Individual Asset Construction

When creating a project in the lab ensure that you navigate to your U: drive. Any reference to a network mapped drive (//userwx/...) will cause an error. Instead each time you create a project navigate to the U: drive and then into the Documents directory and then Unity Projects - create the directory structure if necessary.

For the practical tasks you will be instructed at the start of the practical about the core tasks, extension tasks, and duration for the practical. Core tasks will be detailed in their instruction and should be achievable by all students able to devote time to completing the task. Extension tasks will not have any instructions provided and will reflect the deeper understanding and individual investigation required for high distinction level work. ***Take note of these instructions as submission after the deadline will result in zero marks***.

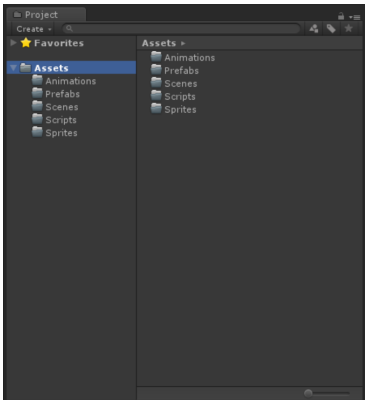
Practical 03 - Tappy Plane

Core Task: 6%
Extension: 4%
Duration: 4 practical sessions (due in practical session of Week 13)

This third and final practical task will require you to implement a game that emulates an existing game that was extremely popular around 2014: Flappy Bird. The simple mechanics but challenging gameplay produced a game that was fun to play on a mobile device but did not require extensive gameplay sessions. The process will guide you through the creation of the game space, demonstrate the use of the engine tools, explore simple sprite based animation, create a paralax effect and the game logic. The steps will be quite brief with the use of screenshots to help guide you. The steps will give you a basic understanding of the game engine, extension work will require individual investigation.

If you run into an issue or a situation that presents you with an error then in the first instance attempt to undo or backtrack your work until you find a stable solution, if you are still not able to resolve the problem then the demonstrators will be able to provide you with some assistance.

1. Start by creating a new project named "TappyPlane". You should save the project to your U: drive and you should use the 2D setting.
2. Once the project has been created Save the scene as TappyPlane. (File menu, Save Scene As ..., TappyPlane)
3. In the Project panel, create the following folders under Assets:
   - Animations
   - Prefabs
   - Scenes
   - Scripts
   - Sprites



4. To begin you will need to add a background to your game world. For this game all of the art assets required are stored in a zip file on FLO that you should extract. The assets were created by Kenney Vleugels and can be freely used under creative commons license. Original assets can be found at `http://kenney.nl/assets/tappy-plane`. The files included in the zip should be all that is required for the core tasks and any extension tasks that you choose to implement.

Drag the background.png sprite from the Prac03 folder and drop it into the Assets/Sprites folder in the Project panel.

5. Click on the background asset in the Sprites folder and ensure that the Texture Type property in the Inspector panel is set to Sprite (2D and UI). If not change it and hit the Apply button.

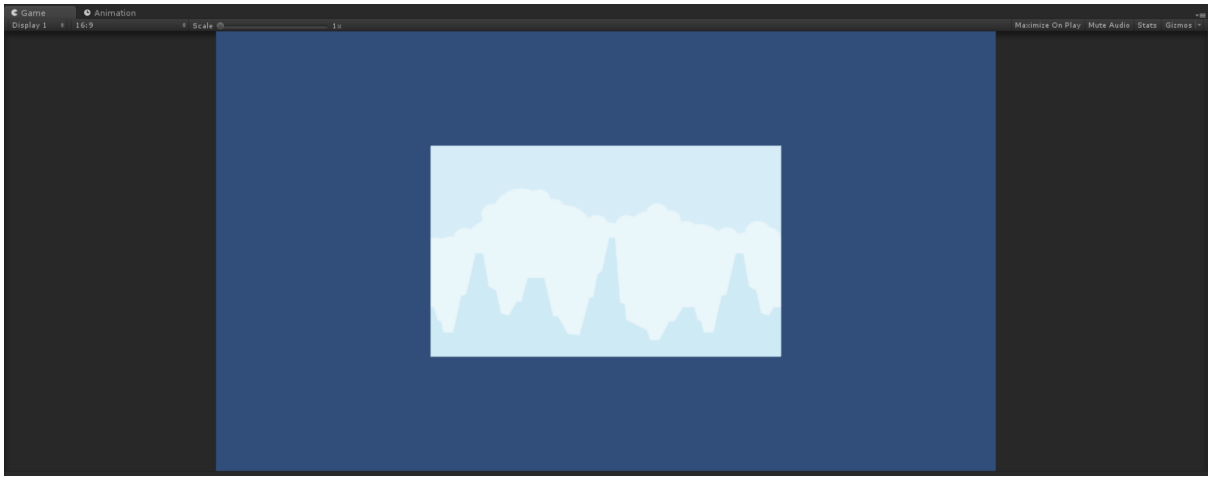
Ensure the Pixels Per Unit property is set to 100. This will mean that the background image (which has the dimensions 800 x 480) will be 8 x 4.8 units in size in the Unity Scene. We will use this value later to ensure that the background image is captured correctly by the camera.

6. Drag and drop the background sprite from the Sprite folder into the Scene tab. With the background game object selected in the Hierarchy panel, go to the Inspector panel and right click on the Transform component and then select Reset Position. Alternatively, you can simply set the X, Y, and Z values of position to 0.

Now we have the background in place but if you look at the Game panel you will see that there is a blue border around the background image (see the image over the page). If we were looking to have the game presented at full screen this would diminish the play experience and will cause issues when we go to animate the background. To fix this we will zoom the camera in.

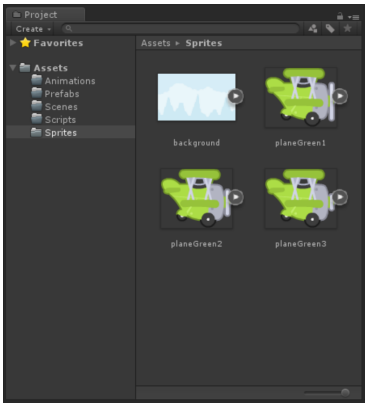
7. From the Hierarchy tab select the Main Camera and change the Size property to 2.2. This Size property is half the vertical size of the camera. Recall that we had set our Pixels Per Unit property to 100 previously which defined our background sprite with 8 x 4.8 Unity dimensions. If we consider the Size property is half the vertical height then we can see that if the Main Camera Size was set to 2.4 this would be the exact vertical units for the existing background sprite. By setting the Size to 2.2 we are actually zooming into the sprite a little - which may help for various phone resolutions.

