COMP4002/G54GAM Lab Exercise 02 – Platformer

10/02/20

This lab exercise involves creating a traditional scrolling two-dimensional platform game (think Mario, Braid), that develops a slightly more sophisticated player controller, and makes use of animated sprites and tile sets. This will involve separating the visuals from the mechanics and controls.

The game should have the following features:

- A character that the player can run left and right, and jump
- Animations that show the character running, or standing still as appropriate
- A camera that follows the character as it moves through the level (to achieve the "scrolling" element of the game)
- Hazards that move up and down for the player to avoid, that when hit restart the game
- A series of platforms that experiment with "platforming" in different ways for the player to navigate

Creating the Environment

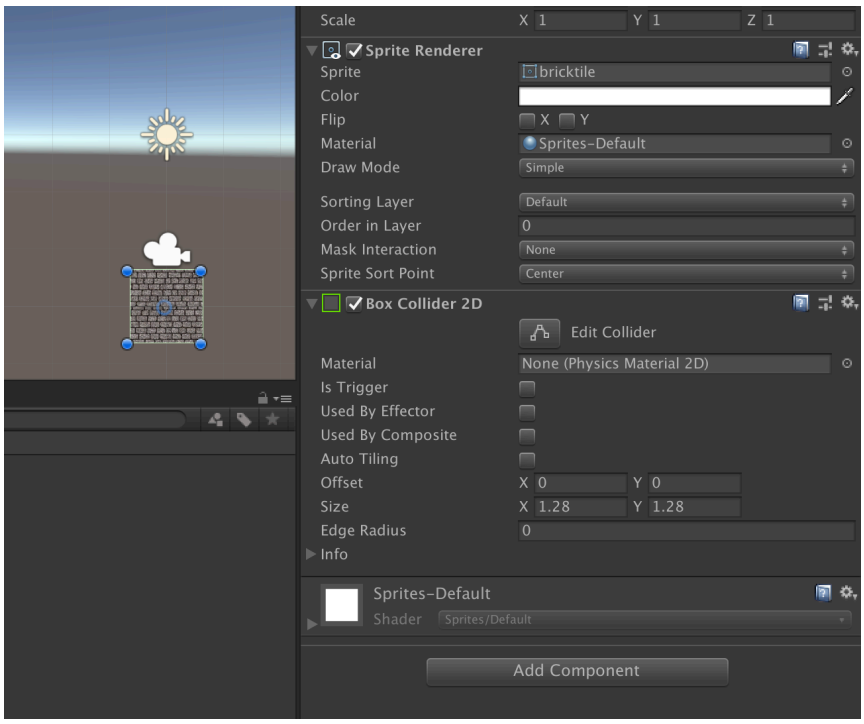
Create a new Unity project, but this time select “2D” as the template. This makes use of Unity’s two-dimensional mode, which essentially removes the z-axis from the scene, leaving just the x and y axes. Clicking the “2D” button in the Scene view automatically switches between a 3D and a 2D view.

The first task is to create a simple environment (some platforms) and the player’s character, and to allow the player to move the character around. Platform games are often derived from simple movements – left, right, and jumping, with gravity bringing the player back down to earth.

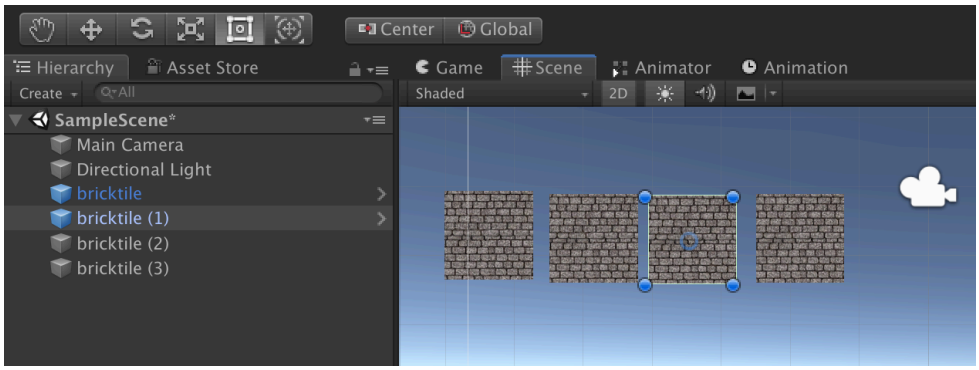
This game will involve sprites instead of models to display the environment and the player, and anything else that will be in the game. Sprites are essentially bitmap textures (i.e. pixel-based art) and the sprites for a game are often of the same size to enable regular tiling of the sprites when creating a level.

