Diptera Transcendence

Technical Write-up

Contents

[Outline 2](#_Toc435822836)

[Creating player 3](#_Toc435822837)

[Camera Movement 5](#_Toc435822838)

[Lanes 7](#_Toc435822839)

[Animating Fly 8](#_Toc435822840)

[Creating Webs 10](#_Toc435822841)

[Spawning Webs 12](#_Toc435822842)

[Fly – Web collisions 14](#_Toc435822843)

[GUI 15](#_Toc435822844)

[Ending the Game 18](#_Toc435822845)

[Particle Effects 19](#_Toc435822846)

[Activating Particle Effects 22](#_Toc435822847)

[Scaling up the Webs 24](#_Toc435822848)

[Multiple Cameras 25](#_Toc435822849)

[Oscillating Fly 28](#_Toc435822850)

[Scrolling Background 29](#_Toc435822851)

[Space 32](#_Toc435822852)

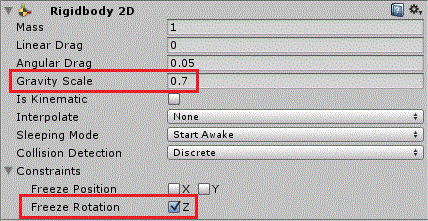
[Stars 33](#_Toc435822853)

# Outline

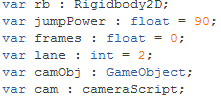
Diptera Transcendence is a game where you play as s fly whose dream is to fly higher than any living thing has ever flown before. The player will mash enter to fly higher and faster, while pressing Q and E to switch between three lanes to dodge obstacles. Rather than having a score the increases as you perform certain actions, this idea of this game is to get to then end as fast as possible. As such, players should aim to have to lowest time possible.

# Creating player

1. In **assets**, create the following five folders: **Images, Sounds, Animations, Prefabs and Scripts** (This is entirely optional, but will help keep things tidy).
2. **Import** the **fly sprite**.
3. Create a **2D sprite game object** called Fly.
4. Set **sprite type** to **multiple**, and **slice sprite automatically.**
5. Assign the **fly sprite** to the **sprite-renderer**.
6. Add a **Rigidbody2D** and a **BoxCollider2D** to the fly.
7. In the **Rigidbody2D** component, set the **gravity scale** to **0.7**, and **check freeze rotation Z.**



1. Click on the **Main Camera** game object. **Add a new script** called **cameraScript** ( we do not do anything with this yet, we only need it because we reference it in the Fly object)
2. Create a **new script** called **playerScript**. Open it in Monodevelop.
3. **Create** the following **variables** (some of these won’t be used until later sections).



rb is the Rigidbody2D of the fly. We need a reference to it to apply forces.

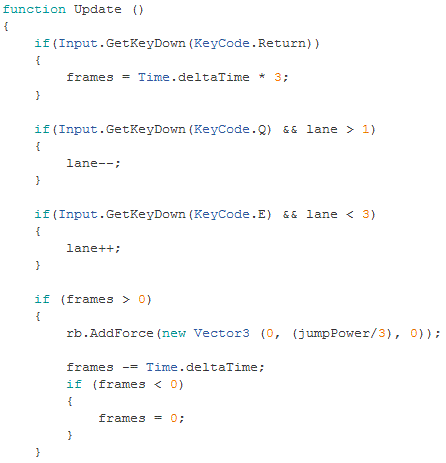
jumpPower is the force we will apply to the fly when the player presses enter.

frames is a variable used to smooth the application of jumpPower.

lane determines which of the three lanes the fly is in.

camObj and cam are used to calculate certain positions.

1. In the **Start()** function, assign **rb**, **camObj** and **cam** as shown below. C:\Users\user\Documents\College Things 2\Game Dev\Diptera Transcendence\Screenshots\screenshot2.gif
2. In the **Update()** function, check if the user pressed **enter**. If so, set **frames** to **Time.deltaTime \* 3**.
3. Check if the user pressed **Q**, and they are in a **lane higher than 1**. If so, **decrement lane**.
4. Check if the user pressed **E**, and they are in a **lane lower than 3**. If so, **increment lane**.
5. Check if **frames** is **greater than 0**. If so, **apply a force** of **(0, jumpPower/3, 0)** to the fly, **reduce frames by Time.deltaTime**. If **frames is less than 0**, **set frames to 0**.

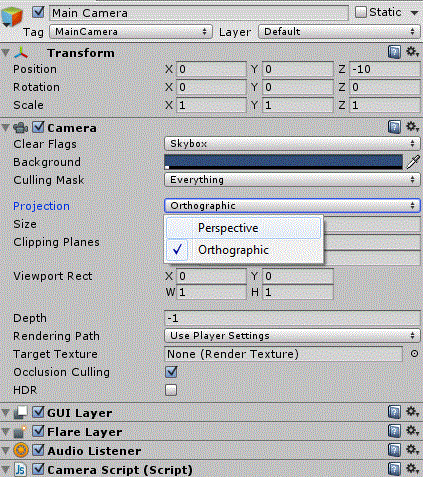


By using frames and jumpPower as we did above, we apply the jumpPower over three frames rather than in a single frame. This lends to smoother visuals in game.

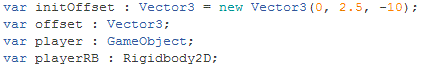
We need to add more features before we can actually use the lane variables, so at the moment there is no visible effect if the player presses Q or E. If they mash enter, they will fly off-screen. This issue will be addressed in the next section.

# Camera Movement

1. Click on the **Main Camera** in the **hierarchy**. Make sure that **projection** is set to **perspective**. This will allow the camera to show farther away images as smaller.



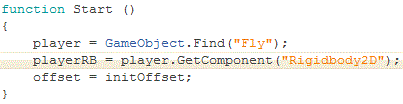
1. Open **cameraScript**.
2. **Create** the following **variables**.



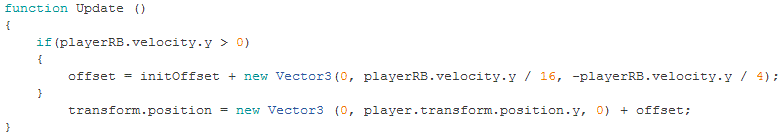
**initOffset** is as the base position for the camera. Assuming that Fly is at 0, 0, 0 This will make it stay about ¾ down the screen.

