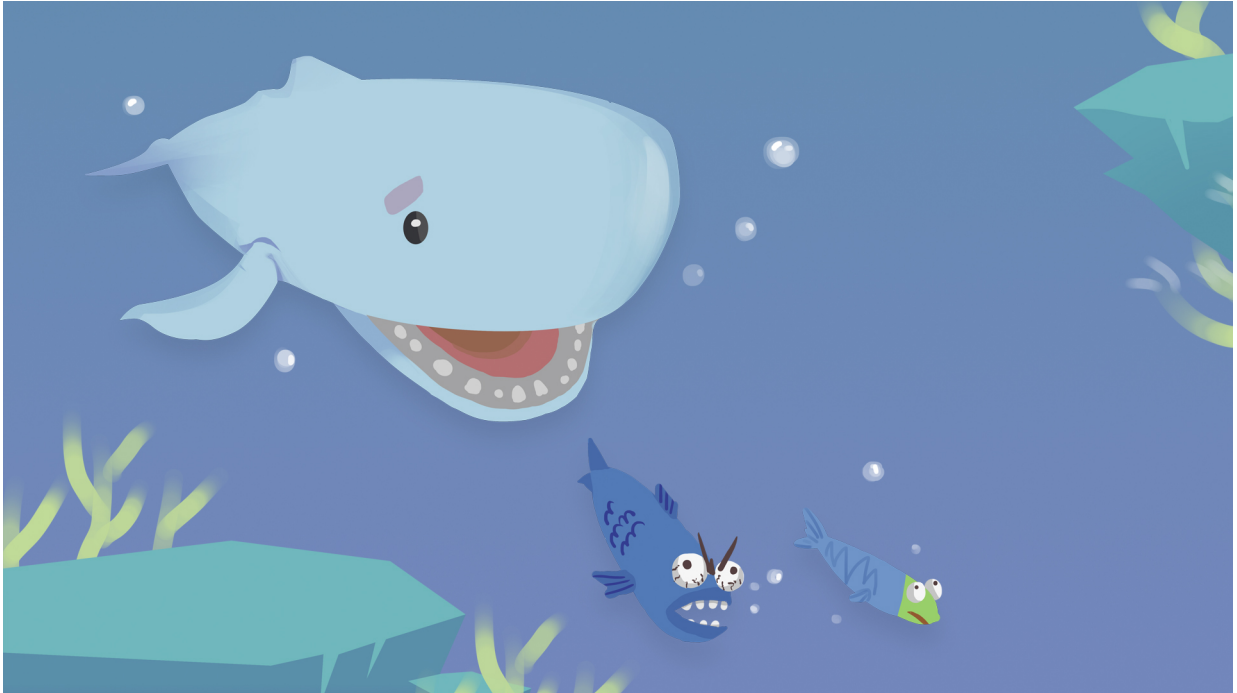

OpenSesame_Docs Documentation

Renke Huang

Dec 19, 2019

1	Video	3
2	About	5
2.1	The Project Members:	5
3	Background	7
4	Contents	9
4.1	Sensor Communication & Tuning	9
4.2	Player Control	13
4.3	Player Health & Score	19
4.4	Objects Spawn & Properties	24
4.5	Environment Objects	32
5	Indices and tables	35



CHAPTER 1

Video

Project overview

CHAPTER 2

About

This is the documentation for the group Open Sesame Project for the HCARD module, in March 2019.

The project is hosted on Github: <https://github.com/Antimony51122/Open-Sesame>

2.1 The Project Members:

Mechanical	<ul style="list-style-type: none">• Federica Spinola• Florence Wu• Vincent Ziliang Song
Sensor Communications	<ul style="list-style-type: none">• Junke Yao• George Rui Zhou
Software & UI Design	<ul style="list-style-type: none">• Renke Huang

With special thanks to: Dr Tomoki Arichi from King's College

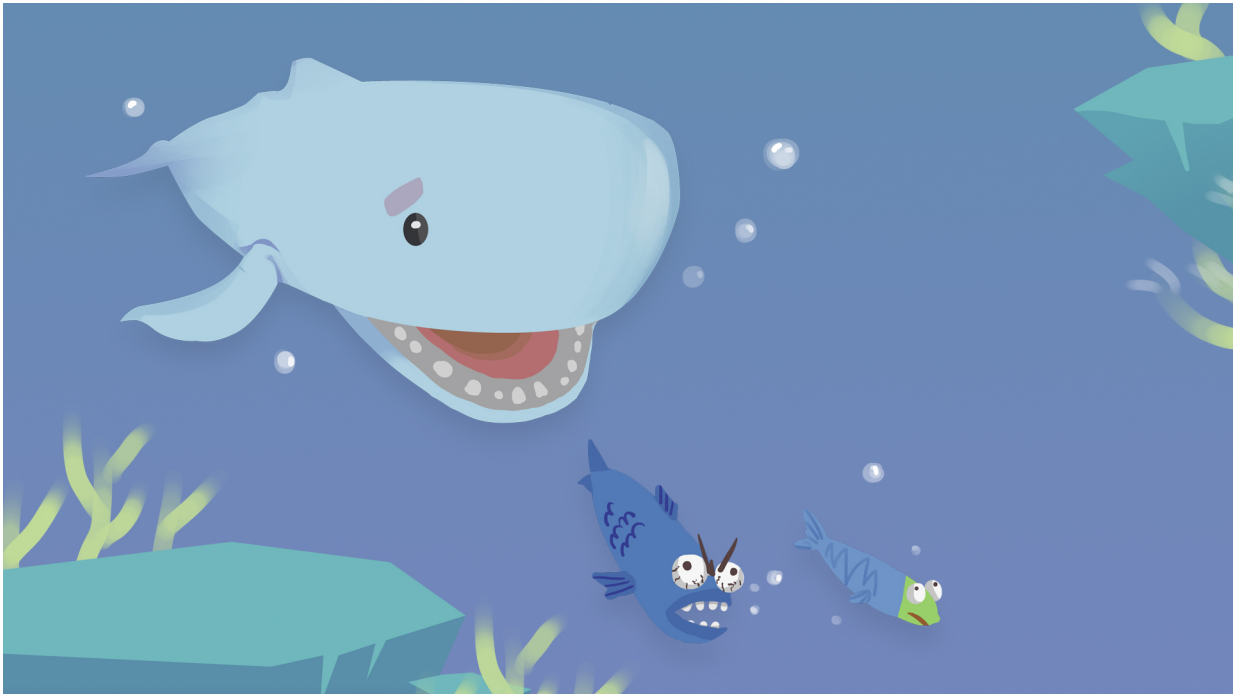
For enquiries on this documentation or source code, please contact rh3014@ic.ac.uk

CHAPTER 3

Background

Cerebral Palsy (CP) is a very common neurological disorder, with over 3 in 2000 babies being diagnosed each year worldwide. The hip is a complex ball-and-socket joint, which primary function is to support the body's weight and to perform the movement of the upper legs. The majority of patient with CP develop spasticity. About 35% of CP patients suffer from hips displacement, which is caused by the involuntary tightening of the muscle. This places an abnormal force around the hip joint and the neighbour muscles, which could lead to the dislocation of the hip. 1 in 3 patients are affected by hip dislocation and some may need surgery, yet little effective interventions have been developed for lower limb rehabilitation despite its potential of prevention. Conventional rehabilitation of the hip requires the assistance of a specialised physiotherapist and thus is usually carried out only few times a week due to high cost and non-motivating nature of the exercises.

In this project a low-cost device with an attractive and fun game interface is designed and developed to help improve lower-limb mobility by guiding the training of the patient's hip joint rotation. The following report describes and discusses in detail the mechanical, the electronic and the software components of the full system.

