Lab - Animation

Goals

- To get familiar with the sprite editor in Unity.
- To learn how to do sprite animation.
- To learn how to create a parallax effect.
- To learn how to do multi-part animation.

Resources

- Animation project (https://altitude.otago.ac.nz/cosc360/Animation) on GitLab.
- Unity tutorials on animation (http://unity3d.com/learn/tutorials/topics/animation)

- Unity Manual on Animation (https://docs.unity3d.com/Manual/AnimationSection.html)
- Tutorials on animation: Unity 2D animation (http://www.raywenderlich.com/61532/unity-2d-tutorial-getting-started), Bone-based 2D animation (http://gamedevelopment.tutsplus.com/series/bone-based-unity-2d-animation--cms-617), 2D platformer animation (https://www.youtube.com/watch?v=FRMy5B3dD_I), and probably many more.

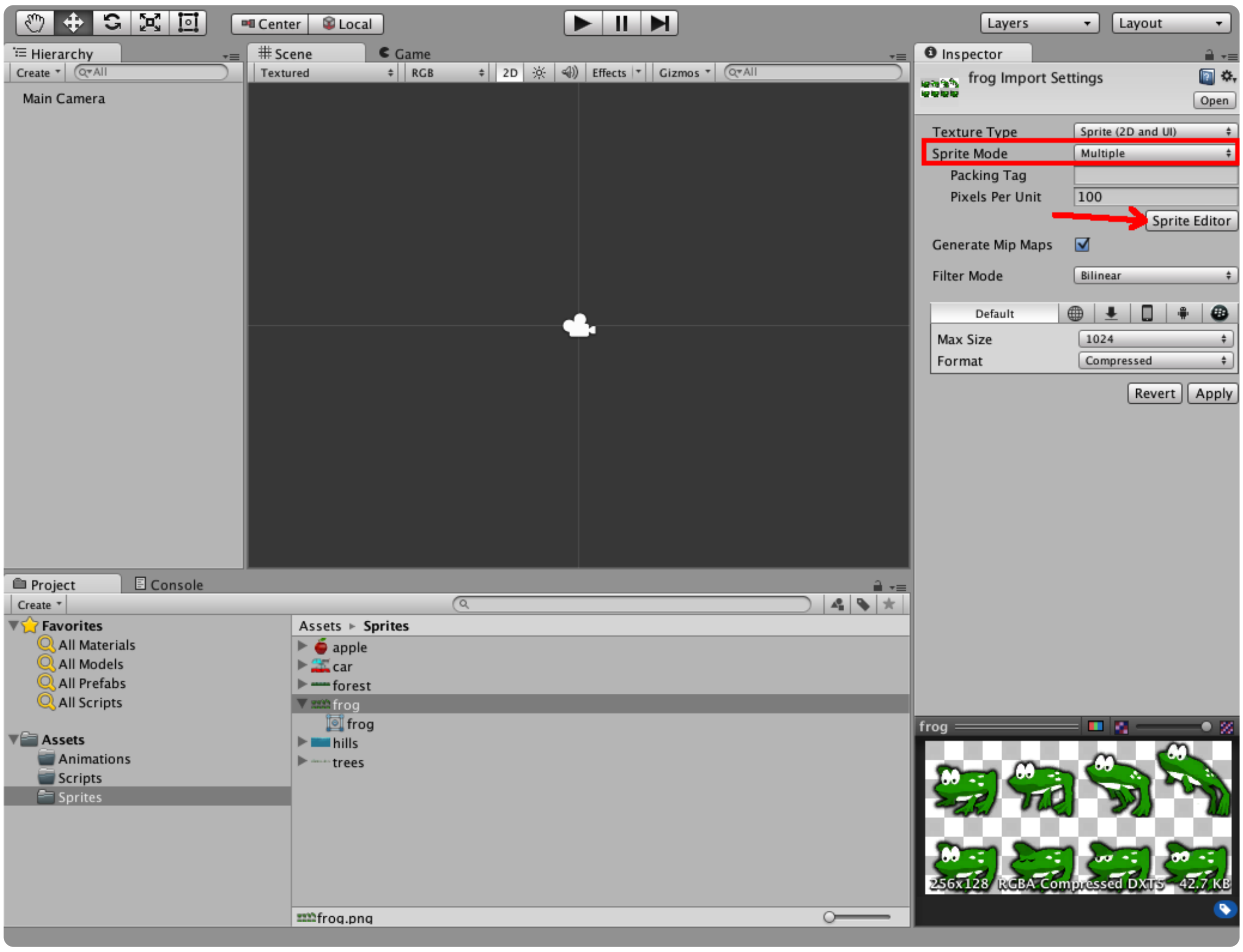
Basic Challenge

Animation is a big part of the game experience. This lab will cover some basic principles and animation the tools available in Unity.

Sprite sheets and animation

Unity provides a number of different tools for creating Animation. The first method this lab is going to cover is a very basic animation, based on rendering cycle of pre-drawn frames in a sprite object.

- Fork the Animation project (https://altitude.otago.ac.nz/cosc360/Animation) on GitLab, then clone your fork to your hard-drive.
- Open the *Animation* project, the directory of the cloned repository, in Unity.
- Save the scene as "Level1".
- The assets for this lab have already been imported into the project. Select *frog.png* from *Assets/Sprites*. In the **Inspector Panel** you should see *Import Settings* for the image and *Sprite Mode* currently being set to *Single*. In the preview window you should see that the image is actually eight sprites showing a frog in different stages of jumping. The image is a sprite sheet - it contains a number of sprites packed into one image. You'll need to tell Unity how to fish out individual sprites from the sheet.
- Set the *Sprite Mode* to *Multiple* and click the *Sprite Editor* button. If you get a message about "Unapplied import settings" just click the *Apply* button.

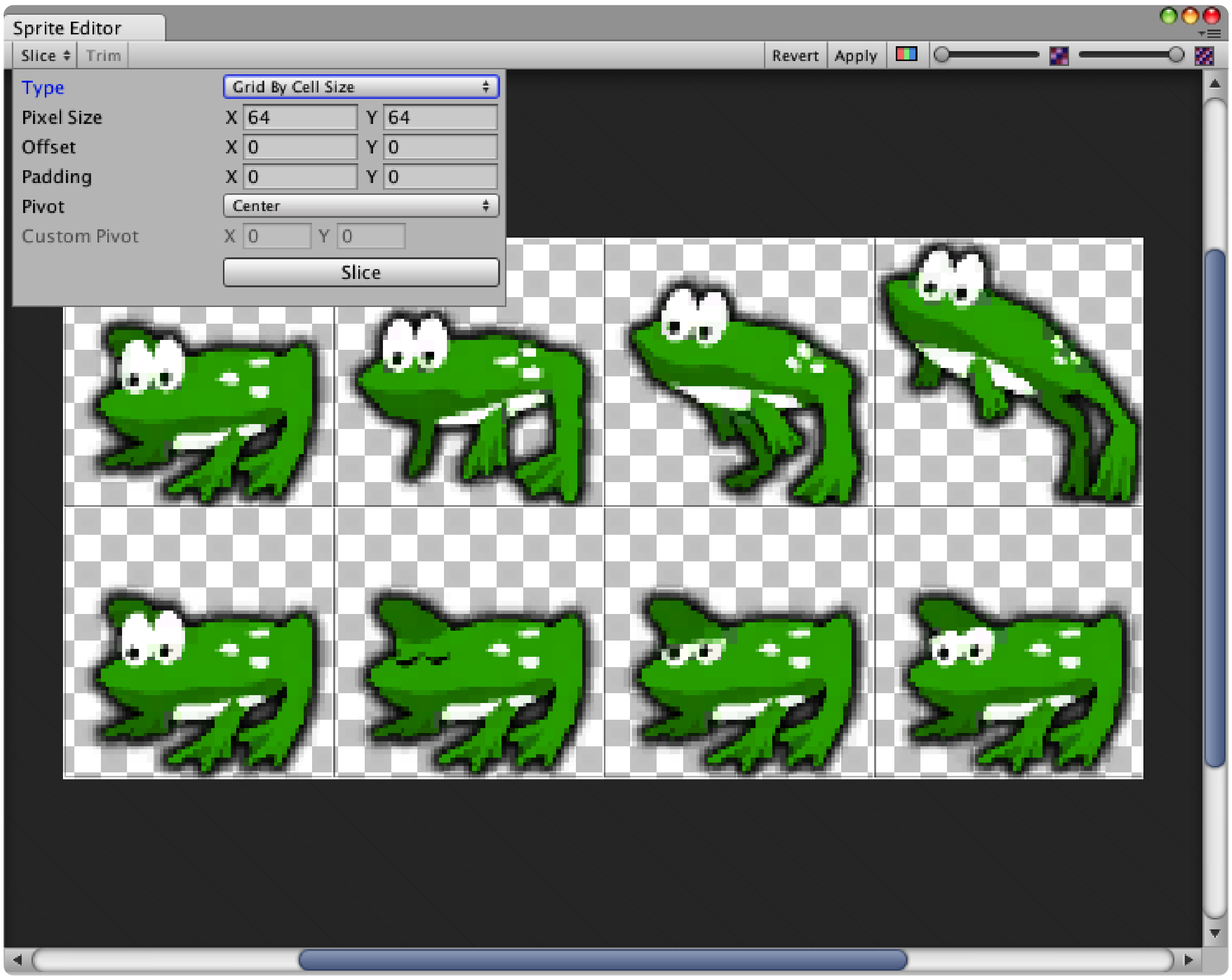


frog Import Settings

Texture Type: Sprite (2D and UI)
Sprite Mode: Multiple
Packing Tag:
Pixels Per Unit: 100
Generate Mip Maps: ✓
Filter Mode: Bilinear
Max Size: 1024
Format: Compressed