Import the images from the asset zip file by dragging them onto the assets tab in Unity. Unity will import the assets as textures, but as it is in 2D mode will also automatically generate sprite assets.

There are two ways to make use of the sprites. Dragging the *bricktile* sprite from the assets folder onto the hierarchy will prompt Unity to create an appropriate GameObject, with a *Sprite Renderer* component. To this we can add a *Box Collider 2D* component – similar to the capsule collider used in the previous exercise, but only operating in two dimensions – to give the sprite a physical as well as visual presence. The box collider is particularly appropriate for square tiles.

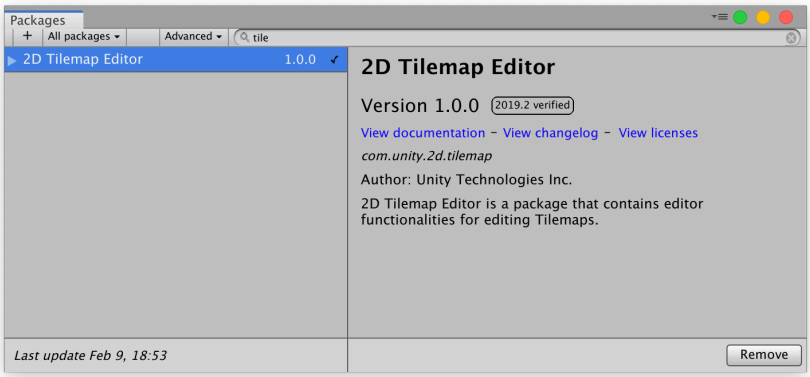


This object can then be dragged into the assets folder to create a reusable *Prefab*, and this can be used to create multiple instances of the sprite in the scene to build a primitive platform.



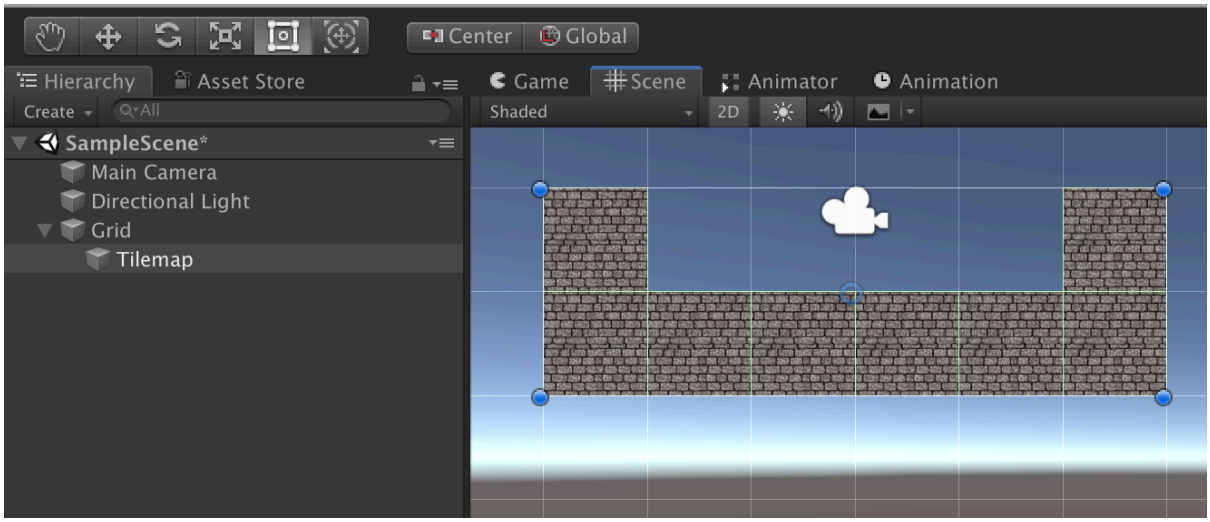
While useful for very simple levels, this is not a practical way to create a complex layout of sprites for a platform game. Instead, a common approach is to think of how to use sprites as *tiles*, that as they are the same size can be laid out uniformly via a grid.

Note – if you are using a newer version of Unity, i.e. 2019, you will likely need to include the Tilemap package via the Package Manager (Window->Package Manager). Search for Tile and click Install to add the package to your project.