**offset** is a **Vector3** we will use to move the camera.

1. In the **Start()** function, assign the **player** and **playerRB** variables. Also, set **offset** to **initOffset**.



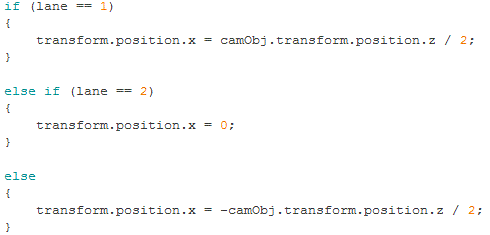
1. In the **Update()** function, check if **playerRB.velocity.y** is **greater than 0**. If so, **set offset** as shown below.
2. Set **transform.position** as shown below.



This code adjusts the cameras position relative to the players speed and position. The y and z values of offset are changed as the player speeds up, and move the camera up the y-axis, and out the z-axis. Effectually, the camera zooms out, and keeps the player relatively low in the view if the player is travelling upwards. After this calculation, the position is adjusted by this offset, and also by how high the player currently is, so that the camera does not get left behind.

# Lanes

1. Now that we have our camera set up, we return back to **playerScript**. At the end of the **Update()** function, check which **lane** the player is and adjust the **transform.position.x** as shown below.

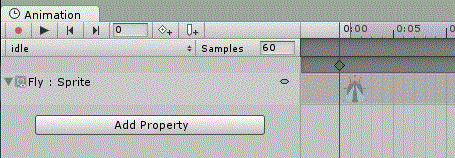


We have to set the position relative to the camera, because if we don’t there will be very little difference between the three lanes at high speed with the camera zoomed out. This code should mean that the player will always be ¼ , ½ , or ¾ , across the screen, regardless of how far out the camera is zoomed.

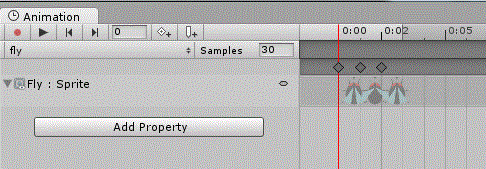
You should probably now test the game to make sure the camera follows the player, zooms out correctly, and that the lanes are working. If you are having trouble making the fly speed up, increase jumpPower in playerScript.

# Animating Fly

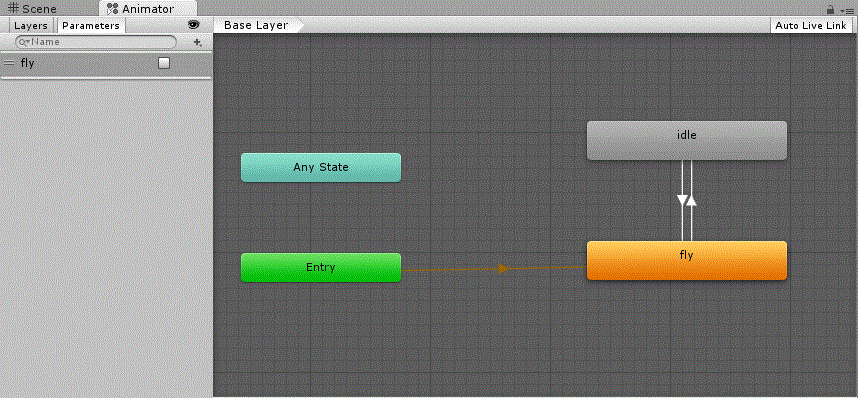
1. With Fly selected, go to window - animation. Create animation “idle”. Set samples to 60, but only use one image, fly\_0.



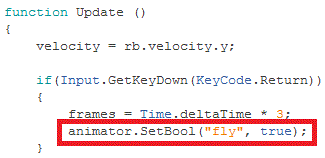
1. Create new animation “fly”. set samples to 30, and use fly\_1, fly\_0, fly\_1.



1. Now go to window - animator. Create a parameter of type boolean called fly. Make a transition from idle to fly and vice versa. Idle to fly should occur when the parameter fly is true. The opposite should occur when it is false.



1. Open playerScript.
2. Declare a variable animator of type Animator, and in the Start function initialize it with animator = GetComponent("Animator");
3. In the Update() function, set the animator Boolean fly to true in if the player hits “enter”



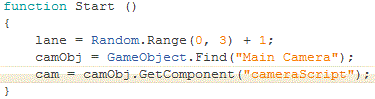
Now, whenever the player first hits enter to speed up the fly, the flies wings will start to flap.

# Creating Webs

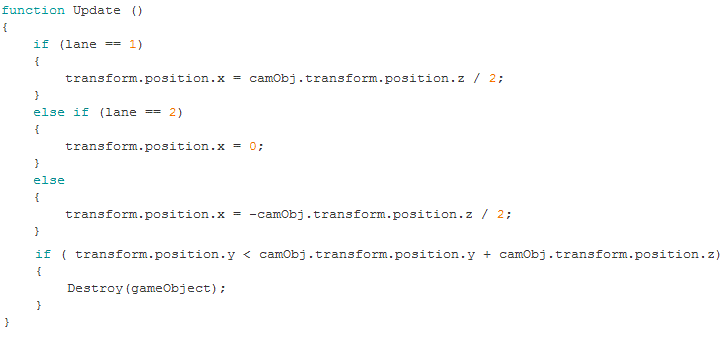
1. **Import** the **web** **sprite**.
2. Create a **2D sprite game object** called **Web**, and assign the **web sprite** to the **sprite renderer.**
3. Add a **Rigidbody2D** and a **BoxCollider2D**.
4. Add a **new script** called **webScript**.
5. Open **webScript** in monodevelop.
6. **Create** the **variables** seen below.

C:\Users\user\Documents\College Things 2\Game Dev\Diptera Transcendence\Screenshots\screenshot9.gif

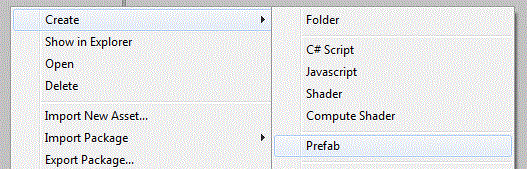
1. In the **Start()** function, assign **camObj** and cam as done previously. **Initialise** **lane** to a **random number** between **1 and 3**.



