Elinging 3 seconds of speed buff. In addition, Elinging enters the invincible mode when riding the bicycle where she can basically knock away the vehicles on the road.

Soldiers get alerted and start chasing Elingling	Turning invincible after collecting the bicycle
	

1.3 Scene Flow

The game starts with a typical simple starting menu, and the player can check for the statistic after completing each set of the game.

Start Menu	Score Page
	

CHAPTER 2

Contents

2.1 Event Handling Pattern

The game design follows a simple observer pattern where event handlers respond when an event occurs. Unity Event Handling system has been based on the delegate type, which specifies a method signature and allows us to pass references to methods. The design pattern is shown in the system diagram below:

2.1.1 Event Manager

The centralised event manager script aims to manage connections between event listeners and event invokers. Therefore objects can interact without creating instances for them to know about each other. The core purpose of the event manager is to reduce the complexity of inflation as the program expands where more and more scripts need to know each other via instances. This idea can be shown in the plot below:

Rather than defining each invoker and corresponding listener, an `enum` of event names has been declared in a separate file to extract all the events and actions of the same data type:

```
public enum EventName {
    HealthChangedEvent,
    SpeedUpActivatedEvent,
    GameOverEvent,
    TimerChangedEvent,
}
```

Then corresponding classes of events have been declared in separate files such as `HealthChangedEvent`:

```
public class HealthChangedEvent : UnityEvent<float> { }
```

Note: For the ease of implementation, I declare all the event as one `float` argument event.

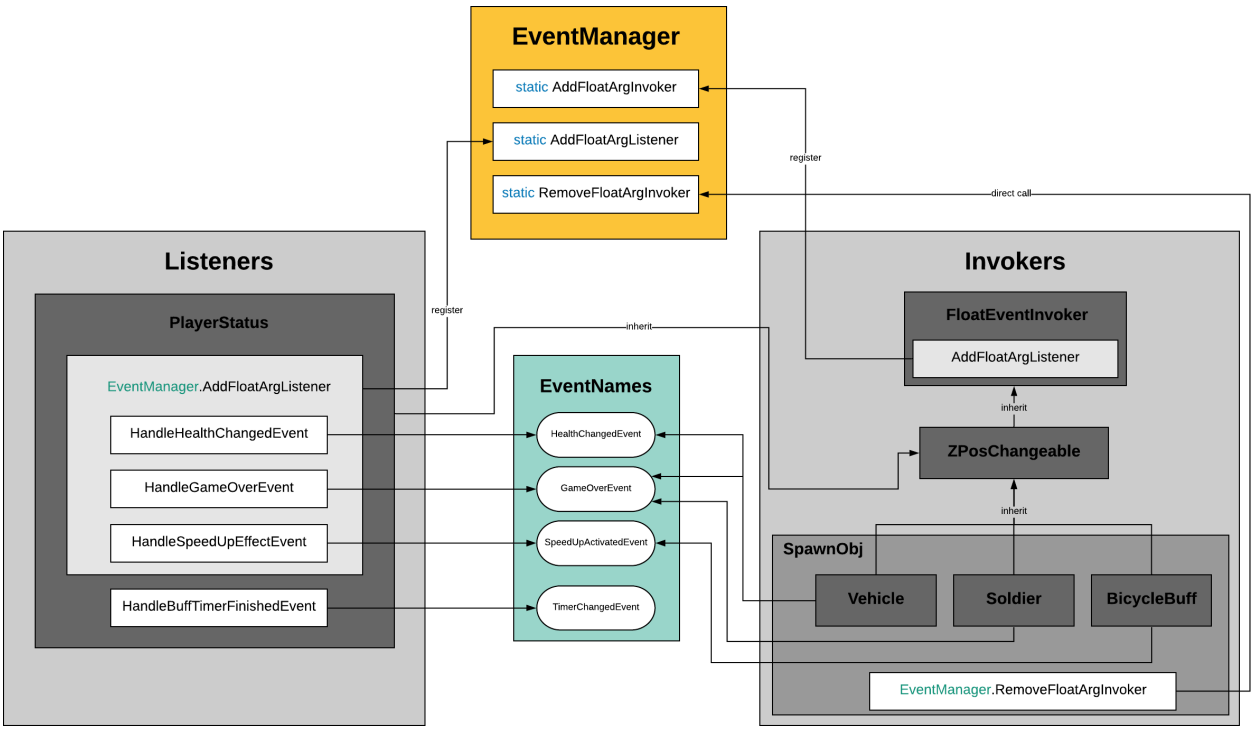
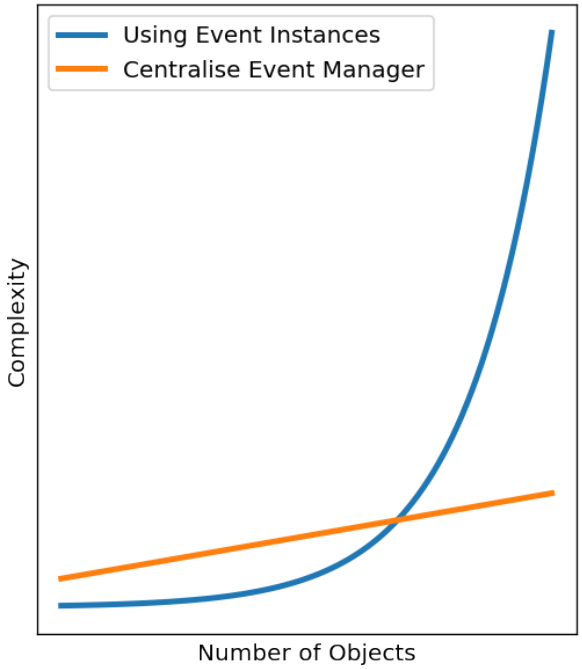


Fig. 1: System Diagram of Event Handling Design Pattern (*ctrl* + + to zoom in)



Then, in the `EventManager` class, lists of invokers and listeners have been declared because we might have multiple invokers for a particular event:

```
private static readonly Dictionary<EventName, List<FloatEventInvoker>> Invokers =
    new Dictionary<EventName, List<FloatEventInvoker>>();

private static readonly Dictionary<EventName, List<UnityAction<float>>> Listeners =
    new Dictionary<EventName, List<UnityAction<float>>>();
```

Then we declare the `Initalize()` method to be called elsewhere when initalising the game session.

We create empty lists for all the dictionary entries, `foreach` goes through each of those four values in `EventName` enumeration. If the dictionary doesn't have that name already, we create new lists for the invokers and listeners. If it already has the name, we clear the list because `Initialize()` method might be called multiple times as we play the game. We don't want to try to add a new list if the dictionary already does contain a particular name, because it throws an exception when trying to add something with the same key as the dictionary already has.

```
public static void Initialize() {
    foreach (EventName name in Enum.GetValues(typeof(EventName))) {
        if (!Invokers.ContainsKey(name)) {
            Invokers.Add(name, new List<FloatEventInvoker>());
            Listeners.Add(name, new List<UnityAction<float>>());
        } else {
            Invokers[name].Clear();
            Listeners[name].Clear();
        }
    }
}
```

After that, we declare the float argument handlers to be called in listeners and invokers:

```
// Adds the given invoker for the given event name with float argument
public static void AddFloatArgInvoker(EventName eventName, FloatEventInvoker invoker)
↪{
    // add listeners to new invoker and add new invoker to dictionary
    foreach (UnityAction<float> listener in Listeners[eventName]) {
        invoker.AddFloatArgListener(eventName, listener);
    }

    Invokers[eventName].Add(invoker);
}

// Adds the given listener for the given event name with float argument
public static void AddFloatArgListener(EventName eventName, UnityAction<float>␣
↪listener) {
    // add a listener to all invokers and add new listener to dictionary
    foreach (FloatEventInvoker invoker in Invokers[eventName]) {
        invoker.AddFloatArgListener(eventName, listener);
    }

    Listeners[eventName].Add(listener);
}
```

Don't forget to add removal functionality of the invoker when the invoker has been destroyed or no longer interacts with and scene objects to increase the code efficiency.

```
public static void RemoveFloatArgInvoker(EventName eventName, FloatEventInvoker␣
↪invoker) {
```

(continues on next page)

(continued from previous page)

```
    // remove invoker from dictionary
    Invokers[eventName].Remove(invoker);
}
```

2.1.2 Invokers

Instead of defining the invokers' properties separately, we firstly define a parent class of invokers `FloatEventInvoker`. Dictionary once again has been utilised to enable us to invoke more than one event. We couldn't just have a field for the `UnityEvent<float>`. We needed to have a dictionary fo `UnityEvents` so that classes can invoke multiple float events.The keys don't have to be strings but any data type, in this case, keys are enumerations and values are float unity events.

```
public class FloatEventInvoker : MonoBehaviour {
    protected Dictionary<EventName, UnityEvent<float>> UnityEvents =
        new Dictionary<EventName, UnityEvent<float>>();

    ...
}
```

Then we define the function that adds the given listener for the given event name:

```
public void AddFloatArgListener(EventName eventName, UnityAction<float> listener) {
    // only add listeners for supported events, `ContainsKey` check for the key
    if (UnityEvents.ContainsKey(eventName)) {
        // get the invoker by putting the key in between square brackets
        UnityEvents[eventName].AddListener(listener);
    }
}
```

Note: This method has been called in `EventManager` class when we we declare the float argument handlers to be called in listeners and invokers.