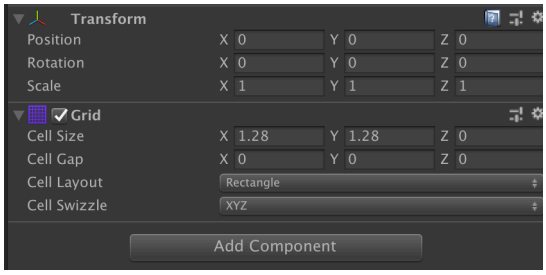
Create a new *Tile* asset (right click in the assets tab, Create->Tile), and assign its sprite to be the bricktile sprite.

In the hierarchy create a *Tilemap* object (right click in the hierarchy, Create->2D Object->Tilemap). This will show a grid that snaps tiles into a uniform layout, but also encapsulates all of the tiles as one object.



With the *Tilemap* selected you can drag the brick *tile* asset onto the grid and create a simple platform. Note that by default the grid has a size of 1 unit, or 100 pixels, and the sprite is 128 pixels square. To fix this, select the grid and set the cell size in the inspect to be 1.28 units.

Dragging tiles one by one onto the map is still not the most efficient mechanism for authoring a level. Open the *Tile Palette* (Window->2D->Tile Palette) and create a new palette. Dragging an existing Tile onto the palette allows it to be selected for painting onto the selected Tilemap, but also erase, fill, paint large areas etc.

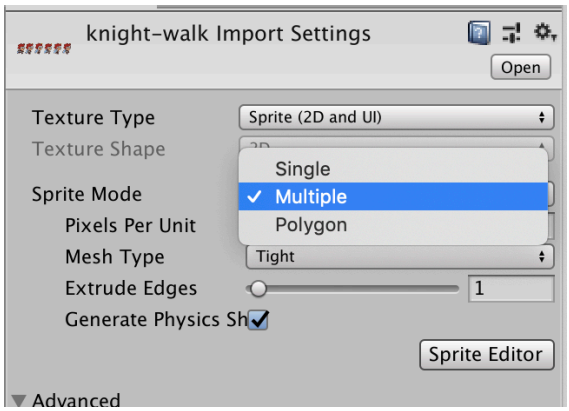


Finally, the tile map needs to have a collision component so that the player can stand on it. Rather than the *Box Collider 2D* component, add a *Tilemap Collider 2D* component to the tile map in the hierarchy. The slight green outline around the tiles when the object is selected indicates the effective collision area.

Moving the Player

The next step is to implement a controller to handle the moving and jumping of the player. It will have an animated sprite, but the details of this will be handled later.

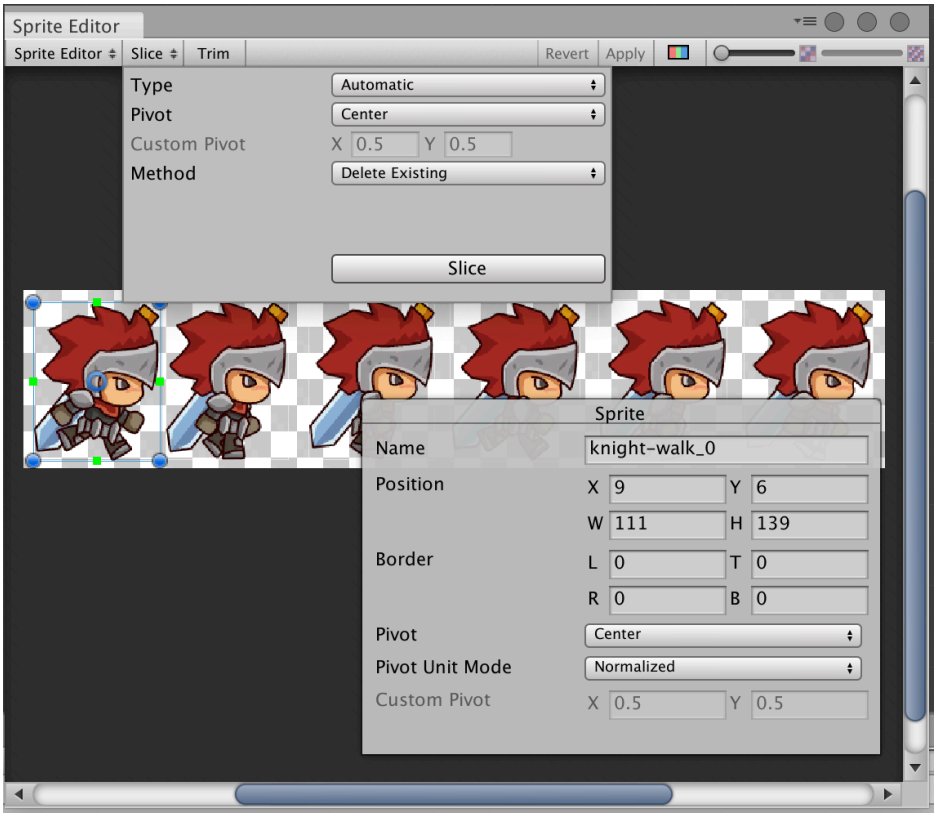
The sprites for the character are actually sprite sheets – or atlases – in that they contain multiple sprites for each frame of animation, rather than having a separate file for each frame. You will find sprite assets for 2D games are generally packaged in this way. However, Unity is currently treating the sprite sheet as a single sprite, and must be told how to extract the frames from the sheet.