1. In **Update()**, we check the **lane** and adjust **position** as done in **playerScript**. We also need to **remove** the **webs** once they pass the player, so we use an **if statement** to check their **y position** relative to the **camera’s position**. In the **if statement**, we need to account for both the **cameras height**, **and how far out it has zoomed** (because the further out it has zoomed, the bigger the area it sees, and thus the webs have to fall further before being offscreen), so we use both its **y** **and z coordinates**.



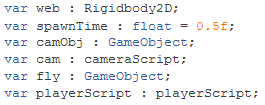
1. Now navigate to the **Prefabs** folder and create a new **prefab**.



1. Drag the **Web** to this new **prefab**. You can now delete it from the **scene**.

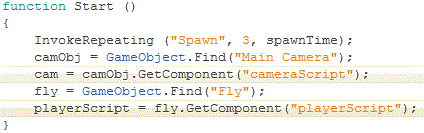
# Spawning Webs

1. **Create** a new **empty game object** called **Spawner**.
2. Add a **new script** called **spawnerScript**. Open it in monodevelop.
3. **Create** the following **variables**.

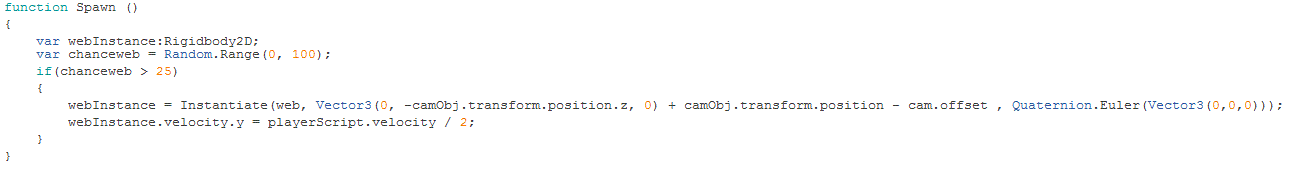


**spawnTime** is used for an **InvokeRepeating** call. It is not necessary, but allows us to adjust the rate of spawning from the inspector instead of needing to open the script.

1. In the **Start()** function we must once again assign the **camObj**, **cam, fly and playerScript** variables. We also call **InvokeRepeating(“spawn”, 3, spawnTime)**.



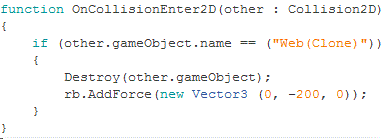
1. **Declare** a **Spawn()** function.
2. **Declare** the variable **webInstance** of type **Rigidbody2D**. Declare the variable **chanceWeb** and set it to a **random number between 0 and 100**.
3. Check if the variable is above a certain number (the higher the number, the lower the chance of a web). **If it is greater, instantiate a webInstance**. It’s **position** should be: **Vector3(0, -camObj.transform.position.z, 0) + camObj.transform.position – cam.offset**.
4. **Set the y velocity** of the new **Web** to **half** the **y** **velocity of the Fly.**



This code essentially makes the webs spawn at the top of the screen by making their y coordinate relative to how far the camera has zoomed out, adding the camera’s current position, and finally subtracting the camera’s offset to place it at the same depth as the Fly. We set the webs velocity relative to the Fly to reduce the speed at which they approach the player, making the game more fair.

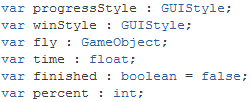
# Fly – Web collisions

1. Open the **playerScript**.
2. Create the **OnCollisionEnter2D(other : collision2D)** function.
3. Check if the **name** of the **other collision** is **“Web(Clone)”.** If so, **destroy** the **web**, then we **add a downward force** to the **player**, to impede their progress.



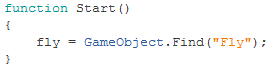
# GUI

1. Import the **textBackground** image.
2. Create an **empty game object**, and name it **GUI**. We will use this to display how far the player has come, and show a victory message when they win, along with time taken.
3. Attach a **new script** to **GUI**, and call it **GUIScript**. Open it in MonoDevelop.
4. **Create** the following **variables**:

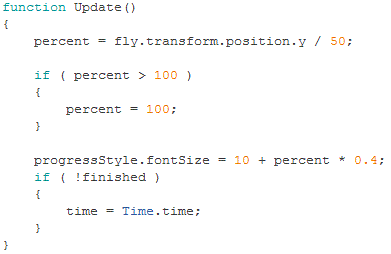


The **GUIStyles** are used to format the GUI elements. Time will be used to store the **players time taken** to reach the end. **finished** will be used to stop time from increasing once the player reaches the end. **percent** will be used to store the players **current progress** (as a percentage). We will say the player has reached the end once they **achieve a y position greater than 5000.**

1. In the **Start()** function, we must **initialise** the **fly** variable.



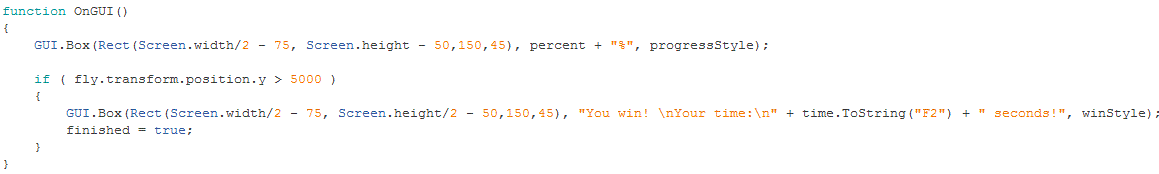
1. In the **Update()** function, we need to keep **percent** and **time** up-to-date (before updating **time**, we check to make sure the **player hasn’t finished yet**). **If percent is greater than 100, set it to 100**. We will also increase **progressStyle.fontSize** relative to **percent**.



The game ends when the player reaches 5000 on the y-axis, so to get their progress as a percentage, we **divide their y position by 50**.