If your Game panel looks a bit different to the images seen here then click on the drop down menu next to Display 1 and change the resolution to 16:9. This should force the background sprite to consume all of the available space as seen in the image here.



Using this resolution tool you can create your own custom resolutions for various types of projects.

8. Now we need to add in our player character sprite. From the Prac03 folder you downloaded from FLO add three of the plane sprites to the Sprites folder in your Project panel. You are free to choose any colour plane but they should all be of the same colour (I would recommend leaving the red one for one of the extension tasks).

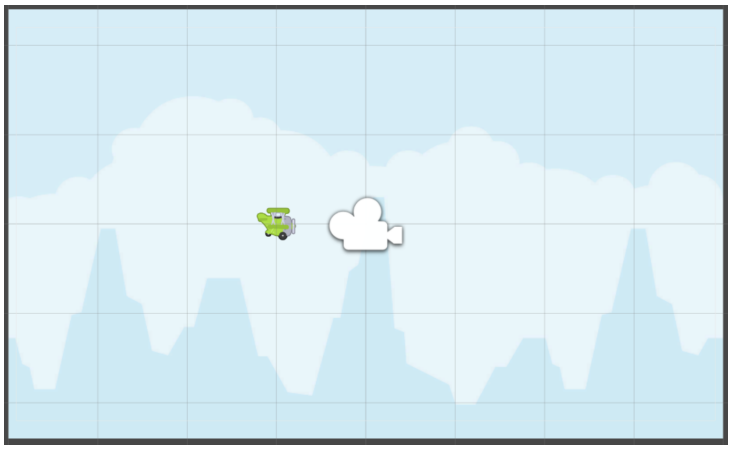


9. Click on the first plane sprite and then hold down the Ctrl key and select each of the other plane sprites. With all three plane sprites selected drag and drop them onto the Scene panel. A new dialog window will open and ask you to save your animation. Navigate to the Animations folder and change the file name to PlaneFlying. Click save.

You should now have two new items in the Animations folder in the Project panel as well as a plane game object in the Hierarchy. At this point change the name of your plane sprite in the Hierarchy panel to Plane. You may notice that the Plane doesn't display in the Scene or Game view; to correct this we will create some layers to our scene.

10. Click on the background sprite in the Hierarchy panel and then in the Inspector panel click on the Sorting Layer property Default. Click on Add Sorting Layer. Click on the + symbol and name the new layer Background. Click and drag the newly created Background layer above the Default layer.

Now click on the background sprite in the Hierarchy again, click on Default in the Sorting Layer property and change this to Background. Your plane should now be visible.



11. The Plane sprite is a little large so we will reduce it to half its size - this should make the gameplay for our game a little easier and provide us with a sense of a larger play space. Click on the Plane in the Hierarchy panel and in the Inspector panel change the Scale values to 0.5, 0.5, 1. While here change the Plane's Position so that it is toward the left of the screen, set Position to -2.5, 0, 0.

12. Save your Scene as MainGame in the Scenes folder, save the Project and then play the game. You should notice that the plane animates.

13. Now we want to add some additional dynamics to our game and have the background scroll to give a sense of the Plane flying across the scenery. To achieve this we will implement a repeating background functionality through code to continuously reuse the background sprite and give the sense of movement.

Open the Scripts folder in the Project panel, right click and create a new Script named `RepeatingBackground` **take note that there is no space between the words.* Double click and open in your code editor.

Our first task is to set up two properties that control how fast the background moves. After the class declaration line (`public class RepeatingBackground...`) declare the following two variables:

```
public float scrollSpeed;

public const float ScrollWidth = 8;
```

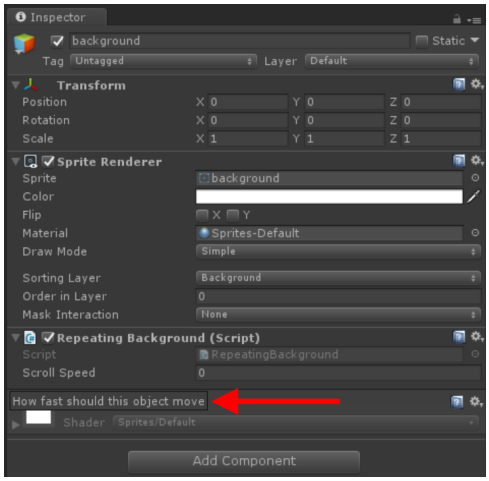
`scrollSpeed` will determine how fast the background will move while `ScrollWidth` will tell our code what size our background sprite is. We have set `ScrollWidth` to be a `const` which tells our code that it is a constant variable and cannot be changed from the value 8.

Save your RepeatingBackground script and return to Unity. Drag the RepeatingBackground script from the Script folder in the Project panel onto the background game object in the Hierarchy panel. Consult the Inspector panel to see the Scroll Speed property in the background script component - you may need to play the game to ensure the changes take place and are displayed properly in the Inspector panel.

To ensure our code is readable and usable within the Unity engine we can add some identification features to the code that are displayed in the Inspector. Return to the code editor and above the line where you declared the scrollSpeed variable add the following line of code:

```
[Tooltip("How fast should this object move")]
```

Save your code again and return back to Unity. If you now hover your mouse pointer over the Scroll Speed property in the Inspector panel the tool tip we just added to our code will appear – see the following image. While this doesn't improve the functionality of our code it does help future developers work with our game. By providing this additional piece of information we are telling people the purpose of this particular property without the future developers trying to discern it from the variable name.