For the children and grandchildren classes of invokers, we use `Vehicle` class as an example, register for `HealthChangeEvent` and `GameOverEvent` in the `Start` method:

```
protected override void Start() {
    ...

    UnityEvents.Add(EventName.HealthChangedEvent, new HealthChangedEvent());
    EventManager.AddFloatArgInvoker(EventName.HealthChangedEvent, this);

    UnityEvents.Add(EventName.GameOverEvent, new GameOverEvent());
    EventManager.AddFloatArgInvoker(EventName.GameOverEvent, this);
}
```

These events have been triggered when colliding with the player, each time colliding with the player, deduct one health point, and when the health point equals 0, trigger the game over event:

```
protected override void OnTriggerEnter2D(Collider2D coll) {
    if (coll.gameObject.CompareTag("Player")) {
        UnityEvents[EventName.HealthChangedEvent].Invoke(1.0f);
```

(continues on next page)

(continued from previous page)

```
        // check for game over
        if (PlayerStatus.Health == 0) {
            UnityEvents[EventName.GameOverEvent].Invoke(0);
        }
    }

    base.OnTriggerEnter2D(coll);
}
```

Finally, don't forget to unregister the invoker using the `RemoveFloatArgInvoker` static method we have talked above, since we don't want the `Vehicle` script hanging around in that dictionary in the `EventManager` after the `Vehicle` game object itself was attached to gets destroyed.

```
protected override void OnDestroy() {
    EventManager.RemoveFloatArgInvoker(EventName.HealthChangedEvent, this);
    EventManager.RemoveFloatArgInvoker(EventName.GameOverEvent, this);
}
```

2.1.3 Listeners

In this game, there is only one current listener listening to all the events which are the `PlayerStatus` class. The listener is where we define the actual functionalities as an event handler, here we define the four event handling functions (the detailed functionality implementation will be discussed in separate sections):

```
// reduces health by the given damage
private void HandleHealthChangedEvent(float damage) {
    ...
}

// boost the player movement speed and turn invincible
private void HandleSpeedUpEffectEvent(float factor) {
    ...
}

// callback this function when buff timer finished
private void HandleBuffTimerFinishedEvent() {
    ...
}

// store the result and go to score page
private void HandleGameOverEvent(float unused) {
    ...
}
```

Then in the `Start` method, we register the event handling functions to the central event manager (the timer event handling follows a different pattern that would be described in below section):

```
void Start() {
    ...

    EventManager.AddFloatArgListener(EventName.HealthChangedEvent,
↪HandleHealthChangedEvent);
    EventManager.AddFloatArgListener(EventName.SpeedUpActivatedEvent,
↪HandleSpeedUpEffectEvent);
```

(continues on next page)

(continued from previous page)

```
    EventManager.AddFloatArgListener(EventName.GameOverEvent,
↪HandleGameOverEvent);
}
```

2.1.4 Timer Event Handling

The event handling pattern for the Customised Timer has been separated from the centralised event manager workflow. Logically the timer is a separate process, thus in a parallel system make it more modular and easier to debug. On the other hand, unlike the `FloatEventInvoker` where one or more float argument unity events could be triggered simultaneously, there should be only one kind of time pattern *time starts* > *time changes* > *time flows* > *time finishes* (as long as we are still in 3-dimensional world without applying Einstein's relativity) thus no need for going through a central event manager as no various kinds of time events need to be flexibly manipulated. In this scenario, back to the plot in the previous event manager session above, going through the event manager is actually more complex than just using timer instances.

In this case, the `CutomTimer` acts as the invoker, we first declare the instance of events in the script without using dictionaries and enumerations:

```
private readonly TimerChangedEvent _timerChangedEvent = new TimerChangedEvent();
private readonly TimerFinishedEvent _timerFinishedEvent = new TimerFinishedEvent();
```

Then we define the function that adds the given listener for the given event name:

```
// Adds the given event handler as a listener
public void AddTimerChangedEventListener(UnityAction<float> handler) {
    _timerChangedEvent.AddListener(handler);
}

// Adds the given event handler as a listener
public void AddTimerFinishedEventListener(UnityAction handler) {
    _timerFinishedEvent.AddListener(handler);
}
```

In the listener which is also the `PlayerStatus` class, we first declare the timer instance and access to the invoker class by getting the `CustomerTimer` component from the game object, we declare the callback event handler in the bottom and add listener for no argument event in the Start method:

```
private CustomTimer _buffTimer;

...

void Start() {
    _buffTimer = gameObject.AddComponent<CustomTimer>();
    _buffTimer.Duration = ConfigUtils.BuffDuration;
    _buffTimer.AddTimerFinishedEventListener(HandleBuffTimerFinishedEvent);

    ...
}

...

// callback this function when buff timer finished
private void HandleBuffTimerFinishedEvent() {
    ...
}
```

2.2 Configuration Data Utilities & Game Initialisation

As a professional practice of game development, we tend to separate all the configuration parameters used in the game in a centralised data management file, usually in *.csv* file thus we can tune the game directly in the separate data file.

Once again, we start with declaring an `enum` of data value names:

```
public enum ConfigDataValueName {
    VertSpeed,
    HoriSpeed,
    BuffFactor,
    BuffDuration,
    MinSpawnIntervalBuff,
    MaxSpawnIntervalBuff,
    MinSpawnIntervalObstacle,
    MaxSpawnIntervalObstacle,
    MinSpawnIntervalSoldier,
    MaxSpawnIntervalSoldier
}
```

After declaring the enumerations comes the main part where we create the `ConfigData` class for all the data manipulations. Firstly, we declare the variable to store the string of data path and declare the value dictionary:

```
private const string ConfigDataFileName = "ConfigData.csv";

private readonly Dictionary<ConfigDataValueName, float> _values =
    new Dictionary<ConfigDataValueName, float>();
```

Then we declare all the public properties to be utilised elsewhere:

```
// using expression-body style
public float VertSpeed                => _values[ConfigDataValueName.VertSpeed];
public float HoriSpeed                => _values[ConfigDataValueName.HoriSpeed];
public float BuffFactor               => _values[ConfigDataValueName.BuffFactor];
public float BuffDuration             => _values[ConfigDataValueName.BuffDuration];
public float MinSpawnIntervalBuff     => _values[ConfigDataValueName.
↪MinSpawnIntervalBuff];
public float MaxSpawnIntervalBuff     => _values[ConfigDataValueName.
↪MaxSpawnIntervalBuff];
public float MinSpawnIntervalObstacle => _values[ConfigDataValueName.
↪MinSpawnIntervalObstacle];
public float MaxSpawnIntervalObstacle => _values[ConfigDataValueName.
↪MaxSpawnIntervalObstacle];
public float MinSpawnIntervalSoldier  => _values[ConfigDataValueName.
↪MinSpawnIntervalSoldier];
public float MaxSpawnIntervalSoldier  => _values[ConfigDataValueName.
↪MaxSpawnIntervalSoldier];
```

After that, we define the main functionality of stream reading. The function should read configuration data from a file. If the file read fails, the object should contain default values for the configuration data. After reading the data, always remember to close the input file and check if the input is `null`. If we close a file that has never been opened, we will get a `NullReferenceException`.

```
public ConfigData() {
    StreamReader input = null;
```

(continues on next page)

(continued from previous page)

```
    try {
        // create stream reader object
        input = File.OpenText(Path.Combine(
            Application.streamingAssetsPath, ConfigDataFileName));

        // populate in names and values
        string currentLine = input.ReadLine();
        while (currentLine != null) {
            string[] tokens = currentLine.Split(',');
            ConfigDataValueName valueName = (ConfigDataValueName)Enum.Parse(
                typeof(ConfigDataValueName), tokens[0]);
            _values.Add(valueName, float.Parse(tokens[1]));
            currentLine = input.ReadLine();
        }
    } catch (Exception e) {
        Console.WriteLine(e);

        // set default values if something went wrong
        SetDefaultValues();
    } finally {
        // if close a file that never even opened, will get NullReferenceException
        if (input != null) {
            input.Close();
        }
    }
}
```

Warning: Beware that the `Application.streamingAssetsPath` variable corresponds to a certain directory *StreamingAssets* for the convenience to deduct redundant hard-coding. However, the *.cvs* file has to be in this directory or otherwise, will trigger the exception.

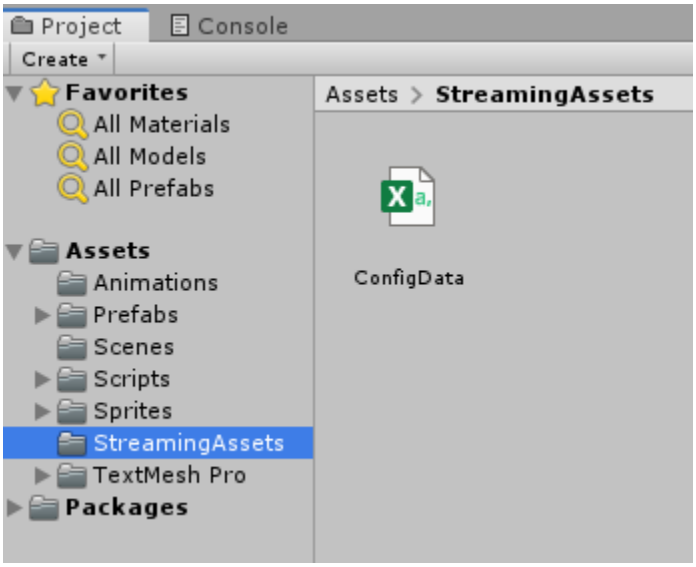


Fig. 2: screenshots of streaming assets path in unity

As a fallback plan if the stream reading fails, we should always declare default values:

```
private void SetDefaultValues() {
    _values.Clear();
    _values.Add(ConfigDataValueName.VertSpeed,                10.0f);
    _values.Add(ConfigDataValueName.HoriSpeed,                0.2f);
    _values.Add(ConfigDataValueName.BuffFactor,               3.0f);
    _values.Add(ConfigDataValueName.BuffDuration,             4.0f);
    _values.Add(ConfigDataValueName.MinSpawnIntervalBuff,     8.0f);
    _values.Add(ConfigDataValueName.MaxSpawnIntervalBuff,     12.0f);
    _values.Add(ConfigDataValueName.MinSpawnIntervalObstacle, 1.25f);
    _values.Add(ConfigDataValueName.MaxSpawnIntervalObstacle, 1.75f);
    _values.Add(ConfigDataValueName.MinSpawnIntervalSoldier,  12.0f);
    _values.Add(ConfigDataValueName.MaxSpawnIntervalSoldier,  20.0f);
}
```