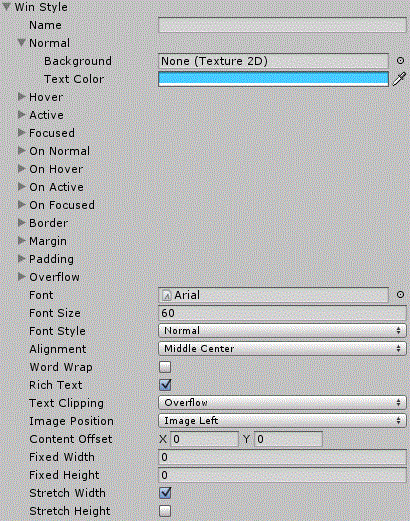
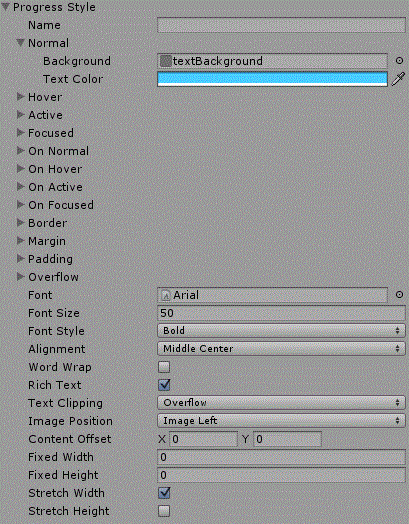
By adding the line of coding that changes **progressStyle.fontSize**, when we create actual GUI elements to display in our game, the size of the font will grow as the player gets further. This will hopefully making the player feel more excited, because people like **big, growing numbers**.

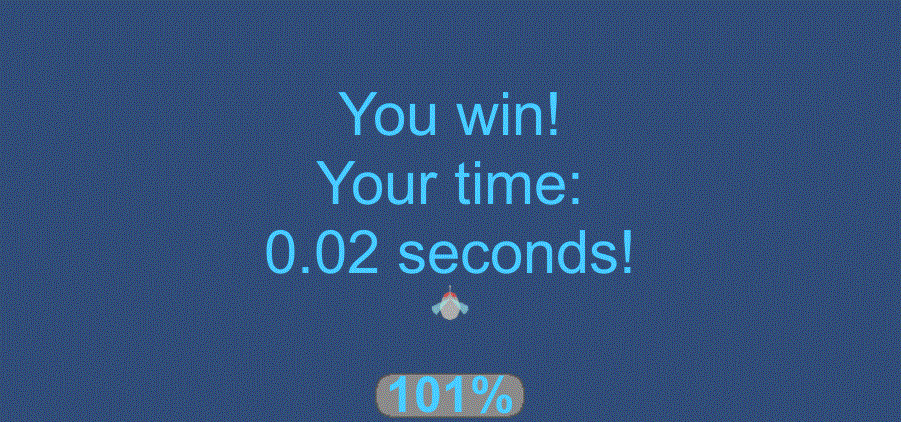
1. **Create** the **OnGUI()** function. In this function, we will **display the players percentage completion**, and once they are **above 5000, a victory message displaying their time** ( we also set **finished** to **true**).



**Screen.width** and **Screen.height** return the **width** and **height** of the **screen** in **pixels**. We use **ToString(“F2”)** to format **time** to **two decimal places**, for a cleaner looking message.

1. Go back to the **inspector**, and edit **winStyle** and **progressStyle**. These can look like whatever you want, so feel free to do your own thing. Below are screenshots showing the settings I used.



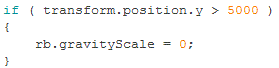


An easy way to test the victory message, is to change the **y position** of the **Fly** directly to **5000** in the **inspector**. **Don’t forget to change it back when you’re done.**

# Ending the Game

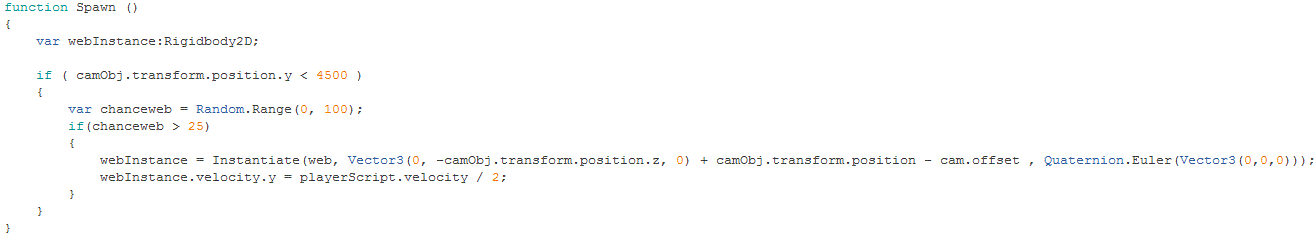
Now that we have a victory message displaying, we should probably make the game stop. Or at the very least, stop spewing webs at the player, and stop gravity from pulling them back down to Earth. To this we need to make a couple of minor changes.

1. Open **playerScript**.
2. In the **Update()** function, we check if the **players y coordinate is above 5000**, and if so, we set the **gravity scale to 0**.



All this does is stop the player being affected by **gravity**.

1. Open the **spawnerScript**.
2. Inside the **Spawn()** function, move the code for spawning webs into **an if statement** that checks the **player is below 4500**.

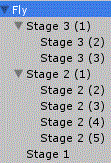


Now the Fly’s life should be very peaceful after they finish the game.