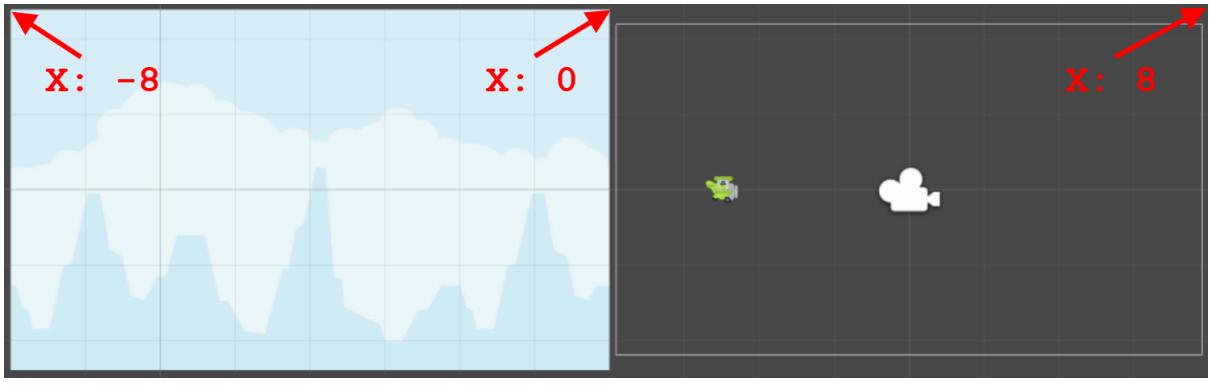


Next we need to define two new functions. For this script we do not need the Start or Update functions so we can delete those. The first function that we will define is the FixedUpdate function – the same as we used in a previous practical. Within this newly created function we will adjust the backgrounds Position properties on the X axis. Each update the background sprite will be shifted to the left. At the end of each move we will then check to see if the background sprite is completely off the screen and if so we

will implement that functionality in a new function. Then finally we set the new position to the adjusted value. For now we want to define the FixedUpdate function with the following code:

```
private void FixedUpdate()
{
   Vector3 pos = transform.position;

   pos.x = pos.x - scrollSpeed * Time.deltaTime;

   if(transform.position.x < -ScrollWidth)
   {
     Offscreen(ref pos);
   }

   transform.position = pos;
}
```

The next piece of code is required to set the location of the background element to the right hand side of the game world ready to be moved during the FixedUpdate cycle. This will be handled in the Offscreen function. When the background sprite has moved completely off the screen it would look similar to the image below. If we wanted to place the background sprite back to the right of the screen we would need to adjust the X position by twice the ScrollWidth, i.e., if we were to adjust X in the image below by adding the ScrollWidth X would be set to 0, however we want the background to appear scrolling so we need to set the X value to 2 times ScrollWidth which will make X equal 8 (or the right hand edge of the screen).



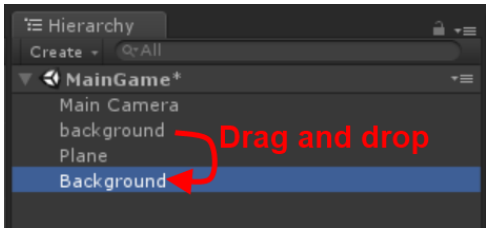
For the Offscreen function we will be reusing it later in the practical for a different functionality, to ensure we have access to this we need to declare this function a little differently to how we have in the past. We need to declare the function as protected rather than private - this means that it is kept hidden from most things except for those that need access to it. We also need to declare the function as virtual so that we can redefine it later when we need to. Add the following code beneath your FixedUpdate function but before the end of the class brace.

```
protected virtual void Offscreen(ref Vector3 pos)
{
   pos.x = pos.x + 2 * ScrollWidth;
}
```

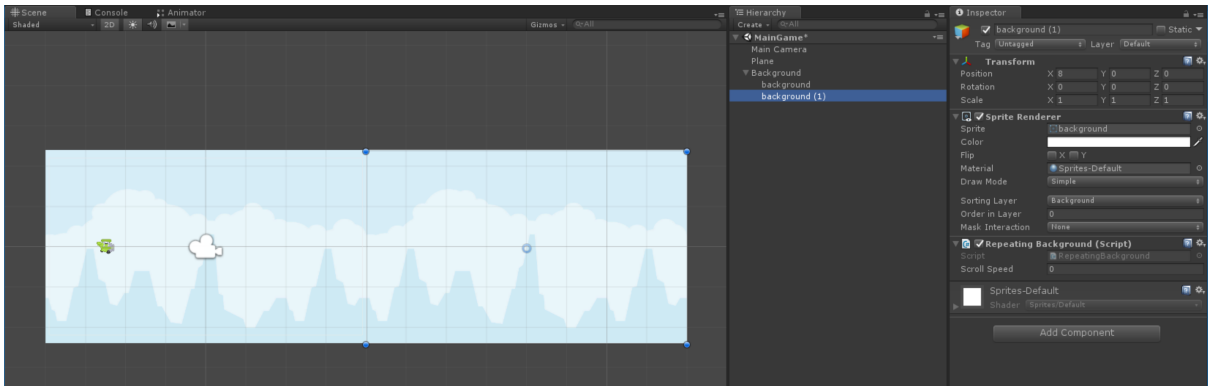
Save your code and return to Unity and play your game. When the game executes you will notice that nothing is happening. Stop the game and then in the Inspector panel change the Scroll Speed property to 5, play the game again and observe the background animating.

14. You will notice that the background will move from the right to the left of the screen and then return to the right side of the screen and repeat. This is the functionality that we are looking for but you will also notice that there is an empty space as the background cycles off the screen. To correct this we will add in another background sprite and have that scroll with our original but offset so that there is always a background sprite visible on the screen. We are going to want to apply this process to a range of background elements so we will use an empty GameObject to contain all these features.

15. Start by creating an empty Game Object in the Hierarchy panel and name it Background - if needed set its position to 0, 0, 0. Now drag and drop the background sprite from the Hierarchy and place it over the Background GameObject so it becomes nested.



Duplicate the background sprite and set the X position of the new sprite to 8. These changes should result in the following:



If you play the game now you should see tha the background will continue scrolling forever. If your background is not scrolling ensure that you have the Scroll Speed value set to 5 for both the background elements. To add depth to our scene we will now introduce foreground elements and implement a parallax effect.

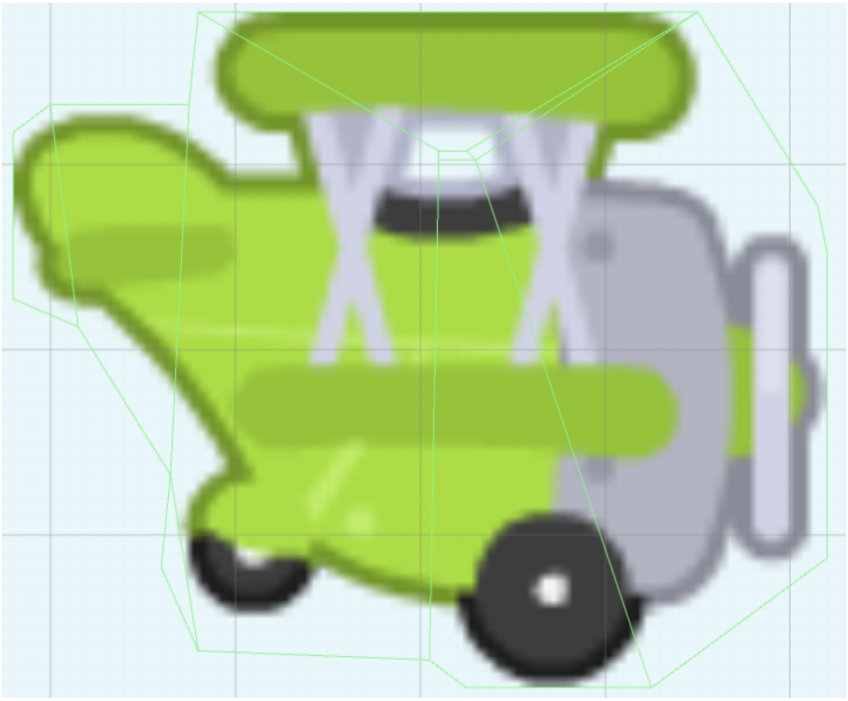
16. Drag and drop the groundGrass sprite from the downloaded Prac03 folder into the Scene panel. Rename it to Ground and set its position to 0, -2, 0. Attach the RepeatingBackground script to Ground and set the Scroll Speed property to 5. Drag Ground onto the Background GameObject. Duplicate Ground and set the second Ground's position to 8, -2, 0.

If you play it now you will see the ground and the background elements are scrolling from right to left. They will continue to scroll until the game ends. However, to enable the parallax effect we typically see items that are further in the background scroll slower than those in the foreground. To correct this set the Scroll Speed property of the background elements to 2.

While we are manipulating the layers of our game world we should ensure the Ground is set to an appropriate sorting leyer. Click on Ground and then in the Inspector panel find the Sorting Layer property. Click on Default then Add Sorting Layer... Click on the + sign and set the new Sorting Layer to Foreground – you list should now look like Background then Default then Foreground. Click on Ground in the Hierarchy panel and then hold Ctrl and click on Ground (1), find the Sorting Layer property in the Inspector panel and change this to Foreground.

17. We will now implement the player behaviour. Select the Plane Game Object from the Hierarchy panel and then in the Inspector panel add a Rigidbody 2D component (Add Component >> Physics 2D >> Rigidbody 2D). Change the Collision Detection property to Continuous. Now if you play the game you will see that gravity now works on the plane and it will fall off the bottom of the screen.

Obviously we don't want the plane to just fall off the world we want it to collide with the ground. To do this we will add a collider to the Plane. We could use a simple box or circle collider but we want to make the collisions as accurate as possible so instead we will use a Polygon Collider 2D. Add a new component to Plane (Add Component >> Physics 2D >> Polygon Collider 2D). Zooming right into the Plane in the Scene panel you can see the green collision lines around the object.



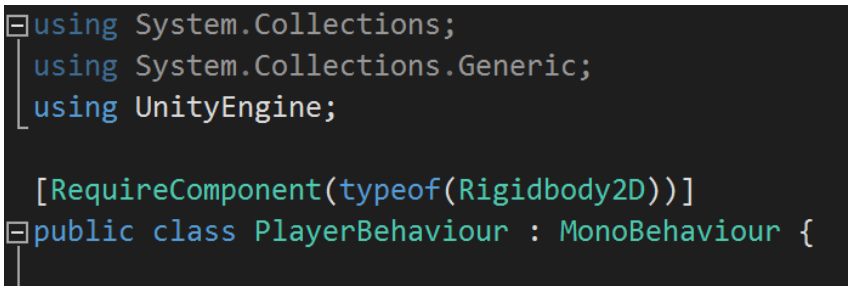
Add the same collision components to the two Ground Game Objects that you have in your game. When you play your game now you should see that the Plane will hit the Ground and will not fall through – you will notice that the game is still not functioning how we would expect though.

18. We will add code to make our Plane bob in the expected fashion for this type of game. Create a new Script in the Scripts folder of the Project panel and name the new script PlayerBehaviour. Double click the newly created script to begin editing it.

To ensure that the Plane (which we will attach this script to) behaves as expected we need to add a defining line to our class declaration. In the line just before `public class PlayerBehaviour...` you will need to add the following piece of code:

```
[RequireComponent(typeof(Rigidbody2D))]
```

If you are using Visual Studio then the top of your code will now look like:



Now we need to define some variables, place the following declarations before the `Start` function.

```
[Tooltip("The force which is added when the player jumps")]
public Vector2 jumpForce = new Vector2(0, 300);

private bool beenHit;

private Rigidbody2D rigidbody2D;
```

These variables will be used to determine how the bobbing action occurs and provide a mechanism that we can use to respond to collisions. Now in the `Start` function we will set values for the beenHit boolean variable and the Rigidbody variable. Add the following code to the `Start` function:

```
beenHit = false;
rigidbody2D = GetComponent<Rigidbody2D>();
```

The next task is to respond to user input. We will let the user play our game either by pressing the space bar or by clicking the left mouse button. To capture this input we will use a selection statement called an 'if' statement and to ensure that we complete all other Update operations first we will implement this code in a `LateUpdate` function.

Delete the existing Update function from your PlayerBehaviour code and add the following LateUpdate function in its place.

```
void LateUpdate () {
   if((Input.GetKeyUp("space")||Input.GetMouseButtonDown(0)) && !beenHit)
   {
     rigidbody2D.velocity = Vector2.zero;
     rigidbody2D.AddForce(jumpForce);
   }
}
```

The code above uses the OR symbol `||`, this means that we are testing to see whether the player has pressed the 'space' key OR the left mouse button. The other aspect of the statement is to work out whether the Plane has been hit. The exclamation mark used before beenHit is the NOT operator and the `&&` symbol represents AND, so we would read the whole statement as:
if the player has pressed the space key OR the left mouse button AND has NOT been hit.

If the above statement equates to true then we set the Planes velocity to zero and add our jump force value – the result is the Plane boobing up in the air briefly and then descending back down.

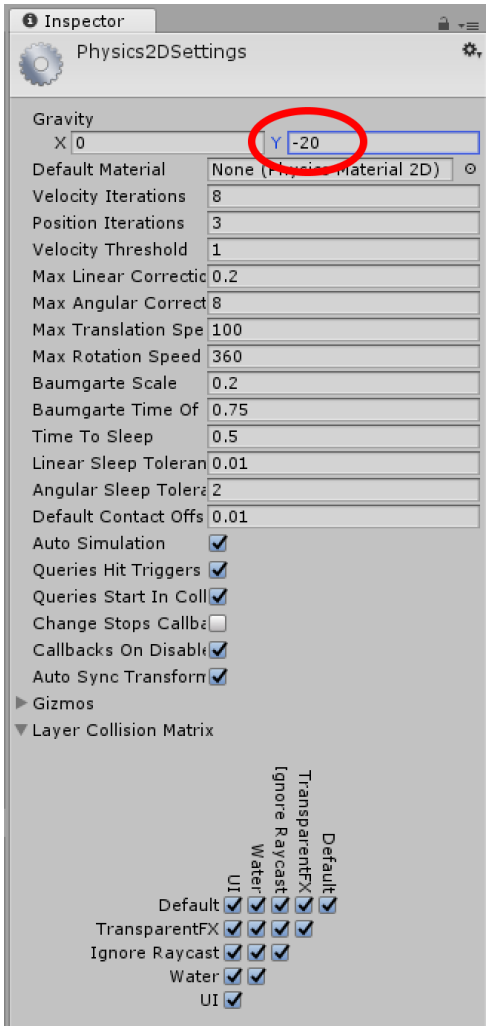
The last step for this piece of code at the moment is to respond to a collision. If the Plane collides with something (at this point this is just the Ground) then we want to set the boolean value to true and we want to stop the Plane propeller animating. We will use an OnCollisionEnter2D function to complete this action.

```
void OnCollisionEnter2D(Collision2D other)
{
   beenHit = true;
   GetComponent<Animator>().speed = 0.0f;
}
```

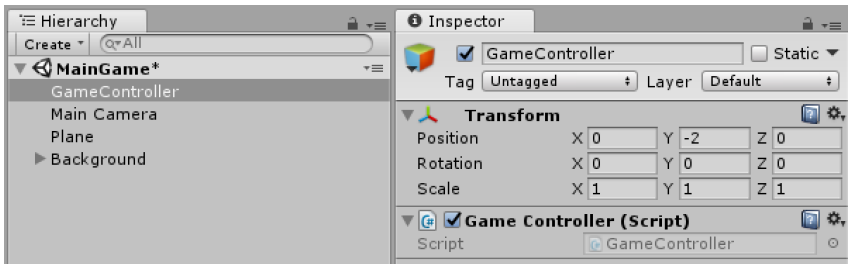
Save your changes and go back to the Unity engine. Attach the PlayerBehaviour script to the Plane, save your Scene and Project and then test the game. You should now see your Plane jump briefly into the air when you press the space bar or click the left mouse button. You may need to click in the Game panel once you start the game to ensure that the Game panel in the current focus for your system.

19. Before we continue on you will probably notice that the jump action is faster or more aggressive than you would ideally like for this type of gameplay. To correct this go to the Edit menu then Project Settings and select Physics 2D. In the properties that appear change the Gravity Y value to -20.

20. Now we want the game to respond correctly to the player crashing their Plane, that is, stop the background elements from scrolling. To complete this functionality we will create a new Script called GameController and define a variable that our other scripts will be able to use. The variable will control the speed of the repeating background elements. We will use the GameController class later as well for other game features.



Start by creating a new empty GameObject in the Hierarchy panel and name it GameController, drag it to the top of the Hierarchy list so it is easy for us to identify and access later on. Now add a new C# Script component and name it GameController.



Drag the newly created script from the Assets folder into the Scripts folder in the Project panel. Double click on the GameController script to edit it. We will define a new variable of type float and name it speedModifier and then set its value in the Start funciton to 1.0f. Add the following variable declartion just above the Start function:

```
[HideInInspector]
public static float speedModifier;
```

The initial statement will (as it suggests) hide the variable speedModifier in the Inspector. This means that it will be available to the program and other classes but the player cannot edit nor see its values in the Inspector. Now set the value of speedModifier to 1.0f in the Start function

```
void Start() {
   speedModifier = 1.0f;
}
```

Now that we have implemented this code we need to get the other classes within our game to make use of this new property. Open the `PlayerBehaviour` script and in the `OnCollisionEnter2D` function after the `beenHit = true;` statement add the following line of code:

```
GameController.speedModifier = 0;
```

What this is doing is telling our game that when the Plane collides with something set the speedModifier to 0. We can use this value within the RepeatingBackground class to set the position of our background elements and essentially stop any movement.

Open the RepeatingBackground class and find the FixedUpdate function. Within this function find the line:
`pos.x = pos.x - scrollSpeed * Time.deltaTime;` Modify this line of code so it looks like the following:

```
pos.x = pos.x - scrollSpeed * Time.deltaTime * GameController.speedModifier;
```

The execution of this sequence of changes that you have just made results in the following:

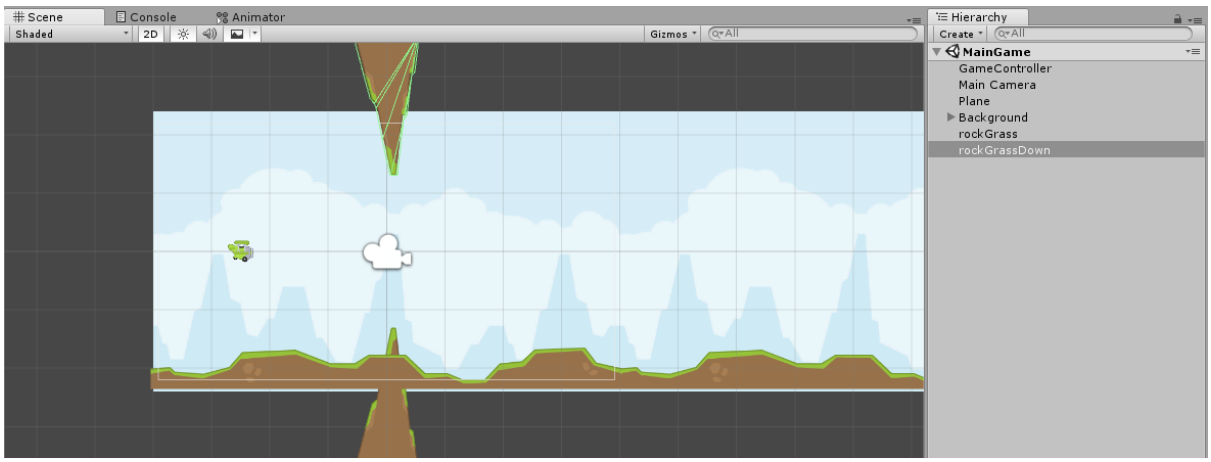
- When the game starts the speedModifier is set to 1
- While the game is playing the RepeatingBackground elements will move from right to left based on the scroll speed and modified by the speedModifer value of 1 (anything multiplied by 1 will retain its value)
- When the Plane collides with something the speedModifier is set to 0
- In RepeatingBackground the pos.x value is incremented by the scrollSpeed mulitplied by the new speedModifier value of 0 (any value mulitplied by zero will always result in 0)
- This will mean that the pos.x value will not change.

Save all changes to the various script files you have just edited, save the Scene and Project and then play the game to test the new funcitonality.

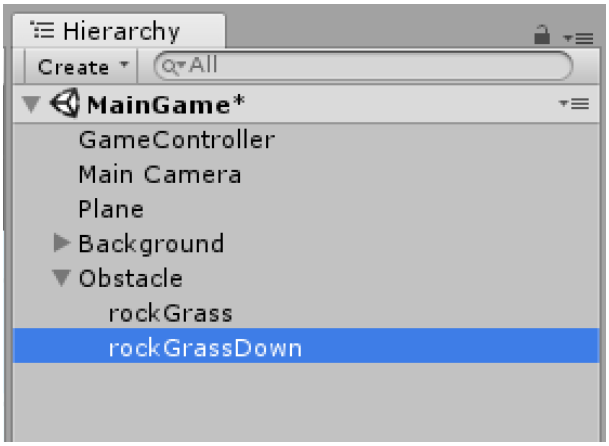
21. Now We will add some additional challenge to the game by including obstacles for the player to dodge. In the Prac03 folder drag and drop the rockGrass and rockGrassDown files into the Sprites folder in the Project panel.

Drag the rockGrass sprite into the Hierarchy panel and set its Position to 0, -2.5, 0. Add a Polygon Collider 2D component to it.

Drag the rockGrassDown sprite into the Hierarch panel ad set its position to 0, 2.5, 0. Add a Polygon Collider 2D component to it.



Now create a new empty GameObject in the Hierarchy panel and name it Obstacle, set its Position to 0, 0, 0. Drag and drop the rockGrass and rockGrassDown GameObjects into it.



We need the obstacles to move like the RepeatingBackground but we need to remove the obstacles from the game once they are off the screen. To do this we can Destroy the game object. Create a new Script called ObstacleBehaviour and save it in the Scripts folder. Open it in your editor to make some changes. Rather than starting from a blank canvas we will be making use of similar functionality already defined in our RepeatingBackground class. This will mean that the declaration for our new ObstacleBehaviour class will look a little different. Implement the following code **ensure that you replace MonoBehaviour with RepeatingBackground.*

```
public class ObstacleBehaviour :  RepeatingBackground {
   protected override void Offscreen(ref Vector3 pos)
   }
     Destroy(this.gameObject);
   }
}
```

Save your script and return back to Unity. Attach the new script to the Obstacle GameObject in the Hierarchy panel and set its Scroll Speed property to 5. Save the Scene and Project then run your game. You should see the Obstacle move to the left and then if you monitor the Scene view of your game you will see that once the Obstacle is out of the bounds of the screen it will be removed from the game. This mechanic will allow us to reuse this GameObject without overloading our system.

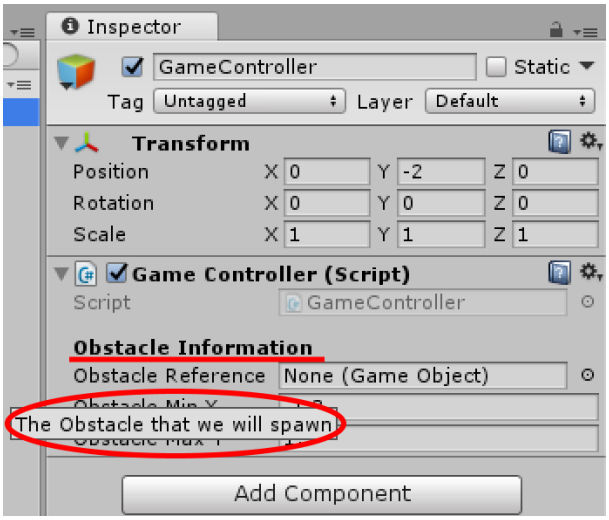
22. Dodging one obstacle for the game is not overly challenging so we want to be able to continuously spawn the Obstacles while the game is running. We will set the Obstacle as a Prefab and then set up code in our GameController script to spawn new Obstacles as needed.

Drag and drop the Obstacle GameObject from the Hierarchy panel into the Prefabs folder in the Project panel. Now click on the Obstacle GameObject in the Hierarch panel and hit Delete to remove it. Now we can implement the code to spawn the Obstacles in the GameController Script. Open the GameController script and define the following variables:

```
[Header("Obstacle Information")]

[Tooltip("The Obstacle that we will spawn")]
public GameObject obstacleReference;

[Tooltip("Minimum Y value used for obstacle")]
public float obstacleMinY = -1.3f;

[Tooltip("Maximum Y value used for obstacle")]
public float obstacleMaxY = 1.3f;
```

With the code implemented above we can see a new directive has been used. The Header directive will provide additional information within the Inspector panel about the properties associated with the script. The Header text is displayed in the Inpsector panel, see the image below demonstrating the header text (underlined) and the tool tip (circled).

While you are still in your GameObject code create a new function named CreateObstacle with the following statement:

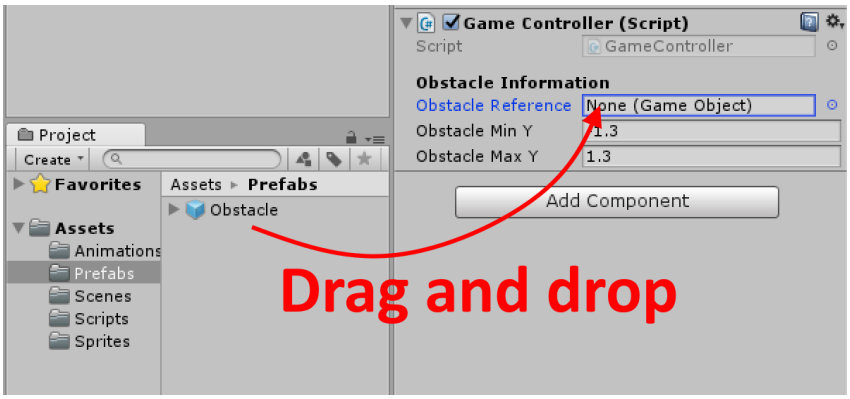



```
void CreateObstacle() {
   Instantiate(obstacleReference,
     new Vector3(RepeatingBackground.ScrollWidth,
     Random.Range(obstacleMinY, obstacleMaxY),0.0f),
     Quaternion.identity);
}
```

The last step in the code is to run the function we have just created at set times during the execution of our game. In the Start function add the following line of code:

```
InvokeRepeating("CreateObstacle", 1.5f, 1.0f);
```

This line of code will run the CreateObstacle function after 1.5 seconds and then every 1 second after that. Save the Scripts you have been working on and return to Unity. Select the GameController GameObject from the Hierarchy panel and in the Inspector panel locate the Obstacle Reference property. Drag and drop the Obstacle prefab from the Project panel into this Object Reference property.



Save the Scene and run your game to see the Obstacles spawning. If you hit one of these then the game will stop as it does if your Plane collides with the ground – you will notice

in the Scene panel however that when you crash your Plane the Obstacles will continue to spawn offscreen until your force the game to stop.

23. At the moment the game begins as soon as you hit the play button. We want to control this a little better and give the player an opportunity to prepare themselves before the game's mechanics drag their Plane to a premature demise.

Start by creating a new Script name GameStartBehaviour in the Scripts folder of the Project panel. Open up the file and modify the code so it looks like the following.

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

public class GameStartBehaviour :  MonoBehaviour {

   private GameObject player;

   // Use this for initialization
   void Start () {
     player = GameObject.Find("Plane");
     player.GetComponent<Rigidbody2D>().isKinematic = true;
   }

   // Update is called once per frame
   void Update () {
     if ((Input.GetKeyUp("space") || Input.GetMouseButtonDown(0)))
     {
       GameController controller = GetComponent<GameController>();
       controller.InvokeRepeating("CreateObstacle", 1.5f, 1.0f);

       player.GetComponent<Rigidbody2D>().isKinematic = false;

       Destroy(this);
     }
   }
}
```

The code when applied through the GameContoller code will provide a brief period of gameplay where the Plane slowly descends. The actual gameplay will commence when the player first click the mouse button or presses the space bar. To execute this code through the GameController replace the InvokeRepeating line (located in the Start function after the speedModifier line) in the GameController script with the following:

```
gameObject.AddComponent<GameStartBehaviour>();
```

Save your scripts, save your Scene and then play the game to test the new feature.

24. We will now add in code to make the game restart when the player Plane crashes after a short pause. The player will need to click their mouse button or press the space bar to

start the game again. Create a new Script called GameEndBehaviour and open it in your code editor. Fill out the script with the following statements:

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

public class GameEndBehaviour :  MonoBehaviour {

   private bool canQuit = false;

   // Use this for initialization
   void Start () {
     StartCoroutine(DelayQuit());

     GameController controller =
       GameObject.Find("GameController").GetComponent<GameController>();
     controller.CancelInvoke();
   }

   // Update is called once per frame
   void Update () {
     if((Input.GetKeyUp("space") || Input.GetMouseButtonDown(0)) && canQuit)
     {
       SceneManager.LoadScene(SceneManager.GetActiveScene().name);
     }
   }

   IEnumerator DelayQuit()
   {
     yield return new WaitForSeconds(.5f);
     canQuit = true;
   }
}
```

Pay careful attention to the using statements at the start of the class as we are using a SceneManagement library to load (or reload) the Scene when we die. The final step is to tell our game when this end behaviour should be executed. We can do that within the PlayerBehaviour class – namely when a collision has occured. Open up the PlayerBehaviour script and locate the OnCollisionEnter2D function. At the end of the function add the following code:

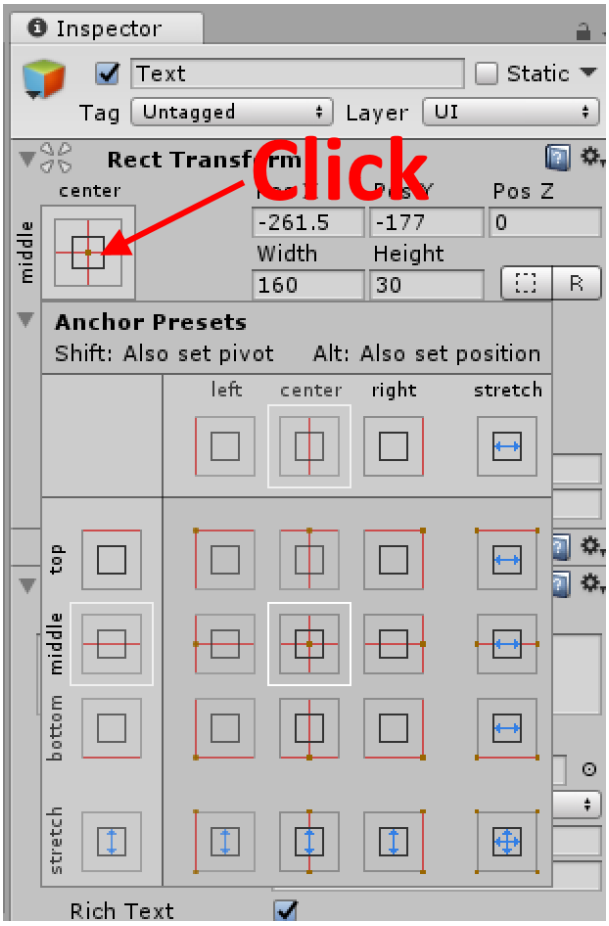
```
if (!gameObject.GetComponent<GameEndBehaviour>())
{
   gameObject.AddComponent<GameEndBehaviour>();
}
```

Once again, save all of your scripts, save the scene and play the game to test the end game functionality. After you have crashed your plane if you wait 0.5 seconds you can press the space bar or the mouse button to restart the game.

