CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 6B – MAKING IT PRETTY VOL 2: ANIMATIONS

DUE: DEMO BY END OF LAB WED MARCH 15
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

INSTRUCTIONS:
This lab is a bit different. You can apply this lab to either Lab 6 or Lab 7, whichever one you have recently finished.

Objective:

To have the students add Animation to their Enemy and import a Model that uses a Blend Tree Animation.

Preparation:

Supplemental Video Tutorials:

https://unity3d.com/learn/tutorials/topics/animation

Background and Theory:

Animation:

As described in Lab 6's *Preparation* section, Animations define the subtle 'small' movement of a Game Object. Typical Animations include those of walking, running, jumping, and even dying. In this Lab, you will apply a walking and dying Animation to your Enemy model and import a model that uses a blend tree.

The most common technique for animating a character is called *skeletal animation*. Like it sounds, this is when a series of 'bones' are setup inside the model. The 'bones' are rigid but move according to the type of animation you are trying to achieve. The mesh that covers the model, on the other hand, is not rigid. It will flex and deform accordingly when a 'bone' is moved.

The process of designing the 'skeleton' for a 3D model is called *Rigging*. The process of defining how the mesh will deform and flex is called *Skinning*. Typically, for a Character 3D Model the workflow goes like this:

1. Design the Character – hand drawn drawings of the character.
2. Model the Character – draw the character digitally in 3D Modelling software.
3. Texture Unwrapping – convert the model into 2D for texturing (discussed in Lab 6).
4. Texturing – texture the model (skin, clothes, etc.). Textures were discussed in Lab 6.

5. Shading – custom Shaders for the character (Shaders were discussed in Lab 6).
6. Rigging & Skinning – define the 'skeleton' for the character and how that skeleton will deform and flex the mesh.
7. Animate – pose the model.
8. Record – record a series of poses to produce a desired animated clip (walking, running, etc.).

As you can see from the above eight steps, character modeling and animating is definitely a serious, time-consuming undertaking.

Animation and Unity:

Unity has two animations systems: *Legacy* and *Mecanim*. The Legacy animation system was used before Unity 4 and is in the process of being phased out. *Mecanim*, based on skeletal animation, is the modern, more advanced animation system. One feature of *Mecanim* is *Retargeting*. Retargeting is the process by which you can define one set of Animations and use it on various humanoid models. This is made possible by Unity's *Avatar* system. This basically maps the skeleton of any humanoid model to a common internal format (called the Avatar).

Animating the Enemy:

If you used the Space Robot Kyle model, you'll find that it doesn't come with any Animations. However, it is humanoid and is fully rigged. That means we can take advantage of *Mecanim's retargeting feature* and apply a humanoid animation to use with it.

Animating any Humanoid model that doesn't come with its own Animations is covered in Tasks 1 and 2.

Animating a Humanoid or Non-Humanoid model that comes with its own Animations is covered in Tasks 3 and 4.

Unity API References for this Lab:

https://docs.unity3d.com/ScriptReference/Animator.html
https://docs.unity3d.com/ScriptReference/Animator.SetTrigger.html
https://docs.unity3d.com/ScriptReference/Animator.SetFloat.html

Tasks:

Tasks Part 1 – Applying a Walking Animation to a Humanoid Model (Robot Kyle)

1. First, you need to setup the Animation System for Space Robot Kyle. Click on the Robot Kyle model under the *Robot Kyle->Model* folder. In the Inspector Tab, you'll see three buttons: Model, Rig, and Animations. Click on the Rig button. Where it says *Animation Type*, change it to **Humanoid**. Hit the *Apply* button. You should see a checkmark appear beside the *Configure…* button.

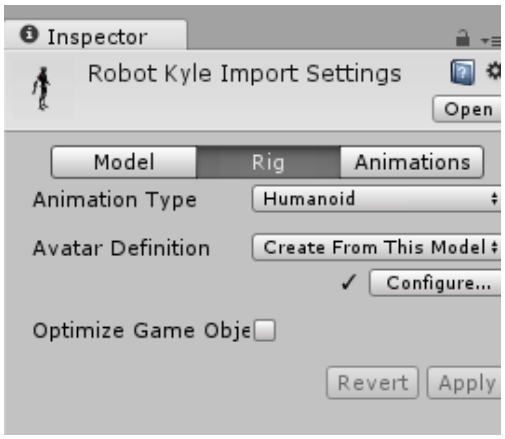

Figure 2: Robot Kyle Import Settings

2. If you click on Configure… button, this puts Unity into *Avatar Configuration mode* and your Scene will disappear and the model will be displayed showing its 'skeleton' in a pose called the *T-Pose*. This is where you can manually configure the mapping between 'bones' (shown in the Hierarchy Tab) and the Avatar body map (shown in the Inspector Tab). This was already done for us automatically (remember the checkmark?) so no changes are needed here.

3. If you click on the *Muscles & Settings* button near the top of the Inspector Tab, you put the Model into its *Stress Test mode*. Here you can play with the sliders to see how the 'bones' and mesh perform (and deform) when putting the model in different poses. When you are done testing, hit the **Done** button in the Inspector Tab.