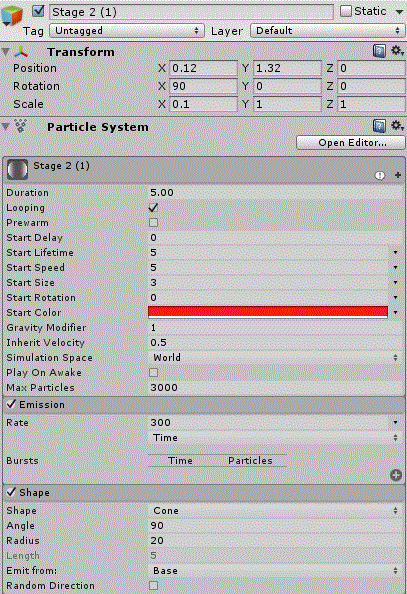
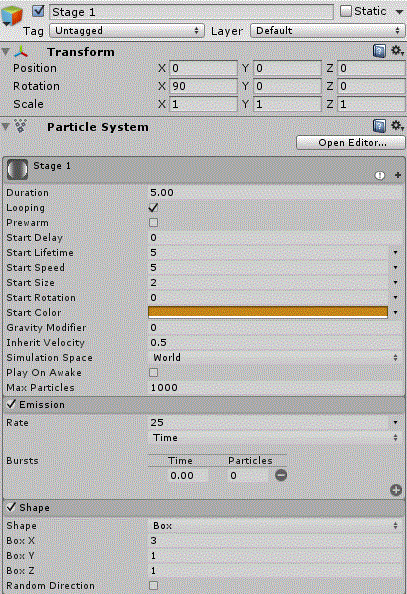
# Particle Effects

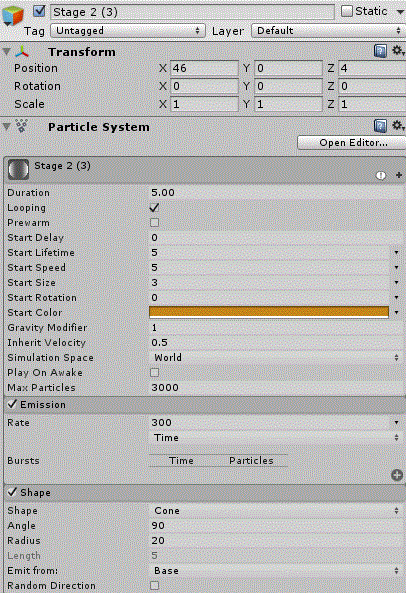
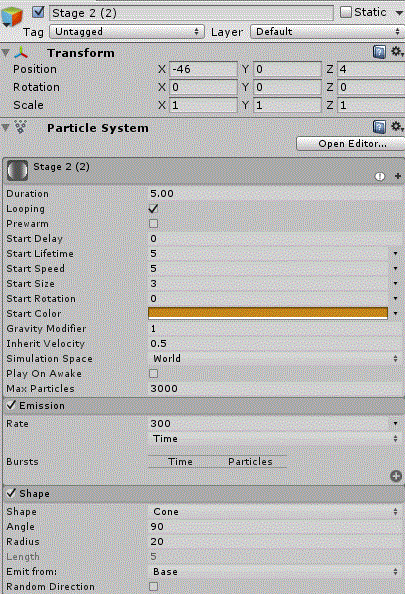
The game is pretty much finished insofar as all the mechanics work. Now all that’s left to do is **spice it up** with graphics such as backgrounds and **particle effects**. The aim of these **particle effects** is to turn the fly into a fireball as he picks up speed and exits the Earth’s atmosphere. Similarly to the GUIStyles, the particle effects should be experimented with, although I will show the settings I used for each one. What **IS** important, is that we have **three separate groups of particle effects**.

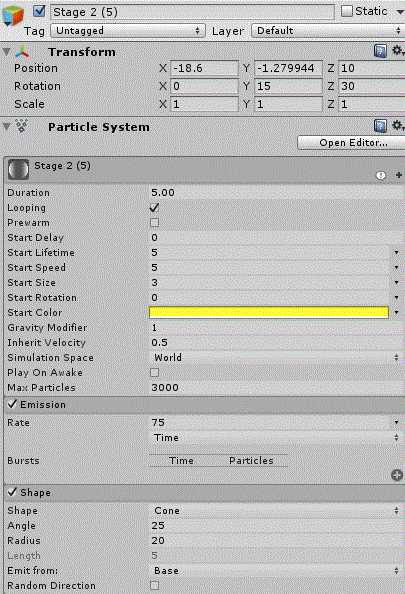
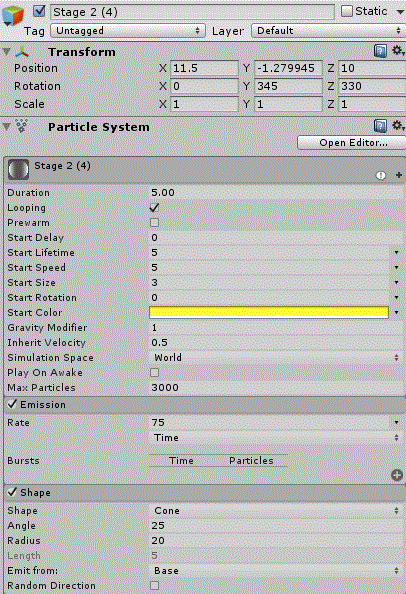
1. In the **hierarchy**, create **three particle systems** as **children** of **Fly**. Name them **stage1, stage2, and stage3**. Each of these can have **children** if you wish, depending on how complex you wish the final effect to look. Below is my set-up.

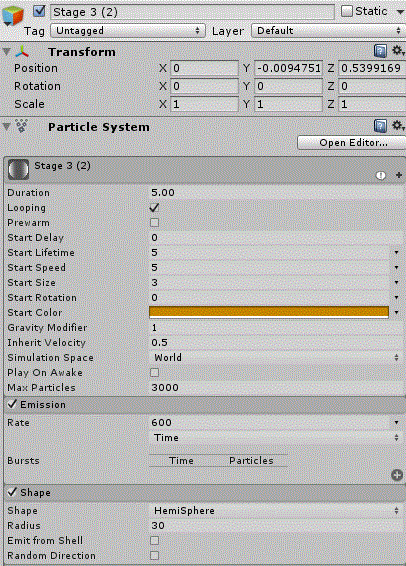
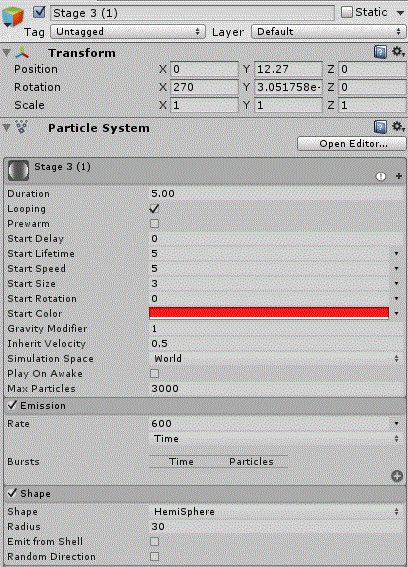