Select the sprite asset for the running character, and set the *Sprite Mode* to *multiple* in the inspector – click *Apply* in the inspector.

Next click on the *Sprite Editor* button in the inspector.

Note – if you are using a newer version of Unity, i.e. 2019, this may prompt an alert that there is no Sprite Editor installed. In this case you will need to install the 2D Sprite package via the Package Manager.



This Sprite Editor interface, via the *Slice* dropdown, allows you to specify how the sheet should be divided into individual frames. For many sheets the *Automatic* setting will correctly divide the sheet, however depending on the sheet and how it was created it may be necessary to specify the sheet layout as a grid by size or number of sprites. Either way, click *Slice*, followed by *Apply,* and you will see that 6 individual sprite assets have appeared in the asset tab under the original sprite sheet.

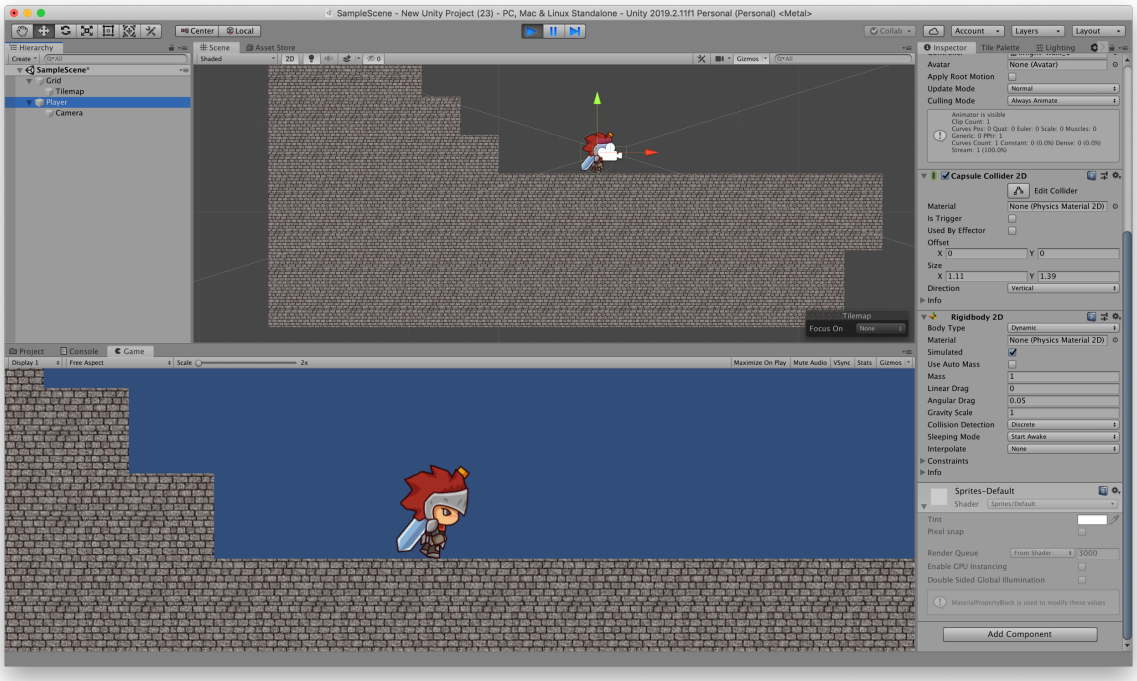
Select these 6 new sprites, and drag the entire collection onto the hierarchy. This will prompt Unity to create a new *Animation* asset – name it something sensible – and also a new game object in the scene that will play out this animation. This game object will form the basis for the player's character.

The next steps are similar to the previous exercise – add physics components to the object to allow it to move, and add a script to handle user input and to manipulate the character.

Perform the following steps:

- Name the new object something sensible – e.g. Player
- Drag the *Camera* onto the Player in the hierarchy, and set its position relative to the Player so that the Player is in the centre of the screen
- Add a *Capsule Collider 2D* component that broadly matches the shape of the sprite
- Add a *Rigidbody 2D* component
- Add a new *Script* component called PlayerController

Check that the various components are working by raising the Player object above the platform and playing the game – it should fall under gravity (via the Rigidbody) and stop when it hits the platform (via the collider and the collider on the tile map).