After declaring the `ConfigData` class, we declare a `ConfigUtils` utility class to declare static variables of each of the parameters, Since these are utility classes we don't need to inherit from the `MonoBehaviour` unity class as we don't want to attach the class to game objects to instantiate it. We just want consumers to access the class directly.

```
public class ConfigUtils {
    private static ConfigData _configData;

    // using expression-body style
    public static float VertSpeed                => _configData.VertSpeed;
    public static float HoriSpeed                => _configData.HoriSpeed;
    public static float BuffFactor               => _configData.BuffFactor;
    public static float BuffDuration             => _configData.BuffDuration;
    public static float MinSpawnIntervalBuff     => _configData.MinSpawnIntervalBuff;
    public static float MaxSpawnIntervalBuff     => _configData.MaxSpawnIntervalBuff;
    public static float MinSpawnIntervalObstacle => _configData.
↪MinSpawnIntervalObstacle;
    public static float MaxSpawnIntervalObstacle => _configData.
↪MaxSpawnIntervalObstacle;
    public static float MinSpawnIntervalSoldier  => _configData.
↪MinSpawnIntervalSoldier;
    public static float MaxSpawnIntervalSoldier  => _configData.
↪MaxSpawnIntervalSoldier;

    // Initialise the config utils, run the initialisation in GameInitializer.cs
    public static void Initialize() {
        _configData = new ConfigData();
    }
}
```

Eventually, we call the `Initialize()` method in `GameInitializer` class where all the functionalities including `EventManager` functionalities from the previous section initialise for the current game session. The `GameInitializer` class should be the first script attached to the *Main Camera* in the Gamplay Scene:

```
public class GameInitializer : MonoBehaviour {
    // Awake is called before Start
    void Awake() {
        // initialise the screen utils
        ScreenUtils.Initialize();

        // initialise the config utils
        // Beware: build on phone device has problem reading streaming assets
        ConfigUtils.Initialize();
```

(continues on next page)

(continued from previous page)

```
        // initialise all event handling functionality
        EventManager.Initialize();
    }
}
```

Warning: Note that the phone build has problems with streaming assets reading functionalities, thus we just use the default values for phone builds.

2.3 Menu & Scene Management

The game consists of 4 occasions in 3 scenes: the main menu, the gameplay scene, the score page after that and a pause scene which is contained in the gameplay scene. As usual, we start the implementation with an `enum` stating all the cases we are caring:

```
public enum MenuName {
    MainMenu,
    Gameplay,
    ScorePage,
    Pause
}
```

Then we create a central scene manager to manage the navigation through the menu system:

2.3.1 Main Menu

```
public static class MenuManager {
    // Goes to the menu with the given name
    public static void GoToMenu(MenuName name) {
        switch (name) {
            case MenuName.MainMenu:
                // go to MainMenu scene
                SceneManager.LoadScene("01_MainMenu");
                break;
            case MenuName.Gameplay:
                // go to gameplay scene
                SceneManager.LoadScene("02_GamePlay");
                break;
            case MenuName.ScorePage:
                // go to score page
                SceneManager.LoadScene("03_ScorePage");
                break;
            case MenuName.Pause:
                // instantiate prefab
                Object.Instantiate(Resources.Load("PauseMenu"));
                break;
        }
    }
}
```

After that, we handle the detail functionalities in each scene. In the main menu scene, we have a button listening for the `OnClick` events for the main menu buttons:

```
public void HandlePlayButtonOnClickEvent() {
    MenuManager.GoToMenu(MenuName.Gameplay);
}
```

In the scene, create the button gameobject and attache the function in the *OnClick* event inspector:

Fig. 3: *Playbutton* in Unity Hierarchy

2.3.2 Score Page

The score page inherits score data from previous gameplay session. We first create a `GameSession` class to declare the static variables we are going to show in the score page:

```
public class GameSession : MonoBehaviour {
    public static int ScoreResult;
    public static int TimeResult;
    public static int BuffCollectedResult;
    public static int BuffMissedResult;

    ...
}
```

In order to let the `GameSession` object get inherited towards the score page, we have to utilise the `DontDestroyOnLoad` method. In addition, We should keep only one single `GameSession` object throughout the game, thus we need to detect and destroy extra `GameSession` object, we find the length of the list of `GameSession` objects, and if it's bigger than 1, that means the current one is the second thus we destroy it. Otherwise, we maintain it towards the next session.

```
void Awake() {
    // Find how many Game Status Objects are there
    // Beware this time using plural FindObjectsOfType<>() because there might be␣
↪multiple
```

(continues on next page)

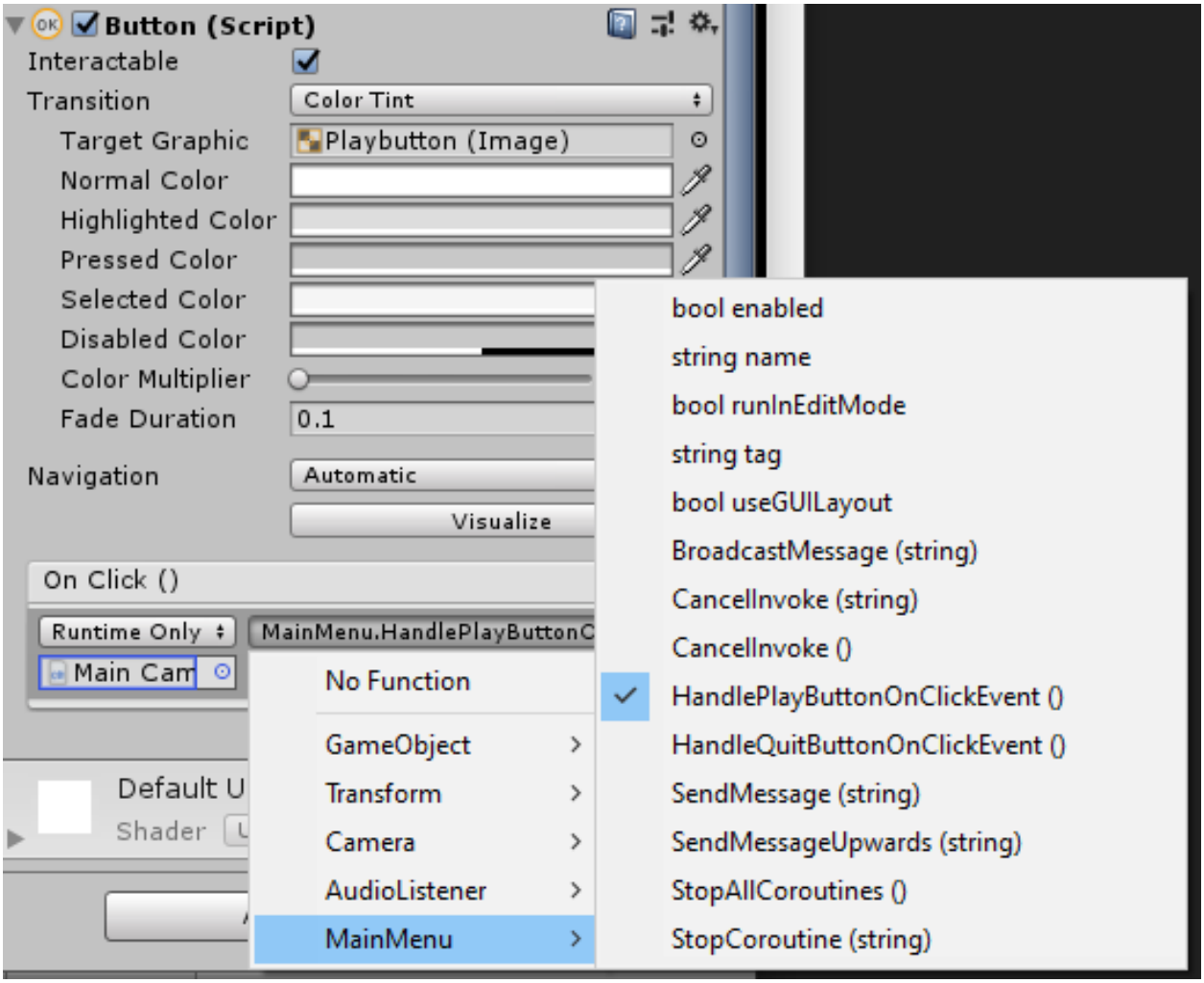


Fig. 4: assigning the `HandlePlayButtonOnClickEvent()` functionality in the inspector

(continued from previous page)

```
    int gameSessionsCount = FindObjectsOfType<GameSession>().Length;

    // gameSessionsCount more than one means this is the second game session
    if (gameSessionsCount > 1) {
        // prevent the issues destroying action come in bit later then activating the␣
↪game object
        gameObject.SetActive(false);
        Destroy(gameObject); // Destroy "yourself" referring to the current Game␣
↪Status
    } else {
        DontDestroyOnLoad(gameObject); // Maintain "yourself" if this is the first␣
↪Game Status
    }
}
```

Don't forget to destroy the current game status when restarting the game:

```
public void ResetGame() {
    Destroy(gameObject);
}
```

In the Gameplay Scene, when one game session has ended, the `HandleGameOverEvent` handler function will be called to update the score data that will be passed on to the next scene.

```
private void HandleGameOverEvent(float unused) {
    GameSession.ScoreResult         = Score;
    GameSession.TimeResult          = (int) TotalPlayTime;
    GameSession.BuffCollectedResult = BuffCollectedCount;
    GameSession.BuffMissedResult    = BuffMissedCount;

    MenuManager.GoToMenu(MenuName.ScorePage);
}
```