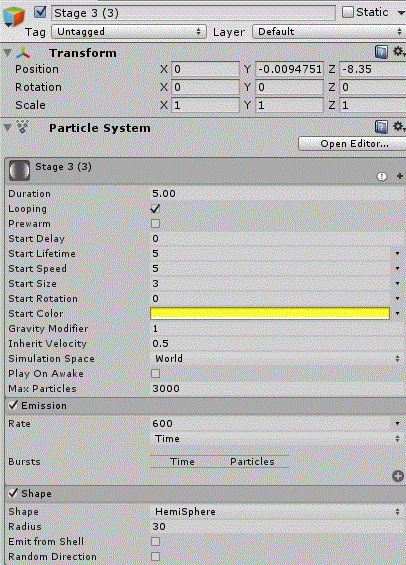
**Stage1** has no children, **stage2** has four children, and **stage3** has two children. Below follows a (long) series of screenshots showing all the different settings I used. Feel free to ignore this, and use simpler settings if you wish. **Play On Awake MUST be unchecked** (can be turned on for testing, but should be turned off afterward).







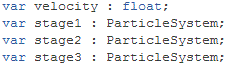




# Activating Particle Effects

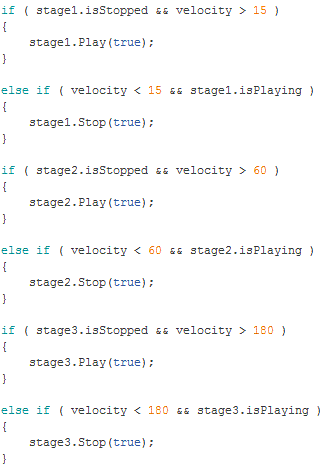
Once we have the three stages of particle effects made, the next step is to activate them one at a time as the fly reaches certain speeds.

1. Open **playerScript**.
2. **Create** the following **variables**:



The **ParticleSystem** variables must be assigned in the **inspector** later. Make sure that it is the **parent** of each stage that is **assigned** to the **variable**.

1. In the **Update()** function, for each stage we must check if the **velocity** is **above a certain amount** and that the **particleSystem** is **stopped**, and if so we **turn it on**. We then check to see if the **player is below a certain velocity** and the **particleSystem** is **playing**, and if so we **turn it off**.



This code causes the **particleSystems** to start **playing** as the **velocity** increases, creating a feeling of speed and progression for the player. **isStopped** and **isPlaying** checks if the system is currently stopped/playing and returns a **Boolean**. .**Play()** and .**Stop()** **play** and **stop** a **particle** **System**. If we pass in **true**, it makes all the **children** of the system **play**/**stop** as well. Passing in **false** means only the **parent** will **stop**.

You should now probably test the game to see if all the particle effects trigger correctly.

# Scaling up the Webs

At high speeds, the **webs** become very small and hard to see, which makes the game somewhat **unfair**. To solve this problem, all we have to do is make the **webs** **bigger** as the **player** **speeds** **up** and the **camera** **zooms** **out**.

1. Open the **webScript**.
2. **Add** the following **variables**:

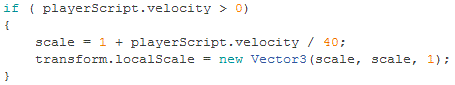
C:\Users\user\Documents\College Things 2\Game Dev\Diptera Transcendence\Screenshots\screenshot44.gif

We need to **player** **script** to access the **velocity** variable in it. **Scale** will hold a value used to **calculate** **the** **new** **scale** of the **webs**.

1. In the **Start()** function, add to following lines of code to **initialise** the **fly** and **playerScript** variables:

C:\Users\user\Documents\College Things 2\Game Dev\Diptera Transcendence\Screenshots\screenshot45.gif

1. In the **Update()** function we check if the **velocity** is **positive**. If so, we set **scale** to **one plus one tenth of velocity**. We then use **transform.localScale** to adjust the **x and y scale** of the **web**.



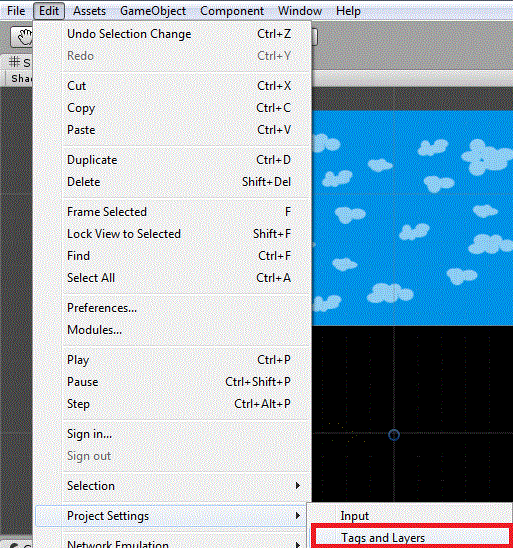
The **webs** should now remain **large** and very **visible** for the entire game, as their **size** **will** **grow** as the **camera** **zooms** **out**.

Quickly test the game to make sure this works correctly.

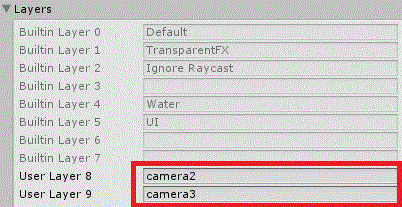
# Multiple Cameras

This game utilises **more than one camera** to achieve certain visual effects. But before we get to these effects, we need to set-up a few new **layers**.

1. Go to **edit – project settings – tags and layers**

****

1. Create two new **layers** and call them **camera2, and camera3**.

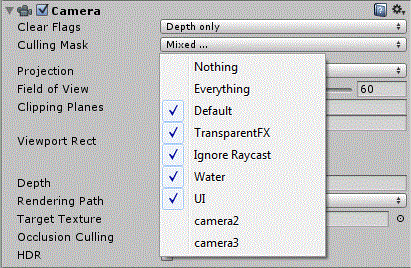
****

1. Now create two new **empty gameObjects**. Call them **camera2, and camera3**. Make them **children** of **Main Camera**.