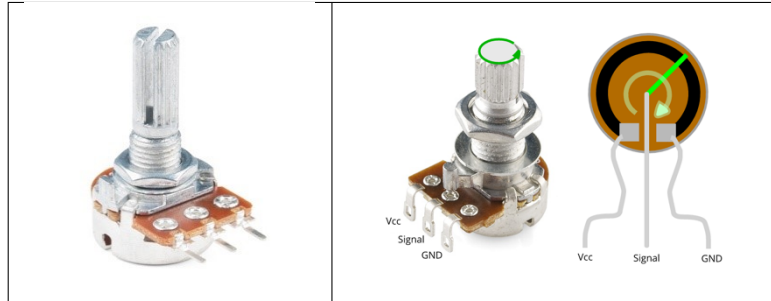


4.1 Sensor Communication & Tuning

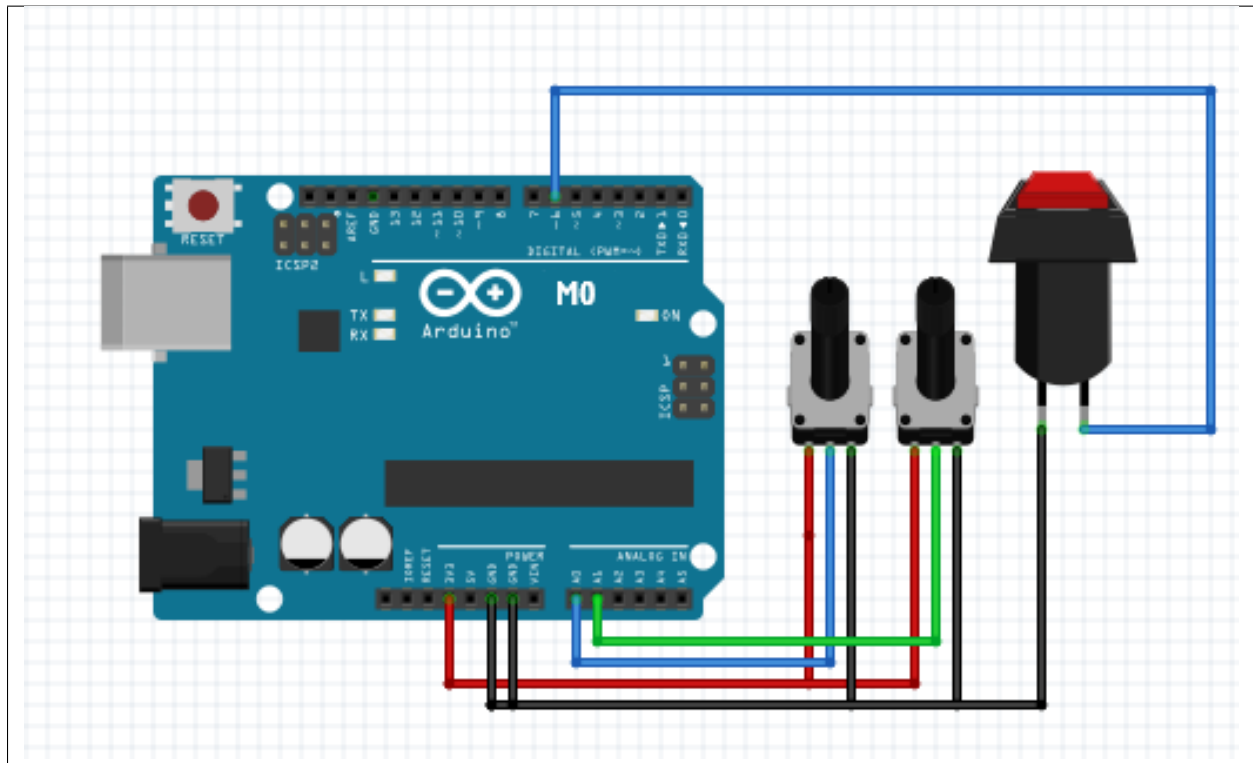
For the whale's movement in the game, three variables are acquired by the electronic system and sent in parallel at each time step. The variables are the two angles of rotation of the patient's legs that control the position and jaw of the whale, and a binary signal that controls the whale's water spray. Two potentiometers were used to detect the rotation of the patient's legs.

The three lugs of the potentiometer correspond, from left to right, to power input, signal output and ground, as shown below:

Rotating the upper part of the potentiometer alters the resistance between the signal output and the power input as shown in the middle figure,



An Arduino M0 board is used to monitor the potentiometer's output as an analog reading in the range of 0 to 1023 from its serial ports A0 and A1, and digitise it.



The potentiometer has a rotation range of 270 degrees, thus the potentiometer's output signal, N , can easily be converted into an angle, θ , in degrees using Equation:

$$\theta = \frac{270 \times N}{1023}$$

The potentiometers are screwed in different orientations as the patient's left and right leg rotate anticlockwise and clockwise, respectively. When the two legs are in their initial position (closed and centred), the potentiometer should be at its 0 position. However, since the legs do not have the limitation of the centre position, unlike the potentiometer, an offset was used to counteract this and avoid mechanical damaging. The offsets for two potentiometers were tested

and calculated in Arduino to get actual leg angles, shown in the equation below, where Reference and Actual each indicates θ with and without offset.

$$\theta_{actual\ left} = \theta_{reference\ left} - 49$$

$$\theta_{actual\ right} = 229 - \theta_{reference\ right}$$

The game version requiring both legs:

- the movement of the right leg corresponds to the opening of the whale's jaw (i.e. open and close)
- the movement of the left leg corresponds the whale's position (i.e. up or down).

The whale can be in either one of two positions. It will descend towards the bottom position when the angle is larger than a threshold and ascend to the top position when the angle is smaller than another threshold. The difference between the two thresholds was implemented to avoid cheating by swinging the leg across a small range. The whale's jaw angle experiences an error that fluctuates between 0 and 0.5 at steady state and accumulates with time. A move average filter was applied to minimise this fluctuation and avoid error accumulation. The whale's spraying action is triggered by a button. The button is connected to the Arduino's M0 digital input pin. The serial monitor reads a 1 when the button is pressed and a 0 when it is released. The entire electronic connections to the Arduino board presented in the M0 figure above.

The full Arduino code has been appended below:

```
/*
Reads an analog input on pin 0, converts it to angle that potentiometers screwing,
and prints the result to the Serial Monitor. Graphical representation is available
using Serial Plotter (Tools > Serial Plotter menu). Attach the center pins of two
potentiometers to pin A0 and A1, and the outside pins to +3.3V and ground.
*/
#include "MovingAverage.h"
// This is the moving average filter from https://github.com/sofian/MovingAverage
MovingAverage average(20);
const int buttonPin01 = 6;
int buttonPush;

// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // initialize digital input and set it as high
  pinMode(buttonPin01, INPUT);
  digitalWrite(buttonPin01, HIGH);
}

// the loop routine runs over and over again forever:
void loop() {
  // when button is not pushed, buttonPush is 0
  buttonPush=0;
  // reset the delta angle each loop preventing cached reference from the previous_
  ↪ loop
  // read the input on analog pin 0:
  int sensorValue_r = analogRead(A0);
  int sensorValue_l = analogRead(A1);
  // Convert the analog reading (which goes from 0 - 1023) to a angle (0 -
  // 270 degree):
  float angle_r = sensorValue_r * (270 / 1023.0);
  float angle_l = sensorValue_l * (270 / 1023.0);
  // when button is pushed, the digital input change to low and buttonPush is_
  ↪ assigned to 1
```

(continues on next page)

(continued from previous page)

```
if (digitalRead(buttonPin01) == LOW)
{
  buttonPush=1;
}
// Get actual angle according to offsetvalue
float actualAngle_r=angle_r-49;
float actualAngle_l=229-angle_l;
// Using filter
float movingAvg_r = average.update(actualAngle_r);
// float movingAvg_l = average.update(actualAngle_l);
SerialUSB.print(movingAvg_r);
SerialUSB.print(",");
SerialUSB.print(actualAngle_l);
SerialUSB.print(",");
SerialUSB.print(buttonPush);
SerialUSB.println();
SerialUSB.flush(); // for completing previous data sending
delay(20);         // in case the previous line doesn't work well
}
```

Tip: As you can perceive from the start, a moving average filter has been applied to the right leg angle due that, the reading from Arduino is very fluctuating, without smoothing the reading using the filter, the whale jaw might move in a creepy pattern due to the reaction to random noises. The filter has only been applied to right leg angle since the left one gives binary output, noise won't be affecting the performance of the patient.

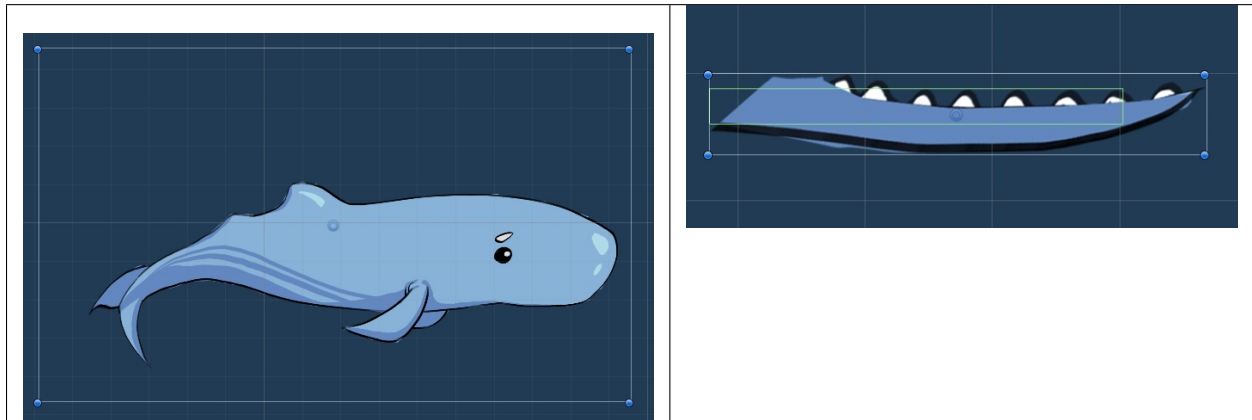


4.2 Player Control

4.2.1 Jaw Rotation

For the convenience of development, the idle of the whale has been divided into two parts:

- the body part (static animation which will not be interacting with any user input)
- the jaw part (which will rotate according to rheostat value)



Then, open the jaw to maximum of 60 degrees and map this to the maximum angle range the patient could open his leg (if the user chose Right leg mode of both leg mode, the system map the jaw open angle onto `angle_r` from Arduino Serial reading and vice versa):

```
// Jaw.cs (... represents other code blocks irrelevant to the current session)

...

[SerializeField] private bool isRightLeg;

private float angleJaw; // whale jaw open angle controlled by leg open angle

...

void Update () {
    ...

    if (isRightLeg) {
        angleJaw = arduinoHelper.angle_r / (calibrationMenu.angleRightConstraint / 60f);
    }
    else {
        angleJaw = arduinoHelper.angle_l / (calibrationMenu.angleRightConstraint / 60f);
    }

    PotentiometerControl(angleJaw);

    ...
}
```

(continues on next page)

(continued from previous page)

```

...

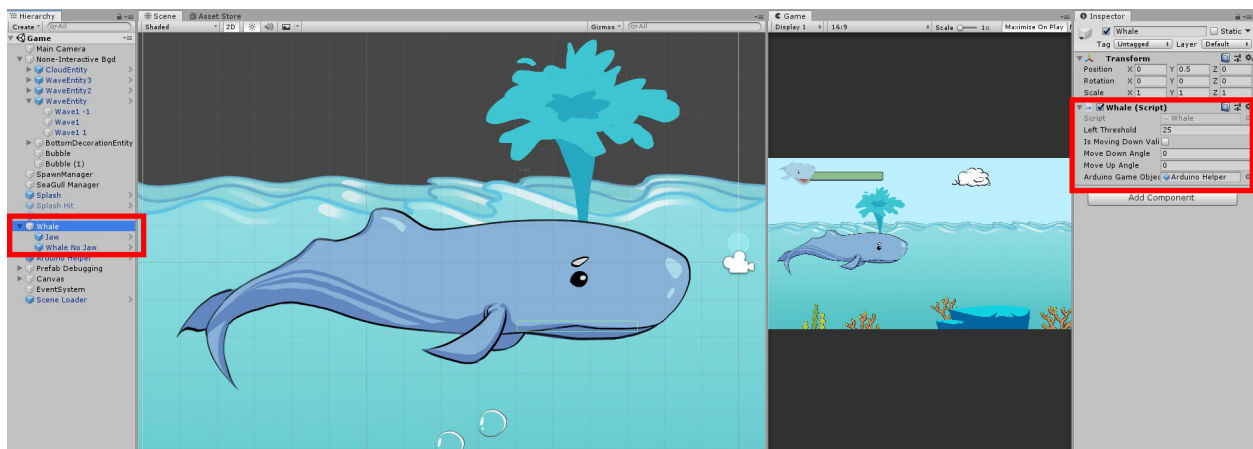
// ----- Arduino Potentiometer Control -----

void PotentiometerControl (float angle) {
    transform.localRotation = Quaternion.Euler(0, 0, -angle);
}

```

4.2.2 Whale Body Movement

In order to make the whale idle including whale body without jaw and the jaw moving at the same time, the two Prefabs have been wrapped in a parent GameObject of Whale and the script has been executed onto the parent object as well:



The implementation of the whale body movement starts with defining the 3 plausible states a current whale could have:

```

// Whale.js

enum State {
    movingDown,
    movingUp,
    stop
}

```

Due that the up and down movements are continuous rather than instantly, the parallel running of up-down movements and other implementations have been processed simultaneously using Asynchronous Programming:

```

// Whale.js (... represents other code blocks irrelevant to the current session)

private void MovementHandler() {
    switch (state) {
        case State.movingDown:
            transform.Translate(
                -Vector3.up * speed * Time.deltaTime,
                Space.World);
            break;
        case State.movingUp:
            transform.Translate(

```

(continues on next page)

(continued from previous page)

```

        Vector3.up * speed * Time.deltaTime,
        Space.World);
        break;
    case State.stop:
        // stop the whale by assign the current position to its position
        transform.position = gameObject.transform.position;
        break;
    default:
        transform.position = gameObject.transform.position;
        break;
    }
}

...

// ----- Change Movements by Manipulating States -----

private IEnumerator MoveDown() {
    if (isMovingDownValid) {
        state = State.movingDown;
        yield return new WaitForSeconds(0.75f); // give 0.75s position translation_
        ↪time
        state = State.stop;

        ...
    }
}

private IEnumerator MoveUp() {
    if (!isMovingDownValid) {
        state = State.movingUp;
        yield return new WaitForSeconds(0.75f);
        state = State.stop;

        ...
    }
}

```

Tip: when moving either up and down, the whale will keep in moving state for 0.75s duration and then switch to stop posture.

In order to prevent the whale from moving downwards when it's already low, or upwards when it's already surfaced, a boolean property of `isMovingDownValid` has been used to check the currnet altitude and constraint the movement of the whale idle only upwards when it's in lower altitude, and only downwards when it's in upper altitude.

```

// Whale.js (... represents other code blocks irrelevant to the current session)

...

private IEnumerator MoveDown() {
    if (isMovingDownValid) {
        state = State.movingDown;
        yield return new WaitForSeconds(0.75f); // give 0.75s position translation_
        ↪time

```

(continues on next page)

(continued from previous page)

```

        state = State.stop;

        // banning the whale from moving further downwards when it's already in lower_
↪position
        isMovingDownValid = false;
    }
}

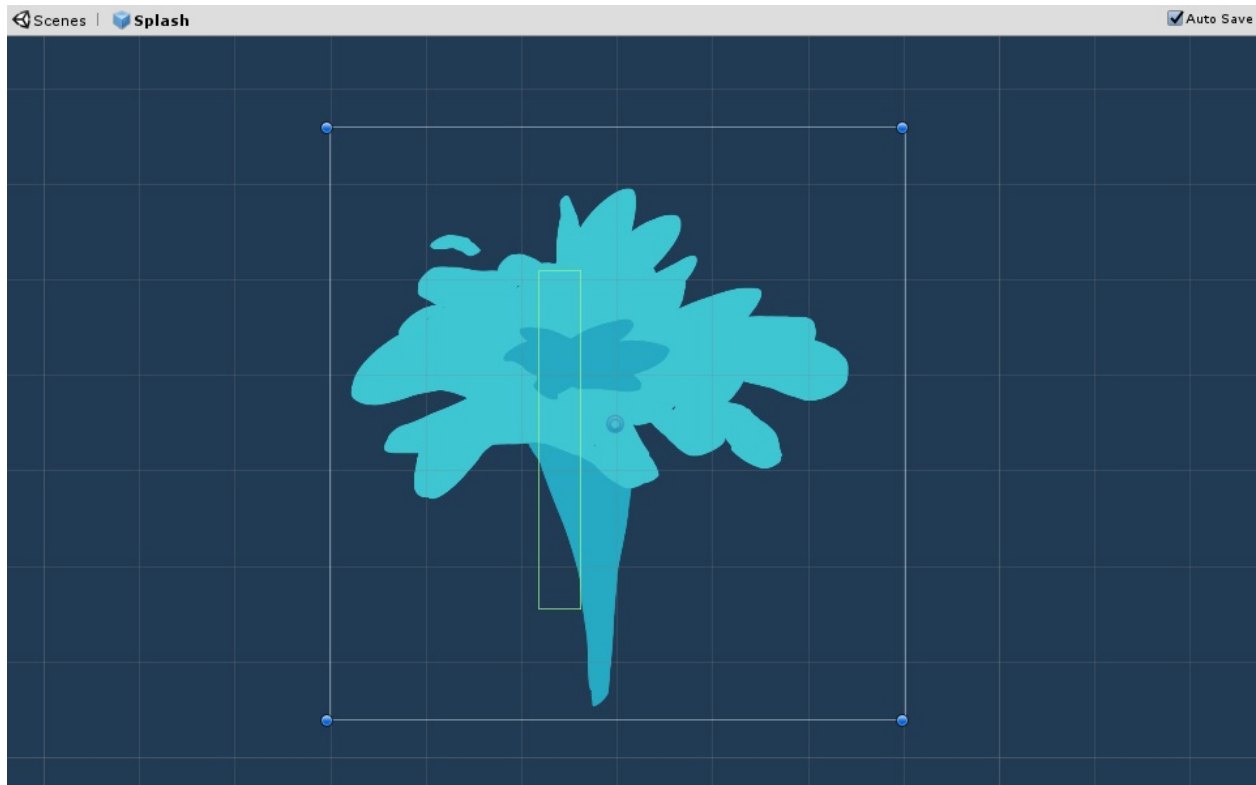
private IEnumerator MoveUp() {
    if (!isMovingDownValid) {
        state = State.movingUp;
        yield return new WaitForSeconds(0.75f);
        state = State.stop;

        // banning the whale from moving further upwards when it's already in higher_
↪position
        isMovingDownValid = true;
    }
}

```

4.2.3 Splash

The splash manipulation has been implemented in a way animations and box colliders of the splash object are pre-defined but hidden as default when the splash has not been triggered:



```

// SplashManager.cs (... represents other code blocks irrelevant to the current_
↪session)

```

(continues on next page)

(continued from previous page)

```

...

void Start() {
    ...

    // initially disable the box collider, animator and sprite render and trigger_
    ↪later
    box2D          = GetComponent<BoxCollider2D>();
    box2D.enabled = false;

    animator       = GetComponent<Animator>();
    animator.enabled = false;

    spriteRenderer = GetComponent<SpriteRenderer>();
    spriteRenderer.enabled = false;

    // initially set the splash activatable to true
    isSplashActivatable = true;
}

...