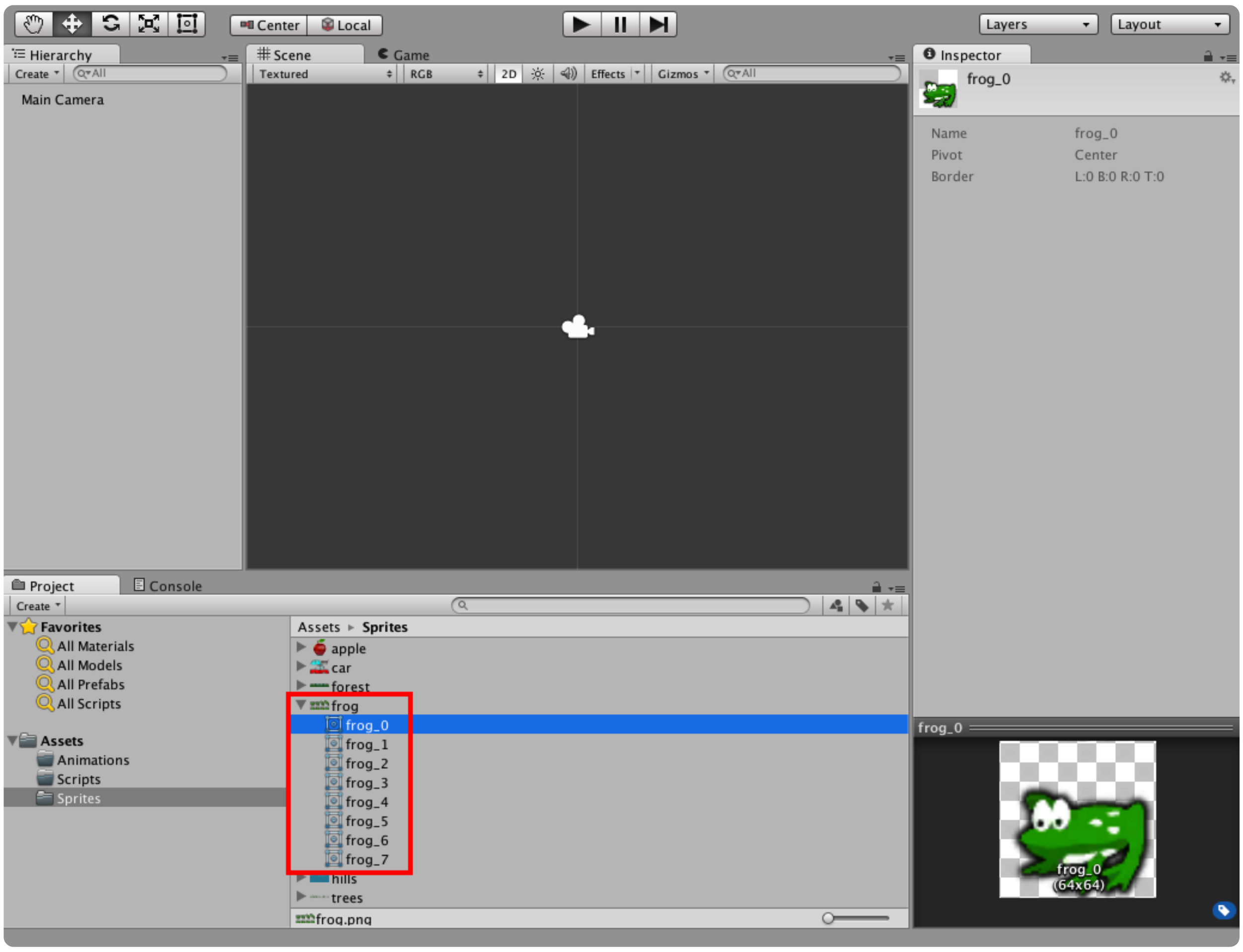
256x128 RGBA Compressed DXT5 42.7 KB

- The *Sprite Editor* window should appear. Expand the *Slice* option and set the *Type* to *Grid By Cell Size*. The *Pixel Size* should by default be set to X=64 and Y=64. Press the *Slice* button. You should see the resulting dividing lines on the sprite sheet.



Type: Grid By Cell Size
Pixel Size: X 64 Y 64
Offset: X 0 Y 0
Padding: X 0 Y 0
Pivot: Center
Custom Pivot: X 0 Y 0

. Press *Apply* and close the *Sprite Editor* window. Now, in *Assets/Sprites* there should be 8 sprites underneath the *frog* image, numbered *frog_0* to *frog_7*. Each sprite is a different frame of the frog animation.



- In the **Hierarchy panel** create a new *2D | Sprite* object. Name it "Frog1". Set its *Sprite Renderer | Sprite* to "frog_0" and its *Transform | Scale* to (2,2,1).

- Create a new C# Script. Name it "Frog1Control" and add the following code:

Frog1Control.cs

```
using UnityEngine;
using System.Collections;

public class Frog1Control : MonoBehaviour {

   // An array with the sprites used for animation
   public Sprite[] animSprites;

   // Controls how fast to change the sprites when
   // animation is running
   public float framesPerSecond;

   // Reference to the renderer of the sprite
   // game object
   SpriteRenderer animRenderer;

   // Time passed since the start of animatin
   private float timeAtAnimStart;

   // Indicates whether animation is running or not
   private bool animRunning = false;

   // Use this for initialization
   void Start () {
      // Get a reference to game object renderer and
      // cast it to a Sprite Rendere
      animRenderer = GetComponent<Renderer>() as SpriteRenderer;
   }


   // At fixed time intervals...
   void FixedUpdate () {
      if(!animRunning) {
         // The animation is triggered by user input
         float userInput = Input.GetAxis("Horizontal");
         if(userInput != 0f) {
            // User pressed the move left or right button

            // Animation will start playing
            animRunning = true;

            // Record time at animation start
            timeAtAnimStart = Time.timeSinceLevelLoad;
         }
      }
   }

   // Before rendering next frame...
   void Update () {

      if(animRunning) {
         // Animation is running, so we need to
      // figure out what frame to use at this point
      // in time

      // Compute number of seconds since animation started playing
      float timeSinceAnimStart = Time.timeSinceLevelLoad - timeAtAnimStart;

      // Compute the index of the next frame
      int frameIndex = (int) (timeSinceAnimStart * framesPerSecond);

      if(frameIndex < animSprites.Length) {
        // Let the renderer know which sprite to
        // use next
        animRenderer.sprite = animSprites[ frameIndex ];
      } else {
        // Animation finished, set the render
        // with the first sprite and stop the
        // animation
        animRenderer.sprite = animSprites[0];
        animRunning = false;
      }
    }
  }
}
```

The `animSprites` variable is an array that holds the set of sprites used for animation. The `framesPerSecond` variable controls how quickly the frames are changed in the animation. The `animRenderer` variable is a reference to the current object's renderer. Its initialised in the *Start()* method by casting `renderer`, a built-in reference to a *Renderer (http://docs.unity3d.com/ScriptReference/Renderer.html)* object, to its subclass, the *SpriteRenderer*. It's always good idea to do one-time computations and initialisations in the *Start()* method, so that its done on scene load, rather than per every frame in the *Update()*.

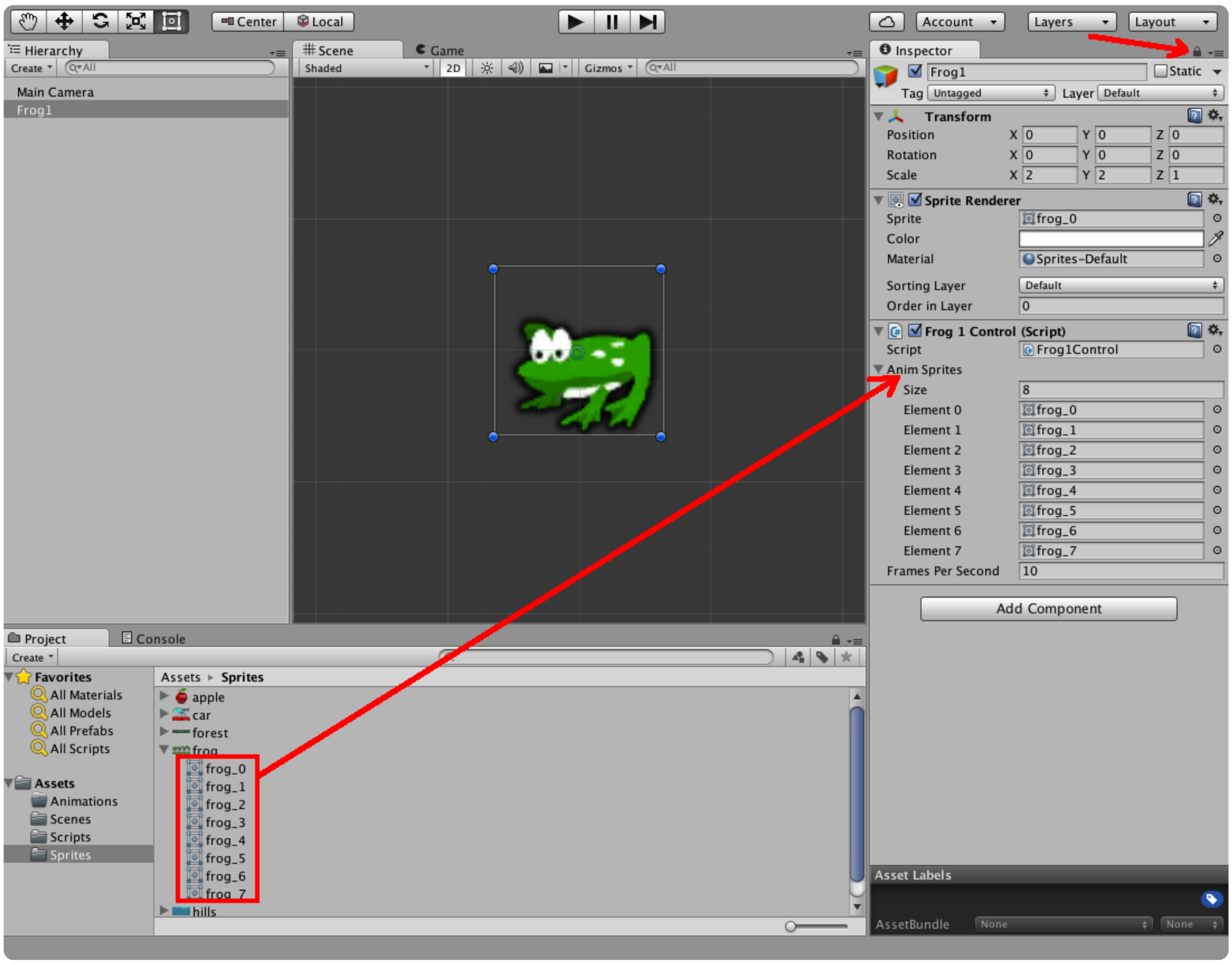
The *FixedUpdate()* function is called at a fixed time interval (recall that *Update()* occurs per every frame). The interval for *FixedUpdate()* is set in *Edit | Project Settings | Time* - right now it should be 0.02 seconds. If a player presses the left or right movement key, when animation is not running, the *animRunning* variable will be set to true, and current time will be recorded as the time at which animation has started.