In the score page, we assign the *TextMeshProGUI* object will be assigned and updated:

```
public class ScorePage : MonoBehaviour {
    // ============================================================================
    // Field Variables
    // ============================================================================

    [SerializeField] private TextMeshProUGUI _textMeshProScore2;
    [SerializeField] private TextMeshProUGUI _textMeshTime;
    [SerializeField] private TextMeshProUGUI _textMeshProBuffCollected;
    [SerializeField] private TextMeshProUGUI _textMeshProBuffMissed;

    // ============================================================
    // Main Loop & MonoBehaviour Methods
    // ============================================================
    void Start() {
        _textMeshProScore2.text        = GameSession.ScoreResult.ToString();
        _textMeshTime.text             = GameSession.TimeResult.ToString();
        _textMeshProBuffCollected.text = GameSession.BuffCollectedResult.ToString();
        _textMeshProBuffMissed.text    = GameSession.BuffMissedResult.ToString();
    }

    ...
}
```

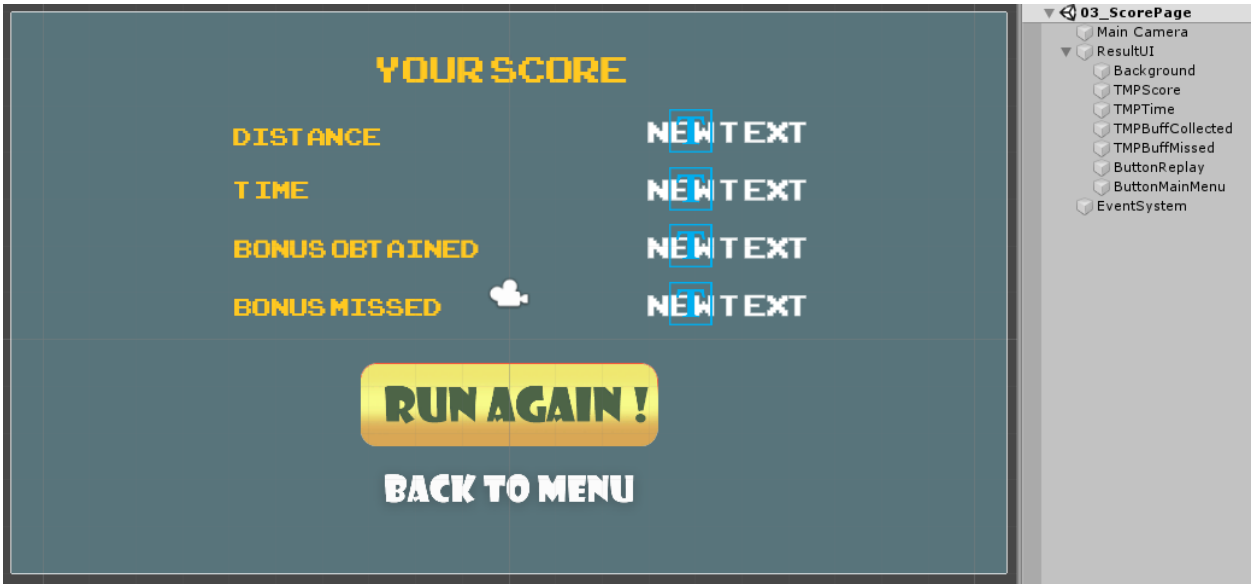
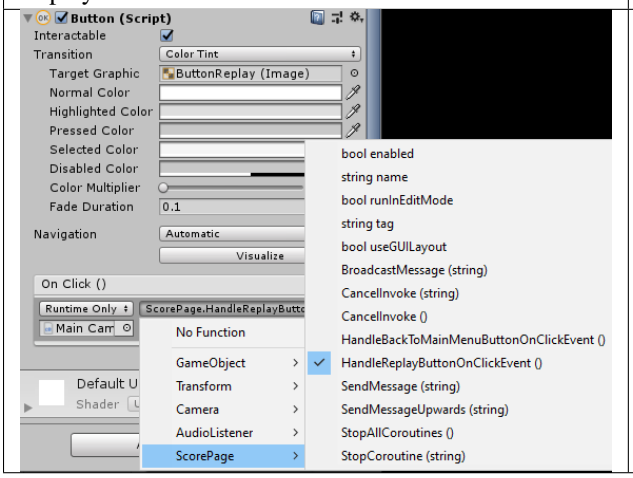
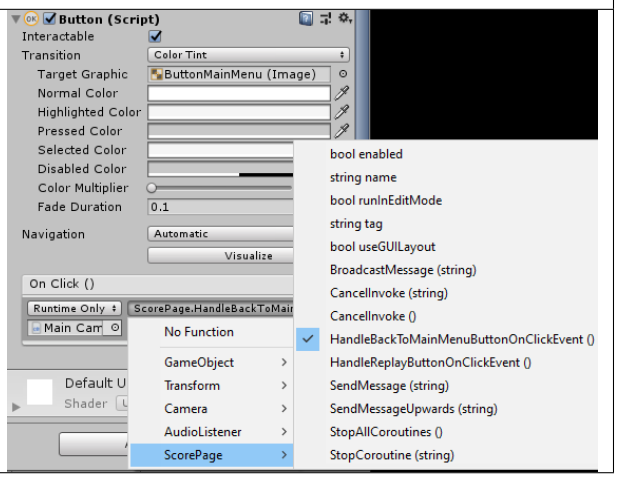


Fig. 5: *TMP* in Unity Hierarchy

Then we assign handler functions to button listening for the `OnClick` events for the score page in the inspector:

```
public void HandleReplayButtonOnClickEvent() {
    DestroyGameSession();
    MenuManager.GoToMenu(MenuName.Gameplay);
}

public void HandleBackToMainMenuButtonOnClickEvent() {
    DestroyGameSession();
    MenuManager.GoToMenu(MenuName.MainMenu);
}
```

Replay the Game	Back to Main Menu
	

Lastly, always remember to destroy the current game session when restarting the game to avoid conflicts in the next loop of game-flow.

```
private void DestroyGameSession() {
    if (FindObjectOfType<GameSession>() != null) {
        // destroy the current Game Session when restarting the game
        FindObjectOfType<GameSession>().ResetGame();
    }
}
```

2.4 The Player

For the purpose of an easier modular approach. The Player's implementation has been divided into two scripts:

- `PlayerControl` which solely handling player's horizontal and vertical movements.
- `PlayerStatus` which handling the player's properties including health, running distance score, invincibility and amount of buffs collected.

2.4.1 PlayerStatus

The manipulation of the status has primarily based on the event handling system, which has been discussed in the previous section. In this section, we focus more on the actual handler functions.

All player status properties have been declared as static field variable at the top and initialised in the `Start()` method:

```
// static fields to describe the player's current condition
public static float Health;
public static int   Score;
public static bool  Invincible;

// power buff collection and miss count
public static int BuffCollectedCount;
public static int BuffMissedCount;

...

void Start() {
    // initialise player health with 3 hearts
    Health = 3;

    // initialise player invincibility mode with false
    Invincible = false;

    BuffCollectedCount = 0;
    BuffMissedCount    = 0;
    TotalPlayTime      = 0;

    ...
}
```

Health & Death

After subscribing to the listening to the `HealthChangedEvent`, we define the actual health handler, the deduction will only trigger when the player is not in `Invincible` mode:

```
private void HandleHealthChangedEvent(float damage) {
    // only deduct health when the player is not invincible
    if (!Invincible) {
        // don't go below zero in health
        Health = Mathf.Max(0, Health - damage);
    }
}
```

Note: As you can see, the function is taking in a float argument which is the health point deducted when triggering the event. This number will be passed in the invoker when the event happens.

We also define the game over handler, which stores the result and go to score page:

```
private void HandleGameOverEvent(float unused) {
    GameSession.ScoreResult         = Score;
    GameSession.TimeResult          = (int) TotalPlayTime;
    GameSession.BuffCollectedResult = BuffCollectedCount;
    GameSession.BuffMissedResult    = BuffMissedCount;

    MenuManager.GoToMenu(MenuName.ScorePage);
}
```

The event triggers when colliding with a `Vehicle` object. In the `Vehicle` script we define the `OnTriggerEnter2D` function as follow. 1 health point will be deducted as the event fires, if the health point reaches 0, trigger the game over event. As you can see that the event invoking requires a float argument passing in which is corresponding to the float argument of `HandleHealthChangedEvent` function above:

```
protected override void OnTriggerEnter2D(Collider2D coll) {
    if (coll.gameObject.CompareTag("Player")) {
        // deduct the health by 1
        UnityEvents[EventName.HealthChangedEvent].Invoke(1.0f);

        // check for game over
        if (PlayerStatus.Health == 0) {
            UnityEvents[EventName.GameOverEvent].Invoke(0);
        }
    }

    base.OnTriggerEnter2D(coll);
}
```

Speed Up & Timer

The speed-up event triggers when colliding with the bicycle object. The implementation requires two functionalities: the speed up and timer to calculate whether the buff time has expired. These have been done by the following two handlers. The `HandleSpeedUpEffectEvent` handler boosts the player movement speed, turn the player into an invincible mode, change the sprite of the player to bicycle riding mode and switch the 2D collider's `isTrigger` property to false thus the player has a real collider volume to crash the vehicles away.

```
private void HandleSpeedUpEffectEvent(float factor) {
    // set invincibility mode to true thus player will not deduct health during␣
↪invincible mode
    Invincible = true;
```

(continues on next page)

(continued from previous page)

```
    // Player movement speed set to buffed state
    PlayerControl.HoriMvtState = HoriMvtState.Buffed;

    // change the sprite animation to riding bicycle in the animator
    _animator.SetBool("OnBicycle", Invincible);

    // set isTrigger property to false so the player can crash away the car and bus
    _capColl2D.isTrigger = false;

    // start the buff timer and exit buffed mode after buff duration
    _buffTimer.Run();
}
```

At the end of the function, we start the timer running, and after a buffer duration, we shut it down. We actualise this by subscribing to the timer invoker, and let the buff timer instance listen to the `HandleBuffTimerFinishedEvent` handler after the buff time has finished:

```
void Start() {
    ...

    // access to the invoker class by getting the CustomerTimer component from the␣
↪game object
    _buffTimer = gameObject.AddComponent<CustomTimer>();
    _buffTimer.Duration = ConfigUtils.BuffDuration;
    _buffTimer.AddTimerFinishedEventListener(HandleBuffTimerFinishedEvent);

    ...
}
```