```

When the button connected to Arduino has been pressed, all 3 components above will be set to `true` and thus make usable basically by calling `ActivateSplash()` method:

```

void ActivateSplash() {
    box2D.enabled      = true;
    animator.enabled   = true;
    spriteRenderer.enabled = true;
    Invoke("DeactivateSplash", splashDuration);
}

```

Note: Since the button stays at state of 1 during being pressed, this state will trigger multiple splashes in a row during the pressing. Therefore, a logic has to be implemented to allow only one splash within 0.5s by setting `isSplashActivatable` to `false` immediately after each splash:

```

public class SplashManager : MonoBehaviour {

    [SerializeField] private float splashDuration = 0.5f;
    private int buttonPressed = 0;

    private bool isSplashActivatable;

    ...

    void Start() {
        ...

        // initially set the splash activatable to true
        isSplashActivatable = true;
    }

    ...
}

```

(continues on next page)

(continued from previous page)

```

// ----- Button Control -----

void ButtonControlSplash() {
    if (buttonPressed == 1) {
        ActivateSplash();

        PreventMultipleSplash();
    }
}

// ----- Enable and Disable Splash Activatable to mitigate splash overlay -----
→ -

void PreventMultipleSplash() {
    // prevent the user from splashing various times within short time
    isSplashActivatable = false;

    // set the splash activatable property back to true after a short delay
    Invoke("SplashActivatable", 0.5f);
}

void SplashActivatable() {
    isSplashActivatable = true;
}

// ----- Splash Manipulations -----

void ActivateSplash() {
    box2D.enabled      = true;
    animator.enabled   = true;
    spriteRenderer.enabled = true;
    Invoke("DeactivateSplash", splashDuration);
}

void DeactivateSplash() {
    box2D.enabled      = false;
    animator.enabled   = false;
    spriteRenderer.enabled = false;
}
}

```

Lastly, the splash can only happen when the whale is surfaced. The information whether the whale is surfaced or not can be retrieved from the Whale class:

```

// SplashManager.cs (... represents other code blocks irrelevant to the current_
→ session)

[SerializeField] private GameObject whaleGameObject;
private Whale whale;

...

void Start() {
    ...
}

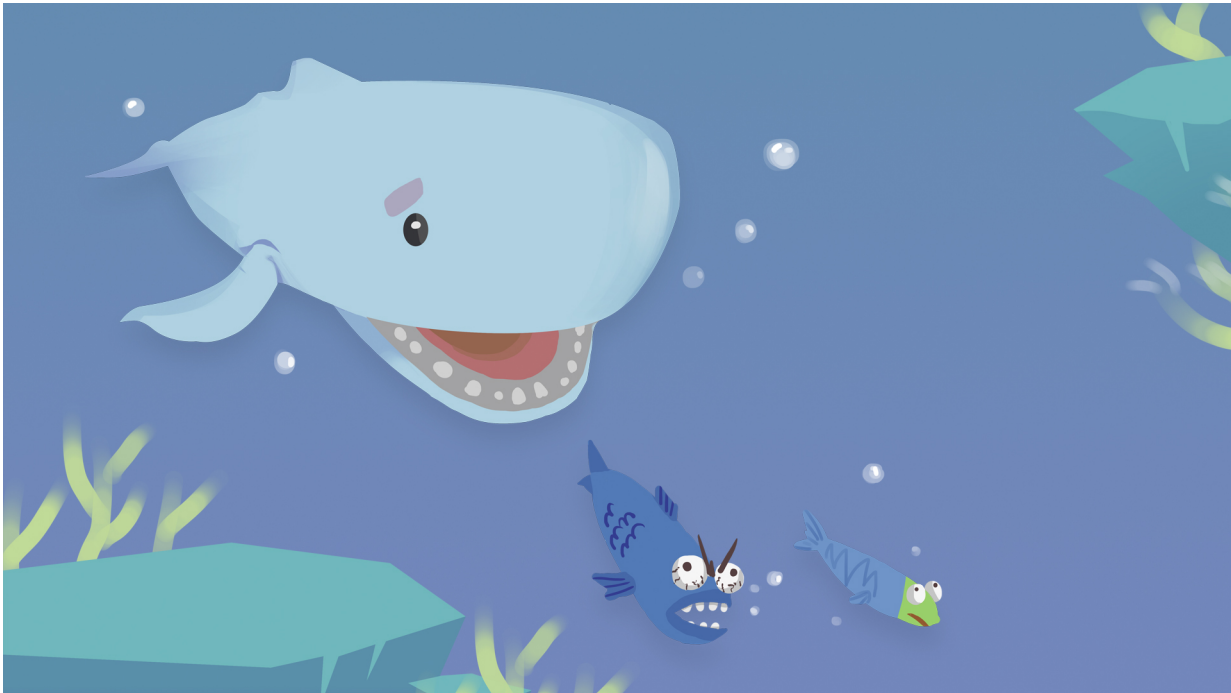
```

(continues on next page)

(continued from previous page)

```
whale = whaleGameObject.GetComponent<Whale>();  
  
...  
}  
  
void Update() {  
    ...  
  
    // determine whether the whale altitude and only trigger at higher position  
    if (whale.isMovingDownValid && isSplashActivatable) {  
        KeyboardControlSplash();  
        ButtonControlSplash();  
    }  
}
```

Tip: Using the property of `isMovingDownValid` of `Whale` class to determine the altitude level of the `Whale`, if it is true, that means the whale is surfaced and thus `Splash` is valid



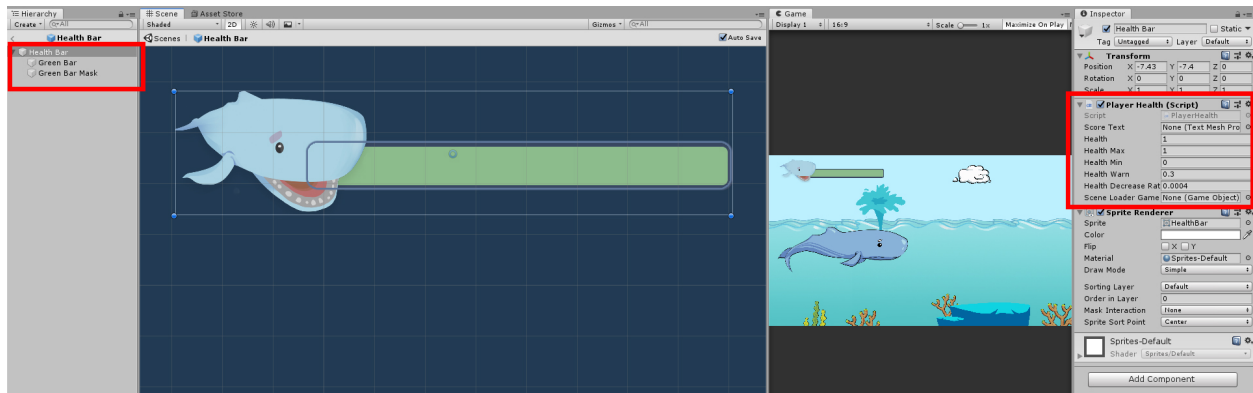
4.3 Player Health & Score

4.3.1 Sprite Manipulations

The Player Health has been shown using 3 layers of sprites:

- main sprite including Whale idle on the top
- an invisible mask on top of the bottom bar

- the bottom bar which represents the actual Health



The manipulation of the appearance of the health bar pursued with a way that rather than vary the size of the green bar, the size of the mask on the green bar has been varied according to the current health.

To implement this, the cached reference of the bar and the bar mask has been defined in prior:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current_
↪session)

private Transform barMask;
private Transform bar;

...

void Awake() {
    barMask = transform.Find("Green Bar Mask");
    bar      = transform.Find("Green Bar");

    ...
}
```

The manipulation of of the size has been implemented using the following function:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current_
↪session)

private void SetSize(float sizeNormalised) {
    barMask.localScale = new Vector3(sizeNormalised, 1f);
}
```

4.3.2 Health Point Manipulations

Firstly, in order to engage the patient to use their legs, the health point constant decreases and can only regenerate by eating fish:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current_
↪session)

...

// ----- Health Manipulations -----
```

(continues on next page)

(continued from previous page)

```
private void ConstantHealthDecrease() {
    if (health > healthMin) {
        health -= healthDecreaseRate;
    }
}

// ----- Eaten Behaviour -----

public void EatSmallFish() {
    health += 0.2f;
    ...
}

public void EatBigFish() {
    health += 0.4f;
    ...
}

...
```

Tip: On the other hand, to prevent the patient from opening the mouth of the whale all the time, a penalty measure has been implemented which is the trash that deduct health points when being eaten:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current_
↪session)

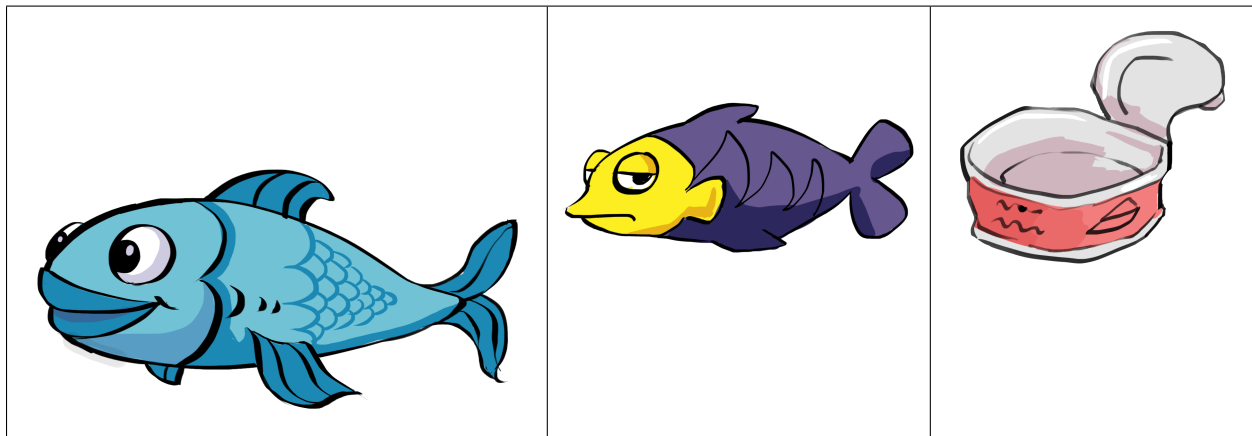
...

public void EatTrash() {
    health -= 0.6f;
}

...
```

All three functions will be called when the `Jaw` collides with each of the corresponding objects. This will be further discussed in “Object Spawn & Their Properties” session.

The Sprites of the three spawned objects has shown below:



There are two constraints on the health points of the player:

- the maximum health point (which is the HP when the player is fully healthy)
- the minimum health where the player die

Since the health calculation has been normalise, the max and min are just 0 and 1:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current ↵  
↵session)  
  
...  
  
[SerializeField] private float healthMax = 1f;  
[SerializeField] private float healthMin = 0f;  
  
...
```

When the health point are higher than the maximum by gaining points from the fish, it will be set back to the maximum. When it's lower than the minimum, the program will directly load the death scene which is the replay scene:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current ↵  
↵session)  
  
...  
  
void Update() {  
    ...  
  
    ConstantHealthDecrease();  
    SetSize(health);  
  
    if (health > healthMax) {  
        health = healthMax;  
    } else if (health <= healthMin) {  
        // player dead, load game over scene to reload  
        sceneLoader.LoadReloadScene();  
    }  
}  
  
...
```

4.3.3 Score

Apart from health point which is the essential factor for the player to be alive, score is another factor the player will be chasing upon.

The appearance of the score uses TextMeshPro UI in the Canvas object:

Then we manipulate it by updating `scoreText.text` in the script:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current ↵  
↵session)  
  
private int score;  
[SerializeField] private TMPro.UGUI scoreText;  
  
...
```

(continues on next page)

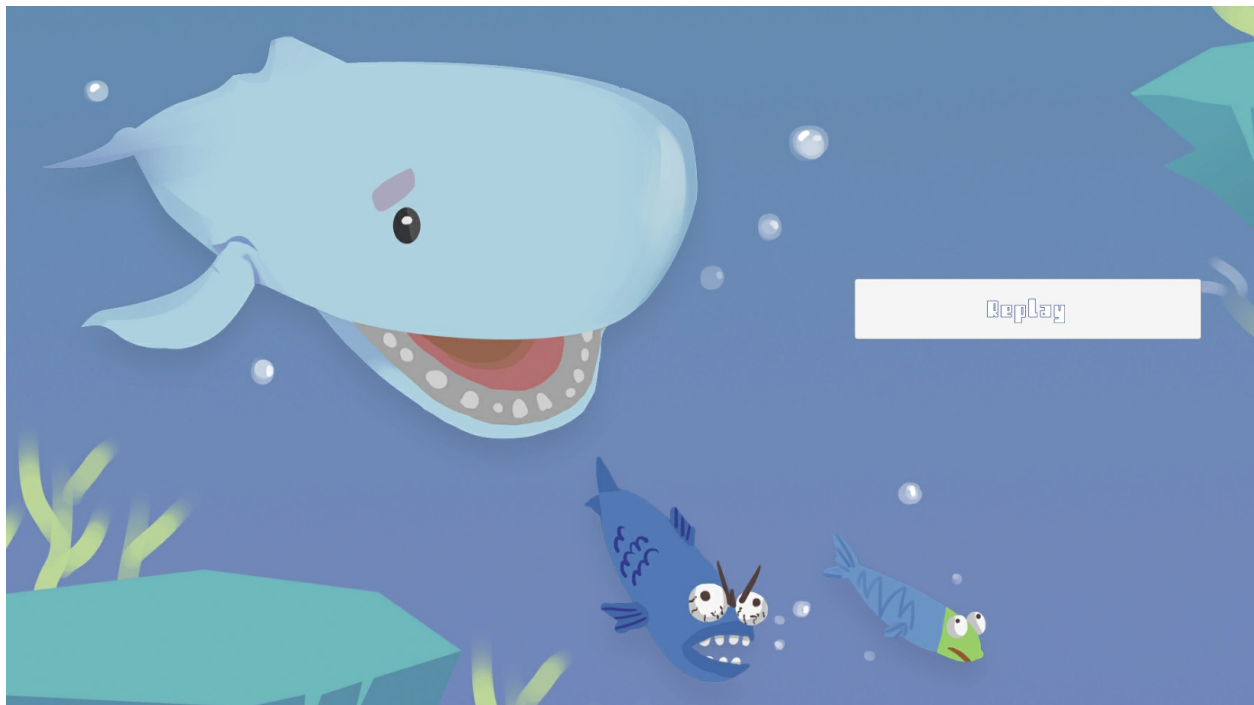


Fig. 1: Reload Scene

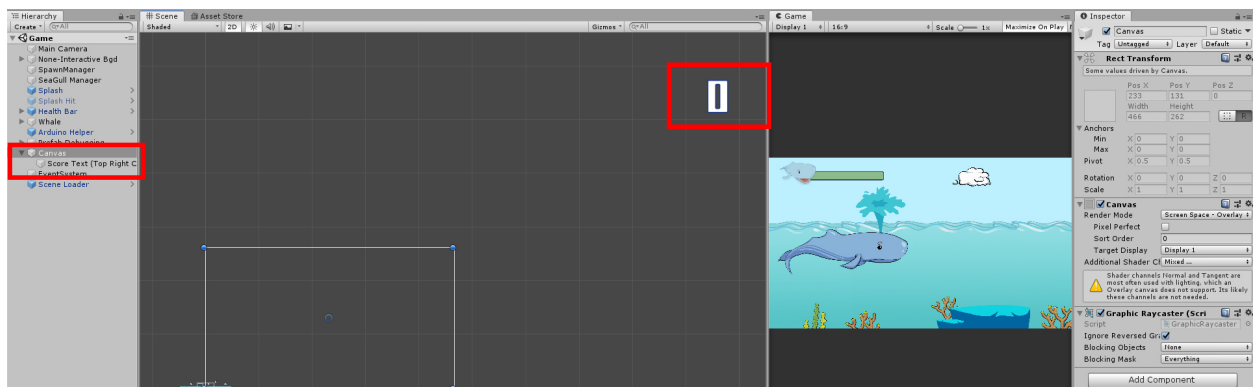


Fig. 2: Score TextMeshPro in Editor

(continued from previous page)

```
void Update() {  
    scoreText.text = score.ToString();  
  
    ...  
}
```

Eating fish and splash the seagull both have effect on the scoring:

```
// PlayerHealth.cs (... represents other code blocks irrelevant to the current_  
↪session)  
  
...  
  
// ----- Eaten Behaviour -----  
  
public void EatSmallFish() {  
    ...  
    score += 20;  
}  
  
public void EatBigFish() {  
    ...  
    score += 40;  
}  
  
...  
  
// ----- Splash SeaGull -----  
  
public void SplashSeaGull() {  
    score += 60;  
}
```

The last one will be triggered when a seagull collide with the `Splash` box collider (which will be further discussed in Object Spawn Section):

4.4 Objects Spawn & Properties

4.4.1 Objects Spawn

The object spawn action has been conducted in two scripts in parallel:

- `SpawnManager.cs` (which takes charge of all items spawn in the sea)
- `SpawnSeaGullManager.cs` (which only takes charge of spawning seagull)

The two classes have no interactions and effect on each other and working simulataniously.

Spawning Sea Objects

The spawn of the sea objects starts since 2s after the beginning of the game, every 2s, a new object will be spawned:

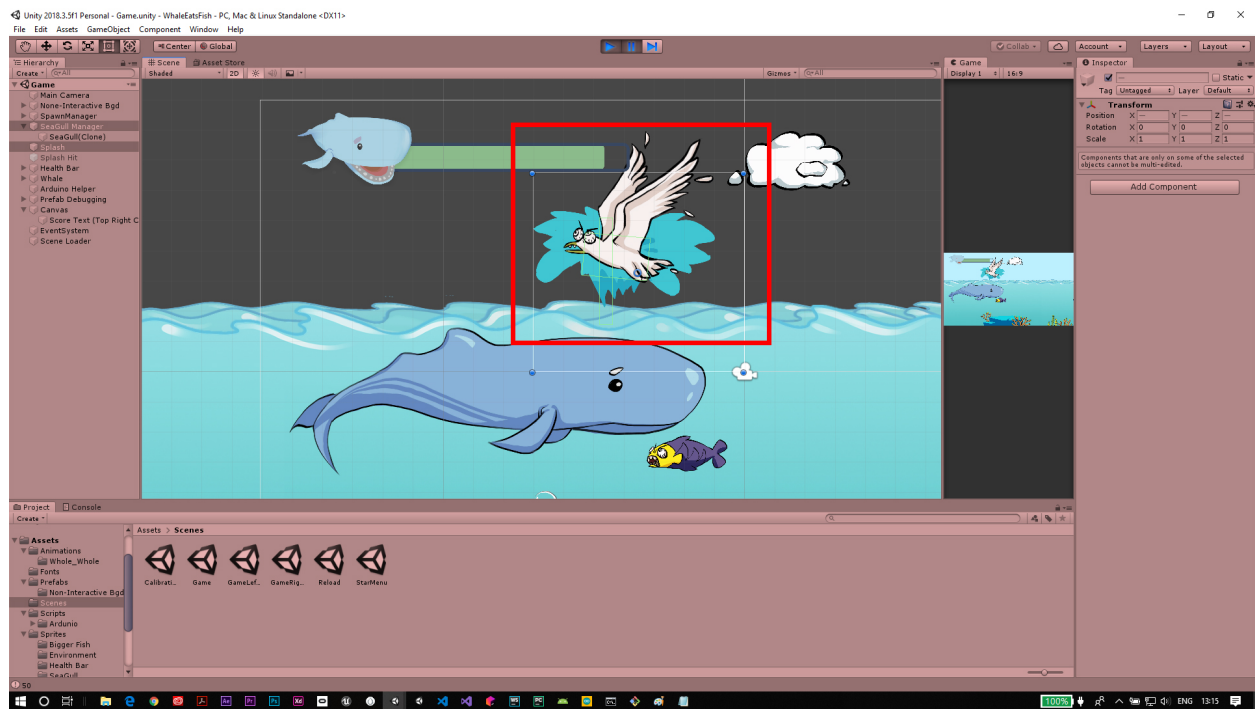
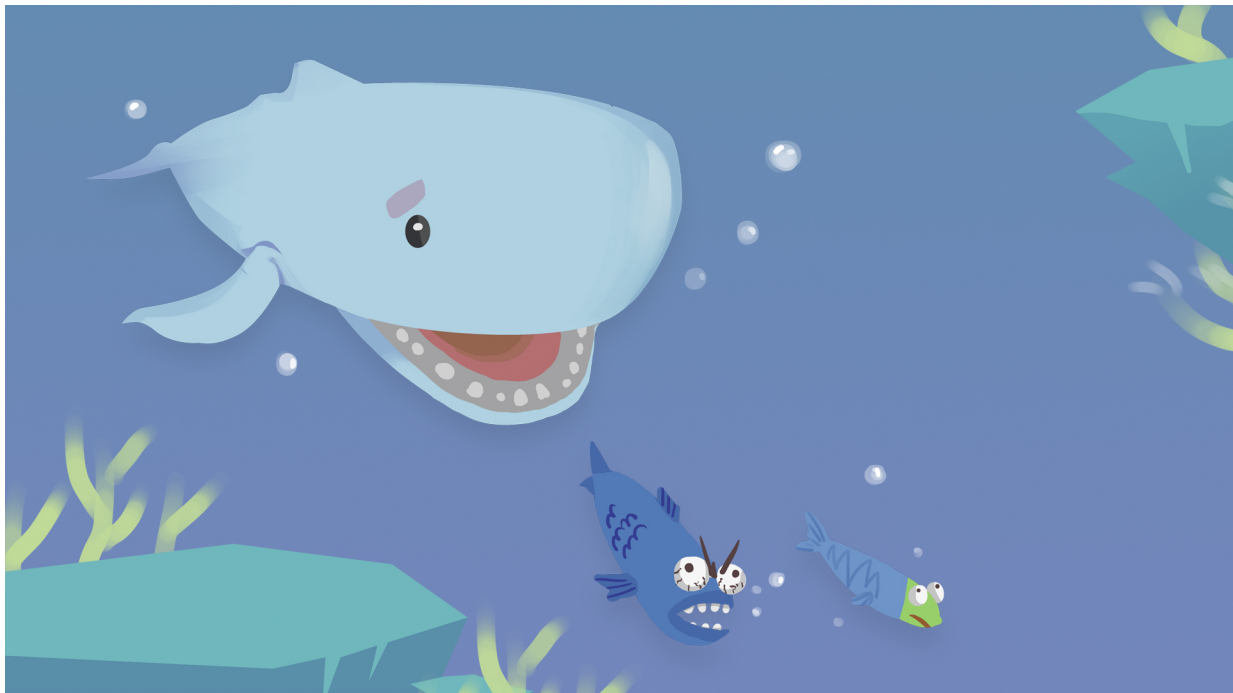


Fig. 3: SeaGull & Splash Box Collider Interaction Scene in Editor



```
void Start() {  
    // trigger spawning new object, starting from 2s, with frequency of once each 2s  
    InvokeRepeating("spawnObject", 2.0f, spawnInterval);  
}
```

In the `SpawnObject()` function, two random values have been generated first to determine which object to spawn and which altitude to spawn that:

```
// SpawnManager.cs (... represents other code blocks irrelevant to the current_  
↪session)  
  
private void SpawnObject() {  
    ...  
  
    // random 1/3 possibility spawning each of the 3 plausible objects  
    // random 1/2 possibility spawning at each of the 2 plausible altitude  
    Random random = new Random();  
    int randomThresholdObject = random.Next(1, 4); // generate a integer number_  
↪between 1, 2, 3  
    int randomThresholdPos = random.Next(1, 3); // generate a integer number_  
↪between 1, 2  
  
    ...  
}
```

Then an if-else statement will firstly determine which object to spawn under the constraint of play mode:

- Under Only “Right Leg” or “Left Leg” play mode: the object will only be spawn in the higher altitude.
- Under “Both Leg” mode, the object will be spawned via two lanes each of 50% chance.

```
// SpawnManager.cs (... represents other code blocks irrelevant to the current_  
↪session)  
  
[SerializeField] private bool isBothLegMode;  
  
...  
  
private void SpawnObject() {  
    ...  
  
    // determine the altitude of the object spawn position, assigning values to_  
↪spawnPos  
    if (isBothLegMode) {  
        if (randomThresholdPos == 1) {  
            spawnPos = spawnPosHigher;  
        } else if (randomThresholdPos == 2) {  
            spawnPos = spawnPosLower;  
        }  
    } else {  
        spawnPos = spawnPosHigher;  
    }  
  
    ...  
}
```

Lastly, an “if-else” statement has been implemented to determine which object being spawned according to the sprite defined as configuration parameters above:

```
// SpawnManager.cs (... represents other code blocks irrelevant to the current_
↳session)

[SerializeField] private GameObject smallFish;
[SerializeField] private GameObject bigFish;
[SerializeField] private GameObject trash;

...

private void SpawnObject() {
    ...

    // determine which object will be spawned at the previous defined altitude
    if (randomThresholdObject == 1) {
        newSpawn = Instantiate(
            smallFish,
            spawnPos,
            Quaternion.identity);
        ...
    } else if (randomThresholdObject == 2) {
        newSpawn = Instantiate(
            bigFish,
            spawnPos,
            Quaternion.identity);
        ...
    } else if (randomThresholdObject == 3) {
        newSpawn = Instantiate(
            trash,
            spawnPos,
            Quaternion.Euler(0, 0, -20f)); // beware the trash spawn has rotation_
↳angle
        ...
    }
}
}
```

Spawning SeaGull

SeaGull spawn on the other hand, is much simpler since there is only one spawn altitude and one plausible object being spawned:

```
// SpawnSeaGullManager.cs (... represents other code blocks irrelevant to the current_
↳session)

void Start() {
    // trigger spawning new object, starting from 2s, with frequency of once each 2s
    InvokeRepeating("SpawnSeaGull", 2.0f, spawnSeaGullInterval);

    seaGull = seaGullGameObject.GetComponent<SeaGull>();
}

...

// sea gull is not part of the fish-trash system and the spawning rate is very low
// thus doesn't need to be wrapped into the above object spawn-destroy system
void SpawnSeaGull() {
    GameObject newSpawnSeaGull;
```

(continues on next page)

(continued from previous page)

```

newSpawnSeaGull = Instantiate(
    seaGullGameObject,
    spawnPosSeaGull,
    Quaternion.identity);
newSpawnSeaGull.transform.parent = transform;
}

```

Constant Leftward Movement

The constant leftward movement of the sea objects pursue with the following logic:

1. when a new object has been spawned, append it to the current spawn manager parent object
2. in each iteration of `Update()` function being called, loop through all the current children of the parent spawn manager object in a for-loop
3. apply a left-ward vector to every single child in the loop

Note: since the child objects of spawn manager could be destroyed due being eaten by the Whale or self-destructed outside the boundary of the screen, the number of items within the spawn manager is varying thus need a agile and flexible approach on a dynamic array instance of collection of all children objects.

```

// SpawnSeaGullManager.cs (... represents other code blocks irrelevant to the current
↳ session)

...

void Update() {
    float displacement = Time.deltaTime * speed;

    // store all children under Spawn Manager in an array
    Transform[] children = transform.Cast<Transform>().ToArray();

    for (int i = 0; i < children.Length; i++) {
        var child = children[i];
        // beware to add Space.World or otherwise default will be Space.Self
        // where rotation angle of the object will be stored as well
        child.transform.Translate(Vector2.right * displacement, Space.World);
    }
}

```

The append of child happend during the creation of each object:

```

// SpawnSeaGullManager.cs (... represents other code blocks irrelevant to the current
↳ session)

private void SpawnObject() {
    // instantiate the next spawn
    GameObject newSpawn;

    ...

    // determine which object will be spawned at the previous defined altitude

```

(continues on next page)

(continued from previous page)

```

    if (randomThresholdObject == 1) {
        newSpawn = Instantiate(
            smallFish,
            spawnPos,
            Quaternion.identity);
        addChildToCurrentObject(newSpawn);
    } else if (randomThresholdObject == 2) {
        newSpawn = Instantiate(
            bigFish,
            spawnPos,
            Quaternion.identity);
        addChildToCurrentObject(newSpawn);
    } else if (randomThresholdObject == 3) {
        newSpawn = Instantiate(
            trash,
            spawnPos,
            Quaternion.Euler(0, 0, -20f)); // beware the trash spawn has rotation_
↪angle
        addChildToCurrentObject(newSpawn);
    }
}

void addChildToCurrentObject(GameObject item) {
    // make the current item a child of the SpawnManager
    item.transform.parent = transform;
}

```

Destroy Objects

If the object spawned hasn't been eaten, it will continue to move left-wards and stack in the spawn manager parent object, which will consume plenty of computer memory and thus harmful for the program.

Therefore, all object will be destroyed if they are outside the left boundary of the screen to save the computational power.

```

// DestroyObject.cs (... represents other code blocks irrelevant to the current_
↪session)

[SerializeField] private float destroyXPos = -18f;

...

void Update() {
    DestroyHierarchy();
}

public void DestroyHierarchy() {
    //Debug.Log(gameObject.transform.position.x);
    if (gameObject.transform.position.x < destroyXPos) {
        Destroy(gameObject);
    }
}

```

4.4.2 Object Properties

Properties of Objects in the Sea

Collision Trigger

Following the last section, the health point manipulations has been triggered in each of the object’s class. The triggering utilise OnTriggerEnter2D () function rather than OnCollisionEnter2D () because we want the object to pass through and trigger the event rather than collide and bounce away. Using small fish as an example:




```
// SmallFish.cs (... represents other code blocks irrelevant to the current session)

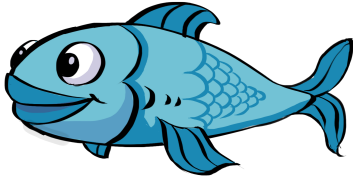
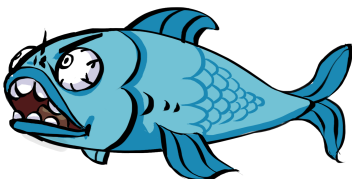
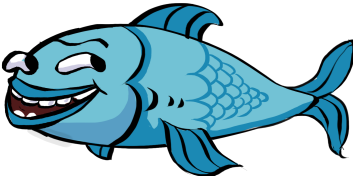
private void OnTriggerEnter2D(Collider2D collision) {
    playerHealth.EatSmallFish();
}
```

Sprite Transition

In order to increase the repetibility of the game by adding more fun factors into the UI design, sprite transitions has been implemented to the two kinds of fishes:

- When the fish the far from the whale, it shows a normal fish
- When the fish is close to the whale but not passed yet, the fish shows a frightened face inspired by rage faces from memes
- When the whale miss eating a fish, the fish shows a grin face

Normal	Frightened	Grin
		

Normal	Frightened	Grin
		

The implementation involves basically getting the component of the sprite renderer and change the correponding sprite which has been pre-defined in the [SerializeField]. The following example uses smalle fish as an example:

```
// SmallFish.cs (... represents other code blocks irrelevant to the current session)
```

(continues on next page)

(continued from previous page)

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

[SerializeField] private Sprite smallFishDefault;
[SerializeField] private Sprite smallFishFrightened;
[SerializeField] private Sprite smallFishLaugh;

...

private void ChangeSprites() {
    if (transform.position.x > -7f &&
        transform.position.x < 4f) {
        // When the fish is close to the jaw but not being eaten yet
        GetComponent().sprite = smallFishFrightened;
    } else if (transform.position.x < -7f) {
        // When the fish passed the Whale, indicating the Whale missed capturing it
        GetComponent().sprite = smallFishLaugh;
    }
}
```

Properties of the SeaGull

The movement of the SeaGull is more complicated than the previous fishes since it involves the a dropping mechanism. This has been implemented using the manipulations of `rigidbody` type of the object.

- When a seagull has been spawned, the `rigidbody` type has been set to `kinematic` where there is no effect of gravity onto the object.
- When the seagull hit with the splash box collider, change the `rigidbody` type to `dynamic` where gravity has an effect on the object and therefore it falls into the water.

```
// SeaGull.cs (... represents other code blocks irrelevant to the current session)

private Rigidbody2D rigidbody2D;

void Start() {
    rigidbody2D = GetComponent<Rigidbody2D>();
    ...
}

...

private void ChangeRigidBodyType() {
    // change to rigidbody type to dynamics thus could use gravity
    rigidbody2D.bodyType = RigidbodyType2D.Dynamic;
}
```

When it falls into water, ignore the gravity again and apply a horizontal left-wards vector onto it for it to flow.

```
// SeaGull.cs (... represents other code blocks irrelevant to the current session)

// get rid off all downwards force and make the object slowly move with water towards_
↪left
private void FlowWithWater() {
```

(continues on next page)

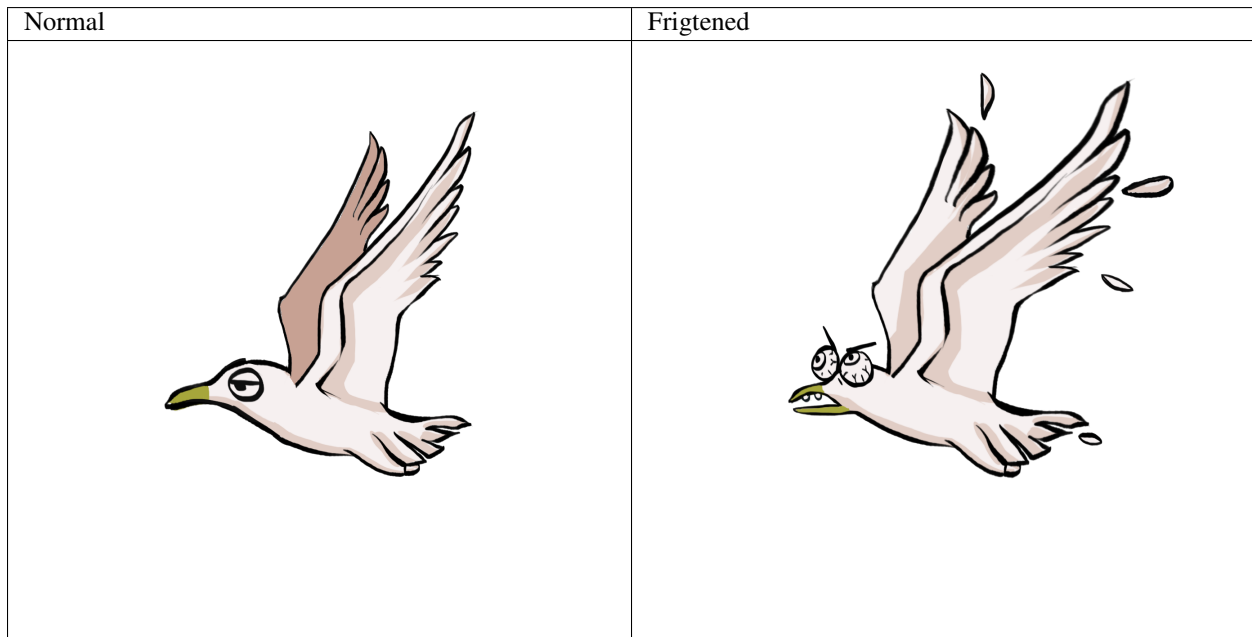
(continued from previous page)

```

if (transform.position.y < 0.5f) {
    rigidbody2D.gravityScale = 0;
    rigidbody2D.velocity = new Vector3(-1.5f, 0, 0);
}

```

The Sprite also changed from the normal one to a frightened one:



```

private void ChangeAnimation() {
    animator.SetBool("IsHitByFlush", true);
}

```

4.5 Environment Objects

4.5.1 Scene Object Leftwards Movement

In order to convey the effect that whale is swimming towards right whilst its relative x-position to the screen boundary maintains, functions need to be defined to let the various objects such as corel, wave and clouds scroll to the left at different speeds which also engaged a parallel effect between further and closer objects.

```

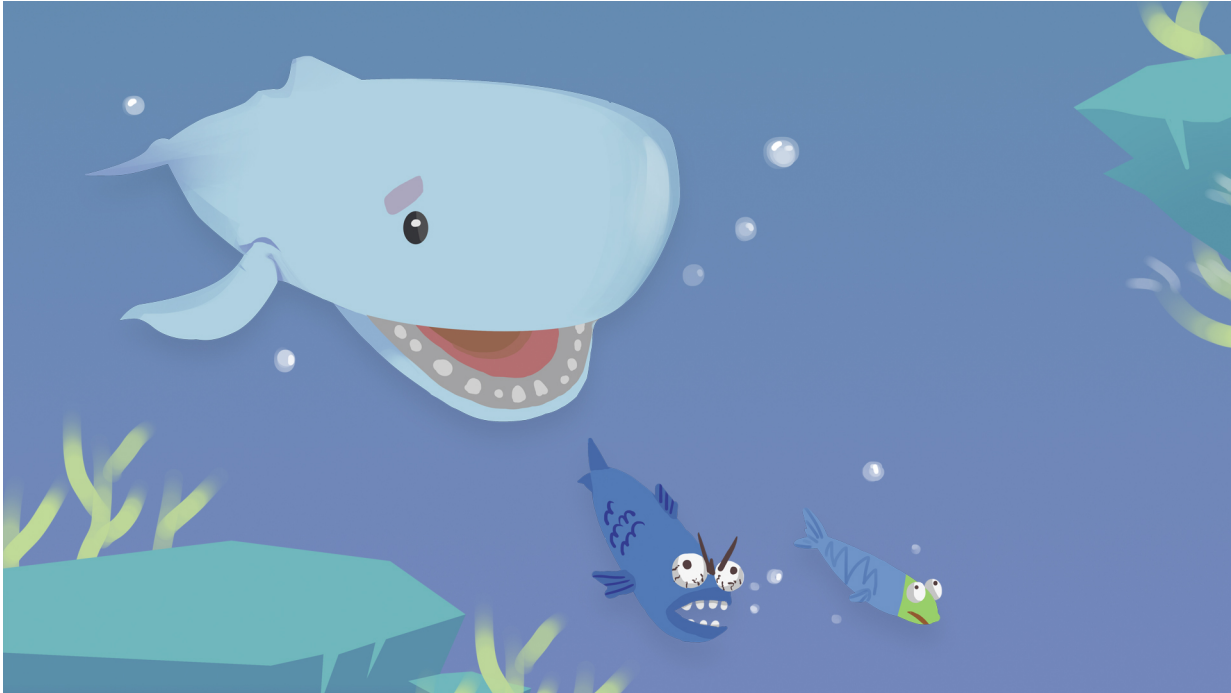
[SerializeField] private float scrollSpeed = -4f;
[SerializeField] private int resetX = -32;

void Start() {
    // override the start position to its initial sprite position
    startPos = transform.position;
}

void Update() {

```

(continues on next page)



(continued from previous page)

```

xPos = transform.position.x;
yPos = transform.position.y;

float displacement = Time.deltaTime * scrollSpeed;
transform.Translate(Vector2.right * displacement);

// when the center of Wave scrolls to one screen width to the left of the
↳original center,
// reset the X of the Wave entity to it's original starting position
if (xPos < resetX) {
    transform.position = new Vector3(startPos.x, yPos, startPos.z);
}

...
}

```

Note: In order to create a constant flow movement of the wave, 3 identical wave sprites with the same width as the screen width have been put in a row. When the center of the WaveEntity which is the container of 3 wave sprites scrolls to one screen width to the left of the original center, reset the X position of the Wave entity to it's original starting position thus create a constance flowing effect.

Tip: An oscillating algorithm has been implemented on the wave entity to mimic the dynamic of real waves. 3 layers of wave entities fluctuate according to various periods

```

// for sine periodic oscillation movement implementation
[SerializeField] private bool isOscillating = false;
[SerializeField] private Vector2 movementVector = new Vector2(0f, 0.25f);

```

(continues on next page)

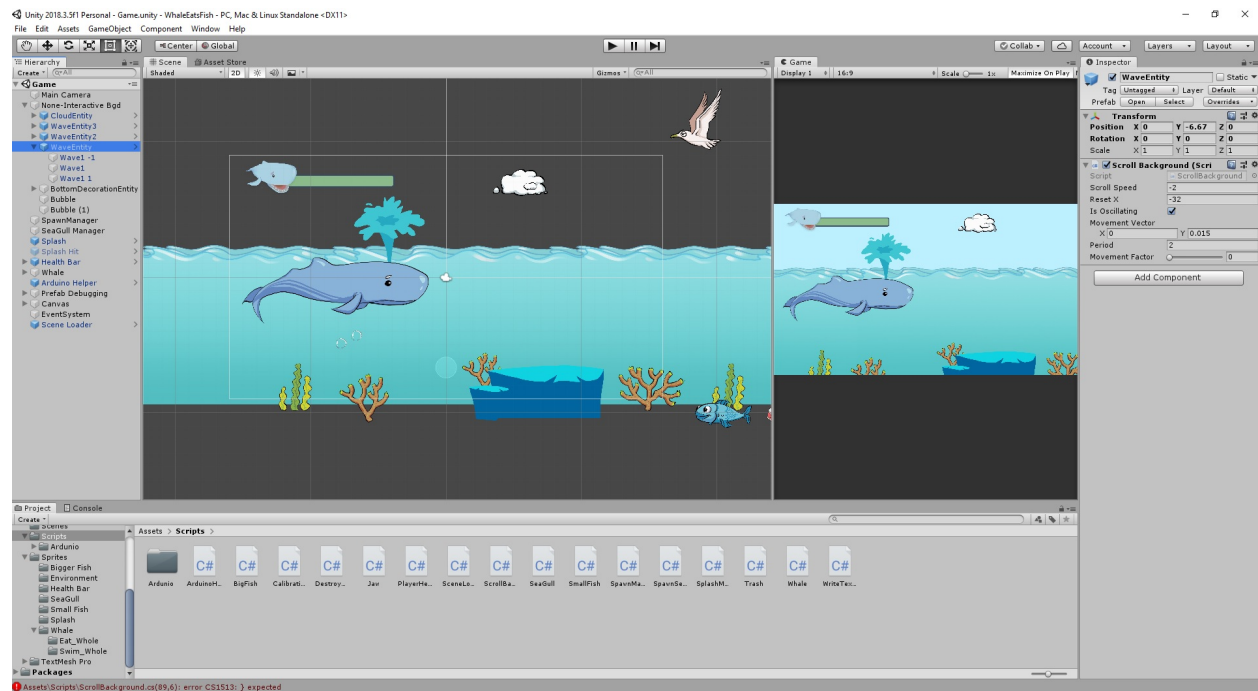


Fig. 4: WaveEntity Object Inspector Screenshot

(continued from previous page)

```

[SerializeField] float period = 2f;

// 0 for not moved, 1 for fully moved.
[Range(0, 1)] [SerializeField] private float movementFactor;

private Vector2 offset;

...

void Update() {
    ...

    // ----- oscillation movement implementation -----
    // protect against period is zero
    // period less than or equal to the smallest thing we can represent (as good as 0)
    if (period <= Mathf.Epsilon) { return; }
    float cycles = Time.time / period; // grows continually from 0

    const float tau = Mathf.PI * 2f; // about 6.28
    float rawSinWave = Mathf.Sin(cycles * tau); // goes from -1 to +1

    movementFactor = rawSinWave / 2f;
    offset = movementFactor * movementVector;

    if (isOscillating) {
        transform.Translate(Vector2.down * offset);
    }
}

```

CHAPTER 5

Indices and tables

- `genindex`
- `modindex`
- `search`