4. If you select Robot Kyle in the Assets window of the Project Tab and click on the small arrow on the right, you'll see the various elements that make up the Robot Kyle 3D Model. The two Robot2s shown are the mesh, the Root is the 'skeleton' and the Avatar, which you just generated in the steps above, is the humanoid mapping for animation retargeting.

5. At this point, our Enemy is ready for some Animations. But first, we need to get some! Fortunately, Unity provides a Standard Assets package that contains all sorts of goodies including a set of humanoid animations. Go up to *Assets->Import Package->Characters*. We don't need all these Assets so click the None button in the Import Package window. Then look for the *HumanoidWalk.fbx* asset in the *Animation* folder of the *ThirdPersonCharacter* folder. Import only the *HumanoidWalk.*fbx file.

6. Once Imported, it will create a Standard Assets folder in your Project. You may want to re-organize this by creating an Animations folder under the root Assets folder and moving

the *HumanoidWalk.fbx* Animation file there.

7. Select the *HumanoidWalk.fbx* Animation file. In the Inspector Tab, click on the Animations button. At the bottom of the Inspector Tab is the *Animation Preview window*. If you click the Play button, you'll see a preview of what the Animation looks like. Note, however, that this isn't using the Robot Kyle model (instead a default model). To see the animation with the Robot Kyle model, click the tiny Avatar button in the lower right of the window (circled in the figure below) and select the *Other…* option.

Figure 3: The Animation Preview Window

8. In the *Select GameObject* window that pops up, select *Robot Kyle*. You should see the model in the *Animation Preview* window change to Robot Kyle! Close the *Select GameObject* window and press Play again in the *Animation Preview* window. Now you can see how the Animation will look with Robot Kyle!

9. We want the Animation to be attached to our Robot Kyle Prefab. To edit Prefabs, we have to place an instance of it in the Scene, edit it, then select *GameObject->Apply Changes to Prefab* to copy the changes back to the Prefab. To start with, place the Enemy Prefab in the Scene.

10. The Enemy Prefab probably has an *Animation* Component. This is a component for the *Legacy* Animation System. Since you are not using this system, delete this Component from the Prefab.

11. Now Add the Animator Component. Press the Add Component button and select *Miscellaneous->Animator*. This is the component used with the *Mecanim* Animation System.

12. Click on the small circle beside the box with the Avatar label. This will bring up the *Select Avatar* window. Select the *Robot Kyle Avatar* and close the window.

13. Now you need what's called an *Animator Controller*. Go up to *Assets->Create->Animator*

Controller. Name it *Enemy AC*.

14. Now select the Enemy Prefab that's in the Scene (that means select the one in the Hierarchy Tab and **NOT** the one in the Asset window). Drag the *Enemy AC* you just made and drop it in the box beside *Controller* in the Animator component.

15. That's all the changes you need to make to the Prefab. Go up to the *GameObject* menu and select *Apply Changes to Prefab*. That should copy the changes you made back to the Prefab so you can delete the instance you've been editing in the Scene (the one in the Hierarchy Tab).

16. Now double click on the *Enemy AC*. This will open the *Animator Tab*.

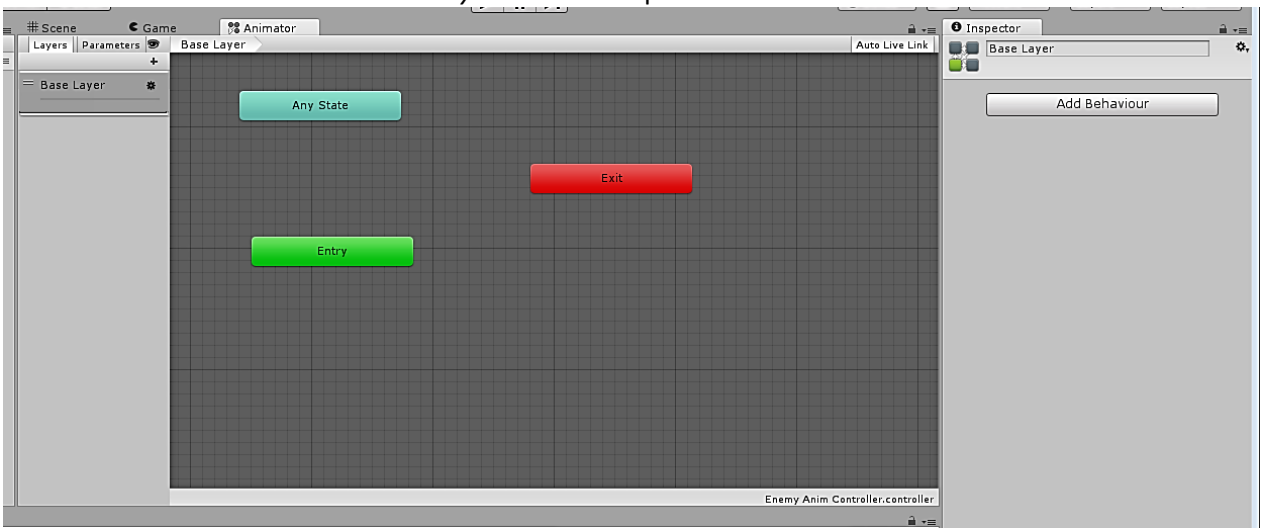

Figure 4: The Animator Tab

The *Animator Tab* is where you load animations and create, arrange, and connect Animation States. As you can see, Animation in Unity is done through a State Machine. Each Animation clip is a state. In the Animator Tab, you connect the different states together in a logical way (for example, going from walk to run to walk again). Your code will define when the Animation State Machine should switch states.