Consider the PlayerController code below. This time, code affecting the movement of the object will be put in the **FixedUpdate** function – we will talk about the architecture behind this in detail in a future lecture, but for now the FixedUpdate function runs at a constant rate to enable the physics simulation to operate most efficiently, whereas the Update function may run at a variable rate, and is more appropriately used for visuals.

We will consider the physics (the movement of the player) first. This function works by checking the player's input in turn in order to calculate a Vector2 velocity, or an overall direction and magnitude to push them in the vertical and horizontal.

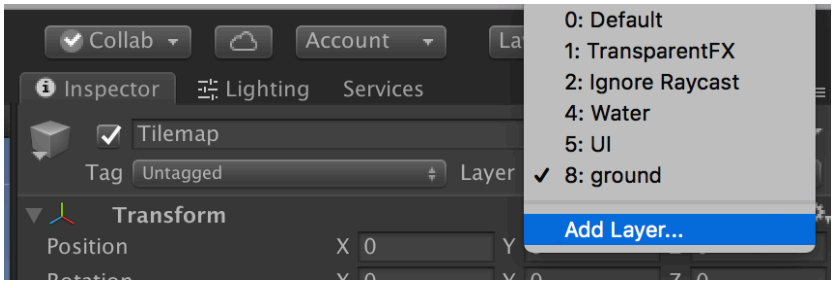
The basic principles are simple – pushing left or right applies a velocity to the rigid body, moving it left or right. Jump applies an upwards velocity. The final addition is only let the player jump once, so they cannot jump while in the air, and to only let them jump again once they have safely landed. This is implemented by setting a variable when the player first jumps, and then testing to see what is directly underneath the player to see whether this variable should be changed, i.e. when they returned to the ground.

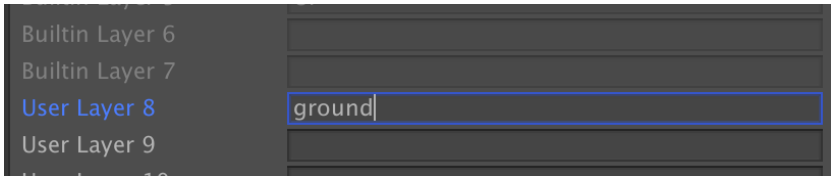
```
bool canJump = true;
int groundMask = 1<<8;

// Update is called once per frame
void FixedUpdate()
{
    // the new velocity to apply to the character
    Vector2 physicsVelocity = Vector2.zero;
    Rigidbody2D r = GetComponent<Rigidbody2D>();
```

```
        // move to the left
        if (Input.GetKey(KeyCode.A))
        {
            physicsVelocity.x -= 2;
        }

        // implement moving to the right for the D key

        // this allows the player to jump, but only if canJump is true
        if (Input.GetKey(KeyCode.W))
        {
            if (canJump)
            {
                r.velocity = new Vector2(physicsVelocity.x, 4);
                canJump = false;
            }
        }

        // Test the ground immediately below the Player
        // and if it tagged as a Ground layer, then we allow the
        // Player to jump again.
        if (Physics2D.Raycast(new Vector2
                        (transform.position.x,
                          transform.position.y),
                     -Vector2.up, 1.0f, groundMask))
        {
            canJump = true;
        }

        // apply the updated velocity to the rigid body
        r.velocity = new Vector2(physicsVelocity.x,
                                             r.velocity.y);
    }
```

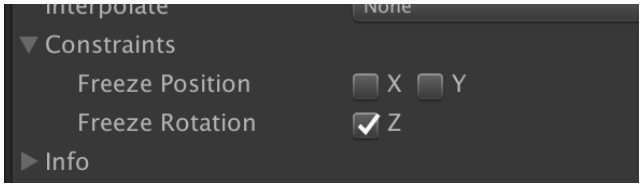
There are two more steps for this to work appropriately.

Firstly, the player will only be able to jump once but then never again, until the ground is correctly tagged as being in a certain *Layer*. Layers allow us to implement different behaviours for different surfaces. Here, the *groundMask* is looking for layer number 8, so we need to label the tile map as such by adding a new layer with this number.





You might also find that the player character rolls around – think about what is happening here. The capsule collider has a round base, which is useful as it allows the collider to be pushed up stairs, and prevents it from snagging on corners, but does mean it is liable to fall over when a force is applied to it. This can be fixed by *constraining*, or freezing the rotation of the collider about the Z axis.