In Update(), if animation is running, the time since the animation has started is computed. The index of the next animation frame to display is computed based on the time since the animation has started, and the frames per second setting. If the computed index is within the range of *animSprites* array, it is used to fetch the next sprite for rendering. Otherwise, the renderer is set to display the first sprite from the array, and the animation is stopped.

- Add the "Frog1Control" script as a component of the *Frog1* game object.

- To set the *Anim Sprites* variable, click on the little lock icon in the **Inspector panel**. This assures that *Frog1*'s properties remain displayed in the panel, even when other game objects/assets are selected in the **Hierarchy/Project panels**. From *Assets | Sprites* select all the frog sprites: "frog_0" to "frog_7" all at once (while pressing SHIFT, select "frog_0" then "frog_7" - everything in between should get selected as well).

Drag them over the *Anim Sprites* variable in the **Inspector panel**. The array should have now 8 entries. Set *Frames Per Second* to 10. Don't forget to unlock the **Inspector panel**.



Frog1 · Static
Tag Untagged · Layer Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	2	2	1

Sprite Renderer

Sprite	frog_0
Color	
Material	Sprites-Default
Sorting Layer	Default
Order in Layer	0

Frog 1 Control (Script)

Script	Frog1Control
Anim Sprites	
Size	8
Element 0	frog_0
Element 1	frog_1
Element 2	frog_2
Element 3	frog_3
Element 4	frog_4
Element 5	frog_5
Element 6	frog_6
Element 7	frog_7
Frames Per Second	10

2. Play the game. The frog should jump in place when you press the left or right arrow key. At the moment it keeps facing to the left, even when pressing the right arrow.

3. Next, you'll be adding the movement. Add the following code to the *Fog1Control* script:

Frog1Control.cs

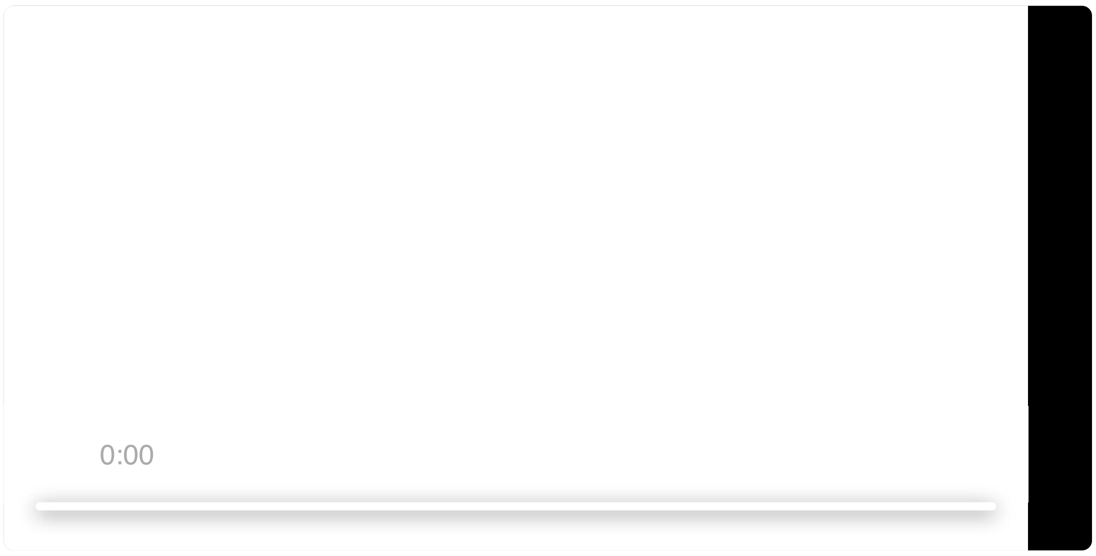
```
using UnityEngine;
using System.Collections;

public class Frog1Control : MonoBehaviour {

   // An array with the sprites used for animation
   public Sprite[] animSprites;

   // Controls how fast to change the sprites when
   // animation is running
   public float framesPerSecond;

   // Reference to the renderer of the sprite
   // game object
   SpriteRenderer animRenderer;

   // Time passed since the start of animatin
   private float timeAtAnimStart;

   // Indicates whether animation is running or not
   private bool animRunning = false;

   // Speed of the movement
   public float speed = 4f;

   // Direction of the movement
   private   float movementDir;

   // Use this for initialization
   void Start () {
      // Get a reference to game object renderer and
      // cast it to a Sprite Rendere
      animRenderer = GetComponent<Renderer>() as SpriteRenderer;
   }

   // At fixed time intervals...
   void FixedUpdate () {
      if(!animRunning) {
         // The animation is triggered by user input
         float userInput = Input.GetAxis("Horizontal");
         if(userInput != 0f) {
            // User pressed the move left or right button

            // Animation will start playing
            animRunning = true;

            // Record time at animation start
            timeAtAnimStart = Time.timeSinceLevelLoad;

            // Get the direction of the movement from the sign
            // of the axis input (-ve is left, +ve is right)
```

```
        movementDir = Mathf.Sign(userInput);
      }
    }
  }

  // Before rendering next frame...
  void Update () {

    if(animRunning) {
      // Animation is running, so we need to
      // figure out what frame to use at this point
      // in time

      // Compute number of seconds since animation started playing
      float timeSinceAnimStart = Time.timeSinceLevelLoad - timeAtAnimStart;

      // Compute the index of the next frame
      int frameIndex = (int) (timeSinceAnimStart * framesPerSecond);

      // The frog in the sprites faces left; when its Scale.X value is positive
      // keep facing left, when its Scale.X value is negative it will face righ
      // the signs for Scale.X correspond to opposite directions of the movemen
      // So, when movementDir is -ve, make sure Scale.X is +ve, and when moveme
      // +ve, makse sure Scale.X is -ve.
      Vector3 localScale = transform.localScale;
      localScale.x = -Mathf.Abs(transform.localScale.x)*movementDir;
      transform.localScale = localScale;

      if(frameIndex < animSprites.Length) {
        // Let the renderer know which sprite to
        // use next
        animRenderer.sprite = animSprites[ frameIndex ];

        // Move the game object only when showing the 3rd and 4th frames
        // (when the frog is in the air)
        if(frameIndex >= 2 && frameIndex <= 3) {
          // Move the game object
          Vector3 shift = Vector3.right * movementDir * speed * Time.deltaTim
          transform.Translate(shift);
        }
      } else {
        // Animation finished, set the render
        // with the first sprite and stop the
        // animation
        animRenderer.sprite = animSprites[0];
        animRunning = false;
      }
    }
  }
}
```

The `speed` variable controls the speed of the movement. The `movementDir` records the direction of the movement (-1 when going left, +1 when going right). The sprites are drawn with the frog facing to the left. To get the frog to face to the right, a horizontal flip of the sprite is required. This can be done by setting the X value of sprite's *Scale* transform to a negative number. The sign of the frog scale ends up being the opposite of the sign of `movementDir`.

The sprites of the frog animation show the frog jumping up, flying through the air for a bit, then landing. To get the frog moving only when its in the air, the movement is applied only during the 3rd and 4th frames (the 2nd and 3rd index in the sprite array) in the *Update()* method.

4. When you play the game, the frog should happily jump left and right now. Note the sign flip in *Transform | Scale* when direction of the movement changes, and the change of *Sprite Renderer | Sprite* when animation is playing.