Three states are created by default as show in the figure above and discussed below:

Entry State - This is the state the SM is in when first executed. It will then transition into the appropriate connected state based on conditions that you define.

Any State – Allows you to create a transition to another state no matter what state the SM is in. We won't be using this for now.

Exit – A transition to this state exits the SM. We won't be using this state either.

17. Drag the *HumanoidWalk.fbx* Animation file into the Animator Tab. A new state will automatically be created called *HumanoidWalk*. A transition from the Entry state will automatically be made. The orange color of the HumanoidWalk state indicates that it is a *default state*, i.e. the state that the SM will always go to when entered. Since we want the Enemy to always be walking, we won't set any specific conditions on this transition – the Enemy will start walking the moment it is placed in the Scene.

18. With the *HumanoidWalk* state selected, check the *Foot IK* option over in the Inspector Tab. IK stands for Inverse Kinematics. When an Animation plays, it changes the positions of the limbs of the 3D model including the feet. This can cause the feet to slightly go into the ground. By checking this option, Unity will slightly re-position the model so that the feet fall convincingly on the floor without going through it!

19. Play the Scene! You should see the Robot Kyle Enemy walking now!

Tasks Part 2 – Adding a Death Animation to a Humanoid Model (Robot Kyle)

1. One little problem you may have noticed, the Animation continues playing even when an Enemy is shot. The Tween still executes but the Enemy is still walking! Let's add a more advanced Animation for when the Enemy dies. In D2L, download and unzip the *Standing_React_Death_Forward.zip* file. Import the *fbx* file into your Project. This animation comes from mixamo.com.

2. In the Assets window, click on this Animation and in the Inspector Tab click on Rig. Change the Animation Type to *Humanoid* and click the *Apply* button (you did the same thing back in Step 1 for the Robot Kyle model).

3. Now click on the Animations Tab. In the Animation Preview window, you can play this Animation to see what will look like (**Note:** you'll see a Warning in Unity about the Animation having import warnings. You can disregard this). There is one change you need to make here. In the Inspector Tab, check the box that says *Bake Into Pose* under Root Transform Position (Y). When you do this, you'll notice that the Average Velocity Y value goes to 0. Basically, checking this options means that the Animation won't affect the Y position of the model so that it doesn't end up going through the floor or floating when it dies (when an Animation influences motion, it is called *Root Motion*). Make sure to hit the *Apply* button.

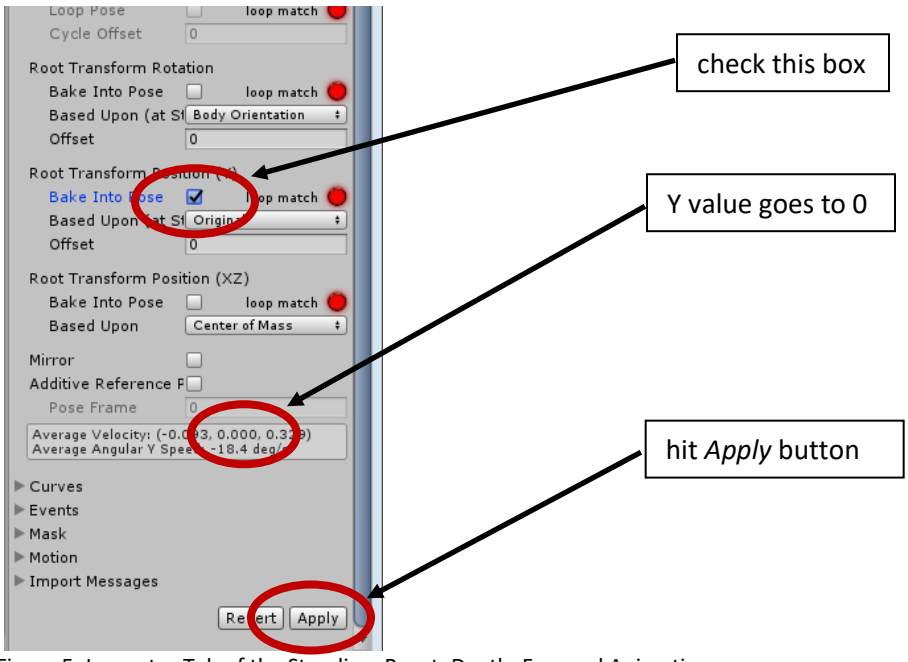

Figure 5: Inspector Tab of the Standing_React_Death_Forward Animation

Note: The red bulb that you see for each of the Root Transforms indicates that the beginning and end pose do not match. If they did, the bulb would turn green. This is only important if you are trying to loop the Animation (which we aren't for this one).

4. Now double click on the Enemy AC Animation Controller to open it in the Animator Tab. Drag the *Standing_React_Death_Forward* Animation into the layout area.

5. Give this State a name. In the top box in the Inspector Tab, type *StandingDeath* and hit Enter. Next, check the Foot IK box, just like you did for the *HumanoidWalk* state.

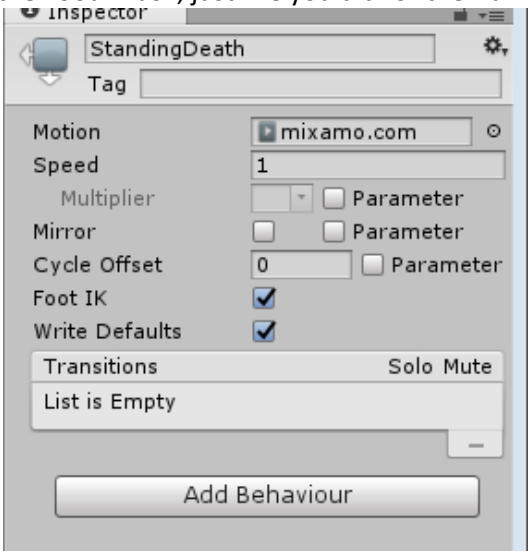

Figure 6: Inspector Tab of the StandingDeath State

6. Right click on *HumanoidWalk* and select Make Transition. Drag the arrow to the *StandingDeath* State.

7. Now click on the arrow. You can play the Animation in the Animation Preview Window. What you'll see now is the combination of the two animations – going from walking to dying. In the Inspector Tab, un-check where it says *Has Exit Time*. This ensures that the death animation will start immediately once in the *StandingDeath* state.

8. The next thing you need to setup is the condition that will put the Animation Controller from the HumanoidWalk state into the StandingDeath state. To begin with, you need to create a Parameter. On the top, left side of the Animator Tab, click on the Parameters Tab:

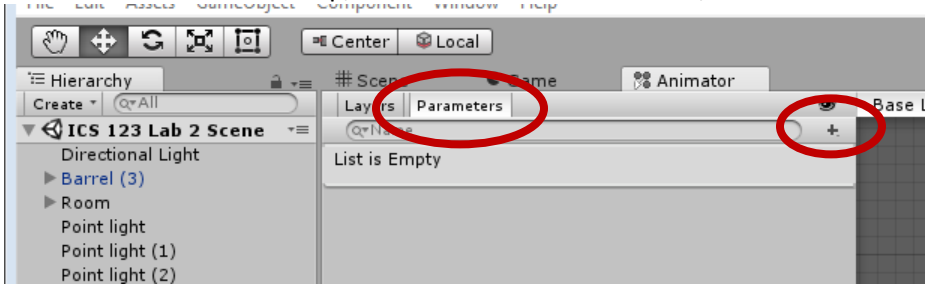

Figure 7: Animation Parameters Tab

9. Next, click on the small '+', circled in the figure above. A drop-down will appear; select Trigger. Name it *Die*.

10. In the Layout area, click on the arrow between *HumanoidWalk* and *StandingDeath* again. In the Inspector Tab, there's a section called Conditions. Click on the '+' at the bottom right of this area. Select the '*Die*' parameter.

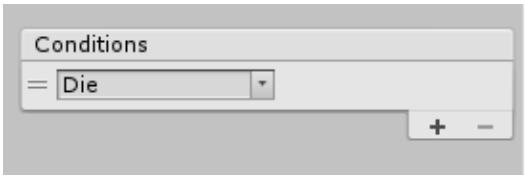

Figure 8: Trigger for the HumanoidWalk -> StandingDeath Transition

This sets the condition to make this transition happen. When the Die Trigger is set, a transition from *HumanoidWalk* to *StandingDeath* will be made.

11. Double click on the *ReactiveTarget.cs* Script to open it **MonoDevelop**. In the `ReactToHit()` method, add the following code before the call to the Coroutine:

```
Animator enemyAnimator = GetComponent<Animator> ();
if (enemyAnimator != null) {
    enemyAnimator.SetTrigger ("Die");
}
```

The code is pretty easy to follow. We get a reference to the Animator component and ensure it isn't null. Then we set the *Die* trigger to cause the Animation state transition.

12. Since we now have an animation for the death sequence, comment out the iTween in the `Die()` coroutine. Save the Script and Play your Scene!

13. One thing you'll notice right away is that the Animation doesn't get a chance to finish before the dying Enemy is removed from the Scene. This is due to the coroutine destroying the Game Object before the Animation has completed. There's a couple of different ways you can handle this. One way would be to time the Animation and change the *WaitForSeconds* parameter of the coroutine. But there's a much slicker way that doesn't even require a coroutine! Click on the *Standing_React_Death_Forward.fbx* file in the Assets Tab.

14. In the Inspector Tab, click on the arrow beside where it says Events. In the Animation Preview window, drag the red play bar so that the window says Frame 110 (it doesn't have to be exact, frame 108 or 109 is also fine). Now hit the icon shown in the figure on the next page.

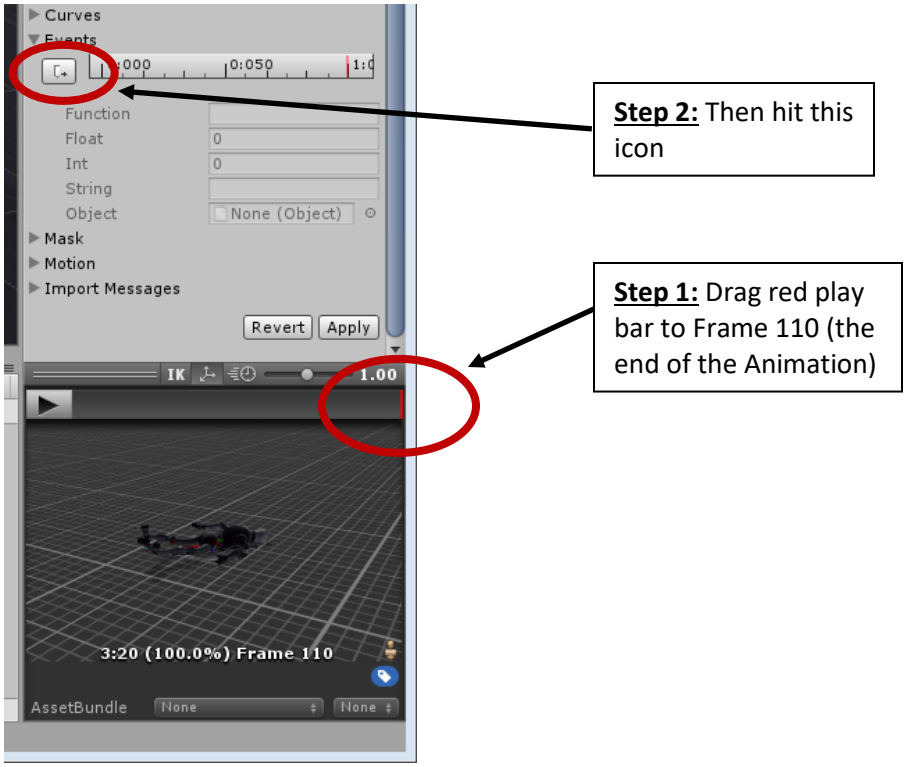

Figure 9: Adding an Event

15. Name the Function *DeadEvent*. Then, beside the Object label, drag the Enemy Prefab in the Assets window onto the box. Hit the *Apply* button. This means that when this Animation completes its sequence, it will generate an Event which calls the *DeadEvent* function.

16. Back in *ReactiveTarget.cs*, add the following function:

```
private void DeadEvent () {
    Destroy (this.gameObject);
}
```

17. Since this will take care of destroying the game object after the Animation is done, you no longer need the Coroutine. Comment out the call to it in the `ReactToHit()` method.

18. Now play your Scene again and enjoy the nicely Animated Robot Kyle (or other Humanoid model) Enemy!

Tasks Part 3 – Animating a Model that comes with its own Animations and Blend Trees

1. This time, we'll add a more complex model that comes with its own Animations. In this case, it will be non-humanoid. The following directions will also work for humanoid models that also come with their own Animations. For this example, a free model called Iguana was chosen. You can use this or another model that has its own Animations.

2. We're going to be selective in the Import Unity Package Window for the Iguana. This particular model uses a feature called LOD. LOD stands for *Level of Detail*. Essentially what this does is use two different meshes for the same model. One mesh is quite detailed, the other more basic. The idea is that when the model is close to the camera in the Scene, use the detailed mesh. When the model is distant from the camera, use the basic mesh. This improves performance and reduces the load on the graphics hardware. We won't use this feature for our project, but it is good to be aware of what it is. We'll instead use the model in this package that always uses the less detailed mesh. In *the Import Unity Package Window*, select the following items as shown in the figure below:

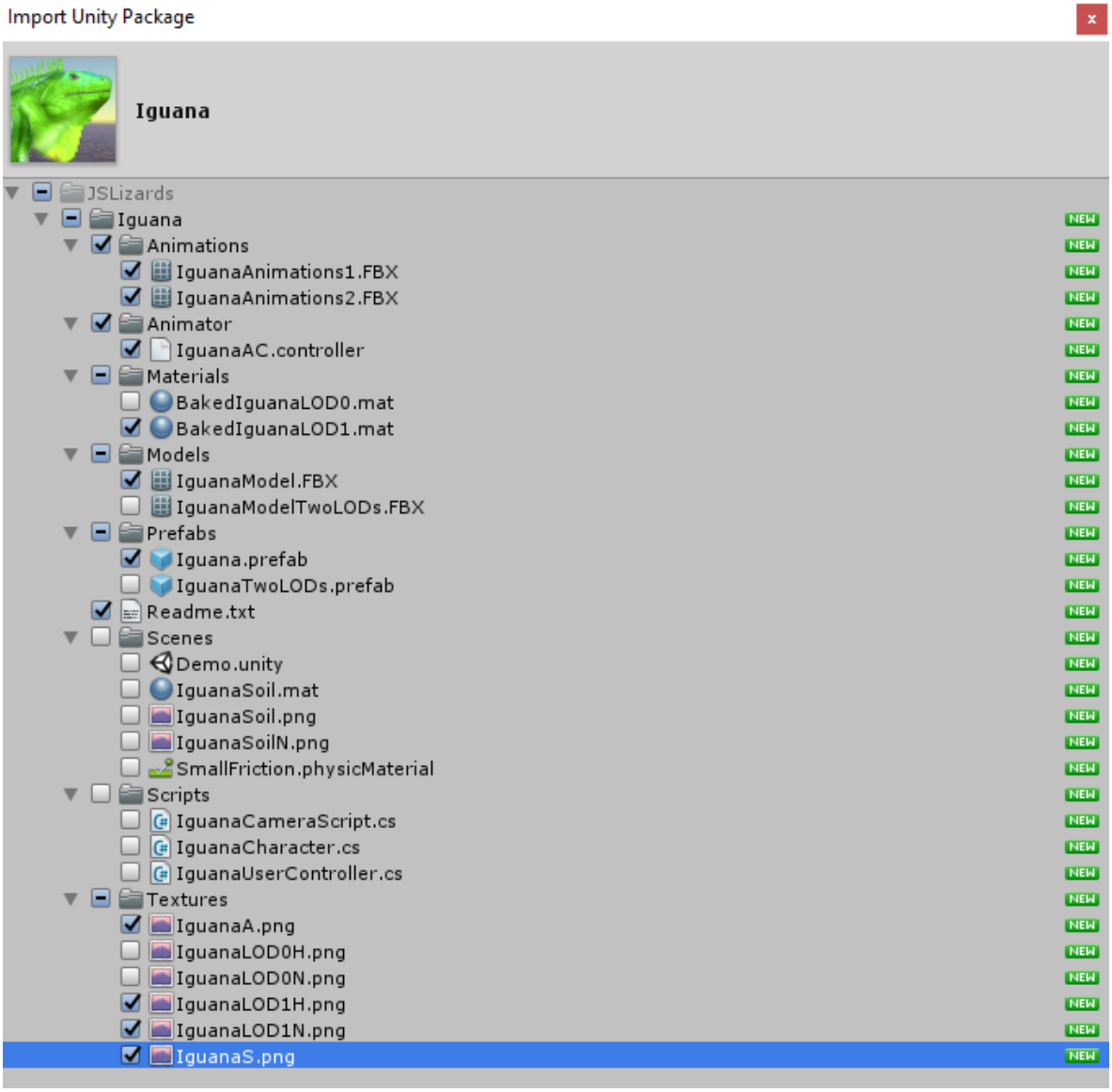

Figure 10: The Iguana Import Unity Package Window

All the Animations for this model are in two files. It also comes with an Animation Controller. Under Materials, we de-select the LOD0 one which is used on the detailed mesh we aren't importing. We also de-select the LOD Model and Prefab. De-select the Scenes since we are importing this into our own Scene. Also, de-select the Scripts since these were designed for the Iguana to be controlled by the Player. Finally, de-select the LOD0 textures. All other Assets are imported.

3. After Importing, you may want to re-organize the Iguana's folders. This is up to you but I recommend you keep the Iguana files together. The next step is to drag the *Iguana Prefab* into the Scene for editing. Change the X, Y, and Z Scale to 2 (no bigger!).

4. With the Prefab selected from the Hierarchy Tab (the one in the Scene), remove the missing Script components (the two at the bottom) and the Rigidbody Component. As always, when making changes to an instance of a Prefab, select *GameObject->Apply to Changes to Prefab*.

5. You'll notice that this is *full-fledged prefab* – it has a collider, animator and script attached. One thing that is different in this model's Animator Component is the *Apply Root Motion* box is checked. That means that the Animation itself will control the movement of the model (instead of a Script, like with the Enemy Prefab) but if you play your Scene, the Iguana doesn't move. Double click on the *IguanaAC* to open its Animation Controller in the Animator Tab.