Change and test the various velocity parameters to achieve the style of movement that you think is best.

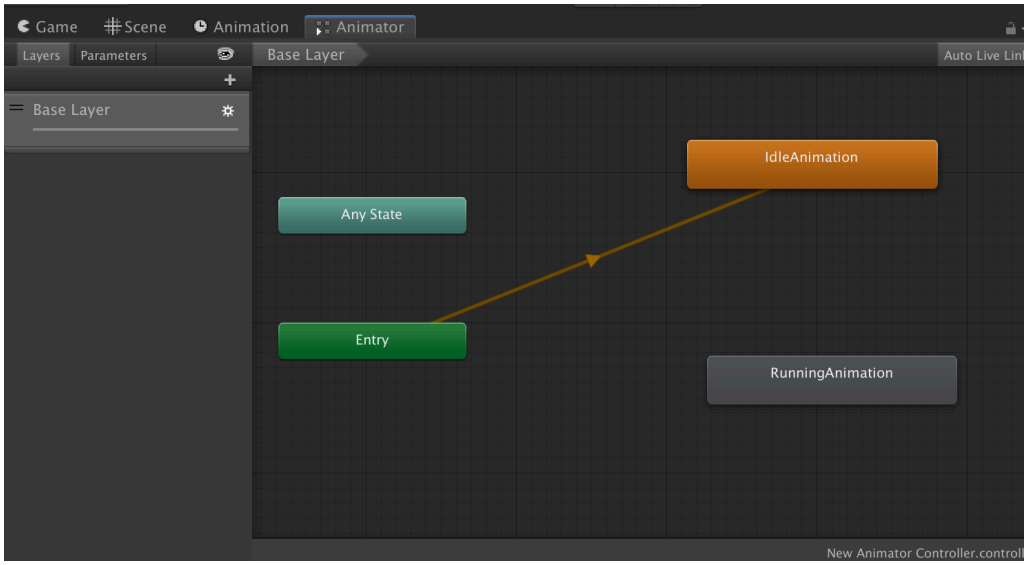
Animation

In the previous exercise the ship and asteroids were given interesting visual effects by using models, and by rotating the rigid body that in turn manipulated the model. In this exercise, we will further *decouple* the visual effect of the player from the movement of the underlying object by using animation. Animations purely affect the visual representation of the object, and do not manipulate the underlying physics representation.

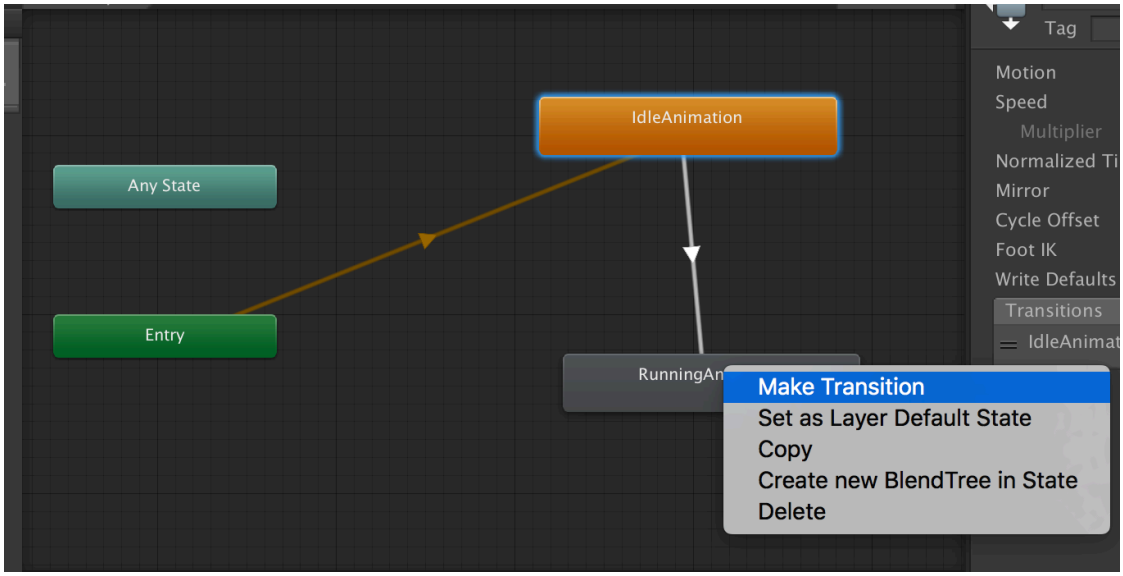
This is an important concept – it allows the physics simulation to be an approximation, which makes it feasible to simulate many objects at a reasonable framerate – while affording complex graphical effects. The capsule component approximates the shape of the running character, while the animation gives the impression of a more detailed activity.