The actual timer finish handler has been declared as follow, the player exit the invincible mode, the sprite has been changed back to running, the 2D collider's `isTrigger` property switches back to true thus the player will no longer have a collision volume, and will go through other objects when colliding and lastly change the horizontal moving state back to normal.

```
private void HandleBuffTimerFinishedEvent() {
    Invincible = false;

    // change the sprite animation to riding bicycle in the animator
    _animator.SetBool("OnBicycle", Invincible);

    // set isTrigger property back to true thus the player won't physically interact␣
↪with vehicles
    _capColl2D.isTrigger = true;

    // Player movement speed set back to normal state
    PlayerControl.HoriMvtState = HoriMvtState.Normal;
}
```

The event triggers when colliding with a `BicycleBuff` object. In the `BicycleBuff` script we define the `OnTriggerEnter2D` function as follow. We trigger the event by invoking the `SpeedUpActivatedEvent``and increment the ``BuffCollectedCount` property of the `PlayerStatus` class.

```
protected override void OnTriggerEnter2D(Collider2D coll) {
    if (coll.gameObject.CompareTag("Player")) {
```

(continues on next page)

(continued from previous page)

```
        // TODO: this float argument here is actually unused, make it useful
        UnityEvents[EventName.SpeedUpActivatedEvent].Invoke(ConfigUtils.BuffDuration);

        // add to buff collected count when the buff destroys due to being collected
        PlayerStatus.BuffCollectedCount++;

        // buff object disappears after the player collects it
        Destroy(gameObject);
    }
}
```

2.4.2 Sprites Manipulation

The pixel art style Elingling avatar extracted the most significant features from Elina's portrait, the glasses, the hairstyle and the overall cute looking.

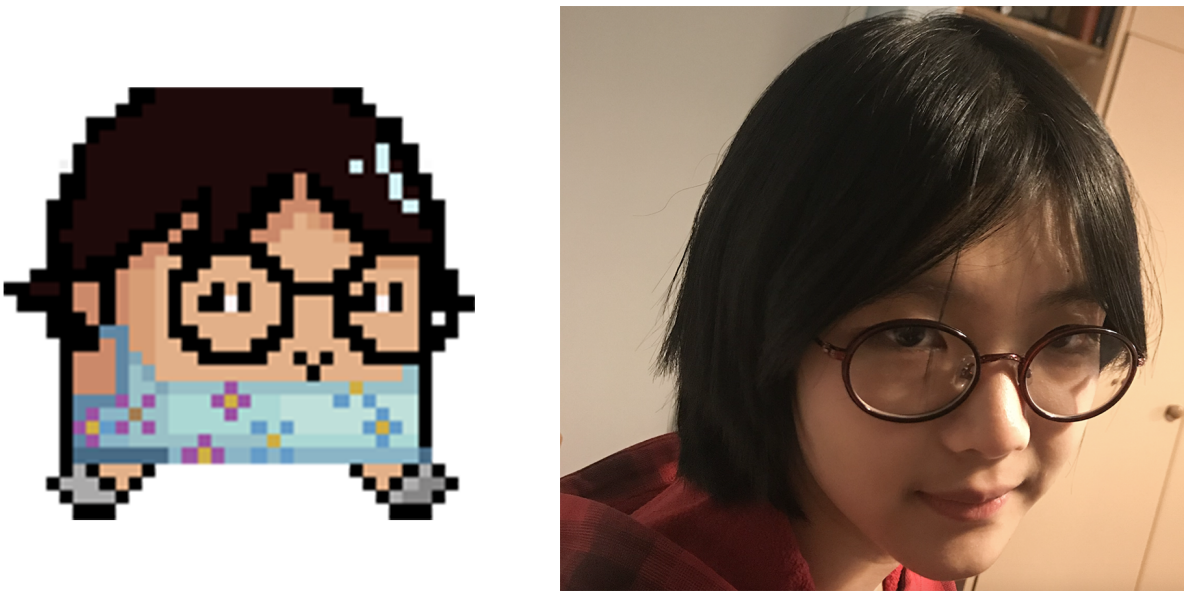


Fig. 6: Elingling vs Elina

When switching horizontal movement state, the sprite has to switch to corresponding ones. This has been accomplished using the Unity Animator. The transition logic between animations is simply actualised by manipulating the `OnBicycle` boolean variable which has been shown in the above functions.

Then the sprites can swtich between the following two animations correspondingly:

Running	Cycling

2.4.3 Player Control

The implementation of player control starts with defining enumerations of horizontal and vertical states in separate files:

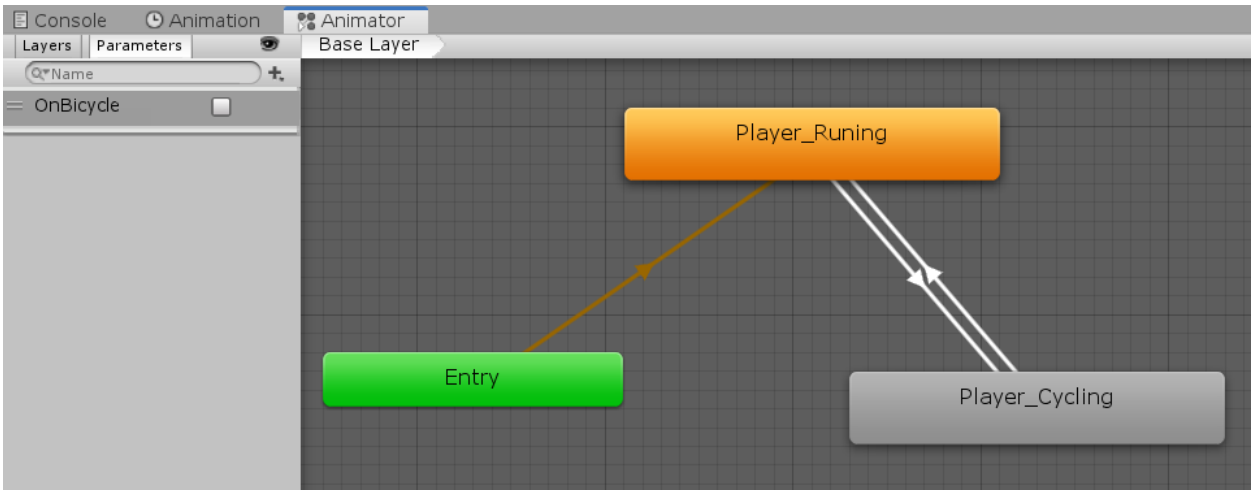


Fig. 7: Unity Animator

```
// An enumeration of the horizontal movement states
public enum HoriMvtState {
    Normal,
    Buffed
}

// An enumeration of the vertical movement states
public enum VertMvtState {
    MovingDown,
    MovingUp,
    Still
}
```

Then we initialise the vertical movement states of the player as `Still` and horizontal movement state as `Normal` respectively.

```
public static VertMvtState VertMvtState;
public static HoriMvtState HoriMvtState;

...

void Start() {
    ...

    // initialise the vertical movement state with still where the player keeps the␣
↪altitude
    VertMvtState = new VertMvtState();
    VertMvtState = VertMvtState.Still;

    // initialise the horizontal movement state with normal where the player keeps␣
↪steady speed
    HoriMvtState = new HoriMvtState();
    HoriMvtState = HoriMvtState.Normal;

    ...
}
```

Vertical Movement

The actual effect of switching between vertical movement states has been handled by `VertMvtHandler` function utilising the unity `transform.Translate()` built-in function

```
private void VertMvtHandler() {
    switch (VertMvtState) {
        case VertMvtState.MovingDown:
            transform.Translate(
                -Vector3.up * _vertSpeed * Time.deltaTime,
                Space.World);
            break;
        case VertMvtState.MovingUp:
            transform.Translate(
                Vector3.up * _vertSpeed * Time.deltaTime,
                Space.World);
            break;
        case VertMvtState.Still:
            // stop only the vertical speed by setting the vertical component to 0
            transform.Translate(
                Vector3.up * 0,
                Space.World);
            break;
        default:
            //transform.position = gameObject.transform.position;
            transform.Translate(
                Vector3.up * 0,
                Space.World);
            break;
    }
}
```

In order to prevent the player from moving outside the screen boundaries, we introduce a clamping position function:

```
private void CalculateClampedY() {
    // remember to add z pos, if using Vector2, the z pos will go back to 0
    // where the player will be behind the background
    Vector3 playerPos = transform.position;

    if (playerPos.y > _playerMvtUpperLimit || playerPos.y < _playerMvtLowerLimit) {
        playerPos.y = Mathf.Clamp(
            playerPos.y,
            _playerMvtLowerLimit,
            _playerMvtUpperLimit);

        transform.position = playerPos;
```

The actual player control follows two discipline: keyboard control or phone controls. The program will detect first which platform the game is currently running on. The Phone version of the game utilises the accelerometer of the phone. When the phone has been tilting over a certain degree, the player will go towards the up direction and vice versa while the keyboard version follows the most basic simple control of using *w* and *s* to control the avatar to move up and down.

```
void Update() {
    // using phone gyroscope & accelerometer input when running as phone apps
    if (Application.platform == RuntimePlatform.Android ||
        Application.platform == RuntimePlatform.IPhonePlayer) {
```

(continues on next page)

(continued from previous page)

```
        PhoneSensorControl();
    } else {
        // using keyboard vertical input axis when running on any other platform
        KeyboardControl();
    }

    VertMvtHandler();
    HoriMvtHandler();
    CalculateClampedY();
}

// ----- Phone Sensor Control -----
private void PhoneSensorControl() {
    if (Input.acceleration.y < -0.60) {
        VertMvtState = VertMvtState.MovingDown;
    } else if (Input.acceleration.y > -0.25) {
        VertMvtState = VertMvtState.MovingUp;
    } else {
        VertMvtState = VertMvtState.Still;
    }
}

// ----- Keyboard Control -----
private void KeyboardControl() {
    if (Input.GetKey(KeyCode.S)) {
        VertMvtState = VertMvtState.MovingDown;
    } else if (Input.GetKey(KeyCode.W)) {
        VertMvtState = VertMvtState.MovingUp;
    } else {
        VertMvtState = VertMvtState.Still;
    }
}
```