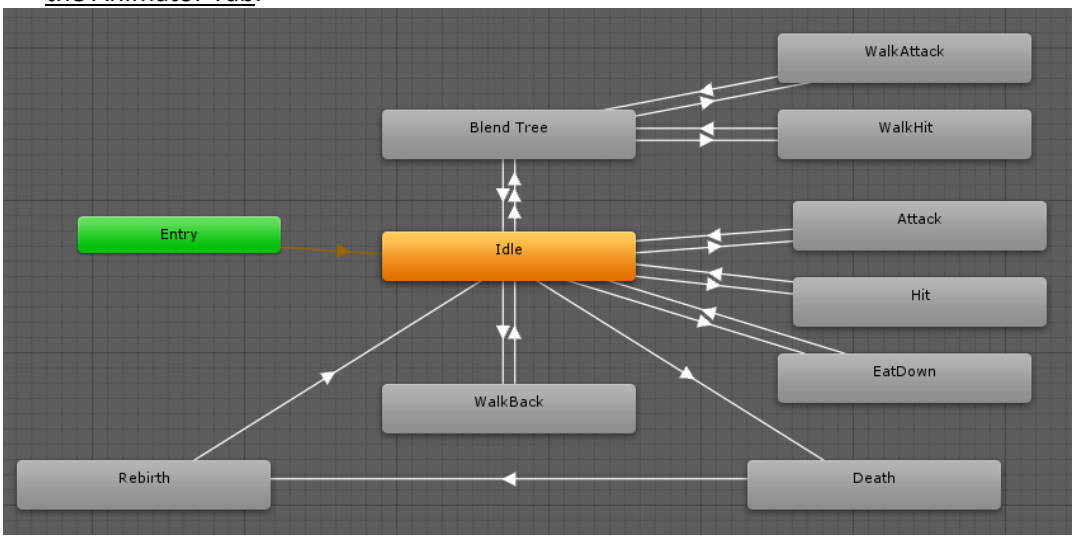

Figure 11: The Definitely More Complicated Iguana Animation State Machine

Right now, the Iguana is stuck in Idle, the default state. The Idle animation for the Iguana is, well, not moving. You'll also notice a state called Blend Tree. Blend Tree States allow you to mix animations together and appear as one smooth blended animation. The best way to understand this is to see an example. Double click on the Blend Tree state in the Animation layout area. This will open up the Blend Tree and should look like the figure on the following page.

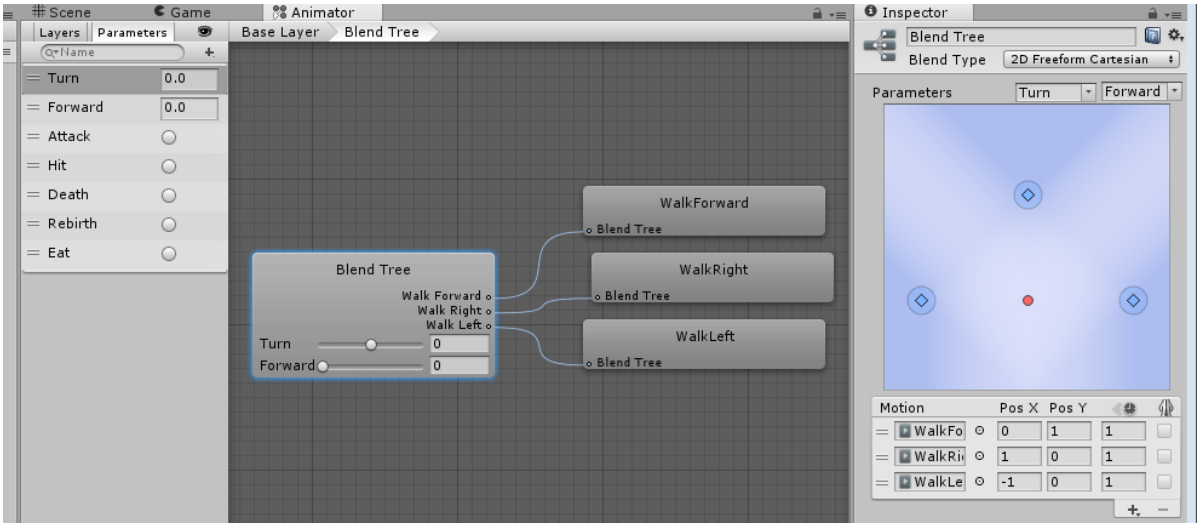

Figure 12: Blend Tree for the Iguana's Walking Animation

6. There's a lot of information in the above figure. First, on the left-hand side are the Parameters. The first two are floats and they are used for the Walking Blend Tree Animation. The others are all Triggers for transitions into other states. On the right-hand side is a 2D graphical realization of the blend tree. At the top, you can see the two parameters: *Turn* and *Forward*. This maps the Turn parameter to the X axis and the Forward parameter to the Y axis. At the bottom, are the motion fields (animations are called motion fields in Blend Trees). An X,Y value of 0,1 and the *WalkForward* animation clip plays. A 1,0 value and the *WalkRight* animation plays. A -1,0 value and the *WalkLeft* animation plays. You can test this by loading up the Iguana model in the Animation Preview Window, hitting Play, and adjusting the Sliders in the Root Blend Tree node.

7. The blending magic happens when you don't use those absolute values. For example, set the sliders to 1,0. This activates the *WalkRight* animation and the Iguana will turn right in place. But if you set the *Forward* slider to a value of 0.5, the Iguana actually walks a bit forward and turns, increasing the radius of its turn. The two animations are blended together!

8. Just like with the Animations in Tasks 1 and 2, all this has to be controlled through a Script. Create a new C# Script and call it *WanderingIguana*. Attach this Script to the Iguana in the Scene and apply the change to the Prefab.

9. In the *WanderingIguana.cs*, add the following class scope variables:

```
public float iguanaSpeed = 3.0f;
public float obstacleRange = 9.0f;

private Animator iguanaAnimator;

private float turn = 0.0f;
```

The first two variables are identical to what was used in WanderingAI. The `iguanaAnimator` variable will store a reference to the Animator instance. The `turn`

variable will store the current turn value based on the Animation blend tree (a value between -1 and 1).

10. In the `Start()` method, add the following statement (next page):

```
iguanaAnimator = GetComponent<Animator> ();
```

We use our trusty `GetComponent` generic method to get a reference to the Animator.