As with the running animation, slice the idle animation sprite sheet into its 6 constituent sprites. Drag these 6 sprites together into the hierarchy to create a new animation asset, as before. The object created in the scene can be deleted.

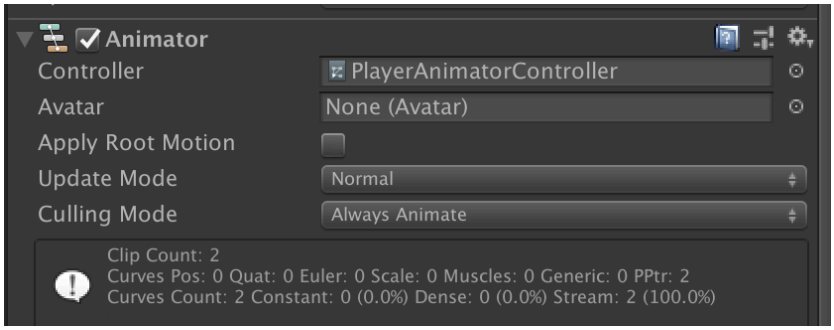
Based on whether the user is pressing the left or right key we are going to change the state of an *Animator Controller* object, which in turn will play out the appropriate animated sprite for the character.



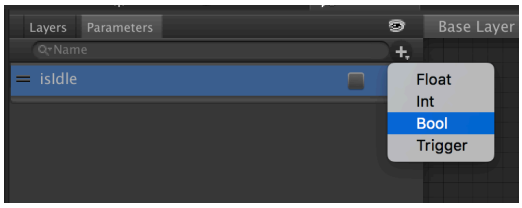
Create an *Animator Controller* asset. Double clicking this will open the Animator tab, and onto this drag the two *Animation* assets you have created – the running animation and the idle animation. These are the two states that the Animator Controller can be in.

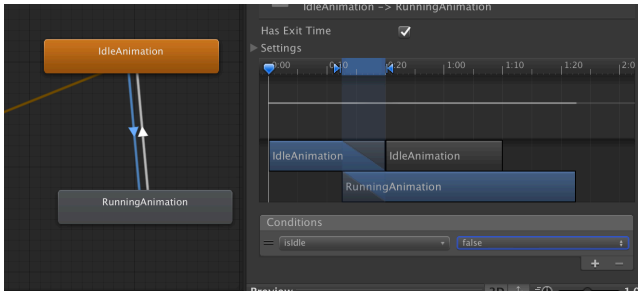


Right click on one of the animations and *make transition* to connect it to the other. Repeat this step to create a transition back to the first animation. Returning to the Player object, set the controller in the Animator component to be the controller asset you've just made.



If you now play the game you should see the character transition from idle to running and back again, as the animator transitions between the two animation states. This is because there are no conditions to say when the animator should transition, and as such it is continually doing so.





The last step here in the animator is to create a parameter that can be controlled programmatically, and using this specify conditions as to when to transition. Create a *bool* parameter called *isIdle*, and create a condition for each of the transitions so that the animator should change state to running when *isIdle* is false, and vice versa. These conditions can be added in the Inspector as seen on the right.

Finally return to the PlayerController script, where the task is to implement the Update function to control the appearance and animation of the character in response to what is going on in the FixedUpdate function.

```
bool isIdle;
bool isLeft;
int isIdleKey = Animator.StringToHash("isIdle");

private void Update()
{
    Animator a = GetComponent<Animator>();
    a.SetBool(isIdleKey, isIdle);

    SpriteRenderer r = GetComponent<SpriteRenderer>();
    r.flipX = isLeft;
}
```

Modify the FixedUpdate function to set the variables isIdle and isLeft accordingly. isIdle should initially be set to false, but then true if either A or D is pressed.

Hazards

There are a few ways in which you could introduce a hazard for the player to avoid – in particular by thinking about how collision events between the character and other objects in the scene might trigger the scene to restart. The most obvious would be to create a game object with no visible sprite, but a large Box Collider 2D that would kill the player on contact rather than allowing them to fall forever if they miss a jump.