Horizontal Movement

The horizontal movement simply utilises the unity `transform.Translate()` built in function:

```
private void HoriMvtHandler() {
    switch (HoriMvtState) {
        case HoriMvtState.Normal:
            transform.Translate(Vector3.right * _horiSpeed);
            break;
        case HoriMvtState.Buffed:
            transform.Translate(Vector3.right * _horiSpeed * _buffFactor);
            break;
    }
}
```

The movement function has not much to discuss, what interesting is if the player runs towards the right of the screen, how to keep it always on the screen. This functionality has been accomplished in `EnvObjLoop` script which has been attached to the main camera. We create an instance of the player in the script and let the camera keep tracking of the player's position:

```
public class EnvObjLoop : MonoBehaviour {
    ...
```

(continues on next page)

(continued from previous page)

```
    [SerializeField] private GameObject _player;

    ...

    void Update() {
        // let the camera follow the player and locate the player at x = -5 of the␣
↪screen
        transform.position = new Vector3(
            _player.transform.position.x + 5,
            transform.position.y,
            transform.position.z);
    }

    ...
}
```

2.4.4 Z-Position of the Sprites

A weird scenario of smaller objects (player and soldier) run on top of the larger objects (vehicles) arises when the objects run into each other. In this case, the smaller object should be behind the larger object, however, it runs on top since the z-position of the smaller object is closer to the camera than the larger one. In order to tackle this problem, we add one child class in between the `FloatEventInvoker` and the `SpawnObj` -> the `ZPosChangeable` class.

In this class, we set all vehicles spawned to be originally at the same level of z-position and determine the player and soldier sprite's z-pos by comparing with the colliding vehicle objects, and if its position should look closer to the camera, we set its z-position closer to the camera than the vehicles.

```
protected virtual void OnTriggerEnter2D(Collider2D coll) {
    // only affect the player and the soldiers
    if (gameObject.name == "Player" || gameObject.name == "Soldier(Clone)") {
        // when the Player's or Soldier's center y-pos is lower than
        // - Bus's center y-pos by more than 1 unit
        // - Car's center y-pos by more than 0.25 units
        if ((coll.gameObject.name == "Bus(Clone)" &&
             transform.position.y < coll.transform.position.y - 1.0f) ||
            (coll.gameObject.name == "Car(Clone)" &&
             transform.position.y < coll.transform.position.y - 0.25f)) {
            // set z-pos of player to closer than the vehicles
            transform.position = new Vector3(transform.position.x, transform.position.
↪y, -3);
        }
    }
}
```

When the smaller object's collider leave the larger object's collider, we set its z-position back to original:

```
void OnTriggerExit2D(Collider2D coll) {
    // only affect the player and the soldiers
    if (gameObject.name == "Player" || gameObject.name == "Soldier(Clone)") {
        if (coll.gameObject.CompareTag("Vehicle")) {
            transform.position = new Vector3(transform.position.x, transform.position.
↪y, -1);
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.5 Interactive Game Elements & Spawning

We have four kinds of elements that need to be spawned: the `Vehicle`, the `Soldier`, the `BicycleBuff` and `EnvObj` each has a corresponding spawning class. The spawners and spawned objects inheritance hierarchy can be shown in the system diagram below, in each hierarchy the class has declared some `protected` method that defined essential functionality patterns that to be utilised and modified by the child classes:

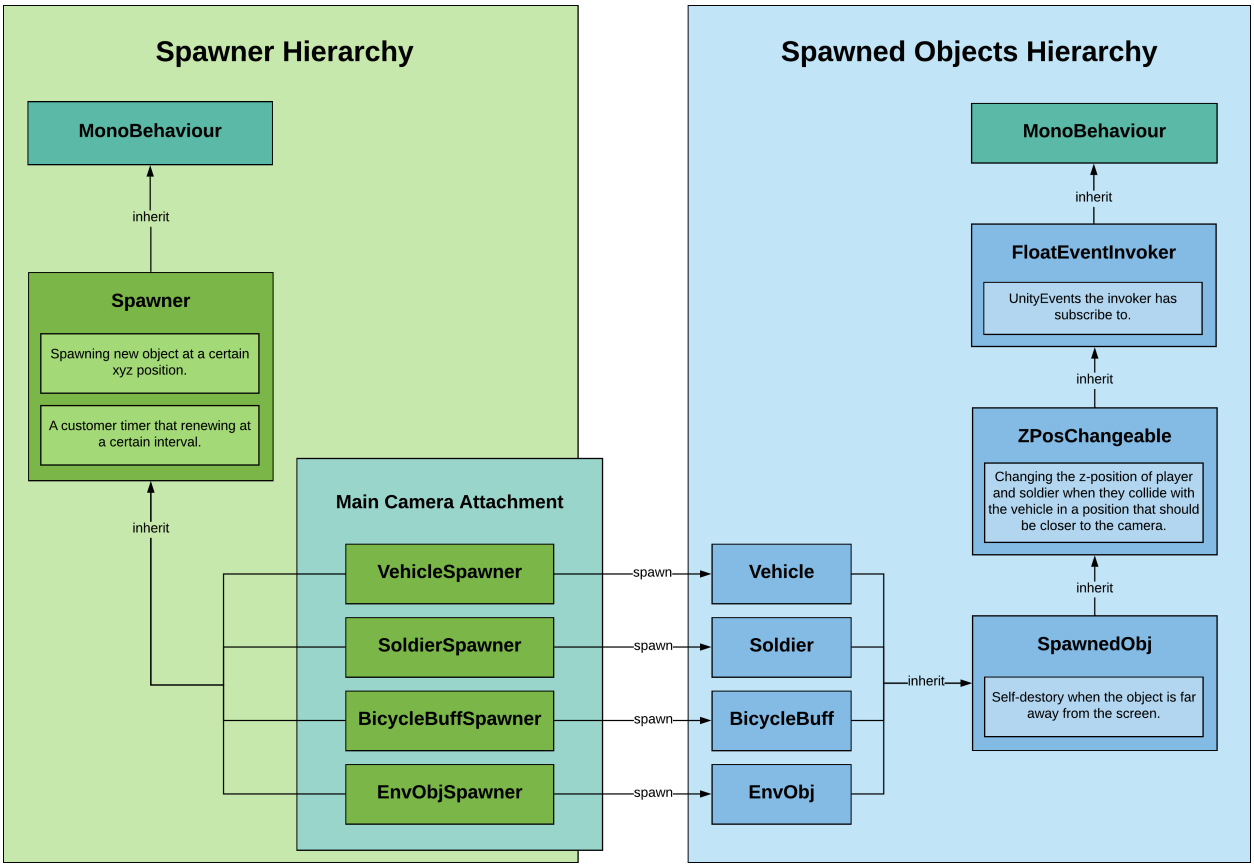
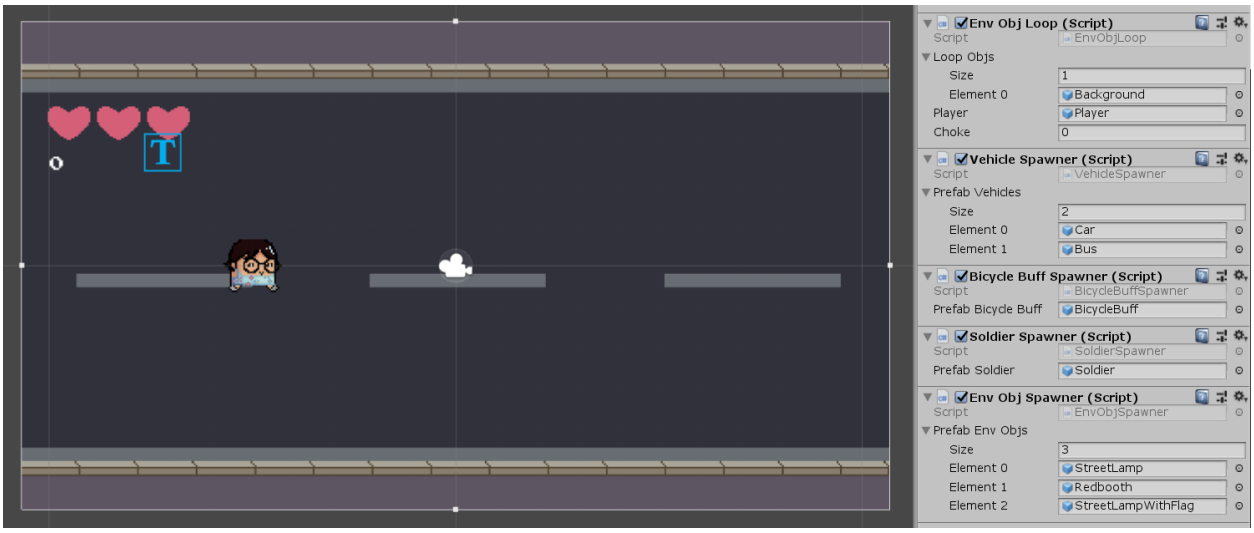


Fig. 8: System Diagram of Spawning Inheritance Hierarchy (*ctrl* + + to zoom in)

2.5.1 Spawners

The essence of the inheritance shown above is to maximise the reusability of functionalities of the same patterns. In the case of the spawners, all four end-user spawners are attached to the `Main Camera` which can be shown in the below screenshot:

They are inherited from the same parent spawner class where the xyz spawning positions and the interval for a custom timer to renew has been defined. These functionalities will be modified and reused in all four end-user spawners. The `Soldier` and `BicycleBuff` class have just made modifications on the prefab to instantiate, timer interval and



spawn position without changing the functionality pattern thus doesn't need to be discussed. Here we will only discuss the new things child classes have added when inheriting.

Vehicles

There are two kinds of vehicles, but they behave in the same way, thus we only change the sprite rather than changing the properties of the game object, thus we start with declaring the field variable:

```
[SerializeField] private GameObject[] _prefabVehicles = default;
```

Then we modify the original `Update` function to randomly choose which object to spawn. We also need to modify the timer since when the player is in a buffed state, she's running 3 times faster, thus vehicles need to be generated 3 times faster:

```
protected override void Update() {
    if (CustomTimer.Finished) {
        SpawnNewObj(_prefabVehicles[Random.Range(0, _prefabVehicles.Length)]);

        // when in buffed state, spawn the obj at 3 times frequency
        CustomTimer.Duration = PlayerControl.HoriMvtState == HoriMvtState.Buffed
            ? Random.Range(
                ConfigUtils.MinSpawnIntervalObstacle / 3,
                ConfigUtils.MaxSpawnIntervalObstacle / 3)
            : Random.Range(
                ConfigUtils.MinSpawnIntervalObstacle,
                ConfigUtils.MaxSpawnIntervalObstacle);
        CustomTimer.Run();
    }
}
```

The vehicles will encounter another issue of whether generating in the top lane or bottom lane, this will be handled in the `Vehicle` script that will be discussed down below.

Environmental Objects

The environmental objects will face the same issue of lane choice as the vehicle does. Since environmental objects are not interacting with the player, we turn to simplify the `EnvObj` class and squeeze all the functionalities in the environmental objects spawner script.

Same as the vehicle spawner, we declare a list of game objects as prefab pool, but this time we create two key-value pairs to store the random environmental object and lane choices:

```
// --------------- Serialized Cached References ---------------

[SerializeField] private GameObject[] _prefabEnvObjs = default;

// --------------- Config Params ---------------

private VehicleLane _vehicleLane;

private List<KeyValuePair<GameObject, float>> _envObjs =
    new List<KeyValuePair<GameObject, float>>();

private List<KeyValuePair<VehicleLane, float>> _laneChoices =
    new List<KeyValuePair<VehicleLane, float>>();
```

In the `Start` method, we assign each environmental object and lane choices with a certain probability of occurring. This has been actualised using the custom `Probability.RandomEventsWithProb` method which will be discussed in later sections:

```
protected override void Start() {
    _envObjs = new List<KeyValuePair<GameObject, float>> {
        new KeyValuePair<GameObject, float>(_prefabEnvObjs[0], 60),
        new KeyValuePair<GameObject, float>(_prefabEnvObjs[1], 20),
        new KeyValuePair<GameObject, float>(_prefabEnvObjs[2], 20),
    };

    _laneChoices = new List<KeyValuePair<VehicleLane, float>> {
        new KeyValuePair<VehicleLane, float>(VehicleLane.Top, 20),
        new KeyValuePair<VehicleLane, float>(VehicleLane.Bottom, 80),
    };

    base.Start();
}

protected override void Update() {
    if (CustomTimer.Finished) {
        // using reusable separate function from Probability Utility class
        SpawnNewObj(Probability.RandomEventsWithProb(_envObjs, 100));

        // when in buffed state, spawn the obj at 3 times frequency
        CustomTimer.Duration = 2;
        CustomTimer.Run();
    }
}
```

2.5.2 Spawned Objects

The `FloatEventInvoker` and `ZPosChangeable` classes have been discussed in previous sections. The most important functionality the `SpawnedObj` class has declared and can be applied to all children spawned objects is the

self destroy functionality where spawned objects destroy themselves when they are too far away from the left boundary of the screen. They will no longer be able to interact with any of the existing game objects in the screen, but they still occupy memory spaces, thus needs to be eliminated:

```
// when the obstacle is 1.5 screen width behind the player, destroy itself
// setting to 1.5 screen width to avoid bugs caused when deploying on phones
protected virtual void DestroySelf() {
    float xPosSelf   = gameObject.transform.position.x;
    float xPosPlayer = PlayerControl.PlayerTransform.position.x;

    // calculate the x distance between position of obstacle itself and the player
    if (xPosSelf - xPosPlayer < 3 * ScreenUtils.ScreenLeft) {
        Destroy(gameObject);
    }
}
```

Vehicles

The implementation of the `Vehicle` class starts with the lane choice:

```
public enum VehicleLane {
    Top,
    Bottom
}
```

```
public class Vehicle : SpawnedObj {
    private Rigidbody2D    _rb2D;
    private SpriteRenderer _spriteRenderer;

    ...

    private VehicleLane _vehicleLane;
```

The event trigger and self-destroy invoker removal functionalities have been discussed in previous sections, in this section, we only discuss the setting direction according to lane choice functionality.

We first choose the lane by utilising the built-in `Random.Range` function. Then if the lane choice is top, spawn on top lane range, otherwise, spawn on bottom lane range. We place the vehicle to the corresponding initial position and make the vehicle start moving by adding force onto the `rigidbody2D` component. Finally, we decide on the sprite direction.

```
private void SetLaneAndDirection() {
    int enumLen = System.Enum.GetNames(typeof(VehicleLane)).Length;
    _vehicleLane = (VehicleLane) Random.Range(0, enumLen);

    if (_vehicleLane == VehicleLane.Top) {
        transform.position = new Vector3(
            transform.position.x,
            Random.Range(_topLaneBot, _topLaneTop),
            transform.position.z);

        _rb2D.AddForce(new Vector2(100, 0)); // moving towards right

        // flip the sprite horizontally to make the vehicle face right
        _spriteRenderer.flipX = true;
    } else {
```

(continues on next page)

(continued from previous page)

```
        transform.position = new Vector3(
            transform.position.x,
            Random.Range(_botLaneBot, _botLaneTop),
            transform.position.z);

        // add force to initialise the vehicle movement
        _rb2D.AddForce(new Vector2(-200, 0)); // moving towards left

        // don't flip the sprite horizontally to so the vehicle faces left
        _spriteRenderer.flipX = false;
    }
}
```

Vehicles towards left without sprite flipping
Vehicles towards right with sprite flipping

Soldier

Apart from event handling functionalities, we have discussed in previous sections, the interesting part about `Soldier` class is the chasing functionality.

Initially, the soldier is standing still, as long as the x-position of the main character is bigger than that of the `Soldier` which means it's on the right of the `Soldier`, it will start the chasing:

```
private void StartChasing() {
    if (!_isRunning && PlayerControl.PlayerTransform.position.x > transform.position.
↪x) {
        _isRunning = true;
        _animator.SetBool("IsRunning", _isRunning);
    }
}
```

The actual chasing involves calculating the direction from the soldier towards the main character and normalise it. Then adding the force towards the normalised direction to consistently chasing down the player:

```
private void Chasing() {
    if (_isRunning) {
        _rb2D.velocity = Vector2.zero;

        // calculate direction to the player and moving towards it
        Vector2 direction = new Vector2(
            PlayerControl.PlayerTransform.position.x - transform.position.x,
            PlayerControl.PlayerTransform.position.y - transform.position.y);
        direction.Normalize(); // normalise it to make it a unity vector

        // because we set the speed to zero previously, adding the force with the␣
↪original
        // impulse force with the normalised direction we have just calculated will
        // make the game object moving at the same speed as before
        _rb2D.AddForce(direction * _impulseForce, ForceMode2D.Impulse);
    }
}
```

Analogous to the player animation switch, he sprite switching of the solider has been accomplished using the Unity Animator as well. The transition logic between animations is simply actualised by manipulating the `IsRunning` boolean variable which has been shown in the above functions.

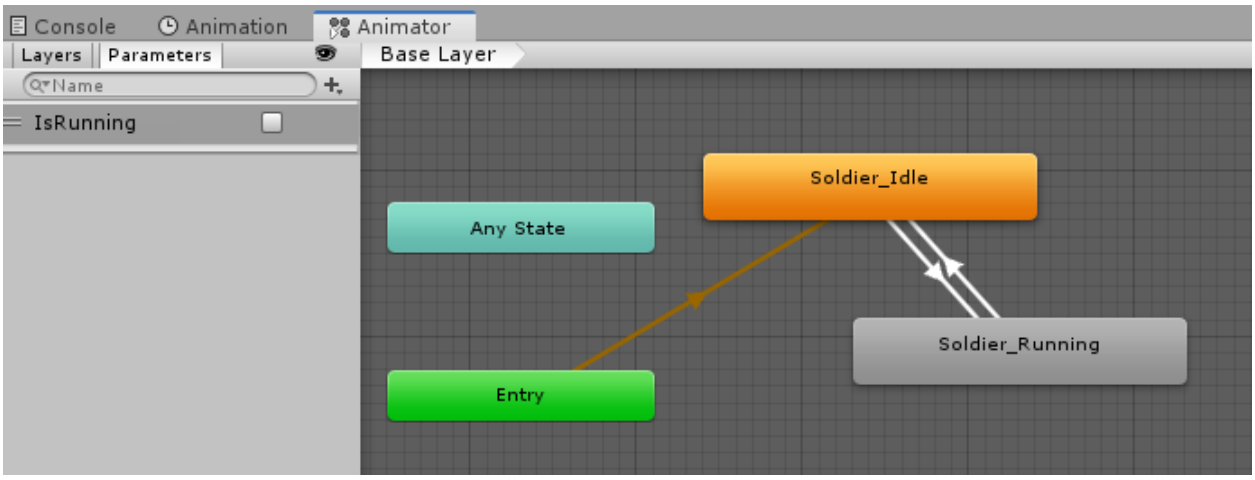


Fig. 9: Unity Animator

Soldier Chasing Elinging

2.6 Background Environment

Since the game is potentially an endless running game, it's crucial to provide an endless running pattern. It's not possible to pre-create the entire map which is long enough and let the player running on since this essentially is not endless and will also occupy huge memory spaces. In order to create smooth endless transitioning, we need to have a set of the same background element that the left side of it can connect with the right side and reuse this background element repeatedly when the player is running towards the right.

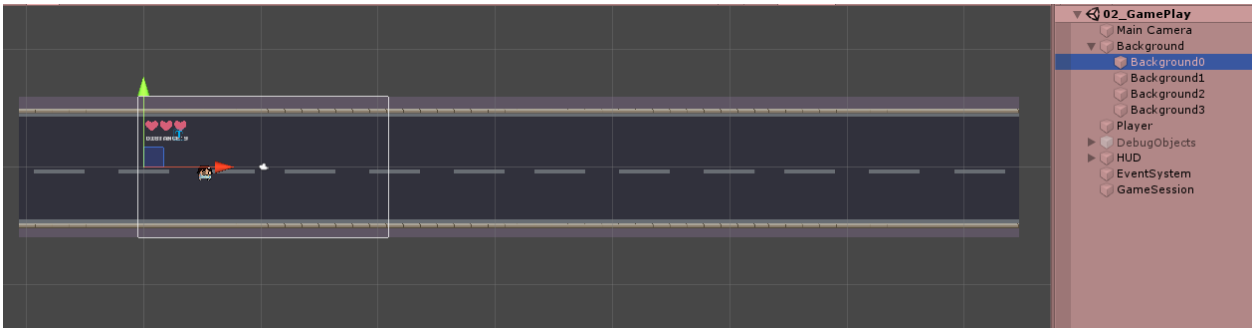


Fig. 10: Background Repetition

As you can see in the hierarchy tab in the above screenshot, when the game is running, we have 4 background elements in a row. Essentially, when the player is running towards the right, we take the last element which just left the screen the player just run over, and we move it to the right as you can see in the below screenshot, `background0` has now moved from the left which is behind the player to the right which is in front of the player.

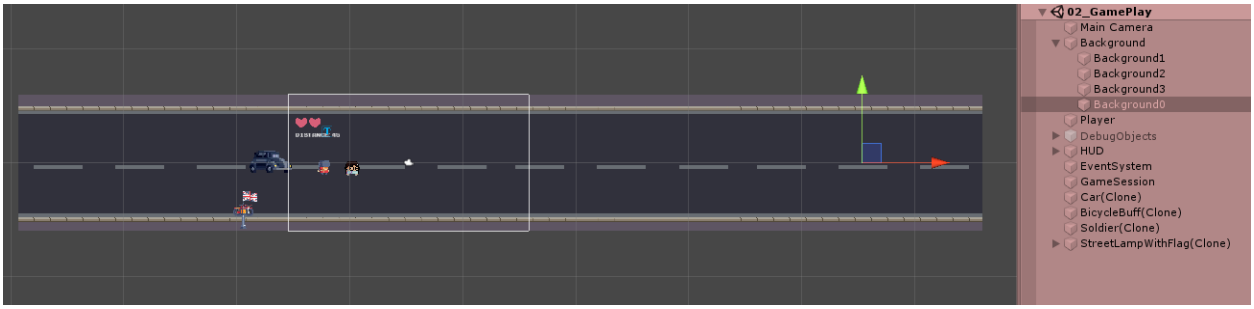


Fig. 11: Background Repetition 2

Then we just keep looping the same pattern and create a smoothly transitioning endless running pattern.

All of the above logic has been singly implemented in one file `EnvObjLoop`. We start with declaring all the background objects we want to loop through and store the screen boundary configuration parameter:

```
[SerializeField] private GameObject[] _loopObjs;

...

private Vector2 _screenBounds;
```

Then we create a function to load all the objects we want to loop to fill the screen. We firstly figure out the width of the current sprite, ten we calculate how many of the clones wee need to fill the width of the screen, after that we start instantiating the clones and add it as the child:

```
private void LoadChildObjects(GameObject obj) {
    // figure out the width of the current sprite by
    // fetching the horizontal value of the boundary box of the sprite
    float objectWidth = obj.GetComponent<SpriteRenderer>().bounds.size.x - Choke;

    // how many of the clones we need to make to fill the width of the screen
    // Mathf.Ceil makes sure we have enough objects to fill the width
    // "+ 2" are safety measure precautions for android devices
    int childrenNeeded = (int)Mathf.Ceil(_screenBounds.x * 2 / objectWidth) + 2;

    // clone the project objects so we have a mold as a reference
    GameObject clone = Instantiate(obj) as GameObject;

    // clone all child objects as reference (instead of just using obj) because
    // as we start adding children objects to obj those child objects will be cloned␣
↪as well
    // instead, we need a copy of obj to use for each child
    for (int i = 0; i <= childrenNeeded; i++) {
        GameObject c = Instantiate(clone) as GameObject;

        // set the clone as the child object of the parent object
        c.transform.SetParent(obj.transform);

        // space out these one after each other
        c.transform.position = new Vector3(
            objectWidth * i,
            transform.position.y,
            obj.transform.position.z);
```

(continues on next page)

(continued from previous page)

```
        c.name = obj.name + i;
    }

    Destroy(clone);
    Destroy(obj.GetComponent<SpriteRenderer>());
}
```

After the step of creating and fulfilling, we need to tackle the re-positioning. We first check if the camera has passed the edge of either the left-most child or the right-most child and re-position the children object accordingly.

Important: Beware that since the position of each child has been specified using the centre of the object when performing the calculations, we need to deduct or add half object with to reach the left-most or right-most boundary.

```
private void RepositionChildObjects(GameObject obj) {
    // be careful with `GetComponentsInChildren` rather than `GetComponentInChildren`
    Transform[] children = obj.GetComponentsInChildren<Transform>();

    // check if the camera extends past to the edge of either the first or the last␣
↪child
    // and re-position the children accordingly
    // check there are more than one child in the list
    if (children.Length > 1) {
        //Debug.Log(children.Length);

        // what we really care about is the first and the last child
        GameObject firstChild = children[1].gameObject; // [1] because [0] is the␣
↪parent object
        GameObject lastChild = children[children.Length - 1].gameObject;

        // transform position is at the centre of the object, so add or subtract half␣
↪the width
        float halfObjectWidth = lastChild.GetComponent<SpriteRenderer>().bounds.
↪extents.x - Choke;

        // detect if camera is exposing the right edge of the background element
        // "4 *" are safety measure precautions for android devices
        if (transform.position.x + 4 * _screenBounds.x > lastChild.transform.position.
↪x + halfObjectWidth) {
            // move our first child to the end of the list
            firstChild.transform.SetAsLastSibling();

            // set the position of the first child to be at right edge of the last␣
↪child object
            firstChild.transform.position = new Vector3(
                lastChild.transform.position.x + halfObjectWidth * 2,
                lastChild.transform.position.y,
                lastChild.transform.position.z);
        } else if (transform.position.x - _screenBounds.x < firstChild.transform.
↪position.x - halfObjectWidth) {
            // reverse of the above circumstance
            // move last child to the first of the list
            lastChild.transform.SetAsFirstSibling();

            // set the position of the last child to be at left edge of the first␣
↪child object
```

(continues on next page)

(continued from previous page)

```
            lastChild.transform.position = new Vector3(
                firstChild.transform.position.x - halfObjectWidth * 2,
                firstChild.transform.position.y,
                firstChild.transform.position.z);
        }
    }
}
```

2.7 Utility Classes

As a professional practice of software engineering, we tend to extract all utility classes which are not inheriting from the Unity `MonoBehaviour` and contain functionalities that could be repeatedly used in separate files. Then the scripts handling gameplay implementations could just import and use these files like external packages. In this game, apart from the `CustomTimer` class, we have another two utility classes serving these purposes in a similar pattern. Since these are just utility classes with static methods that we can directly utilise, I won't go into details how these functionalities have been implemented.

The first one is the `ScreenUtils` class which contains static properties of the coordinates of the 4 edges of the screen:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Provides screen utilities
public static class ScreenUtils {
    #region Fields

    // cached for efficient boundary checking
    private static float _screenLeft;
    private static float _screenRight;
    private static float _screenTop;
    private static float _screenBottom;

    #endregion

    #region Properties

    // Gets the left edge of the screen in world coordinates
    public static float ScreenLeft {
        get { return _screenLeft; }
    }

    // Gets the right edge of the screen in world coordinates
    public static float ScreenRight {
        get { return _screenRight; }
    }

    // Gets the top edge of the screen in world coordinates
    public static float ScreenTop {
        get { return _screenTop; }
    }
```

(continues on next page)

(continued from previous page)

```
    // Gets the bottom edge of the screen in world coordinates
    public static float ScreenBottom {
        get { return _screenBottom; }
    }

    #endregion

    #region Methods

    // Initialises the screen utilities
    public static void Initialize() {
        // save screen edges in world coordinates
        float screenZ = -Camera.main.transform.position.z;

        Vector3 lowerLeftCornerScreen  = new Vector3(0,            0,
↪screenZ);
        Vector3 upperRightCornerScreen = new Vector3(Screen.width, Screen.height,
↪screenZ);
        Vector3 lowerLeftCornerWorld   = Camera.main.
↪ScreenToWorldPoint(lowerLeftCornerScreen);
        Vector3 upperRightCornerWorld  = Camera.main.
↪ScreenToWorldPoint(upperRightCornerScreen);

        _screenLeft   = lowerLeftCornerWorld.x;
        _screenRight  = upperRightCornerWorld.x;
        _screenTop    = upperRightCornerWorld.y;
        _screenBottom = lowerLeftCornerWorld.y;
    }

    #endregion
}
```

Another one is the `Probability` class which helps to handle a set of events and assign a set of probabilities to each of them and let them randomly happen:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Probability {
    public static T RandomEventsWithProb<T>(
        List<KeyValuePair<T, float>> items, float totalProb) {
        // pick random value with in range the sum of all occurence probabilities
        float randomValue = Random.Range(0, totalProb);
        float cumulative  = 0;

        foreach (KeyValuePair<T, float> item in items) {
            cumulative += item.Value;
            if (randomValue < cumulative) {
                return item.Key;
            }
        }

        return default;
    }
}
```

2.8 Indices and tables

- genindex
- modindex
- search