**C:\Users\user\Documents\College Things 2\Game Dev\Diptera Transcendence\Screenshots\screenshot58.gif**

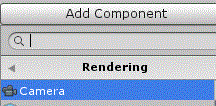
We do this because we want to make sure they stay at the **same** **position** **as** **Main** **Camera**.

1. Go to **Main Camera**. Set the **Clear Flags**  to **Depth Only**. In **Culling Mask**, make sure **every layer EXCEPT camera2 and camera3 is checked**.

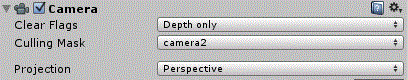


The **Culling** **Mask** basically means that whatever **layers** you have **checked** are the only **layers** that **camera** will **render**. So now, the **Main** **Camera** will **not** **render** any **objects in the camera2 or camera3 layers**.

1. Go to **camera2**. **Add a camera component** (by **add component – rendering – camera**).



1. Set the **Clear Flags**  to **Depth Only**.
2. Set the **Culling Mask**  to **camera2**.
3. Set the **Projection**  to **Perspective**.



1. Carry out the **last four steps** again for **camera3**, expect set **Culling Mask** to **camera3**, and **Projection** to **Orthographic**.

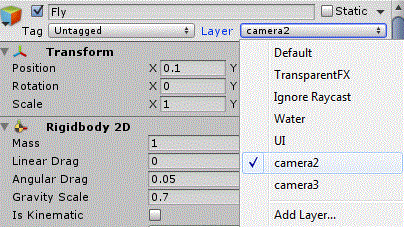
****

The cameras settings are different, as each will be used for different effects, which will be seen later.

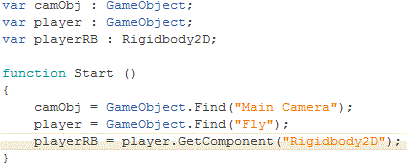
# Oscillating Fly

The first effect we will apply, is to make it appear as if the **Fly** is oscillating (or shaking) as he flies through the sky. This is intended to emphasise the velocity and kinetic energy of the Fly.

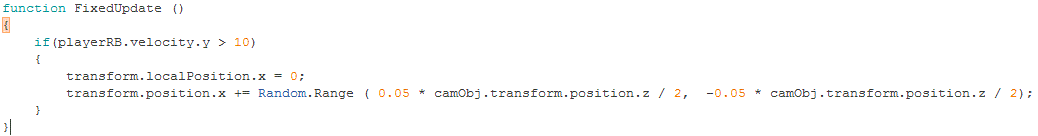
1. Set the **Fly’s Layer** to **camera2**. Unity will ask if you want to **apply this change to Fly’s children**. Click yes.



1. Go to **camera2**. Attach a **new script** to it, called **Oscillation**.
2. **Declare** the following **variables** and **initialise** them as shown.



1. In the **Update()** function, **check if the Fly’s velocity is greater than an amount.** If so, **set localPosition.x** to 0. Then **set localPosition.x** to a **Random number relative to Main Camera’s z position**.

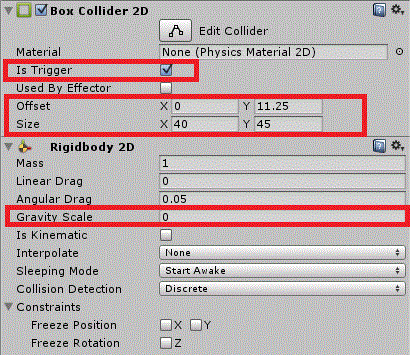


This code **centers** the **camera**, then **moves it out a small amount** relative to the **camera’s** **zoom** every frame. Since this **camera** only **renders** the **Fly** **and it’s children**, this gives the effect of the fly shaking violently one it passes a certain speed.

# Scrolling Background

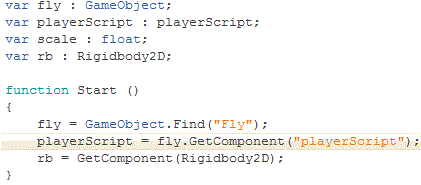
This **scrolling** **background** will work **differently** to the type used in previous practicals. It will **NOT** be handled with an animation, as the **position** AND **speed** of scrolling is **dependent** **on** the speed and location of the **Fly**.

1. **Import** the **sky** image.
2. Create a **2D sprite GameObject**, call it **sky**. **Assign** the **sky** **sprite** to it. Set it’s **x and y scale** to **4**.
3. Create a **2D sprite GameObject** as a **child** of **sky**, call it **sky2**. **Assign** the **sky** **sprite** to it. Set it’s **y** **position** to **22.5** (so that it is directly above it’s parent).
4. Add a **BoxCollide2D** and **Rigidbody2D** to **sky**. **Check** **isTrigger**, and set **Gravity** **Scale** to **0**. Edit the **size** of the **BoxCollider** so that it **covers** **BOTH** **sky** and **sky2**.



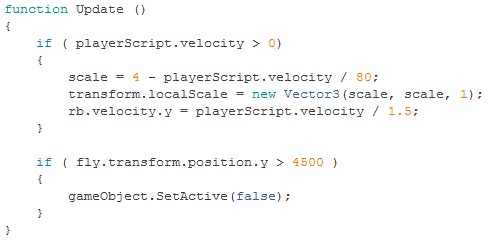
By **checking** **isTrigger**, **sky will not react to the physics of a collision**, and the **collider** **will only be used for collision functions in scripts**, rather than for physics simulation.

1. Set the **layer** to **camera3**.
2. Add a **new script** to **sky**, and call it **sky**.
3. **Declare** and **initialise** the **variables** as seen below.

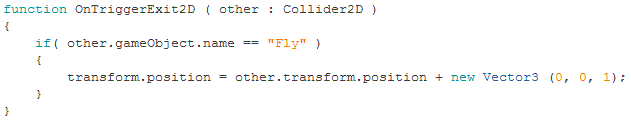


**Scale** is a variable that we will use to change the **scale** of the sky (since the **camera** we are using to **render** it is **orthographic**, **it will not get smaller as we zoom out**. Instead we change it’s scale in the script, so that it appears to zoom out, but at a lower rate, as the sky would realistically do).