25. The last task remaining for the core of our Tappy Plane game is to track the player's score. For this type of game the score will be incremented for each obstacle that is successfully evaded. To display the score we need to use Unity's UI functionality.

Go to the Game Object menu, then to UI and click on Text. Double click on the Canvas object in the Hierarchy menu – you should notice it zooms out so that your game world is very small in the Scene panel.

Select the Text object in the Hierarchy panel and find the Anchor Presets component in the Inpector panel. With the Anchors Preset menu visible hold down the Alt and Shift

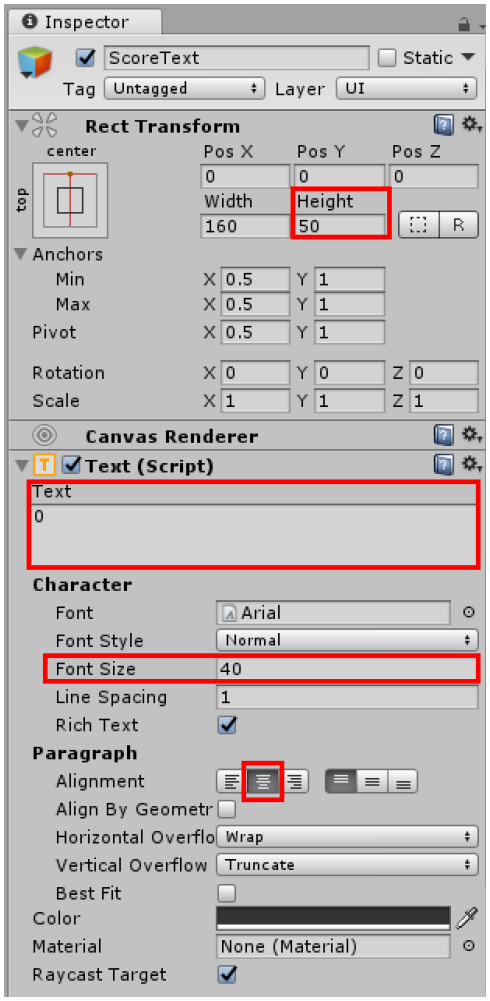


keys on the keyboard and select the top centre option. Now change the name of the GameObject to ScoreText.

Finally, adjust some of the properties of the ScoreText object:

- Set the Rect Transform Height property to 50
- Set the Text value to 0
- Set the Horizontal Alignment to Centered
- Set the Font Size to 40

Now you need to tell your game how to access and update the Score. Open the GameController code and add the following using statement to the top of your code:




```
using UnityEngine.UI;
```

Then you will need to add variables to the class so that the score can be recorded and retrieved. Add the following code to the variables for the class:

```
private static Text scoreText;
private static int score;

public static int Score
{
   get { return score; }
   set
   {
     score = value;
     scoreText.text = score.ToString();
   }
}
```

Finally, add the following lines of code to the end of the Start function:

```
score = 0;
scoreText = GameObject.Find("ScoreText").GetComponent<Text>();
```

Save your scripts and return to the Unity engine. We will use a box collider to determine when we have successfully evaded an obstacle. We will place a box collider into the space in the centre of the Obstacle and then set a trigger to determine when we enter that space. Upon entering the space we will increment the score.

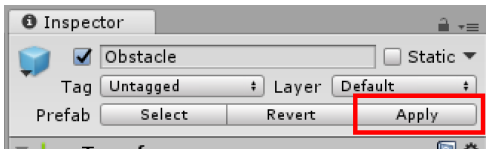
Open the Prefabs folder in the Project panel and drag and drop the Obstacle prefab into the Hierarchy panel.

Add a Box Collider 2D component to the Obstacle (Add Component >> Physics 2D >> Box Collider 2D). Set the Size property to 0.5, 3 and check the Is Trigger option.

Open the ObstacleBehaviour script and add the following function:

```
public void OnTriggerEnter2D(Collider2D collision)
{
   GameController.Score++;
}
```

Save the script and return to Unity. In the Inpector panel under the Prefabs section click on the Apply button.



Finally, delete the Obstacle from the Hierarchy panel, save the Scene, save your Project and test your working game.

Congratulations on getting to the end of the core tasks for these practicals!

Show your demonstrator the completed game and have your work marked off.

For the extension tasks you can complete any of the following, up to a maximum of 4%:

- (2%) Create multiple midground layers to display clouds sweeping past the player - the clouds can appear both in front of or behind the plane. The clouds should appear at "random" heights above the centre line of the game world, i.e., a cloud shouldn't apppear just above the ground.
- (2%) Create an enemy asset that will fly in the opposite direction to the player. The enemy plane does not behave with the same physics as the player - the enemy will fly in a straight line. If the player collides with the enemy then the game should stop as it currently does when the player collides with an existing obstacle.
- (2%) Finish the game experience by restricting the player to three lives. Once the player crashes three times display a game over screen. Use a static image of the plane to represent the lives remaining and display them at the top right of the screen.

- (2%) Create a start screen that has a single button to start the game. Once the game starts provide a countdown of three for the player to get prepared and then start the gameplay as indicated with the core steps above.
- (2%) Develop dynamic difficulty that increases the speed of the game based on the number of points scored by the player. It is suggested that for each 10 points the player earns increase the speed and therefore difficulty of the game appropriately (increase game speed by 1.5x for each 10 points earned).

Show your demonstrator the completed extension tasks and have it recorded.