11. **Below** `Update()` add the following method:

```
private void Move(float x, float y) {
    if (iguanaAnimator != null) {
        iguanaAnimator.SetFloat ("Turn", x, 0.2f, Time.deltaTime);
        iguanaAnimator.SetFloat ("Forward", y, 0.2f, Time.deltaTime);
    }
}
```

The method is fairly straight forward. We get the x and y values and set the turn and forward parameters for the Animation (recall that *Apply Root Motion* is checked in the Animator so the Animation will do all the moving). The third parameter to `SetFloat` is called the *damptime*. This allows for a smooth transition between different values of x and y (transitioning between two points).

12. **In** the `Update()` method, add the following code:

```
1       iguanaAnimator.speed = iguanaSpeed;

2       Ray ray = new Ray (transform.position, transform.forward);
3       RaycastHit hit;
        // Spherecast to determine if need to turn
4       if (Physics.SphereCast (ray, 0.5f, out hit)) {
5           if (hit.distance < obstacleRange) {
                    // Must've hit wall or other object, turn
6                   Move (turn, 0.1f);
7                   if (Mathf.Sign(turn) == 1) {
                        // +ve turn, keep turning right
8                       turn = Random.Range (0.05f, 1.0f);
                    } else {
                        // -ve turn, keep turning left
9                       turn = Random.Range (-1.0f, -0.05f);
                    }
            } else {
10              float forward = Random.Range (0.05f, 1.0f);
                // assign new random +ve/-ve turn value
11              turn = Random.Range (-1.0f, 1.0f);
12              Move (0.0f, forward);
            }
        }
```

Figure 13: The Update() method of WanderingIguana

Explanation of Code:

Line 1: This will set the speed of the Animation. Since this is a public variable, the value can be changed right in Unity. This will make the Iguana move faster since we are using root motion. Note that you can change the Animation speed for the Enemy from Tasks 1 and 2 but it won't change how quickly it moves because in that case, we are using the Transform component not the Animator to control movement. Still, you might want to change it to better reflect how fast the Enemy moves.

Line 2 + 3: Generate a ray from the Iguana's front position. You did the exact same thing with the Enemy. `hit` will store information on what the ray ends up hitting.

Line 4: Cast a sphere along the ray to detect other objects in the Scene just like what you did with the Enemy.

Line 5: If the Iguana detects an Object ahead, it will need to turn.

Line 6: This is where the blending happens. The Iguana will turn and move forward at the same time. This causes the *WalkForward* and either the *WalkLeft* or *WalkRight* Animations to be blended. The Forward value is set at `0.1f` to ensure a small turn radius.

Line 7: Detect whether `turn` is currently positive (which according to our Blend Tree means a turn to the right). `Mathf.Sign()` is a function that returns a 1 if the float parameter is 0 or above.

Line 8: Keep turning the Iguana to the right. To do so, generate a random value between 0.05 and 1. If you look back at the Blend Tree, this will make sense! The 0.05 (instead of 0) is to prevent the Animator SM from going back to Idle (see the Conditions in the Blend Tree).

Line 9: In this case, the Iguana was previously turning left so keep it turning left until the Obstacle is out of the way. This time, the random value is from -1 to -0.05.

Line 10: Generate a random number for the forward movement. The minimum value of 0.05 ensures the Iguana doesn't go back into the Idle state.

Line 11: In this case, the Iguana doesn't need to turn but we assign a new random number to the `turn` variable for the case when it will need to turn in the future (either to the left or right).

Line 12: Move the Iguana forward. No Animation blending happens here because it is only walking forward.

13. Try playing the Scene. Watch how the Iguana's Animation seamlessly blends for walking turns!

Tasks Part 4 – Adding the Model from Task 3 to the Scene

The Model from Task 3 won't be an enemy target. It will act as some extra eye candy for the Scene. That being said, you should add several of them to your Scene programmatically like you did for the Enemy. Here are some hints:

- All you need to do is modify *SceneController.cs*.
- You'll need a `SerializeField` variable to hold a reference to the Iguana Prefab and will need to make the connection in Unity.
- You'll need an array to hold each of the copies of the Prefab.
- Insert **5-10** instances of the Model in your Scene.
- Have them spawn somewhere in your Scene but make sure it isn't where the Enemy spawns, where you are, or where a wall is. Use the Iguana you placed in the Scene from Task 3 to figure out the coordinates of a good spawning place. Then remove it from the Scene.
- Since these won't be dying and respawning, fill the array in the `Start()` method (don't forget to instantiate the array). Instantiate each copy of the Prefab, place it in the Scene, and give it a random angle to start with. That's it!

Submission:

Demo your Scene to the Lab Instructor:

- **Show that you have applied walking and death animations to your Enemy.**
- **Show that you have utilized a blend tree for an additional model that you place in your Scene.**
- **Show that 5-10 copies of the additional non-target model from Tasks 3 and 4 are in the Scene.**

Project Idea:

In this lab, you were only required to implement the blend tree of the Iguana. There were a bunch of other states that weren't used. For part of your project mark, you could implement the rest of those states. In that case, the Iguana would be an additional enemy in the Scene.