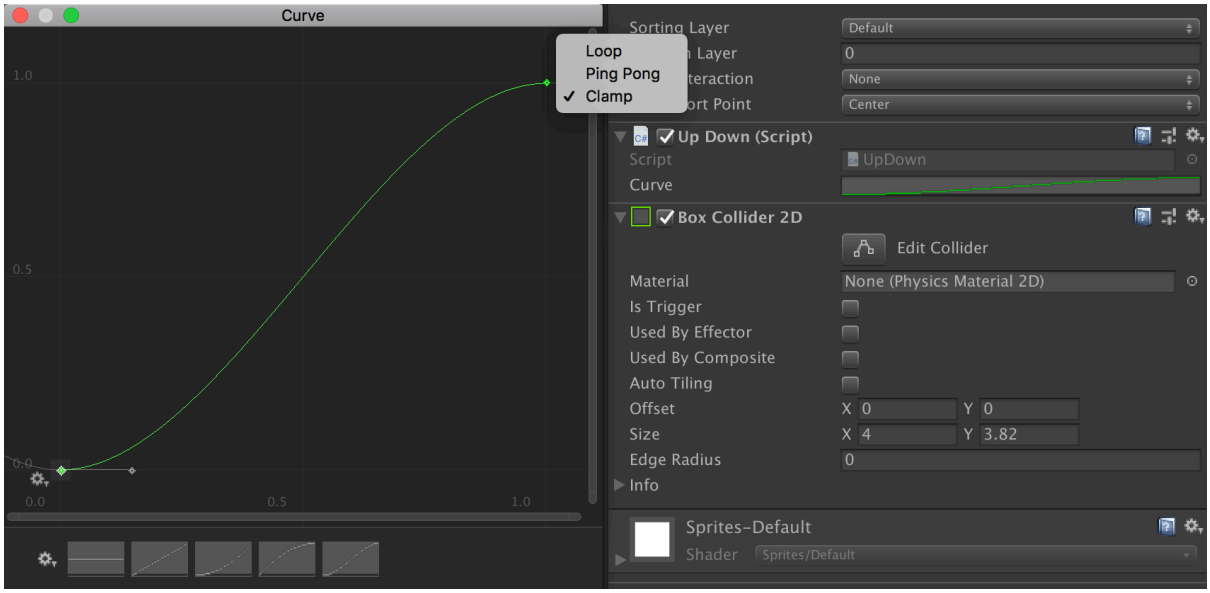
Extending this, an appropriate hazard sprite (in this case the asteroid) can be simply animated to move along a prescribed path, and also kill the player if they touch it.

```
public AnimationCurve curve;
```

```
// Update is called once per frame
void Update()
{
    transform.position = new Vector3(transform.position.x,
        curve.Evaluate((Time.time % curve.length)),
        transform.position.z);
}

private void OnCollisionEnter2D(Collision2D collision)
{
    Application.LoadLevel(Application.loadedLevel);
}
```

Here the curve variable is an *AnimationCurve* exposed to the inspector, and the curve editor can be used to specify a simple sinusoidal movement. There is no need for a Rigidbody 2D component here as the position of the object is being directly set, rather than moved by forces, but a Box Collider 2D is required to generate collision events between player and the hazard. Note also that we must use OnCollisionEnter2D rather than its 3D counterpart.

https://docs.unity3d.com/ScriptReference/AnimationCurve.html



The magnitude and frequency of the movement of the hazard are set in the curve editor, or in the script if you prefer. Setting the *Ping Pong* option at each end of the curve via the gear icon will turn it into a repeating sine wave.

Exercises

Finally, put all of the components together. Construct a simple linear level that requires the player to navigate over a number of platforms while jumping over or avoiding moving hazards. Try to think about the design of the level and implement a range of different platform challenges:

Safe Jumps

What is an ideal arrangement of platforms that the player can make reasonably easily, for example where they do not have to start exactly on the platform edge.

Difficult horizontal jumps
The opposite – what is an arrangement of platforms that requires perfect coordination of running and jumping? What happens if you include a hazard?

Leaps of faith
Construct a jump from which there is no ground reference, meaning that players cannot see the platform to aim for until they've committed to the jump.

Climbing
Construct a ladder of platforms that the player must climb, but where a single mistake will make them fall to the bottom again.

Pattern recognition
Construct a sequence of moving hazards moving at different speeds (i.e. different curves, or parameter multipliers), so that the player must understand and learn the sequence and wait for the right time to move over/under the hazards.

The screenshot at the start of this document has the same game mechanic as implemented above, but I have used a different sprite sheet texture for the character, and a different tile set. This gives a very different visual style to the game, including reusing the character from the game "Braid" with an additional animation state for "falling", and an animated hazard.

http://opengameart.org/content/magic-cliffs-environment

http://www.davidhellman.net/braidbrief.htm