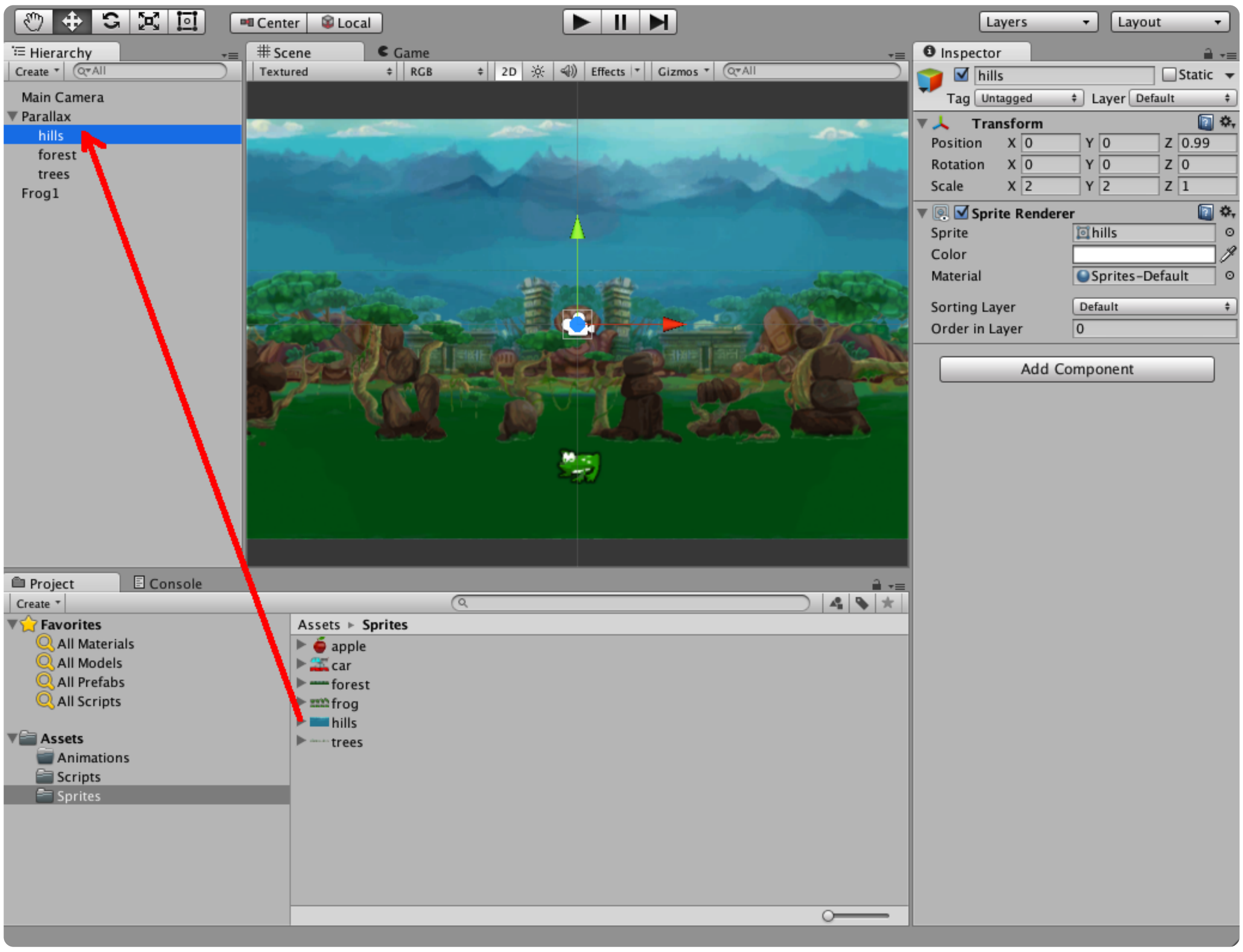


Parallax

Next, you will add a few background layers, make the camera follow the frog, and have different layers move at different speeds to create an illusion of depth. It might seem strange to *fake* depth when in fact Unity has a full 3D engine underneat. However parallax is far easier to setup than the fully fledged 3D environment, and it is quite sufficient for many types of 2D games.

5. First, *Create Empty* game object in the **Hierarchy panel** and name it *Parallax*. Make sure it's *Transform | Position* is set to (0,0,0) and *Transform | Scale* to (1, 1, 1).

6. Next, add three sprite objects to the scene for the background layers based on "hills.png", "forest.png", and "trees.png". There's a fast way of doing it. Drag the images, one by one, from *Assets | Sprites* over to the **Hierarchy panel** - the sprite object with the same name as the image will get created, with its *Sprite Renderer | Sprite* property already set. Place the *hills*, *forest* and *trees* sprite objects under the *Parallax* game object in the hierarchy.



hills   Static
Tag Untagged   Layer Default

Transform
Position X 0 Y 0 Z 0.99
Rotation X 0 Y 0 Z 0
Scale X 2 Y 2 Z 1

Sprite Renderer
Sprite hills
Color
Material Sprites-Default
Sorting Layer Default
Order in Layer 0

2. For the *hills* sprite object, set its *Transform | Position* to (0,0,0.99) and *Transform | Scale* to (2,2,1).

3. For the *forest* sprite object, set its *Transform | Position* to (0,-2,0.7) and *Transform | Scale* to (2,2,1).

4. For the *trees* sprite object, set its *Transform | Position* to (0,-0.5,0.2) and *Transform | Scale* to (3,3,1).

5. Parallax movement will be applied to the horizontal axis. The camera will track the frog, so that the frog remains centred on the horizontal axis while moving. Set the *Frog1*'s new position by setting its *Transform | Position* to (0,-3.35,-1).

6. Create a new C# Script. Name it "Parallax" and paste the following code:

Parallax.cs

```
using UnityEngine;
using System.Collections;

public class Parallax : MonoBehaviour {

   public void Move (Vector3 cameraShift) {

      // Move the camera with the shifting
      // object
      Camera.main.transform.Translate(cameraShift);

      // For every child of the parallax game object
      // determine the degree of lag behind the camera
      // and adjust its shift
      foreach (Transform child in transform) {
         // Take the Z value to ind
         // the camera's shift
         // It is roughly rleated to distance.
         float keepUp = child.position.z;
         if(keepUp < 0) {
            // Z value of 0 corresponds to 0 distance from the camera
            keepUp = 0;
         } else if(keepUp > 1) {
            // Z value of 1 means infinite distance from the camera
            keepUp = 1;
         }
         // Object shift in world coordinates is a degree of
         // camera's shift
         Vector3 backgroundShift = cameraShift * keepUp;
         child.Translate(backgroundShift);
      }
   }
}
```

The script implements a *Move()* function, where the *Main Camera* and children of the script's game object track movement passed in as the `cameraShift` argument. The *Main Camera* describes the location of a virtual camera looking at the 'world' in the scene. It can be moved like any other object in the scene, and in this case it undergoes a translation given by the `cameraShift` vector. This is so that the camera stays centred on the frog, even though the frog has moved.

This script is meant to be attached as a component to the *Parallax* game object. The loop in the *Move()* function iterates over all of its children, applying a degree of shift based on each child's Z position. The Z value of 0 (or less) is taken to mean zero distance from the camera. The Z value of 1 (or more) is taken to mean an infinite distance from the camera. Anything between 0 and 1 is taken as a proportion of some non-zero distance. The 0 to 1 range was chosen, because it relates the fraction of shift that the *child* needs to undergo in order to produce the parallax effect.

First, let's think of parallax in terms of movement with respect to the camera. Things that are close to the camera undergo a large shift, and things that are far in the distance remain almost stationary. It's just like looking sideways when driving - trees by the road go by fast, mountains in the distance seem not to move at all.

This is exactly what is happening in the script above, except the camera itself is moving and all the shifts are expressed in the world coordinates. When the camera moves by `cameraShift`, things that are to remain stationary with respect to the camera (infinity distance) need to undergo the same movement in the world coordinates. Thing that are to move with respect to the camera (closer distance), need to shift with some lag behind the camera in the world coordinates.

In the scene, the *hills* game object is positioned at Z=0.99 (almost infinity) and so it will undergo 99% of camera's shift in world coordinates. The *forest* game object is at Z=0.7, and so it will undergo only 70% of the camera's shift in the world coordinates. The *trees* game object is positioned at Z=0.2, and so it will undergo 20% of the camera's shift in the world coordinates.

2. Add "Parallax" script as a component of the *Parallax* game object. From now on, if you want a game object to undergo a parallax movement, just make it a child of *Parallax* and set it's Z position to a value between 0 and 1.

3. You need to let the "Parallax" script know how much *Frog1* has moved at any given moment, so add the following code to the "Frog1Control" script:

Frog1Control.cs

```
using UnityEngine;
using System.Collections;

public class Frog1Control : MonoBehaviour {

   // An array with the sprites used for animation
   public Sprite[] animSprites;

   // Controls how fast to change the sprites when
   // animation is running
   public float framesPerSecond;

   // Reference to the renderer of the sprite
   // game object
   SpriteRenderer animRenderer;

   // Time passed since the start of animatin
   private float timeAtAnimStart;

   // Indicates whether animation is running or not
   private bool animRunning = false;

   // Speed of the movement
   public float speed = 4f;

   // Direction of the movement
   private   float movementDir;

   // Reference to parallax game object
   public GameObject parallaxObj;

   // Refernce to the script component of parallax
   // game object
   private Parallax parallaxComp;

   // Use this for initialization
   void Start () {
      // Get a reference to game object renderer and
      // cast it to a Sprite Rendere
      animRenderer = GetComponent<Renderer>() as SpriteRenderer;

      // Get the reference to the script component of
      // parallax game object
      parallaxComp = parallaxObj.GetComponent<Parallax>();
   }

   // At fixed time intervals...
   void FixedUpdate () {
      if(!animRunning) {
         // The animation is triggered by user input
         float userInput = Input.GetAxis("Horizontal");
```

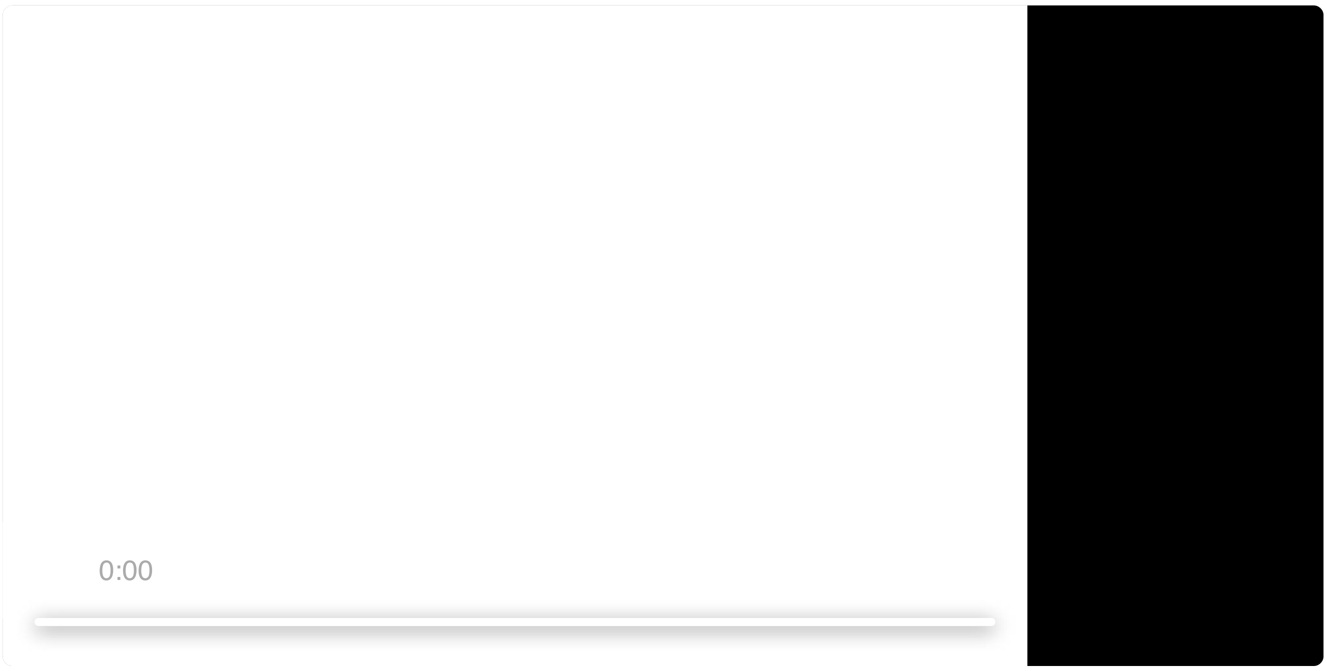
```
            if(userInput != 0f) {
                // User pressed the move left or right button

                // Animation will start playing
                animRunning = true;

                // Record time at animation start
                timeAtAnimStart = Time.timeSinceLevelLoad;

                // Get the direction of the movement from the sign
                // of the axis input (-ve is left, +ve is right)
                movementDir = Mathf.Sign(userInput);
            }
        }
    }

    // Before rendering next frame...
    void Update () {

        if(animRunning) {
            // Animation is running, so we need to
            // figure out what frame to use at this point
            // in time

            // Compute number of seconds since animation started playing
            float timeSinceAnimStart = Time.timeSinceLevelLoad - timeAtAnimStart;

            // Compute the index of the next frame
            int frameIndex = (int) (timeSinceAnimStart * framesPerSecond);

            // The frog in the sprites faces left; when its Scale.X value is positive
            // keep facing left, when its Scale.X value is negative it will face righ
            // the signs for Scale.X correspond to opposite directions of the movemen
            // So, when movementDir is -ve, make sure Scale.X is +ve, and when moveme
            // +ve, makse sure Scale.X is -ve.
            Vector3 localScale = transform.localScale;
            localScale.x = -Mathf.Abs(transform.localScale.x)*movementDir;
            transform.localScale = localScale;

            if(frameIndex < animSprites.Length) {
                // Let the renderer know which sprite to
                // use next
                animRenderer.sprite = animSprites[ frameIndex ];

                // Move the game object only when showing the 3rd and 4th frames
                // (when the frog is in the air)
                if(frameIndex >= 2 && frameIndex <= 3) {
                    // Move the game object
                    Vector3 shift = Vector3.right * movementDir * speed * Time.deltaTim
                    transform.Translate(shift);

                    // Do the parallax shift
```

```
               if(parallaxComp) {
                  parallaxComp.Move(shift);
               }
            }
         } else {
            // Animation finished, set the render
            // with the first sprite and stop the
            // animation
            animRenderer.sprite = animSprites[0];
            animRunning = false;
         }
      }
   }
}
```

Variable `parallaxObj` is a reference to the *Parallax* game object, which will be initialised later in the **Inspector panel**. `parallaxComp` is a reference to Parallax script component of the *Parallax* game object - its initialised int the *Start()* method. During the *Update()*, whenever the frog is about to undergo a shift in position, the *Move()* method of the Parallax is executed to move the camera with the frog, and get the background parallax going.

4. In the **Hierarchy panel** select the *Frog1* game object and drag the *Parallax* game object to *Frog1*'s *Frog1Control (Script) | Parallax Obj* variable.

5. When you play the game, the frog should remain stationary (with respect to the camera) and different aspects of the background should move at different speeds (the hills moving too, although barely)



At this point, if you venture too far to the left or right, the backgrounds will move completely out of camera's view. To fix this you would need to configure continuously scrolling backgrounds. One way of achieving this is to have two copies of each background placed side by side in the scene. The background images must be done in such a way that placing two copies side by side forms a continuum (as is the case for the background images in this lab). As the camera moves, the two copies of the same background image sprite can leap-frog each other (by changing their corresponding game object's positions) so that one is always in camera's view. There are a number of tutorials on how to achieve this; and so it is left as an exercise to do on your own.

Animations and Animator Controller

Did you notice how the frog blinks after each jump? Wouldn't it be nice to have the blinking occur independently of jumping, at random times, to make it seem more natural? For this, you need some kind of state machine that can switch rendering to the blinking sprites at random during idle times, and to the jumping sprites whenever there is movement due to user input. It's possible to implement such a state machine using the animation method you've just seen. However, in this part of the exercise, you are going to take advantage of other animation tools that the Unity Editor provides.

6. Duplicate *Frog1* game object. Rename the duplicate to *Frog2*.

7. Disable *Frog1* by unsetting the check bock in its **Inspector panel** properties. The object will remain in the hierarchy, but won't be used in the scene.

8. Now, back to *Frog2*. Remove its *Frog1Control* script component. Add a new component: *Miscellaneous | Animator*.

9. In *Assets | Animations* right-click and select *Create | Animator Controller* to create new animator controller. Name it *Frog2*. Drag the newly created controller to *Frog2* game object's *Animator | Controller*.

0. Animator controller is a state machine that controls when to play what animation for its game object. If you double-click on the *Frog2* animator controller an *Animator* tab should open alongside the *Scene* and *Game* views. Right now the state machine has three default states: *Enter*, *Exit* and *Any State* (if you can't see the states in the Animator window, press ALT, or middle-mouse wheel, and move the window around).

1. Time to create some some animations. From the main menu, select *Window | Animation*. It should open window/tab, where animations can be created. To keep the screenshots relatively small for this lab, I keep my Unity workspace small, and so the best place for me to put the *Animation* is in the **Project panel** along the *Project*, *Console* views. Generally, you want the *Animations* window placed such that you can the *Scene* or *Animator* tabs at the same time.

Animation vs Animator

The difference between *Animation* and *Animator* views is that the first is used to create animation sequences, whereas the latter is used to implement the state machine that controls those animations.

2. Make sure that *Frog2* is selected in the **Hierarchy panel**. In the *Animation* window, click on the *Create* button to create a new animation clip..

3. When prompted, name the new clip *frog_idle* and make sure to save it into *Assets / Animations*. Since the *Frog2* game object was selected in **Hierarchy panel** while the new animation was created, two things will happen. First, the new animation gets created and loaded into the *Animation* view. Second, the new animation becomes a new state in *Frog2*'s animation controller.

4. For the next few steps, it's best to drag the *Animation* view tab to where the *Scene*, *Game* and *Animator* tabs are, so that *Animation* and *Project* tabs can be seen at the same time. Set *frog_idle*'s *Samples* to 10 (that means 10 frames per second). If the "Samples" setting is missing from your Animation window (check the image below for where it should appear), click on the cogwheel icon in the top right-hand corner of the Animation panel and select "Show Sample Rate".

5. Drag the *frog_0* sprite from *Assets / Sprites* to the animation timeline in *Animation* view, positioning it at time 0:00. Do it again, the same sprite, positioning it at time 1:00. This animation won't do much, just keep

showing the same sprite.

6. Create a new animation clip (by expanding that window where it says *frog_idle* in the *Animation* view, and selecting *Create new Clip*). Name the animation *frog_blink* and make sure it's saved to *Assets | Animations*.
7. Select *frog_5* through *frog_7* sprites in *Assets | Sprite* and drag everything to the animation timeline for the *frog_blink* animation. Make sure the sprites are placed at times: 0:0, 0:1, and 0:2.

8. If you want to preview the animation, you need to be able to see the *Scene view* and the *Animation* tab simultaneously, so drag the *Animation* tab back to the **Project Panel**. Press the play button in the *Animation* view (not the play button that plays the game) to see a continuous play of the animation in the scene. If you have trouble locating the play button, follow the arrow in the screenshot below.

9. Create a new animation clip. Name the animation *frog_jump* and save it to *Assets | Animations*.
10. Select *frog_1* through *frog_4* sprites from *Assets | Sprites* and drag everything to the animation timeline for the *frog_jump* animation. Make sure the sprites are placed at times: 0:0, 0:1, 0:2, and 0:3. When you preview the animation clip, the frog should be now jumping in place.

Setting up the state machine

You've got a set of animation clips. Now it is time to set up the state machine for transitions between different clips.

1. If you take a look at the *Animator* view, you should see three states corresponding to the new animation clips: *frog_idle*, *frog_blink* and *frog_jump*. These clips got associated with the *Frog2* Animator Controller because *Frog2* game object was selected in the **Hierarchy panel** when you were creating them. The orange state is the starting state. You can change the starting state by right-clicking on a state and selecting *Set As Layer Default State*. Leave the *frog_idle* as the default state.

2. You'll need two parameters for triggering transitions from one state to another. Transitions can be triggered by different types of variables - for this project you're going to use a *Trigger* variable, which can be set from a script. There's a *Parameters* tab in the top-left side of the *Animator* view. Click on it, then click the + sign twice, each time selecting *Trigger* as the type of the variable. Name the variables *Blink* and *Jump*. The box beside each variable indicates its default value - keep it as unset.

3. To create a transition from *frog_idle* to *frog_blink*, right-click on the *frog_idle* state. Select *Make Transition* - you will get a transition arrow from *frog_idle* state to your mouse pointer. Mouse over *frog_blink* and click. The transition arrow should go now from *frog_idle* to *frog_blink*.
4. Click on the transition arrow itself to see the properties of the transition in the **Inspector panel**. By default transitions occur automatically after certain time. Un-tick the "Has Exit Time" attribute and expand the *Settings* attribute. Set the *Transition Duration* and *Transition Offset* to 0 - the transition will be instantaneous,

when triggered. To set up the trigger, scroll down the properties until you see *Conditions*. Click on the plus sign to create new condition and set it to *Blink*. The transition will take place when the *Blink* variable is triggered.

5. Make a transition from *frog_blink* back to *frog_idle*, this time leave the *Has Exit Time* set - that means transition from Blink state back to Idle will occur automatically after certain time. Expand the *Settings* attribute and set *Exit Time* value to 1. This setting represents the progress of the state when the transition is to occur. The setting of 1 correspond to 100%, so after the Blink animation runs its course, it will transfer back to Idle. Again, set the *Transition Duration* and *Transition Offset* to 0.
6. Make a transition from *frog_idle* to *frog_jump*. Disable *Has Exit Time*, set the *Transition Duration* and *Transition Offset* to 0, and set the *Condition* to the *Jump* trigger.
7. Make a transition from *frog_jump* back to *frog_idle*. Leave *Has Exit Time* enabled, set the *Exit Time* to 1, and the *Transition Duration* and *Transition Offset* to 0.
8. Just in case the player tries to move the frog while it's in the blink state, create a transition from *frog_blink* to *frog_jump* triggered on *Jump*.

9. The triggers need to be set from the script. You'll start with the frog blinking. Create a new C# Script called "Frog2Control" and paste the following code:

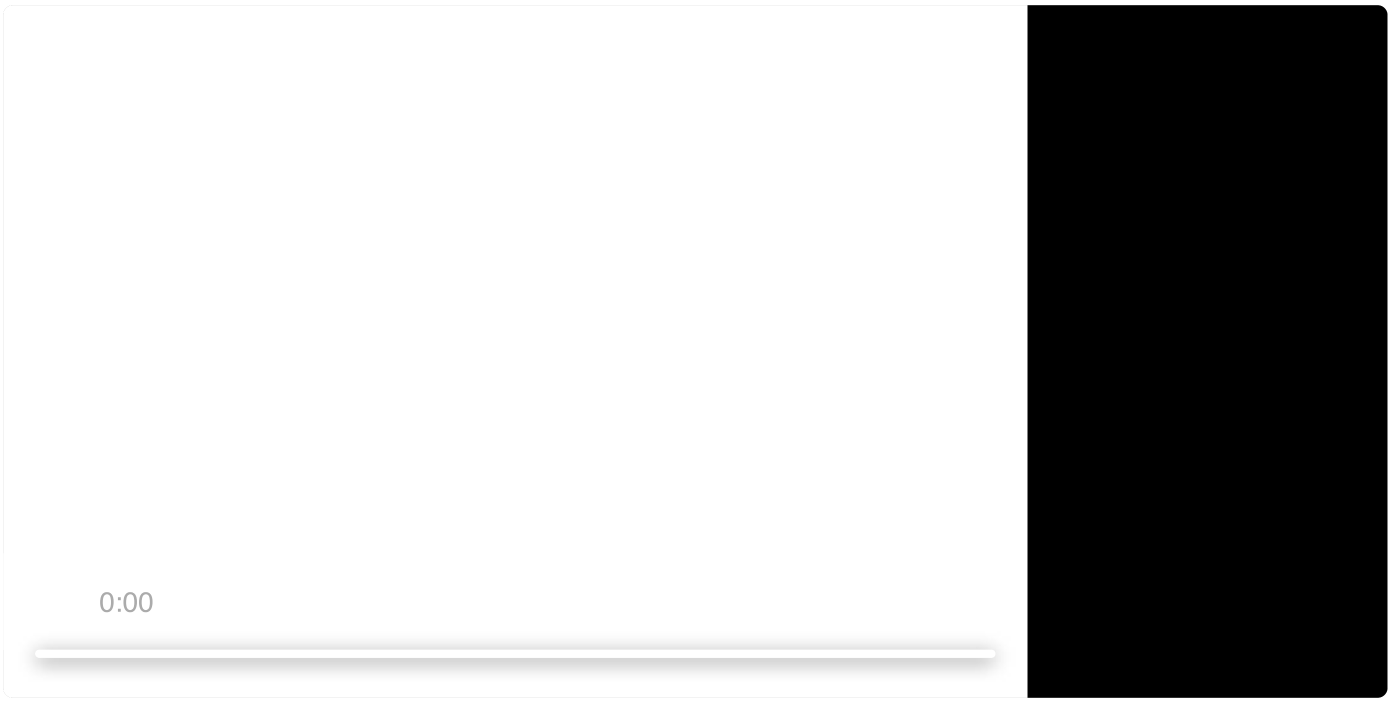
Frog2Control.cs

```
using UnityEngine;
using System.Collections;

public class Frog2Control : MonoBehaviour {

   // Reference to animator component
   Animator anim;

   // Probability of blinking
   float blinkProb = 0.005f;

   // Use this for initialization
   void Start () {
      // Initialise the reference to the Animator component
      anim = GetComponent<Animator>();
   }

   // At fixed time intervals...
   void FixedUpdate () {

      // Draw a random value between 0 and 1, if it's less
      // than probability of blinking, set the Blink trigger
      // of the Animator
      float randomSample = Random.Range(0f, 1f);

      if(randomSample < blinkProb) {
         anim.SetTrigger("Blink");
      }
   }
}
```

This script is meant to be a component of the *Frog2* game object, so on *Start()*, a reference to game object's *Animator* component is fetched. This reference is required for setting the trigger parameters in the Animation Controller. In the *FixedUpdate()* the "Blink" trigger is set at random times.

Add the "Frog2Control" as a script component of the *Frog2* game object.

When you play the game now, the frog should blink once in a while. If you watch the *Animator* view during play, you can see the state machine in action. If the blinking interval doesn't seem right, try different values for `blinkProb` variable in the *Frog2Control* script to change the probability of blinking.



2. Scripting the frog move is a bit more complicated. Again, you don't want the frog to move throughout its entire jump state, but only when it's in the air. You'll need to add events to the animations, which can inform scripts about the stage of the animation that the state machine is at.

3. Add the following code to the *Frog2Control* script:

Frog2Control.cs

```
using UnityEngine;
using System.Collections;

public class Frog2Control : MonoBehaviour {

   // Reference to animator component
   Animator anim;

   // Probability of blinking
   float blinkProb = 0.005f;

   // Speed of the movement
   public float speed = 4f;

   // Direction of the movement
   float movementDir;

   // Keep track of time when frog is moving
   // and when it is in air
   bool moving  = false;
   bool inair = false;

   // Use this for initialization
   void Start () {
      // Initialise the reference to the Animator component
      anim = GetComponent<Animator>();
   }

   // At fixed time intervals...
   void FixedUpdate () {

      // Draw a random value between 0 and 1, if it's less
      // than probability of blinking, set the Blink trigger
      // of the Animator
      float randomSample = Random.Range(0f, 1f);
      if(randomSample < blinkProb) {
         anim.SetTrigger("Blink");
      }

      if(!moving) {
         float userInput = Input.GetAxis("Horizontal");
         if(userInput != 0f) {
            // User pressed the move left or right button

            anim.SetTrigger("Jump");
            moving = true;
            // Get the direction of the movement from the sign
            // of the axis input (-ve is left, +ve is right)
            movementDir = Mathf.Sign(userInput);

            // The frog in the sprites faces left; when its Scale.X value is positi
            // keep facing left, when its Scale.X value is negative it will face ri
            // the signs for Scale.X correspond to opposite directions of the movem
            // So, when movementDir is -ve, make sure Scale.X is +ve, and when mover
            // +ve, makse sure Scale.X is -ve.
            Vector3 localScale = transform.localScale;
            localScale.x = -Mathf.Abs(transform.localScale.x)*movementDir;
            transform.localScale = localScale;
         }
      }
   }

   // Update is called once per frame
   void Update () {
      if(inair) {
         // Move the game object
         Vector3 shift = Vector3.right * movementDir * speed * Time.deltaTime;
         transform.Translate(shift);
      }
   }

   public void SetInAir() {
      inair = true;
   }

   public void ClearInAir() {
      inair = false;
   }

   public void ClearMoving() {
      moving = false;
   }
}
```

The **`speed`** and **`movementDir`** variables do the same things as they did in *Frog1Control* script. The two boolean flags: **`moving`** and **`inair`** indicate when the frog is in the jump state and when it is actually off the ground while in the jump state.

The user input triggers a new jump only when frog is not **`moving`**. This means that player can't change direction of the frog midair. Non-zero input (pressing the move left or right button) switches the *moving* flag to true; sets the frog's X scale so that it faces the direction of the movement, and sets the "Jump" trigger of the animator so that it goes into the *frog_jump* state. The moving flag will be cleared by *ClearMoving()* method (which will be invoked from the animation).

The game object is translated based on the movementDirection only when the frog is in air. That flag is set and cleared by *SetInAir()* and *ClearInAir()* methods, which will be invoked from the animation.

1. To invoke script methods from animation, you need to set animation events and link them to appropriate methods. Go to *Animation* view. Make sure *Frog2* game object is selected in the **Hierarchy panel** and select the *frog_jump* clip.

5. Click the *Add event* button (the pointer icon with the plus sign shown in the screenshot below). An event should be added to the timeline - drag that icon over time 0:01 (the second frame). If you click the event icon in the timeline its properties will display in the Inspector panel. Unity scanned all the script components of the current game object (*Frog2*) for public methods. In the Inspector Panel, under Animation Event -> Function select the *SetInAir()* method. The pointer icon should appear now somewhere in the timeline (above the sprites) indicating when the event takes place. Create another event, select *CleanInAir()* method for it and place it over time 0:03 (the fourth frame).

6. Now select the *frog_idle* animation clip. Create another animation event, link it to the *ClearMoving()* method, and drag it over the time 0:00 in the clip timeline.

7. When you play the game now the frog should hop on command and blink (once in a while) when left idle. The controls are a bit sticky (if you don't let go of the arrow key fast enough, the frog will make another jump). If you'd like to fix that, it's left as an exercise.

8. If you'd like you can also add parallax movement (just follow the example from *Frog2Control*).

Multi-part animation

Multi-part animation is a technique where an object undergoing the animation is composed of several sprites, with each undergoing independent animation. In this part of the tutorial you will animate a car, so that its body moves independently of its wheels, and the wheels rotate when the car is in motion.

9. Disable the *Frog2* game object in the **Hierarchy panel**.

10. Take a look at the *car.png* asset in *Assets | Sprites*. It's another sprite sheet. Set its *Sprite Mode* to *Multiple*, and click the *Apply* button. Then click on the *Sprite Editor* button.

11. Slice the sprite using the *Automatic* setting. The slicer generally does an excellent job when individual sprites are disjoint objects. Click *Apply*.

2. Create an empty game object. Name it *Car*. Set it's Position to (0,0,-1).

3. Create another empty game object. Name it *Body*. Drag it so that its the child of *Car* and set its *Position* to (0,0,0).

4. While the *Sprite Editor* will splice the sprite sheet in the same way, the resulting naming order of the sprites may not be always the same. Therefore, in the figure above, I gave names to different sprites in the sheet. In your assets they will be named *car_x*, where x is a number between 0 and 5. Take the *car_x* sprite corresponding to the *main_body* sprite in the image above. Drag it into the **Hierarchy panel** under the *Body* game object and name it *main_body*. Set its *Position* to (0.57, -2.38, 0).

5. Take the *car_x* sprite corresponding to the *front_window* in the image above. Drag it into the **Hierarchy panel** under the *Body* game object and name it *front_window*. Set its *Position* to (-0.26, -1.64, 0). It should fit nicely into the window gap of the *main_body*. Also, click on the box by the *front_windows*'s *Color* property. A new window should appear where you can choose the colour and opacity of the sprite. Set the Alpha value (opacity) to 140, so that the window becomes semitransparent.

6. Take the *car_x* sprite corresponding to the *rear_window*. Drag it into the **Hierarchy panel** under the *Body* game object and name it *rear_window*. Set its *Position* to (1.63,-1.61,0). Also set its *Alpha* to 140.

7. Drag on of the wheel sprites (doesn't matter which one) into the **Hierarchy panel** under the *Car* game object and name it *front_wheel*. Set its *Position* to (-1.71,-4.09,0).

8. Drag the other wheel sprite into the **Hierarchy panel** under the *Car* game object (but not the *Body* game object). Name it *back_wheel* and set its *Position* to (2.44,-4.03,0).

9. In the *Scene* view, you should see the assembled car with the game object hierarchy as shown in the screenshot below (the order of the objects under a given parent does not matter).

0. Previously, you started animation by creating an *Animator Controller* and linking it to an *Animator* component of the game object that was going to undergo the animation. This was to show all the components that make up an animation. However, Unity provides a shortcut, where few steps are done automatically upon creation of the first animation clip for a given game object. Make sure the the *Car* object is selected in the **Hierarchy panel** and the *Assets | Animations* folder is selected in the **Project panel**. Open the *Animation* view and create a new clip. Name it *car_idle* and save it under *Assets | Animations*. A number of things will happen all at once. Along with the animation, a new Animation Controller, called *Car*, is created in *Assets | Animations*. The *Car* game object gets an *Animator* component and it's *Controller* property gets linked to the *Car* animation controller.

1. If you open the *Animator* view, you'll notice it is loaded with the new *Car* animator controller and it's got the *car_idle* state.

2. Return to the *Animation* view and configure the *car_idle* animation. This time, instead of using different sprites for different frames, you are going to create animation by changing properties of a single sprite. This is done by specifying certain points in time (called keyframes) and setting the properties of the game

object for those keyframes. As the animation is played, the properties will get interpolated between values at keyframes.

3. In the *Animation* view, with *car_idle* animation selected, click on the *Add Property* button. A window will pop-up. It will give you a choice of the property you want to animate for the game object associated with the animation, along with all of its children. On that list, expand the *Body* game object and click on the + icon beside *Transform | Position*.

4. An entry named *Body : Position* should be added to the animation (above the *Add Property* button). There will be two keyframes in the timeline (marked with little diamond icons) position at time 0:00 and 1:00. If you don't see two diamonds, perhaps you zoomed in or zoomed out too much in the timeline - to change the zoom, scroll the mouse when hovering over the timeline.

5. Click on the record button. The Position property of the Transform component in the Inspector panel should be highlighted in red. This means that when you change these values, you're not affecting the current position of the object in the scene, but are setting the changes you want to occur during the animation.

6. Using the controls by the record button (see image below for reference) you can move the white vertical line between different key-frames. In record mode the white line indicates where you want to make changes in the animation.

7. While still in the record mode, move the white line to the left-most keyframe (one at 0:00). Make sure the *Transform | Position* is set to (0,0,0)

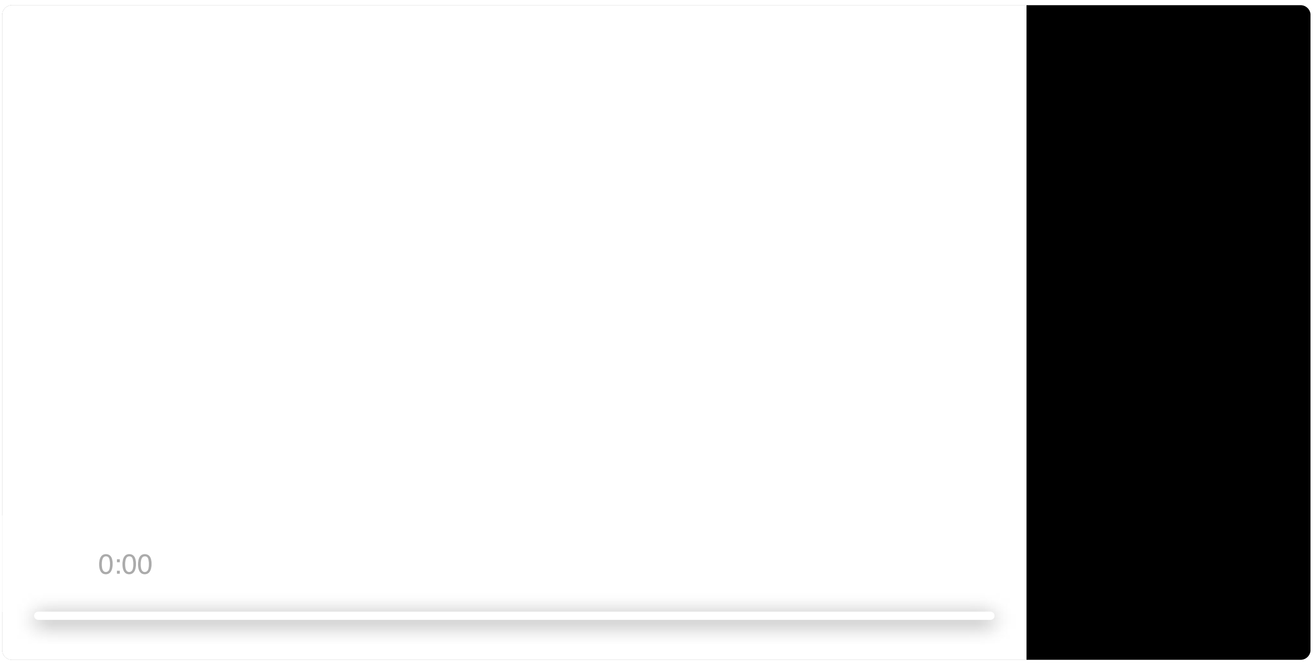
8. Using the control buttons move the white indicator line to the next keyframe (one at 1:00). The *Transform | Position* of the *Body* object in the **Inspector panel** should be still highlighted in red. Set the *Position* of that keyframe to (0,0.05,0).

9. Next, to add another keyframe, double-click in the timeline at 2:00 on the line for *Body | Position* curve.

10. If the keyframe doesn't fall exactly at 2:00, you can drag the diamond to adjust it. Using the control buttons, move the white indicator line to the 2:00 keyframe and set the *Position* to (0,0,0).

11. Disable the keyframe recording mode by de-pressing the record button.

12. To summarise: the animation takes the Y position of the *Body* game object from 0 to 0.05, and then back to 0. This will affect all the children of that object, so the entire body (except for the wheels) will move up and down. This will constitute the entire animation for the idle state. It's meant to give the impression that the engine is on, but the car is not moving.



3. Create another animation (by clicking on the two arrows besides the *car_idle* entry in the *Animation* view and selecting *Create New Clip*). Name it *car_moving* and save it to *Assets | Animations*.

4. Add a property and select *Body | Transform | Position* again. Once again, two keyframes should appear at 0:00 and 1:00 . Add another keyframe at 2:00, and set the *Position*s for consecutive keyframes to (0,0,0), (0,-0.1,0), and (0,0,0). Just like in the previous animation, this will make the car's body bob up and down, thought this time with a bit more displacement.

5. Add another property to the *car_moving* clip. Yes, you can animate many things during the same animation. This time select the *front_wheel | Transform | Rotation* property. Again, configure the animation to have three keyframes at 0:00, 1:00 and 2:00. This time it should be the *front_wheel | Transform | Rotation* property that shows up in red in the **Inspector panel** when you click on these keyframes. Set the *Rotation* in the consecutive keyframes to (0,0,0), (0,0,180) and (0,0,360). The wheel will be rotating about the Z axis by 360. Why rotate about the Z axis? That's the axis that points away from you into the screen, so a rotation about that axis is a rotation in the plane of the camera's view (the XY plane).

6. Add another property to the animation clip. This time select *back_wheel | Transform | Rotation* property. Set it up the keyframes and rotations in the same way as you did for the *front_wheel*. Your *car_moving* animation should look like the screenshot below:

7. If you play the animation (using the play button just above the *car_moving* clip) you should see the car bobbing up and down while the wheels are rotating.

8. Next, you need to set up the state machine in the *Animator* view. Switch to the *Animator* view. The *Car2* animator controller should be loaded there with two states corresponding to the two animations you just created. The *car_idle* should be the default state (since it was created first).

9. Create a new animator parameter - make it an *Int* and name it *speed*.

0. Create a transition from *car_idle* to *car_moving*, un-tick *Has Exit Time* and set the *Condition* to *speed NotEquals* 0.

1. Create a transition from *car_moving* to *car_idle*, un-tick *Has Exit Time* and set the *Condition* to *speed Equal* 0.

2. Create a new C# Script. Name it *CarControl* and paste in the following code:

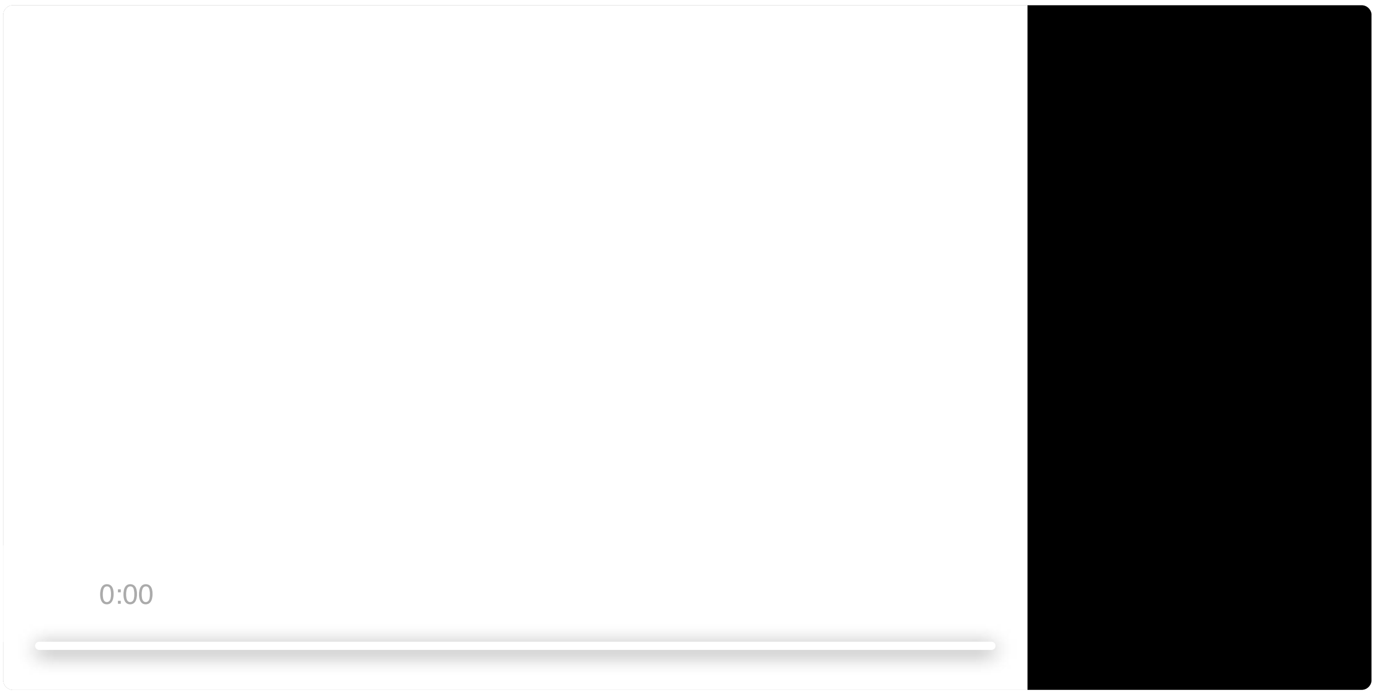
CarControl.cs

```
using UnityEngine;
using System.Collections;

public class CarControl : MonoBehaviour {

   // Reference to animator component
   Animator anim;


   // Use this for initialization
   void Start () {
      // Initialise the reference to the Animator component
      anim = GetComponent<Animator>();
   }

   // Update is called once per frame
   void Update () {
      // Get user input
      float speed = Input.GetAxis("Horizontal");
      // Set the animator variable speed
      anim.SetInteger("speed", (int) speed);

      if(speed != 0f) {
         // The car in the sprites faces left; when its Scale.X value is positive i
         // keep facing left, when its Scale.X value is negative it will face right
         // the signs for Scale.X correspond to opposite directions of the movement
         // So, when movementDir is -ve, make sure Scale.X is +ve, and when movemen
         // +ve, makse sure Scale.X is -ve.
         Vector3 localScale = transform.localScale;
         localScale.x = -Mathf.Abs(transform.localScale.x)*Mathf.Sign(speed);
         transform.localScale = localScale;
      }

      // Move the car (if speed is zero it will remain stationary)
      Vector3 shift = Vector3.right * speed * 2 * Time.deltaTime;
      transform.Translate(shift);
   }
}
```

The script takes user input from the *GetAxis* method as the **`speed`** of the object. This value is what the "speed" variable in the animator gets set with (to control the changes in state). The sign of the **`speed`** determines the horizontal flip of the entire object and it is proportional to the translation of the game object in the scene.

3. Add the *CarControl* script component to the *Car* game object.

4. When you play the game, the car should be animated, bobbing up and down in idle, wheels rotating when moving. For the video shown below, the *Transform | Scale* of the *Car* game object was set to (0.5,0.5,1) to make it smaller, and the *Car* was positioned at *Transform | Position* (0,-2,0).



OK, it doesn't look so great. The wheels slip quite noticeably when you start and stop. That's because GetAxis("Horizontal") returns a float value between -1 and 1, and the *speed* variable, that controls the sate change of the animation, is an integer. As a result, non-zero user input between -0.5 and 0.5 converts to integer of 0. This means that the object might be moving a bit while the animation state from *car_idle* to *car_moving* doesn't get triggered just yet. The same goes for stopping. Also, the flip from going left to going right is a bit abrupt. The scene could possibly use another animation state for switching directions of the car movment. But at this point, it's just refinement of the details and building up of the complexity based on what you just learned. If you'd like to see more, there's a nice online tutorial (http://gamedevelopment.tutsplus.com/series/bone-based-unity-2d-animation--cms-617) with an example of more complex animation, and far more better looking results.

Assessment

Show your work to the demonstrator for assessment.

Things to try

- Add parallax to the scripts controlling *Frog2* and *Car2*'s movement - should be fairly straight forward, based on how it was done in the *Frog1Control* script.
- Fix the controls of *Frog2* so that it doesn't do the occasional extra hop when the player let's go of the controls near the end of the previous hop.

- Fix the controls of *Car* so that there is no wheel slipping when starting and stopping movement.
- Play with animations of other properties of game objects. For instance you can animate the colours of the *hills* sprite to go slowly more orange over time simulatig a sunset. Or, if you needed some objects in the game that grab player's attention (there's an apple sprite in *Assets / Sprites*) you could animate their Scale to get them to pop out (from a zero scale value) to some size.

Intermediate Challenge

- Learn how to use Timeline in Unity (https://docs.unity3d.com/Manual/TimelineSection.html), create one complex cut scene (or few simpler ones) and implement it (them) in Unity's Timeline. The storyboard should be rich (not just stick figures on a black background) and follow a chosen theme. All artwork should be yours. You can have some animation in there if you wish. It's perfectly fine to use the cut scene in your team's game.

Assessment

Show your work to the demonstrator for assessment.

Master Challenge

- Find a tutorial and learn how to *bake* models into sprites using a 3D modelling tool such as Blender. Create nice animations using the *baking* technique. It's perfectly fine to create animations that later will be used for your main game project.

  or

- There are many ways of creating nice and fluid animations. Come up with your own and use it to develop awesome animations. If you need some inspiration take a look at what Jordan Mechner did for the original Prince of Persia (http://www.museumofplay.org/blog/chegheads/2014/10/jordan-mechner-collection-documents-revolution-in-game-graphics).

  or

- Devise and complete your own Animation Master Challenge - just check with the lecturer or the demonstrator first, whether the scope of the work will be sufficient for the awarded skill points.

Assessment

Show your awesome animations to the demonstrator for assessment.