1. In the **Update**() function, check **if** the **Fly’s** **velocity** is **greater** **than** **0**. If so, **set** **scale** to **4** (the value we set in the inspector) **– the Fly’s velocity / 80**. Set the **localScale** using the **scale variable as the x and y values**. We then set the **sky’s y velocity** to the **Fly’s y velocity / 1.5**. This ensures the sky moves upward, but slower than the fly, giving a sense of distance.
2. Check **if** the **Fly** is **above** **4500** (near the end of the game). If so, **disable** the **sky** (we will add something in it’s place shortly).



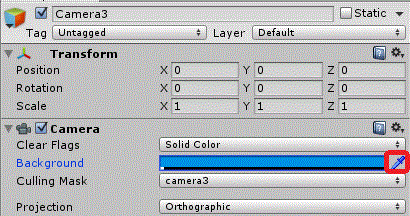
1. **Create** **OnTriggerExit2D**() function.
2. Check if **other**.**name** is **Fly**. If so, set the **sky’s** **position** to the **Fly’s** **position** (**+1 on the z axis**).



**OnTriggerExit2D**() is called whenever a **collider** marked as a **trigger** **leaves** **contact** **with another collider**. We use this to move the sky upward once the fly passes it.

If you test the game now, you will notice that this loop is **NOT** perfect, and gaps can be seen. To solve this:

1. Go to **camera3**.
2. Set **Clear** **Flags** to **Solid** **Color**. Use the **color** **picker** to set the colour to the **blue of the sky**.

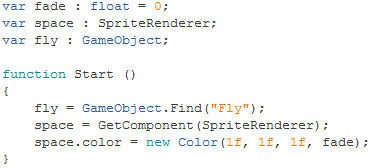


This should make the background colour the same as the sky colour, masking the gaps in the looping.

# Space

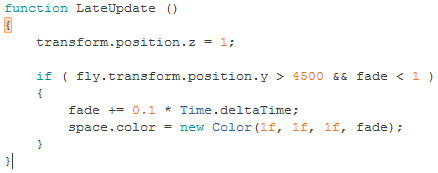
Finally, we’re adding outer **space** to the game. This is the fly’s ultimate goal, and is what **will appear when the** **sky** **is** **disabled**.

1. **Import** the **space** image.
2. Create a **2D** **sprite** **gameObject**. Call it **space**. **Assign** the **space** **image** to it in the sprite renderer.
3. Make **space** a **child** of **Main** **Camera** (so that it follows the camera).
4. Set the **layer** to **camera3**.
5. Set it’s **x and y scale to 4**.
6. Create a **new** **script** and call it **space**.
7. **Declare** and **initialise** the **variables** shown below.



We will use the value of **fade**, and the **SpriteRenderer** to **fade in the space sprite**. At the start, the colour of the **SpriteRenderer** is set to **1f, 1f, 1f, fade (which is 0).** These values are **RGB**, and **alpha** (**opacity**). By having the value corresponding to **opacity as 0, we make the image invisible**.

1. Create a **LateUpdate**() function. This function is **called AFTER update**.
2. Set the **z** **position** **to 1**.
3. Check if the **fly** is **above** **4500**, and that **fade** is **less** **than** **1**. If so, **increase** **fade** by **0.1 \* Time.deltaTime**, and **set** **space.color to 1f, 1f, 1f, fade.**

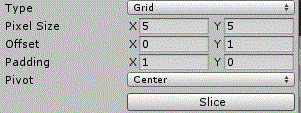


This causes space **to fade in once Fly is above 4500**. By **increasing** **fade** **each** **frame**, and using it to **set the color**, we get a **gradual** **fade** **effect**. We set the **z position to 1 each frame**, because as a **child** of **Main** **Camera**, space would move with it, and by moving on the **z axis**, block out everything else. This lets it keep an appropriate **depth**. You should now test the game to make sure this works.

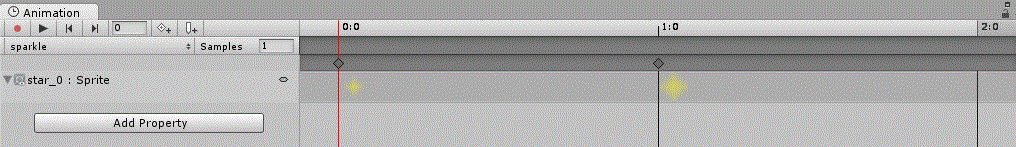
# Stars

Space, at the moment, is just a dark void. This is boring. To make it a bit more exciting and rewarding for the player, we will add some twinkling stars.

1. **Import** the **star** image.
2. Set **sprite** **type** to **multiple**, and **slice**. Unfortunately, the **transparency** in this image **causes some trouble and automatic slicing doesn’t work right**. Instead, use these settings:

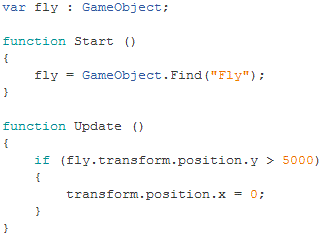


1. Create an **empty** **gameObject** called **stars**, and make it a **child** of **space**
2. Create **another** **empty** **gameObject** as a **child** of **stars**. Call it **star\_0**.
3. Go to **Window** – **Animation**. Create an **animation** called **sparkle\_0**. Set **samples** to **1**. Use **image** **star\_0** and **star\_1**.



Since this is the only animation for this object, we do not need to do anything with the animator window. This animation should play like a small, slowing sparkling star.

1. Follow the **last** **two** **steps** again for **another** **object** called **star\_1**. This time, **use** **star\_1 then star\_0** in the **animation**. We do this so that all the stars will **not be synchronised**, and will look **more** **realistic**.
2. Position as many of these stars as you like around the **scene** (use **ctrl** **D** to duplicate).
3. Now go back to stars (the parent), and change it’s x position to -5.
4. Create a new script called stars.
5. This script is very short and simple, so copy it out from below.



All this script does is check if the fly is above 5000, and then moves the stars back to 0 on the axis (back into view of the camera).

The **Hierarchy** may be a bit confusing now with all the **parent – child relationships**, so check your hierarchy against this screenshot:

