Tutorials

Before programming anything in Unity, make sure to read "50 Tips for Working with Unity (Best Practices)": http://devmag.org.za/2012/07/12/50-tips-for-working-with-unity-best-practices/

For more information about the underlying mechanics of ADAPT, refer to the ADAPT I3D paper at http://www.seas.upenn.edu/~shoulson/papers/i3d2012.pdf and to articles with the ADAPT tag on my blog: http://blog.ashoulson.com/search/label/ADAPT

Note that these tutorials were written for Unity 3.5, but they should work with Unity 4 and up. Your screen might differ slightly from the interface screenshots.

Tutorial 1: Procedural Animation Blending with Shadows

We will begin by creating a simple character controller and learning how to blend it with one of the included ADAPT shadow controllers. The empty template for this tutorial can be found in the scene `Tutorial1Empty`. Open this file, which begins with a plain scene and nearly empty ADAPTMan character (aside from an animation component).

Creating a "Lean" Controller

Let's create a simple controller that can make the character lean forward or backward. Create a new Unity MonoBehavior called "LeanController". The empty code template should look like this (with some minor formatting differences):

```
using UnityEngine;
using System.Collections;

public class LeanController : MonoBehaviour
{
	// Use this for initialization
	void Start ()
	{

	}

	// Update is called once per frame
	void Update ()
	{

	}
}
```

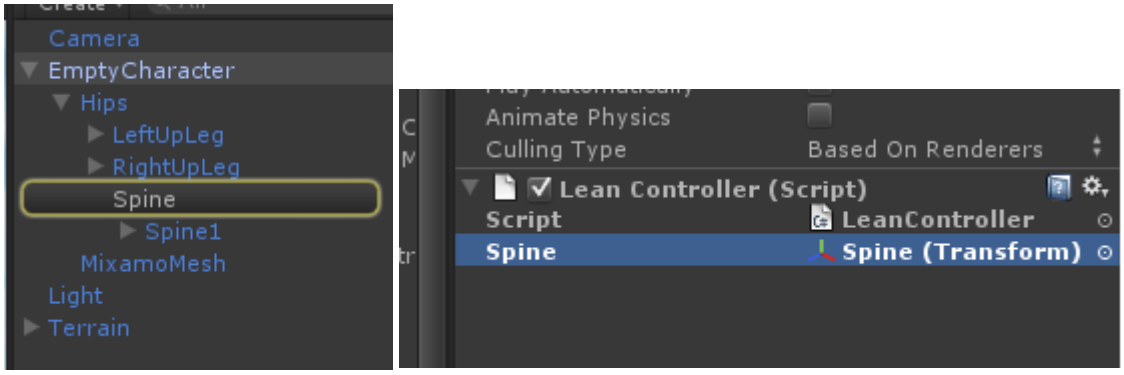
Let's add a parameter called "spine", which will tell the controller where we want the character to bend when we lean forward or backward. We'll assign a value to this field in the inspector in a little bit.

```
public class LeanController : MonoBehaviour
{

	public Transform spine;

	// Use this for initialization
	void Start ()
	{

	}

	// Update is called once per frame
	void Update ()
	{

	}
}
```

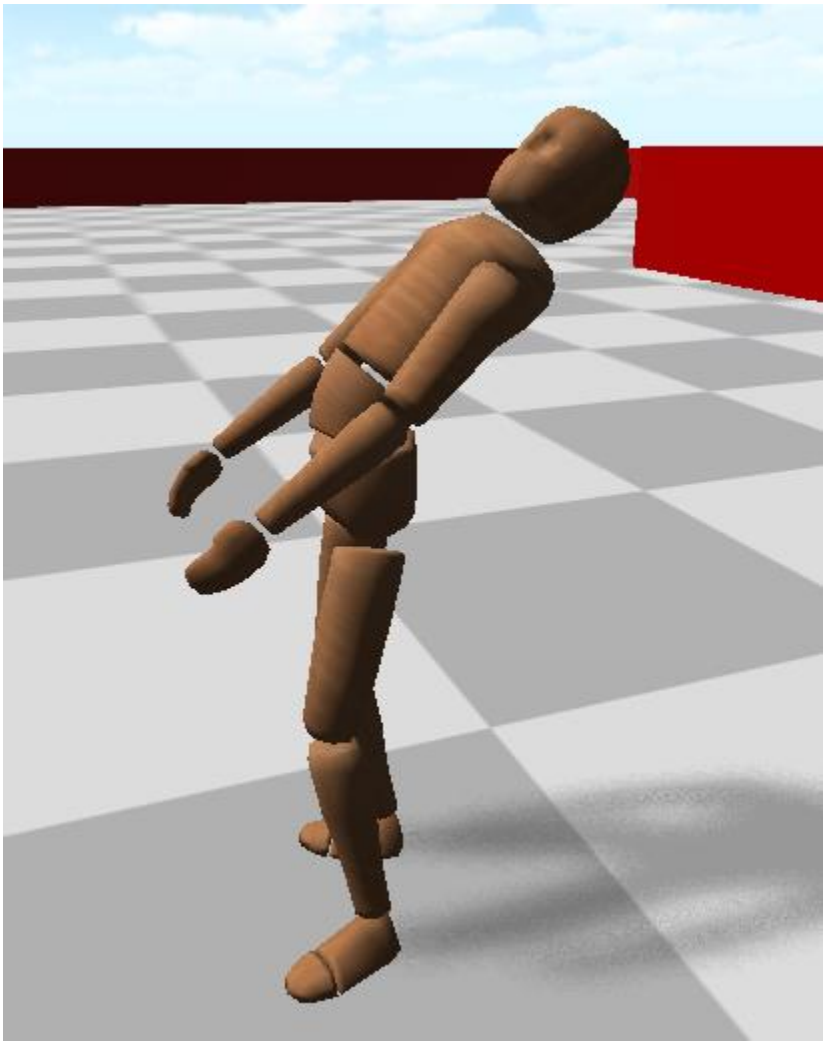
Next, in the update function, we want to rotate that spine bone. We'll use "R" to lean backwards, and "F" to lean forwards. All we're doing is rotating the spine bone along its local x axis when we press either one of these keys.

```
public class LeanController : MonoBehaviour
{

	public Transform spine;

	// Use this for initialization
	void Start ()
	{

	}

	void Update ()
	{
		// Get the current euler angle rotation
		Vector3 rot = spine.rotation.eulerAngles;

		// Detect key input and add or subtract from the x rotation (scaling
		// by deltaTime to make this speed independent from the frame rate)
		if (Input.GetKey(KeyCode.R))
			rot.x -= Time.deltaTime * 50.0f;
		if (Input.GetKey(KeyCode.F))
			rot.x += Time.deltaTime * 50.0f;

		// Apply the new rotation
		spine.rotation = Quaternion.Euler(rot);
	}
}
```

Attach this LeanController script to the EmptyCharacter game object, and drag the "Spine" bone to the spine parameter in the inspector.



Now run the simulation. You can control the camera with the WASD keys, and look around by holding down the right mouse button. Try pressing the R and F keys to make the character lean. (If you'd like, try imposing a maximum and minimum angle for how far the character can lean, but keep in mind that angles wrap around from 359 degrees to zero!)



Making the Controller Work with ADAPT

Now we have a basic character controller in Unity, but this controller currently has nothing to do with ADAPT. We're going to convert it so it can work with the ADAPT system. We need to change a few lines of code:

```
public class LeanController : MonoBehaviour
```
becomes:
```
public class ShadowLeanController : ShadowController
```

```
void Start()
```
becomes:
```
public override void ControlledStart()
```

```
void Update()
```
becomes:
```
public override void ControlledUpdate()
```

Also, change the filename from "LeanController" to "ShadowLeanController".

We're changing the name and inheritance of the class. We're also changing the Start and Update functions from the default functions that Unity automatically calls, to special functions that we will call manually ourselves instead within ADAPT. So our class looks like this:

```
public class ShadowLeanController : ShadowController
{
    public Transform spine;

    public override void ControlledStart()
    {

    }

    public override void ControlledUpdate()
    {
        // Get the current euler angle rotation
        Vector3 rot = spine.rotation.eulerAngles;

        // Detect key input and add or subtract from the x rotation (scaling
        // by deltaTime to make this speed independent from the frame rate)
        if (Input.GetKey(KeyCode.R))
            rot.x -= Time.deltaTime * 50.0f;
        if (Input.GetKey(KeyCode.F))
            rot.x += Time.deltaTime * 50.0f;

        // Apply the new rotation
        spine.rotation = Quaternion.Euler(rot);
    }
}
```

We need to do one more thing. Remember that `spine` variable we were given in the inspector? Now, a `ShadowController` is given its own shadow to manipulate instead of editing the bones in the displayed character model. When we give the `spine` bone to this component in the inspector, we're tying the component to a bone in the display model, which we don't want to change directly. Instead,

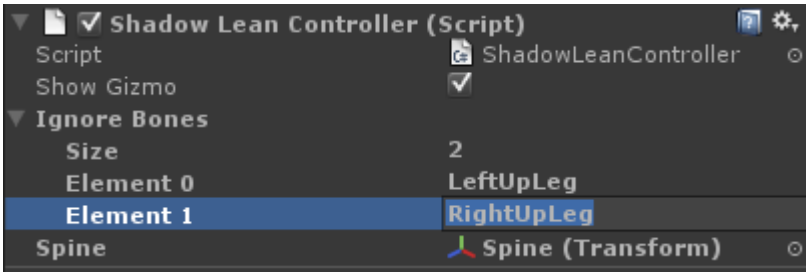
we want to edit the corresponding bone in our shadow. So as soon as we start, we need to take the bone we're given from the display model, and find the corresponding bone in the shadow that's been automatically generated for our `ShadowLeanController` to edit. (You can access that shadow using either the `shadow` or `transform` fields inherited from `ShadowController`. Both of them point to the same thing.) We're going to add this line to our `ControlledStart` function to do that remapping:

```
    public override void ControlledStart()
    {
        // Find the cloned version of the bone we were given in the inspector
        // so that we're editing our own shadow, not the display model
        this.spine = this.shadow.GetBone(this.spine);
    }
```

This line tells our shadow to find the bone that was cloned to correspond the given bone in the display model. We overwrite our spine transform variable so that we'll only ever edit the cloned spine bone instead of the original. Our final class looks like this:

```
using UnityEngine;
using System.Collections;

public class ShadowLeanController : ShadowController
{
    public Transform spine;

    public override void ControlledStart()
    {
        // Find the cloned version of the bone we were given in the inspector
        // so that we're editing our own shadow, not the display model
        this.spine = this.shadow.GetBone(this.spine);
    }

    public override void ControlledUpdate()
    {
        // Get the current euler angle rotation
        Vector3 rot = spine.rotation.eulerAngles;

        // Detect key input and add or subtract from the x rotation (scaling
        // by deltaTime to make this speed independent from the frame rate)
        if (Input.GetKey(KeyCode.R))
            rot.x -= Time.deltaTime * 50.0f;
        if (Input.GetKey(KeyCode.F))
            rot.x += Time.deltaTime * 50.0f;

        // Apply the new rotation
        spine.rotation = Quaternion.Euler(rot);
    }
}
```

Replace the `LeanController` with the `ShadowLeanController` on the character model and make sure it has a reference to the Spine bone like before. Also, we want to tell the class that this controller only affects the upper body, and to not clone the legs for the shadow. We can set this in the inspector:



Every `ShadowController` has its own Ignore Bones array, so add the names `LeftUpLeg` and `RightUpLeg`. This tells ADAPT not to clone the left leg or right leg bones (and their children) when it's creating the shadow for this `ShadowLeanController`. Since we won't ever write to the legs (we're only changing one spine bone), it's more efficient to make the skeleton as reduced as possible. We could cut off more bones and save even more space, but this is enough for now.
If we run the simulation now, the character won't do anything. This is because nothing is calling the `ControlledUpdate` function, and nothing is applying the skeleton from the shadow to the display model. In order to fix that, we need to create a Coordinator. Let's start with a simple empty class that inherits from `ShadowCoordinator`:

```
using UnityEngine;
using System.Collections;

public class TutorialCoordinator : ShadowCoordinator
{
    void Start()
    {

    }

    void Update()
    {

    }
}
```

Note that we're using the default `Start` and `Update` functions here because we want Unity to automatically update this class, and then this class will update each of the controllers. Now, the `ShadowCoordinator` class has a few built-in functions that we need to call at certain points. First, the `ControlledStartAll()` function automatically calls the `ControlledStart()` function of every attached `ShadowController`. You want to make sure to call this function at the end of your coordinator's `Start()` function. Additionally, the function `UpdateCoordinates()` moves the root coordinates (the hips) of each shadow to match that of the display model. This ensures that the shadows and the display model are all in the same place in the world. You usually want to make sure to call this in the beginning of your coordinator's `Update()` function. Here's the class with the recommended default function calls.

```
public class TutorialCoordinator : ShadowCoordinator
{
    void Start()
    {
        // Call each ShadowController's ControlledStart() function
```

```
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();
    }
}
```

Now we need two member fields in our class. The first is a reference to the ShadowLeanController, which we'll set automatically in the Start() function. The second is a ShadowTransform buffer. This is used for storing the position of each bone in a shadow, so we can pass shadow information between ShadowControllers and apply shadows to the display model. We could make new buffers to store these transforms on the fly when we need them, but it's more efficient to pre-allocate the memory for a buffer and just overwrite it each frame.

```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowLeanController lean = null;

    void Start()
    {
        // Allocate space for a buffer for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();
    }
}
```

Now, let's set this coordinator up to update the `ShadowLeanController` and apply the shadow it generates to the display model. We're going to add the following two lines to the `Update()` function under the `UpdateCoordinates()` call:

```
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);
```

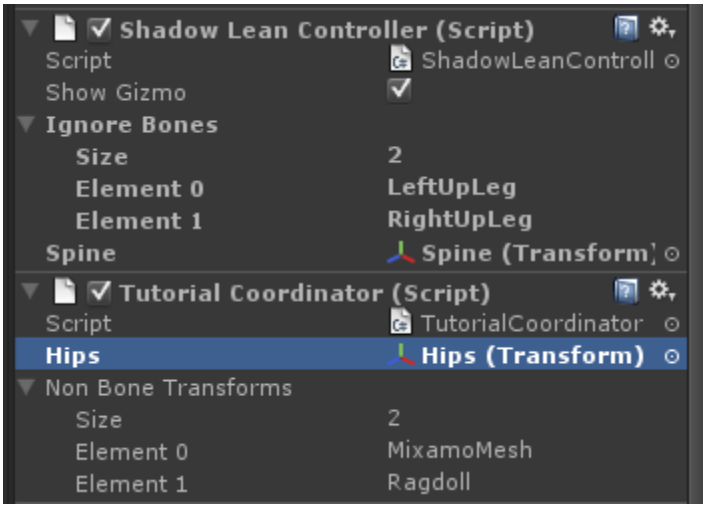
These lines will update the lean controller and write the pose for the shadow into the buffer. Then we need to apply this written pose to the display model. We can do that with this function call:

```
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
```

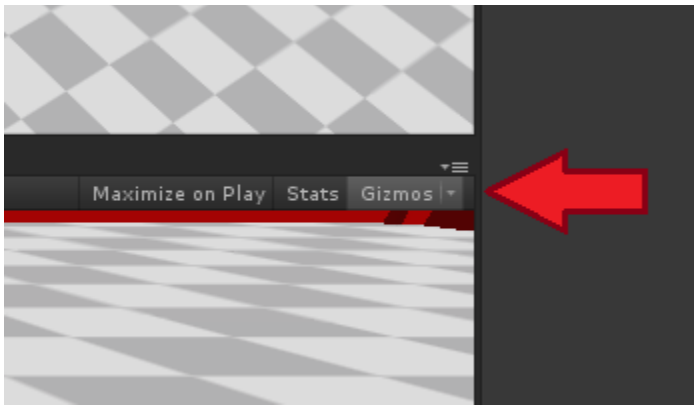
This writes the buffer to the skeleton of the display model, starting at the hips (from `transform.GetChild(0)`). So the final class should look like this:

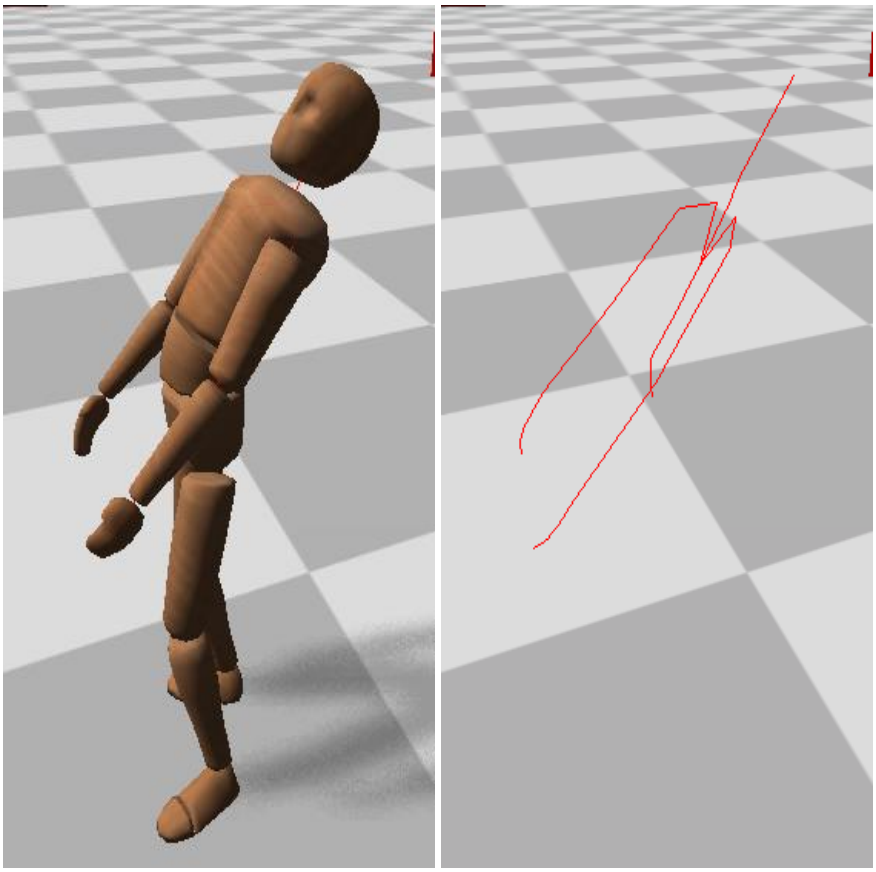
```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowLeanController lean = null;

    void Start()
    {
        // Allocate space for a buffer for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
}
```

Make sure both this coordinator and the `ShadowLeanController` are attached to the character, and give the coordinator a reference to the character's hips. Then run the simulation.



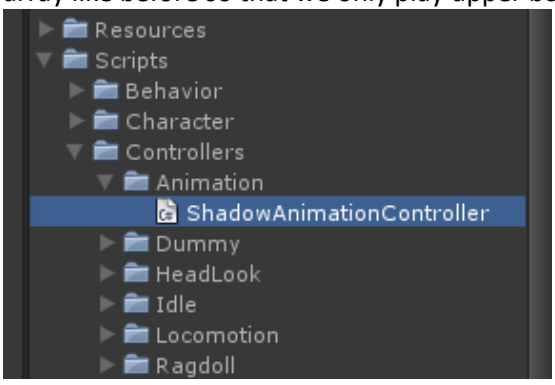
We should be able to lean the character again with R and F. If you enable Gizmos, and press E, you should be able to see the skeleton of the underlying shadow for the lean controller as well.

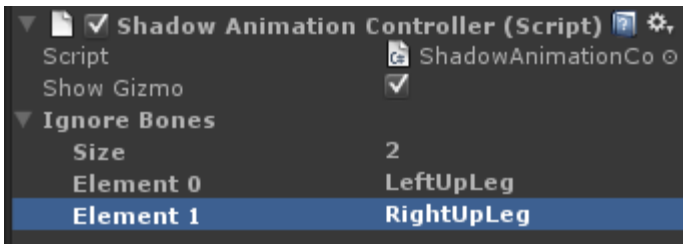




Adding a Second Controller

Now let's blend this controller with another controller from the ADAPT library. Add a `ShadowAnimationController` to the character. You can find it under `Scripts/Controllers/Animation`. Add `LeftUpLeg` and `RightUpLeg` to the Ignore Bones array like before so that we only play upper body animations.





Add a reference to the animation controller in the tutorial coordinator we've been writing.

```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowLeanController lean = null;
    protected ShadowAnimationController anim = null;

    void Start()
    {
        // Allocate space for a buffer for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Get a reference to our animation ShadowController
        this.anim = this.GetComponent<ShadowAnimationController>();

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        ...
    }
}
```

We also have to update the animation controller and encode its shadow to a second buffer. We'll add that second buffer and perform the update and encode steps like we did for the lean controller. Our class should now look like this:

```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowTransform[] buffer2 = null;
    protected ShadowLeanController lean = null;
    protected ShadowAnimationController anim = null;

    void Start()
    {
        // Allocate space for two buffers for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();
        this.buffer2 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Get a reference to our animation ShadowController
        this.anim = this.GetComponent<ShadowAnimationController>();

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
```

```
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);

        // Update the anim controller and write its shadow into the buffer
        this.anim.ControlledUpdate();
        this.anim.Encode(this.buffer2);

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
}
```

Blending Between Two Controllers

Note that we're not actually using that buffer yet. Now we want to be able to blend between the animation controller and our lean controller. First, let's discuss exactly what's happening here. At the beginning of the simulation, the animation controller and lean controller are both given their own clone of the display model's skeleton (we've removed the legs of both so these skeletons are just of the upper body). We need to make sure to manually update each of these ShadowControllers, then we want to extract their shadow skeletons' poses and blend them together. Once we've merged the skeletons, we want to apply the new merged skeleton to the display model so it takes on the combined pose. When we blend, we assign a weight to each skeleton we're blending. The higher the weight (in proportion to every other weight), the more the final skeleton will resemble that given input pose.

Let's start by adding the value we will use for the blend weight. This will be one float value that we'll just interpolate between 0.01 and 0.99. We'll also add code in our Update function to raise and lower this value with Y and H. Here's what our class looks like:

```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowTransform[] buffer2 = null;
    protected ShadowLeanController lean = null;
    protected ShadowAnimationController anim = null;
    protected float weight;

    void Start()
    {
        // Allocate space for two buffers for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();
        this.buffer2 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Get a reference to our animation ShadowController
        this.anim = this.GetComponent<ShadowAnimationController>();

        // Set the weight
        this.weight = 0.99f;
```

```
        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);

        // Update the anim controller and write its shadow into the buffer
        this.anim.ControlledUpdate();
        this.anim.Encode(this.buffer2);

        // Control the weight with the Y and H keys
        if (Input.GetKey(KeyCode.Y) == true)
            this.weight += 2.0f * Time.deltaTime;
        if (Input.GetKey(KeyCode.H) == true)
            this.weight -= 2.0f * Time.deltaTime;
        this.weight = Mathf.Clamp(this.weight, 0.01f, 0.99f);

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
}
```

Now to do the actual blend. This sounds complicated, but is only one function call. Add this right above the `Shadow.ReadShadowData()` function call:

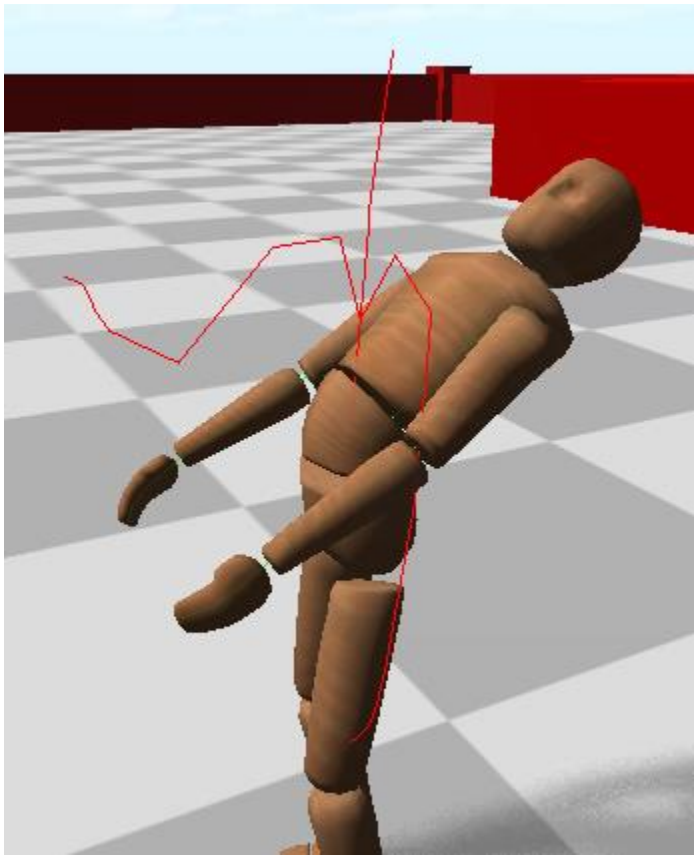
```
        BlendSystem.Blend(
            this.buffer1,
            new BlendPair(this.buffer1, this.weight),
            new BlendPair(this.buffer2, 1.0f - this.weight));
```

This takes the two buffers, blends them using the `weight` value and its inverse, and stores the result back in `buffer1` (this is why it's good to reuse buffers rather than creating new ones). Now, one last thing. Let's make the animation controller actually play an animation when we press the T key. Add these two lines of code right above the `BlendSystem.Blend()` call:

```
        if (Input.GetKeyDown(KeyCode.T) == true)
            this.anim.AnimPlay("dismissing_gesture");
```

Now try running the simulation and play with the R, F, T, Y, and H keys. You should be able to blend into the animation controller, play an animation, and then blend out back into the lean controller. Turn on Gizmos to see the two different skeletons working behind the scenes, too. If you want to keep track of the weight value, put this line of code somewhere in the `Update` function and the value will appear in the console window:

```
Debug.Log(weight);
```



Fixing Strange Behavior

Now, we've got some problems. First, the animation controller is rotating the hips, which moves the legs of the character when we only want to affect the upper body. Second, when we blend towards the animation controller from the lean controller, the character stands upright again. We want to be able to play animations while still leaning forward or backward, right? Fortunately, we can fix both of these problems with one solution using a `Whitelist` filter. When we call the `Encode()` function, we can give the function a filter to ignore bones in the shadow skeleton we're encoding. We can use two kinds of filters. Whitelists take only the given bones and their children, while blacklists take all bones except the given bones and their children. Since the lean controller bends at the Spine bone, let's have the animation controller cut off right above that spine bone and animate everything above it on the body. We'll create a whitelist starting at "`Spine1`" and use it in the encode function. This is very simple. Replace the following line of code:

```
this.anim.Encode(this.buffer2);
```

with this:

```
this.anim.Encode(this.buffer2, new Whitelist<string>("Spine1"));
```

The class should look like this now:

```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowTransform[] buffer2 = null;
    protected ShadowLeanController lean = null;
    protected ShadowAnimationController anim = null;
    protected float weight;

    void Start()
    {
        // Allocate space for two buffers for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();
        this.buffer2 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Get a reference to our animation ShadowController
        this.anim = this.GetComponent<ShadowAnimationController>();

        // Set the weight
        this.weight = 0.99f;

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);

        // Update the anim controller and write its shadow into the buffer
        this.anim.ControlledUpdate();
        this.anim.Encode(this.buffer2, new Whitelist<string>("Spine1"));

        // Control the weight with the Y and H keys
        if (Input.GetKey(KeyCode.Y))
            this.weight += 2.0f * Time.deltaTime;
        if (Input.GetKey(KeyCode.H))
            this.weight -= 2.0f * Time.deltaTime;
        this.weight = Mathf.Clamp(this.weight, 0.01f, 0.99f);

        // Optionally, uncomment this to see the weight value
        // Debug.Log(weight);

        // Play an animation when we press T
        if (Input.GetKeyDown(KeyCode.T) == true)
            this.anim.AnimPlay("dismissing_gesture");

        // Blend the two controllers using the weight value
        BlendSystem.Blend(
            this.buffer1,
            new BlendPair(this.buffer1, this.weight),
```

```
            new BlendPair(this.buffer2, 1.0f - this.weight));

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
}
```

Run the simulation again and hold H for a second to give the animation controller full blend. Now lean back with R and press T to play the animation. Much better, right? But there's still a little problem. Notice that when the animation ends, it leaves the character's arms and head in a slightly offset position. We want the character to return to a neutral pose when the animation is finished, so that we can replay the animation without snapping. If we blend out when the animation finishes, that pose offset will go away, but we have to blend back in to start the animation again. In short, we need to make blending more straightforward.

This is no problem when we use a `Slider`. A slider is a simple object that smoothly slides a float value between 0 and 1 across several frames. This saves us the trouble of having to manually raise and lower the weight values ourselves. We can tell a slider to go to its maximum or minimum value, and it will automatically do so after a second or two automatically. Let's replace the weight value with a weight slider, and change the Y and H keys to use it. Change the following lines:

```
protected float weight;
```

becomes

```
protected Slider weight;
```

```
this.weight = 0.99f;
```

becomes

```
this.weight = new Slider(4.0f);
```
(the 4.0f means that we want the slider to go four times as fast)

```
if (Input.GetKey(KeyCode.Y))
    this.weight += 2.0f * Time.deltaTime;
if (Input.GetKey(KeyCode.H))
    this.weight -= 2.0f * Time.deltaTime;
this.weight = Mathf.Clamp(this.weight, 0.01f, 0.99f);
```

becomes

```
if (Input.GetKeyDown(KeyCode.Y))
    this.weight.ToMax();
if (Input.GetKeyDown(KeyCode.H))
```

```
    this.weight.ToMin();
```

(the slider automatically handles clamping)

```
BlendSystem.Blend(
    this.buffer1,
    new BlendPair(this.buffer1, this.weight),
    new BlendPair(this.buffer2, 1.0f - this.weight));
```

becomes

```
BlendSystem.Blend(
    this.buffer1,
    new BlendPair(this.buffer1, this.weight.Value),
    new BlendPair(this.buffer2, this.weight.Inverse));
```

(Inverse gives 1.0 – the weight value)

Finally, we need to add this line to the top of the `Update` function:

```
this.weight.Tick(Time.deltaTime);
```

So that the weight slider knows how much to change each frame. The class should look like this now:

```
public class TutorialCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowTransform[] buffer2 = null;
    protected ShadowLeanController lean = null;
    protected ShadowAnimationController anim = null;
    protected Slider weight;

    void Start()
    {
        // Allocate space for two buffers for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();
        this.buffer2 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
        this.lean = this.GetComponent<ShadowLeanController>();

        // Get a reference to our animation ShadowController
        this.anim = this.GetComponent<ShadowAnimationController>();

        // Set the weight
        this.weight = new Slider(4.0f);

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
```

```
        this.weight.Tick(Time.deltaTime);

        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);

        // Update the anim controller and write its shadow into the buffer
        this.anim.ControlledUpdate();
        this.anim.Encode(this.buffer2, new Whitelist<string>("Spine1"));

        // Control the weight with the Y and H keys
        if (Input.GetKeyDown(KeyCode.Y))
            this.weight.ToMax();
        if (Input.GetKeyDown(KeyCode.H))
            this.weight.ToMin();

        // Optionally, uncomment this to see the weight value
        // Debug.Log(weight.Value);

        // Play an animation when we press T
        if (Input.GetKeyDown(KeyCode.T) == true)
            this.anim.AnimPlay("dismissing_gesture");

        // Blend the two controllers using the weight value
        BlendSystem.Blend(
            this.buffer1,
            new BlendPair(this.buffer1, this.weight.Value),
            new BlendPair(this.buffer2, this.weight.Inverse));

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
}
```

Try running the simulation. The only difference you'll notice right now is that you only have to press Y and H rather than holding it to fully raise and lower the blend weights. It will continue blending even after you release the key until it's either at 0.01f or 0.99f. Lean the character back a bit with R, then press T and H at the same time to blend in and start the animation simultaneously. When the animation finishes, press Y to blend back to a neutral pose. You're manually doing what we want to automate. Let's make this process fully automatic now and wrap everything up.

Simplifying the Interface

What we're going to do is remove the Y and H keys. Instead of relying on those keys, we'll have the animation controller blend in automatically when the animation begins, and blend out when the animation finishes. Begin by removing the if statements corresponding to the Y and H keys. Replace the if statement corresponding to the T key with this:

```
        if (Input.GetKeyDown(KeyCode.T) == true)
        {
            this.anim.AnimPlay("dismissing_gesture");
            this.weight.ToMax();
        }
```

Finally, we want to check and see if the animation is done playing. If so, fade out the animation controller. So add this if statement right under the last one:

```
        if (anim.IsPlaying() == false)
            this.weight.ToMin();
```

The final `Update` function should look like this:

```
    void Update()
    {
        this.weight.Tick(Time.deltaTime);

        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.lean.ControlledUpdate();
        this.lean.Encode(this.buffer1);

        // Update the anim controller and write its shadow into the buffer
        this.anim.ControlledUpdate();
        this.anim.Encode(this.buffer2, new Whitelist<string>("Spine1"));

        // Optionally, uncomment this to see the weight value
        // Debug.Log(weight);

        // Play an animation when we press T
        if (Input.GetKeyDown(KeyCode.T) == true)
        {
            this.anim.AnimPlay("dismissing_gesture");
            this.weight.ToMax();
        }

        // Fade out the animation controller if we're finished
        if (anim.IsPlaying() == false)
            this.weight.ToMin();

        // Blend the two controllers using the weight value
        BlendSystem.Blend(
            this.buffer1,
            new BlendPair(this.buffer1, this.weight.Value),
            new BlendPair(this.buffer2, this.weight.Inverse));

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
```

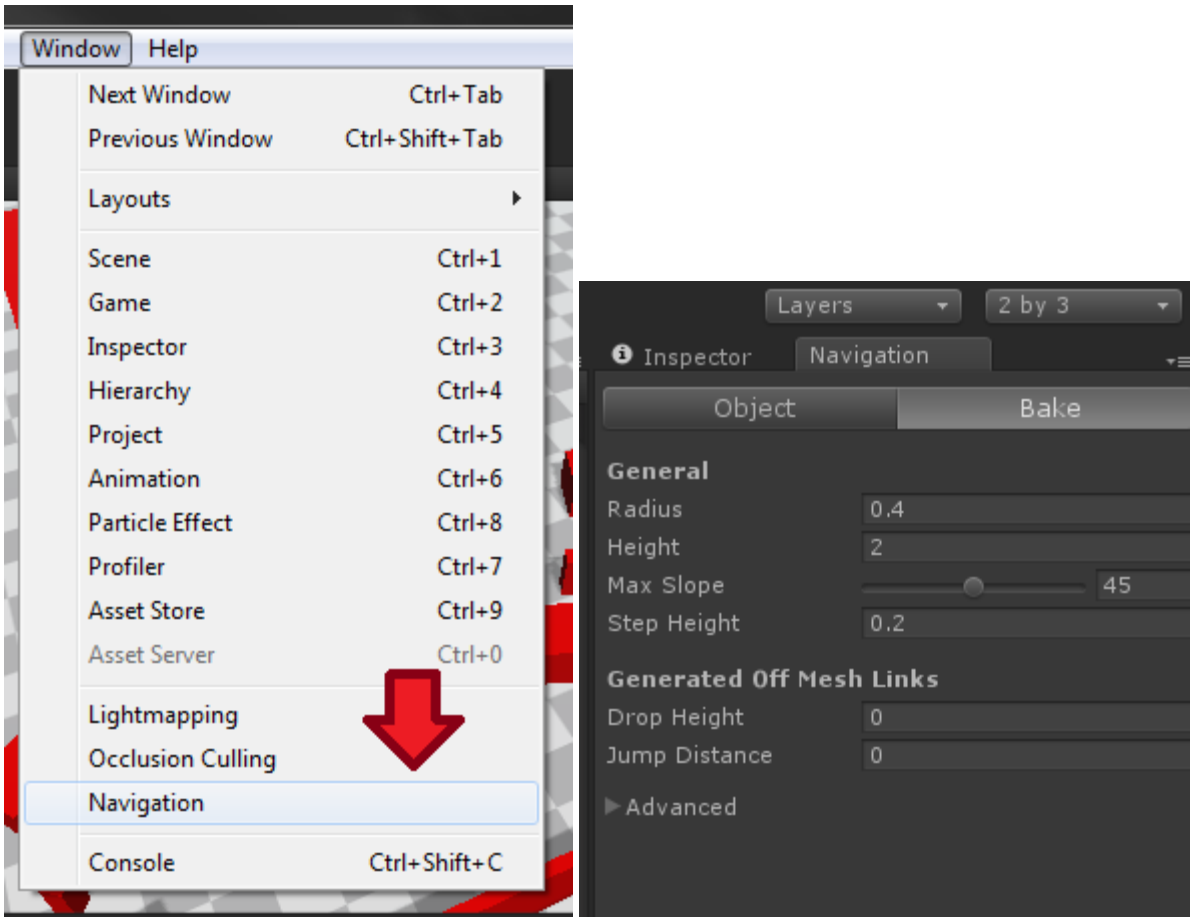
Now you should be able to press T and have the animation controller blend in and out automatically. This should look much better, without any stuttering artifacts or awkward poses. Congratulations, you've successfully created your own character controller and integrated it with a built-in ADAPT controller. In practice, fully-articulated characters are more complex, with multiple controllers like these two working in unison. To see a more complex system with several controllers being blended, look at the `BodyCoordinator` class. The principles here are the same, just applied multiple times to blend at different layers and for different parts of the body.

Tutorial 2a: The Navigation System (Unity)

Now we're going to make the character walk around in the environment. The empty template for this second tutorial can be found in `Tutorial2Empty`. Note that this tutorial uses the Unity navigation system, but the Recast navigation system is also available and sometimes produces better results. Refer to tutorial 2b for that technique.

Building a Navigation Mesh

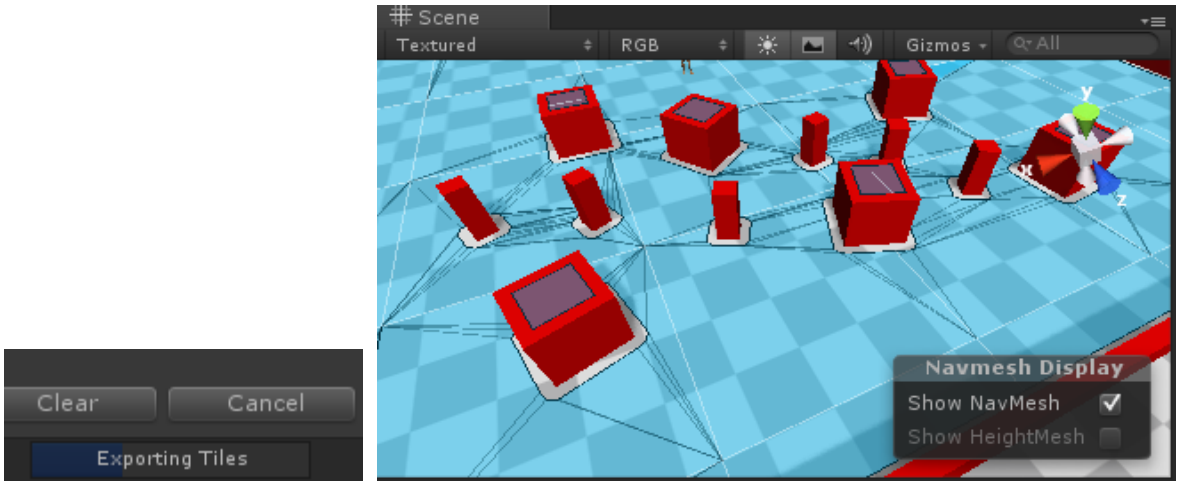
The first thing we need to do, once our environment is built (or in this case, provided to you by the tutorial) is to build the navigation mesh. The navigation mesh (navmesh) is a piece of geometry that contains the information for where the character can walk in the environment. Unity provides the functionality for automatically generating this geometry. You can find it in the Window menu under Navigation. Open this up and you'll see a new tab on your inspector called Navigation. Select that tab, and go to the Bake screen.



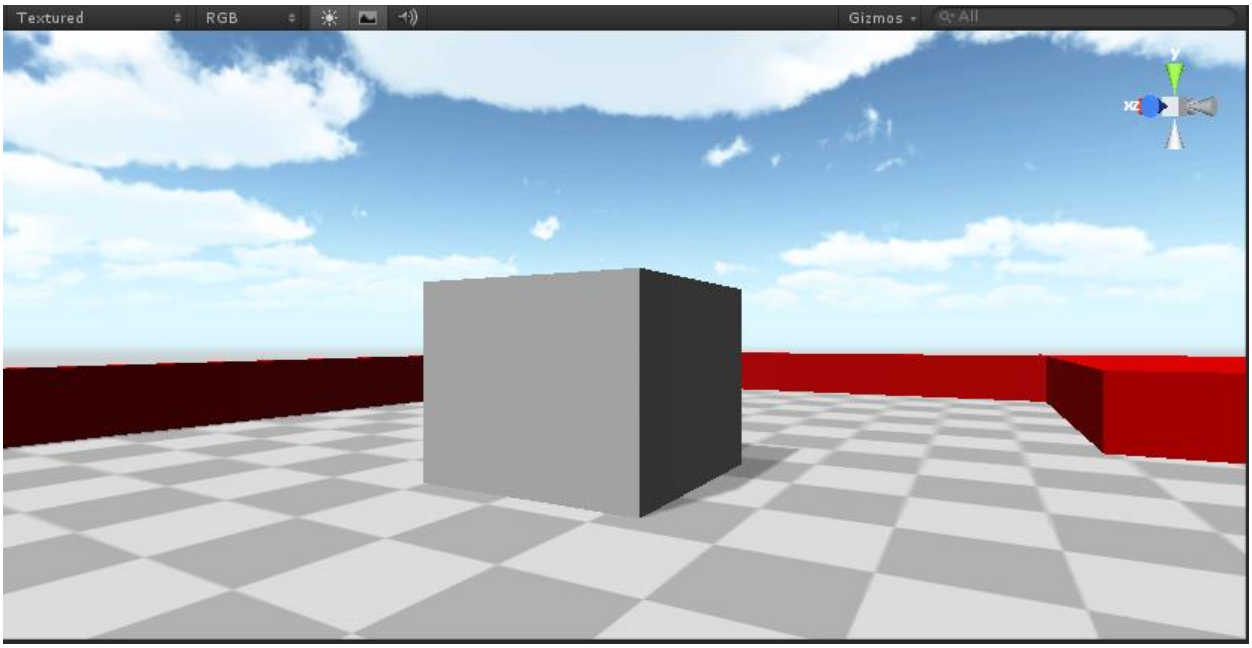
This screen has a few options for tweaking the navigation mesh, which we'll quickly go over:

Radius: The maximum radius of any agent we'll have in the world. For the purpose of navigation, each agent is treated like a cylinder.
Height: The minimum height of any agent.
Max Slope: The maximum incline (an angle, in degrees) that any agent can climb
Max Slope: The maximum step height that any agent can climb

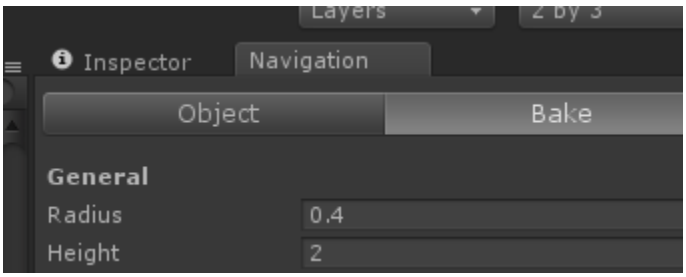
Ignore the others for now. Off-mesh links deal with generating gaps between areas in the navmesh, and the advanced options deal with the detail and fidelity of the navmesh when it's being generated (sacrificing generation speed for quality). Click Clear, and then Bake to create a new navmesh from scratch (this can take a few minutes). You'll see a loading bar at the bottom, and then a blue overlay in the scene view displaying the navmesh Unity generated.

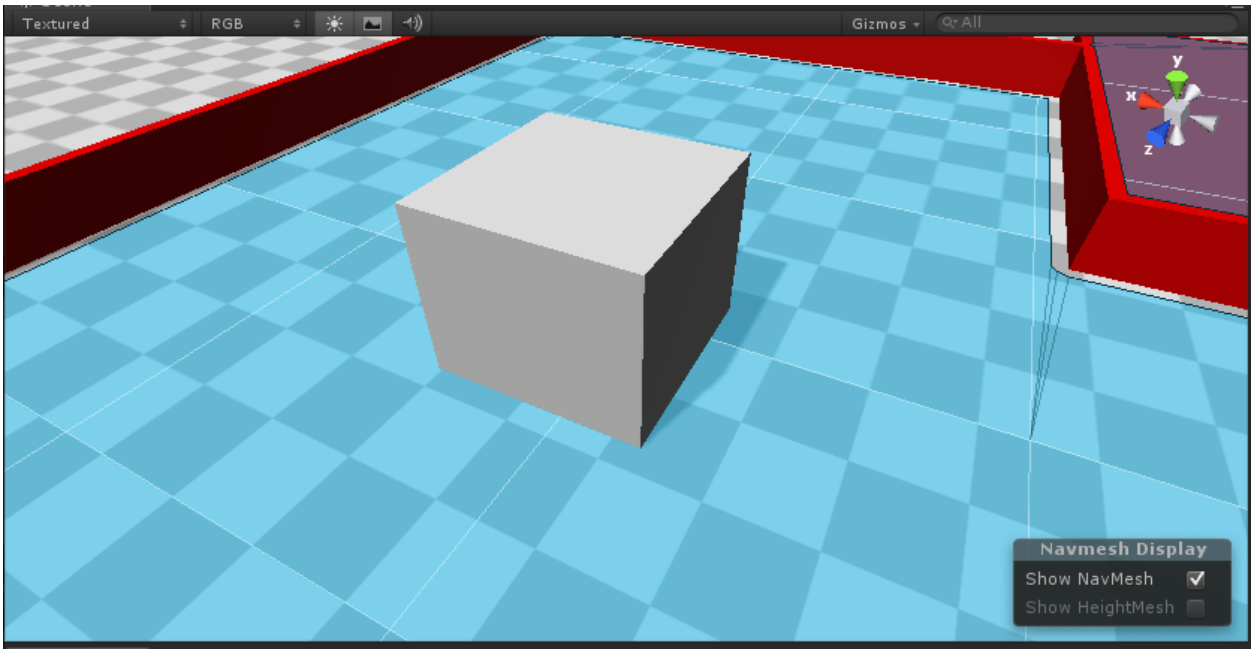


Let's quickly add one more piece of geometry to the environment. Create a new cube gameobject, then scale it and place it somewhere in the scene.

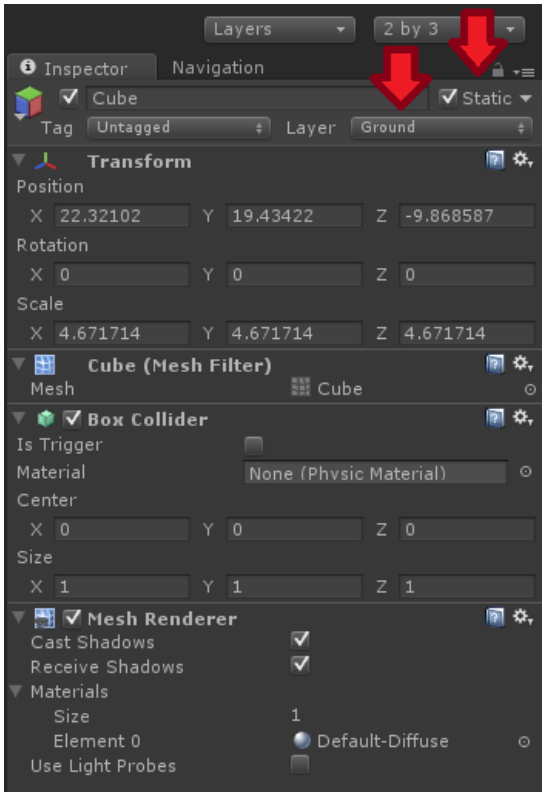


Notice that our navmesh seems to ignore it (if your navmesh isn't visible, go back to the Navigation tab and make sure that that's active):

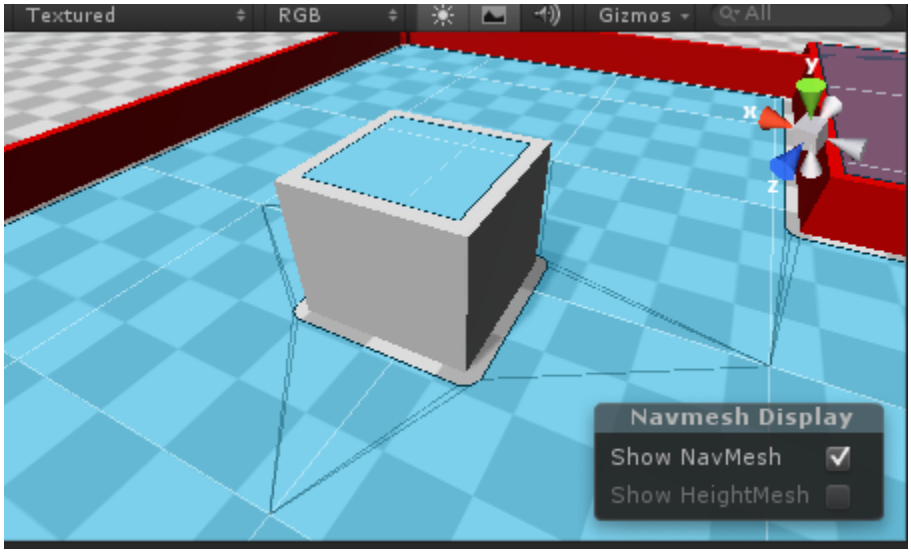




We'll need to generate a new navmesh, but we need to make sure of one thing first. In order for an object to be recognized by the navmesh generator, it needs to be marked as static in Unity. Also, be sure to set its layer to "Ground" (this doesn't affect the navmesh, but other systems in ADAPT will behave better if the ground is annotated). You can do that in the inspector here:



Let's generate the navmesh again like we did before. Make sure that the cube you created is flush with the terrain (or intersects with it a little bit), and that it isn't high enough for an agent to step over.



So, we've generated a navmesh. Let's get an agent to use it. Note that we're going to do this the basic way to show you the underlying mechanics, but Tutorials 3 and 4 talk about much better and more straightforward ways to use the navigation interface (and other components) in a real project.

Creating a Waypoint

We're going to start by making a basic Waypoint object that will tell an agent to approach it when we press a key. Make a new C# file and call it TutorialWaypoint. It should look like this (with some minor formatting differences):

```
using UnityEngine;
using System.Collections;

public class TutorialWaypoint : MonoBehaviour
{
    // Use this for initialization
    void Start ()
    {

    }

    // Update is called once per frame
    void Update ()
    {

    }
}
```

The waypoint needs to know two things. First, we want to be able to configure which key it responds to, so we can have multiple waypoints. Second, it needs to know which character it will be controlling. Let's add these two fields to the top of the class above the `Start` function.

```
    public KeyCode code;
    public UnitySteeringController controller;
```

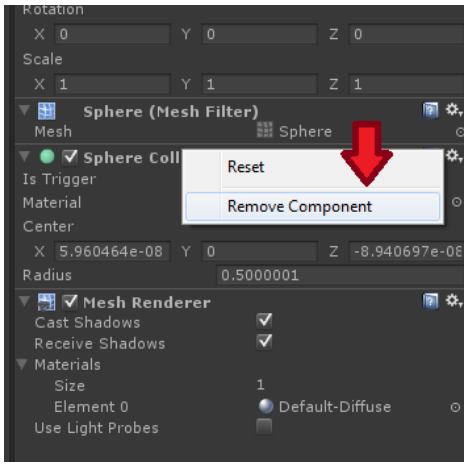
And then the rest is simple. We listen for the key, and if it's pressed, we set the character's navigation target. Just add this to `Update` function:

```
        if (Input.GetKeyDown(this.code) == true)
            this.controller.Target = transform.position;
```

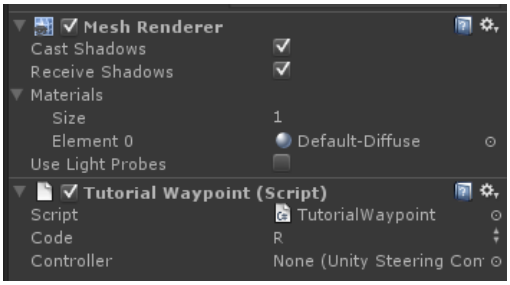
Here's what the final class should look like (you can remove the `Start` function):

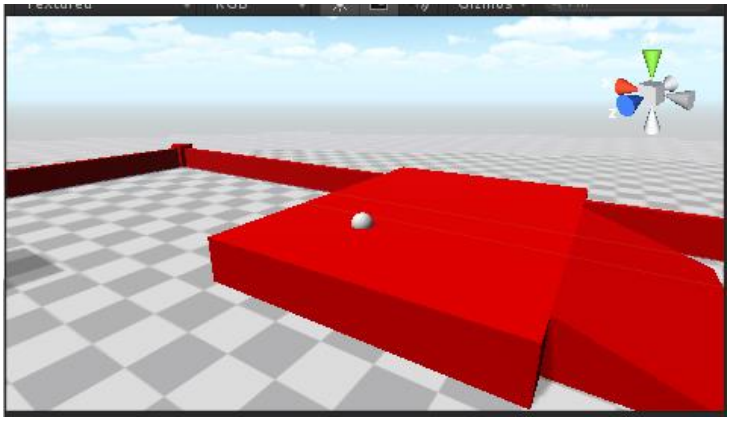
```
public class TutorialWaypoint : MonoBehaviour
{
    public KeyCode code;
    public UnitySteeringController controller;

    void Update ()
    {
        // Listen for the key and set the character's destination
        if (Input.GetKeyDown(this.code) == true)
            this.controller.Target = transform.position;
    }
}
```

Now create a new sphere game object and remove the sphere collider from it.



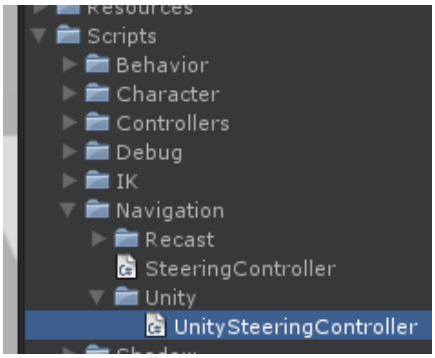
Now attach our new waypoint class to the sphere and place the sphere somewhere in the world. Make sure that's close to the floor and is somewhere the character could reach. You'll also need to set a keycode for it. I chose the R key and placed the sphere on the big red platform. It's probably best to make the sphere go through the floor somewhat.

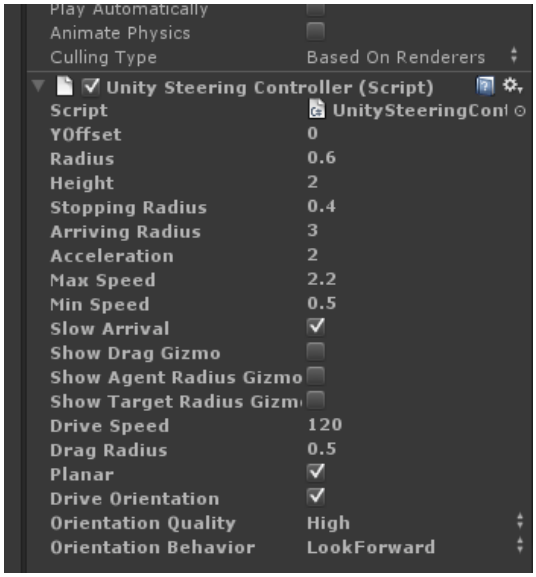




Using the Steering Component

Now let's get the character moving. In order to do so, it needs a UnitySteeringController attachment. Let's drag that onto the character now. You can find it under Scripts/Navigation/Unity.

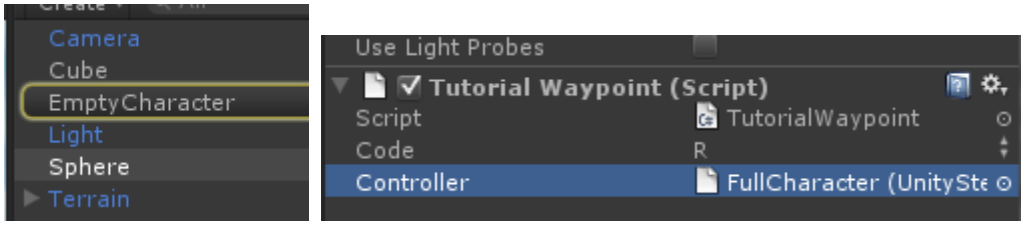




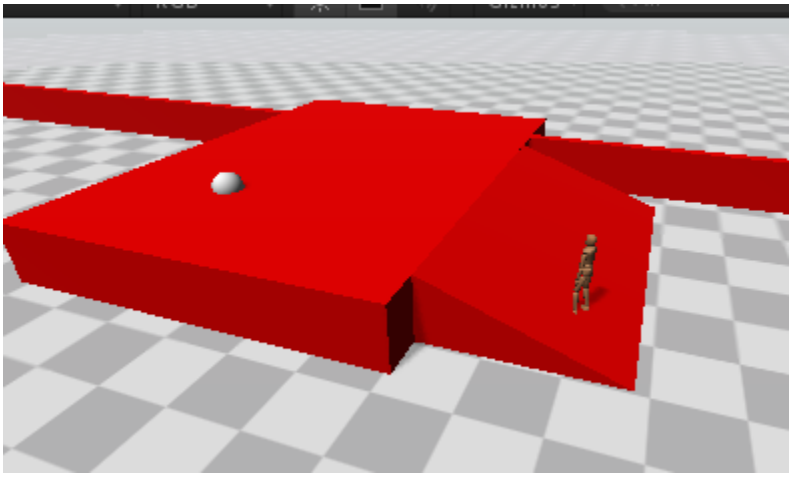
Hopefully most of these parameters are self-explanatory, but let's quickly go over some of the more obscure ones (and ignore the rest for now):

Stopping Radius: When a character is within this radius from the target, it will completely stop.
Arriving Radius: The radius at which the character will begin to slow down before stopping (this is added to the stopping radius). This feature is toggled with the **Slow Arrival** parameter.
Drive Speed: The speed at which we will turn to reorient while walking/running.
Drive Orientation: Turns on or off the ability to turn to face a desired orientation.
Orientation Behavior: Currently we have three options to compute orientation while walking/running. Setting this to "None" allows an external object to set the desired orientation, while any other setting means that the controller will set a desired orientation internally to either look forward or look at the objective.

Now that we've attached a steering controller to the character, let's store a reference to the character in the waypoint we've created. Drag the character into the waypoint's controller field in the inspector.



Now run the simulation and press R (or whatever key you used). The character should glide to the destination and avoid obstacles in the terrain.



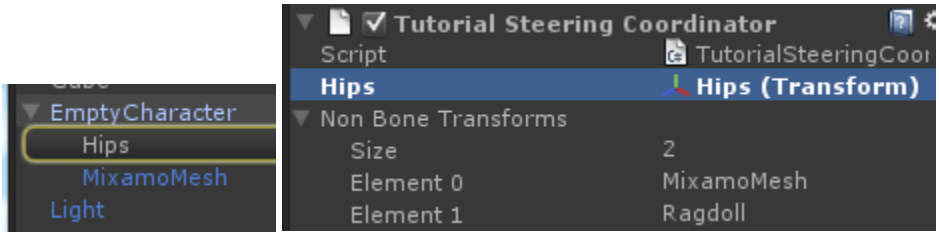
Getting the Character to Walk

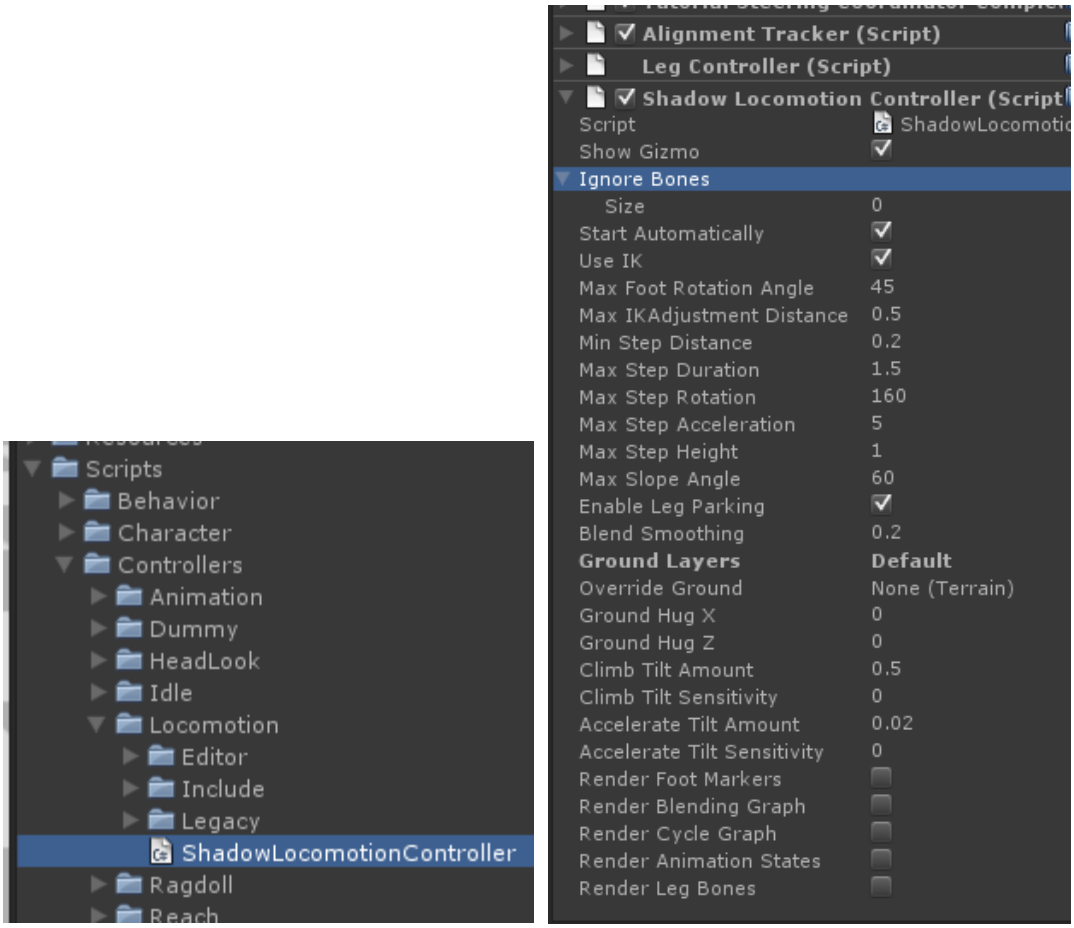
Obviously, our job is not done. Surely our character can't be expected to just levitate everywhere. We need to get the character to walk. For this, we're going to use the `ShadowLocomotionController` and a very simple coordinator based on the one from the first tutorial. Let's start with the coordinator. Create an empty C# script called `TutorialSteeringCoordinator`, and use the following code for it:

```
using UnityEngine;
using System.Collections;

public class TutorialSteeringCoordinator : ShadowCoordinator
{
    protected ShadowTransform[] buffer1 = null;
    protected ShadowLocomotionController loco = null;

    void Start()
    {
        // Allocate space for a buffer for storing and passing shadow poses
        this.buffer1 = this.NewTransformArray();

        // Get a reference to our lean ShadowController
```

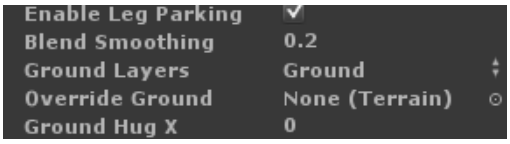
```
        this.loco = this.GetComponent<ShadowLocomotionController>();

        // Call each ShadowController's ControlledStart() function
        this.ControlledStartAll();
    }

    void Update()
    {
        // Move the root position of each shadow to match the display model
        this.UpdateCoordinates();

        // Update the lean controller and write its shadow into the buffer
        this.loco.ControlledUpdate();
        this.loco.Encode(this.buffer1);

        // Write the shadow buffer to the display model, starting at the hips
        Shadow.ReadShadowData(
            this.buffer1,
            this.transform.GetChild(0),
            this);
    }
}
```

This is almost identical to first coordinator we made in the first tutorial. Attach the coordinator to the character along with the ShadowLocomotionController, which you can find under Scripts/Controllers/Locomotion. Give the steering coordinator a reference to the hips of the character.Adding this class will automatically add a few others that we need.

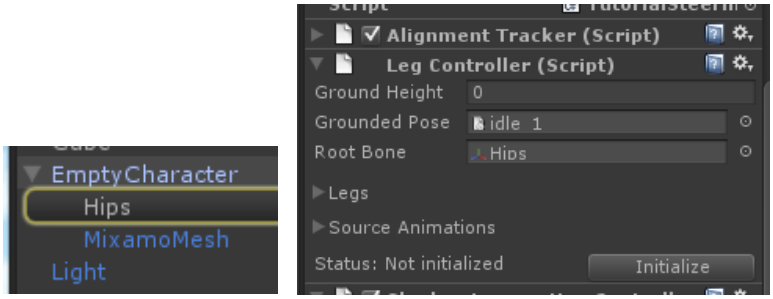




We'll need to configure these components. First, change "Ground Layers" to be "Ground", rather than "Default".

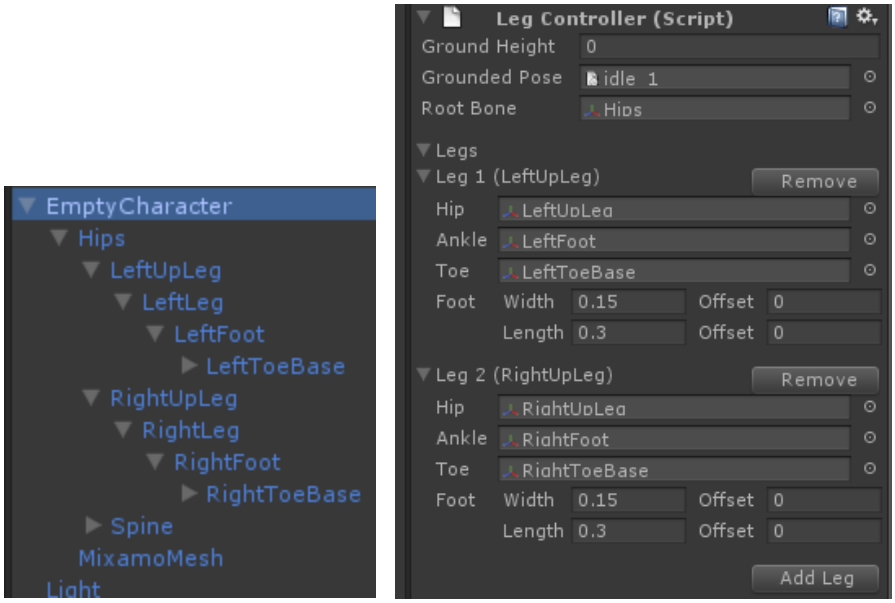


Next, we'll need to tell the leg controller some information about the character. Expand the leg controller component. Set "Grounded Pose" to `idle_1` (use the little dotted circle button to make this easier), and "Root Bone" to the character's `hips` bone.

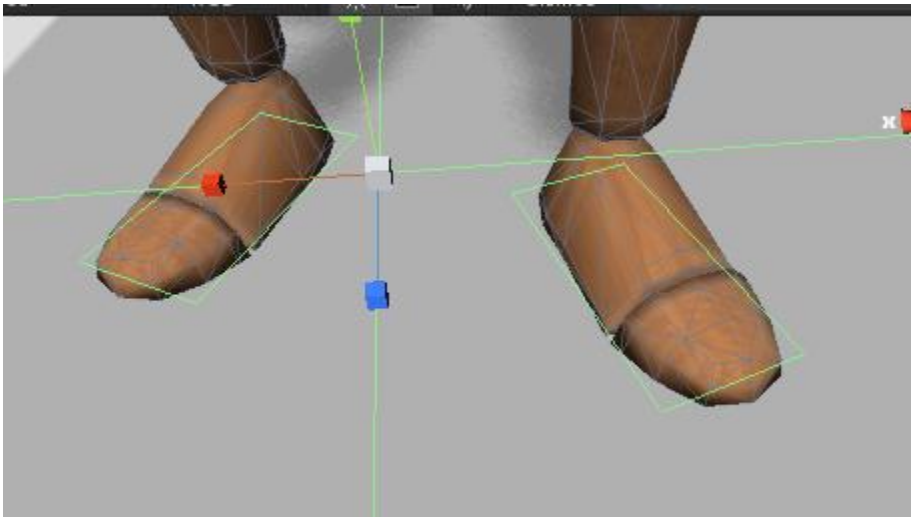


Remember how we had to remap the spine bone in our lean controller to the version in the cloned shadow? The locomotion controller does the exact same thing internally when we give it bone parameters like these as input in the inspector.

Now, let's tell it what bones refer to the legs, and where the feet are. Expand the "Legs" dropdown, and add two legs. Configure them like so:



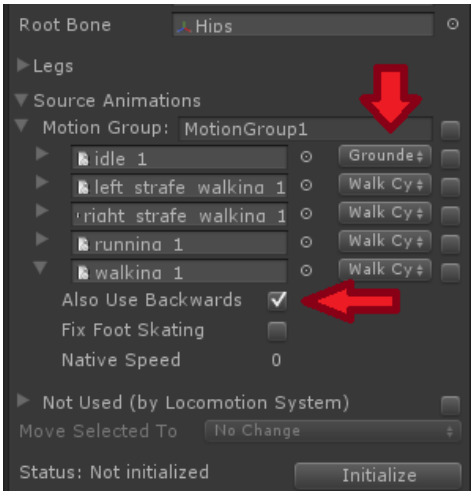
You should be able to see the bottoms of the character's feet in the scene view:



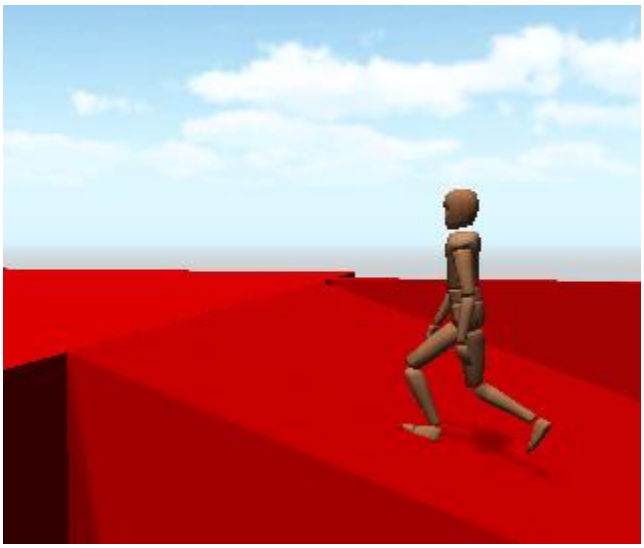
Now, we need to tell the leg controller what animations to use. The locomotion controller works by analyzing these animations and picking a blend of the animations depending on the velocity of the character. Higher speeds will cause the character to play the running animation, or lateral movement will cause the character to use a sidestepping animation. We use a modified version of a locomotion system provided by Unity and developed by Rune Johansen. You can find out more about the system here: http://runevision.com/multimedia/unity/locomotion/. For now, expand the

Source Animations pull-down menu. Select “idle_1”, “left_strafe_walking_1”, “running_1”, “right_strafe_walking_1”, and “walking_1”. With these animations selected, pull down the “Move Selected To” option and pick “New Group”. This tells the system that these animations are the ones we want to consider when walking or running around.

Change `idle_1` from “Walk Cycle” to “Grounded”. This tells the system that the idle pose should be played when we aren’t moving. Also, expand the `walking_1` field and check “Also Use Backwards”.



Finally, click Initialize, and run the simulation. Press R, and watch the character properly walk to the destination waypoint.



Congratulations! You've gotten a character to successfully walk to a destination in ADAPT. Try experimenting with different waypoints in different parts of the environment (using the configurable key code to give different waypoints different keys).

Tutorial 2b: The Navigation System (Recast)

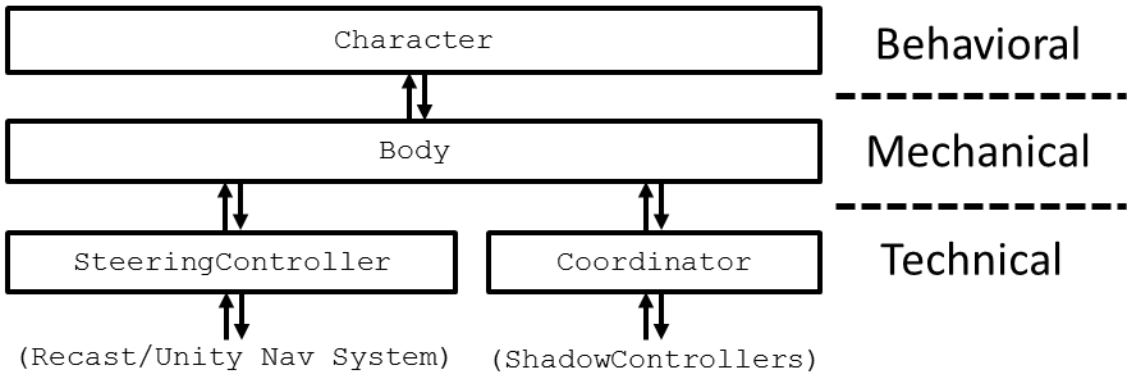
TBD

Tutorial 3: The Body Interface

Up until now, we've been controlling the character at a very low mechanical layer, effectively re-implementing parts of the character coordinator and navigation/locomotion system. In practice, when using ADAPT for a bigger project, we don't want to manually integrate these components. For this reason, ADAPT provides a default character capable of locomotion, reaching, gaze tracking, gesture animations, sitting, and physical response to collisions. Each of these features stems from a shadow

controller dedicated to that function, which we blend together using a more advanced version of the coordinator discussed in Tutorial 1.

The provided default character is organized into layers, with different interfaces offering more intuitive or more specific control of components:



The bottom layers are the "technical" controls of the character. Usually an external component would interact with these systems directly only if it wanted to very specifically modify the behavior of the blending system or a particular choreographer. In general, this would be considered an advanced technique and probably wouldn't be best to start off with.

The middle layer, the "mechanical" layer, contains a set of commands for activating and de-activating shadow controllers, and sending messages to the coordinators to change their targets or behavior. Some of the functions in the `Body` interface include:

```
ReachFor(Vector3 target)
ReachStop()
HeadLookAt(Vector3 target)
HeadLookStop()
AnimPlay(string name)
SitDown()
StandUp()
NavGoTo(Vector3 target)
NavStop(bool sticky = true)
NavSetDesiredOrientation(Quaternion desired)
```

There are also more advanced commands for finer control and feedback. These commands are translated into messages that are sent either directly to the shadow controllers, or to the coordinator itself to know when to blend controllers in and out. The actual commands within the `Body` class are very simple, and should shed some light on how this layer actually performs. Generally, external modules would interact with a character on this layer if they wanted very specific control of the mechanical actions the character performs, as well as being able to handle failure states (can't navigate to a point, too far away to reach for something, etc.) manually. Smart Objects mostly interact with a character using the `Body`, as discussed in Tutorial 5.

The top layer, the behavior layer, is an abstraction of the `Body` that is more suitable for use in behavior trees. Unlike the commands in the `Body`, which occur instantly, the `Character` functions are

designed to “block” until the action either succeeds or fails. All functions in the Character class return a `RunStatus` enumerator, which is discussed more in Tutorial 4. Behavior trees interact with a character at the top layer, as well as any external module which wants to direct the character without having to deal with details like detecting the failure of specific tasks. All commands in the `Character` class use the commands in the `Body` class, but with a more simplified interface.

With the discussion out of the way, this will be a simple tutorial on flexing some of the muscle of the `Body` and using some of its functionality. The empty template for this tutorial can be found in the `Tutorial3Empty` scene file.

Navigation, the “Right” Way

Like before, we will begin by making a waypoint destination for the navigation system. We can edit the waypoint class from before to make a dead simple class for this. Call it TutorialWaypoint2:

```
public class TutorialWaypoint2 : MonoBehaviour
{
    public KeyCode code;
    public Body body;

    void Update()
    {
        if (Input.GetKeyDown(this.code) == true)
            this.body.NavGoTo(transform.position);
    }
}
```

This is almost exactly the same as before, except instead of taking a `UnitySteeringController` and setting its `Target` property, we take a `Body` and use the `NavGoTo` command. We’ve provided a sphere (with no `SphereCollider`) like the one used in the prior tutorial called `NavWaypoint`. Attach the new `TutorialWaypoint2` script to the `NavWaypoint` game object, and give it a reference to the `EmptyCharacter` and a key to use (such as “R”) as we did in Tutorial 2. Run the simulation and test out the waypoint.

Look and Reach

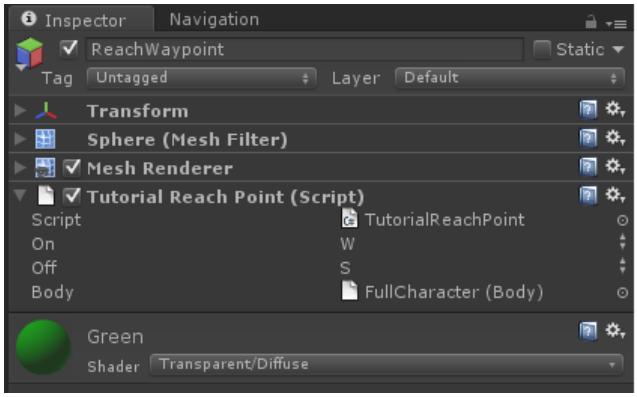
Like navigation, using the `Body` class to perform other activities only requires a few lines of code. We’re going to make two more classes very similar to `TutorialWaypoint`. They’ll look like this:

```
public class TutorialLookPoint : MonoBehaviour
{
    public KeyCode on;
    public KeyCode off;
    public Body body;

    void Update()
    {
        if (Input.GetKeyDown(this.on) == true)
            this.body.HeadLookAt(transform.position);
        if (Input.GetKeyDown(this.off) == true)
            this.body.HeadLookStop();
    }
}
```

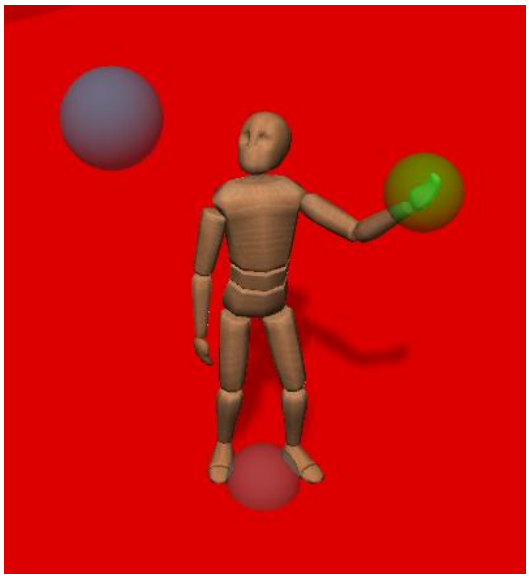
and

```
public class TutorialReachPoint : MonoBehaviour
{
    public KeyCode on;
    public KeyCode off;
    public Body body;

    void Update()
    {
        if (Input.GetKeyDown(this.on) == true)
            this.body.ReachFor(transform.position);
        if (Input.GetKeyDown(this.off) == true)
            this.body.ReachStop();
    }
}
```

The two main commands here are `HeadLookAt` and `ReachFor`, which both just take a Vector3 for input. Note that `ReachFor` will try to reach as close as it can to the target point. Attach the `TutorialLookPoint` script to the `LookWaypoint` gameobject, and the `TutorialReachPoint` script to the `ReachWaypoint` gameobject. Give each component a reference to the FullCharacter and set the on and off keycodes for each script, such as “T” and “G” to turn looking on and off, and “Y” and “H” to turn reaching on and off.



Run the simulation, send the character to the navigation waypoint, and try turning looking and reaching on and off.

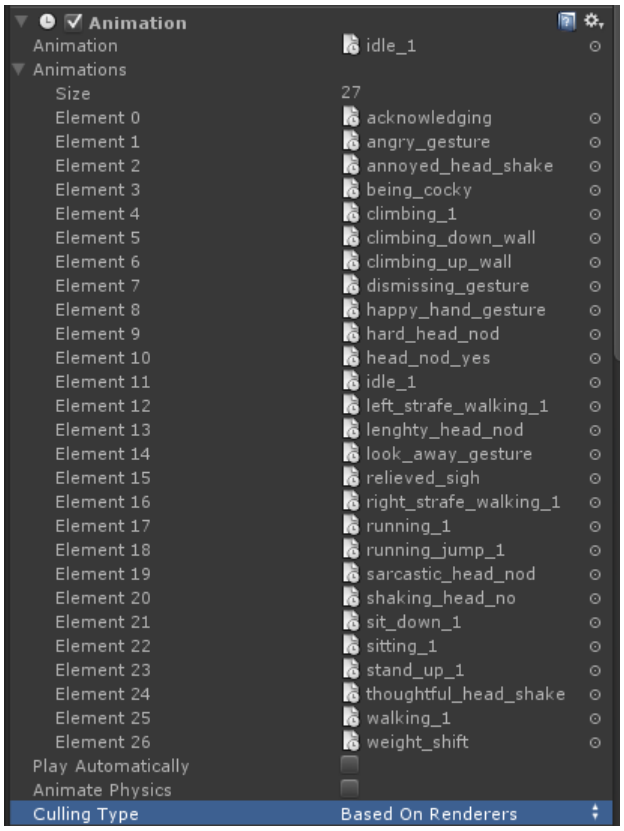


Gestures

Finally, let's make the character play some gestures. We'll make another class to attach to the `FullCharacter` object called `TutorialGestures`. We'll start it out by giving it a `Body` reference and having it fetch the character's `Body` component at the start of the simulation:

```
public class TutorialGestures : MonoBehaviour
{
    protected Body body;

    void Start()
    {
        // Get a reference to the body component
        this.body = this.GetComponent<Body>();
    }

    void Update()
    {
    }
}
```

Now, we want the character to play a few upper body animations. If you select the FullCharacter object, you'll notice it has an `Animation` component with a handful of animations provided (your exact list may differ slightly):



Let's pick two of the more noticeable gestures: "dismissing_gesture" and "being_cocky". We'll bind these to the 1 and 2 keys. Fill in the `Update` function of `TutorialGestures` like this:

```
    void Update ()
    {
        if (Input.GetKeyDown(KeyCode.Alpha1) == true)
            this.body.AnimPlay("dismissing_gesture");
        if (Input.GetKeyDown(KeyCode.Alpha2) == true)
            this.body.AnimPlay("being_cocky");
    }
```

Now run the simulation and press the 1 and 2 keys, as well as the other keys we used earlier to control setting a destination, reaching, and gazing. You can do all of these things simultaneously and the movements should blend with one another.

Now you're really starting to flex some of ADAPT's animation muscle! Experiment with looking and reaching, as well as the other commands in the `Body` class. You can see some demos of the Reach and HeadLook systems in action in the Demos folder. Refer to [TBD] for information on configuring the Reach and HeadLook shadow controllers. Some other controllers like Sitting and the Ragdoll controllers are demonstrated in the Demos folder, though they are more complicated to use.

Tutorial 4: Authoring Behavior with Parameterized Behavior Trees

Armed with a better understanding of the underlying layers of a character, it's time to start writing behavior using TreeSharpPlus, ADAPT's Behavior Tree library. We're going to implement a simple scene first with a single character walking around, and then with a second character for interactions. Before proceeding with this tutorial, here's some required reading/viewing:

Behavior Trees (AIGameDev):

- Part 1: http://aigamedev.com/open/article/behavior-trees-part1/
- Part 2: http://aigamedev.com/open/article/behavior-trees-part2/
- Part 3: http://aigamedev.com/open/article/behavior-trees-part3/

ADAPT uses an extension of Behavior Trees for behavior authoring, and so it is essential to learn what they are and how they work.

Parameterized Behavior Trees (Paper):

- http://www.seas.upenn.edu/~shoulson/papers/mig2011.pdf

PBTs are an extension of behavior trees focusing on controlling multiple characters simultaneously with one single tree. ADAPT and TreeSharpPlus use this concept for its event structure, which makes things like conversations easy to author for multiple agents.

Advanced C# Programming Concepts

- Closures: http://richnewman.wordpress.com/2011/08/06/closures-in-c/
- .NET Iterators and the yield statement (Optional)

Despite being primarily an object-oriented language, TreeSharpPlus exploits some characteristics of C# that make it behave more like a functional languages. In particular, understanding Closures and how they interact with local variables in a function is essential to avoid bugs when calling functions outside of a behavior tree.

In TreeSharpPlus and ADAPT, behavior trees are treated as their own little processes. A behavior tree is automatically ticked on a regular basis (by default, about 20 times per second) by the Behavior Manager. During each tick, the behavior tree evaluates whatever node is currently active, and is told whether that node has succeeded, failed, or is still running and should be ticked again next round. The success or failure of a node, once it's terminated, allows the behavior tree to pick the next node to execute depending on the structure of its control nodes (like sequences and selectors). Nearly every function having to do with running a behavior tree in TreeSharpPlus returns a `RunStatus`, which is an enumerated type containing "Success", "Failure", or "Running". This is how nodes in the tree hierarchy communicate with one another.

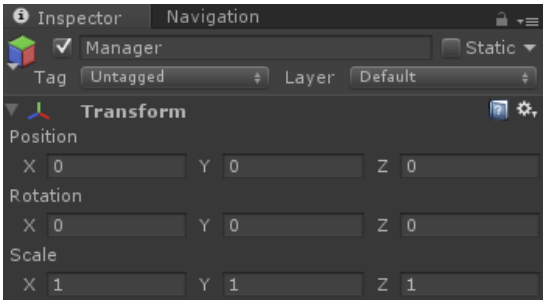
External functions, such as those in the `Character` class, also return a `RunStatus`, which is how those functions can tell the behavior tree whether it terminated or needs to keep running. Most of the leaves of the tree are just "Invoke" nodes that call whatever function they've been assigned and return whatever `RunStatus` that function reported. Other statements, such as assertions, can be converted into a `RunStatus` to express "true" or "false".

One special feature of TreeSharpPlus is that all trees can be terminated. When a tree is terminated, it propagates a special message to any of its active nodes to clean up and end whatever those nodes are doing. Termination messages can come arbitrarily at any time during simulation. Most invoke nodes have two functions: one that they call when ticked, and one that they call to terminate. Like ticking, the process of termination returns a `RunStatus`, so termination can take multiple time steps to finish, and can fail. When designing behavior trees, it is important to remember that trees can be terminated, and to design them so that they clean up properly when this occurs (even if it might take a few moments to fully clean up). The `Behavior` class contains a number of helper functions that pre-construct nodes for you that both perform and terminate properly and safely.

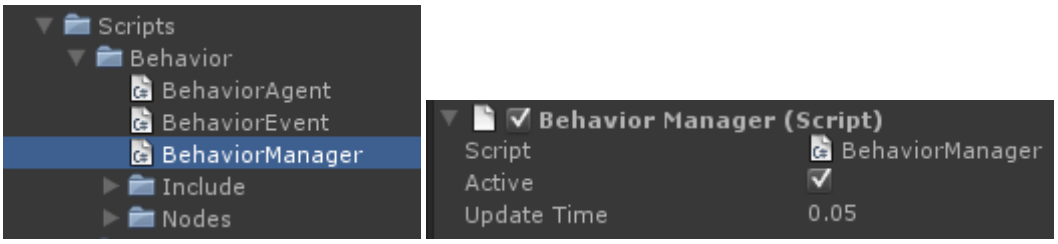
Now, armed with a greater understanding of (P)BTs and C# tricks, let's open Tutorial4Empty.

The Manager

To begin with, the empty scene contains a Wanderer character and three waypoints: Wander1, Wander2, and Wander3. Our simple single-character behavior will make the character randomly walk between these waypoints. To begin with, however, we need `BehaviorManager` in the scene. The `BehaviorManager` handles the scheduling of character behaviors, and the rate at which their trees are "ticked". By default, behavior trees in ADAPT are ticked 20 times a second. During those ticks, the tree will check its status and send out new commands to the characters if necessary. Create an empty game object, change its name to "Manager", and place it at (0, 0, 0) in the scene.



Next, attach a `BehaviorManager` component to it:



Characters will automatically find the behavior manager if there's exactly one in the scene, so we don't need to deal with any pointers or references between the manager and the characters.

Behavior for a Single Character

Now let's make a script to build the behavior tree for the wanderer character. Be sure to import the `TreeSharpPlus` library and inherit from the `Behavior` class. We won't be using the `Update` function, so remove it for now.

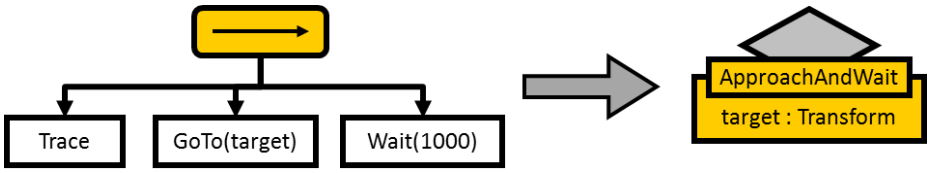
```
using UnityEngine;
using TreeSharpPlus;
using System.Collections;

public class TutorialWanderBehavior : Behavior
{
    public Transform wander1;
    public Transform wander2;
    public Transform wander3;

    // Use this for initialization
    void Start()
    {

    }
}
```

This class will allow us to assign the waypoints in the inspector, so that the tree can use their positions for wandering. Now let's start building the tree. When building behavior trees, it's always good to break the structure of the tree down to the smallest sensible functions possible. To do this, we can create a number of functions that take parameters and return a `Node` at the root of the sub-tree. These functions take the place of a Lookup node in a tree diagram. Here's an example function, which uses the prefix "ST_" in its function name to indicate that it returns a sub-tree.

```
    protected Node ST_ApproachAndWait(Transform target)
    {
        return new Sequence(
            new LeafTrace("Going to: " + target.position),
            this.Node_GoTo(target.position),
            new LeafWait(1000));
    }
```

As the name implies, this sub-tree will approach a target and wait. The function constructs a sequence with three child nodes. The first is a trace node which will write output in the Unity console. The second

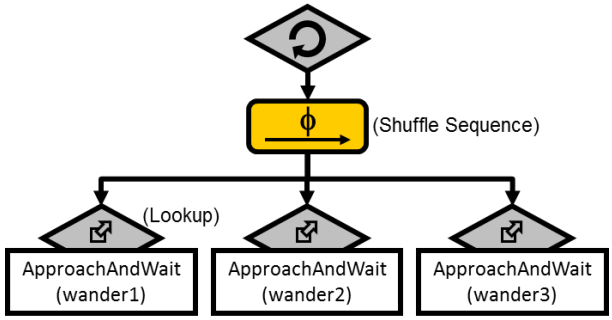
is a “GoTo” node, which is constructed by a helper function in the `Behavior` class. (Take a look at the Node_GoTo() function to see exactly what it’s returning -- you’ll need to understand Closures to see how it works). Most pre-constructed node functions like `Node_GoTo` and `Node_Gesture` contain the correct behavior for what to do both when the node is ticked, and when it’s terminated, so it’s best to use them whenever possible. Finally, we have a wait node which will wait for 1,000 milliseconds. Here’s an illustration of the subtree:



Next, we want to use this approach-and-wait subtree to cause the character to move to the waypoints. Since this is a random wander, we want to randomize the order in which they appear in the tree. Additionally, since this is a character’s individual behavior tree, we never want it to terminate. So what we will have is a SequenceShuffle node wrapped in a DecoratorLoop. Here’s what the function looks like:

```
    protected Node BuildTreeRoot()
    {
        return
            new DecoratorLoop(
                new SequenceShuffle(
                    ST_ApproachAndWait(this.wander1),
                    ST_ApproachAndWait(this.wander2),
                    ST_ApproachAndWait(this.wander3)));
    }
```

And here’s an illustration:

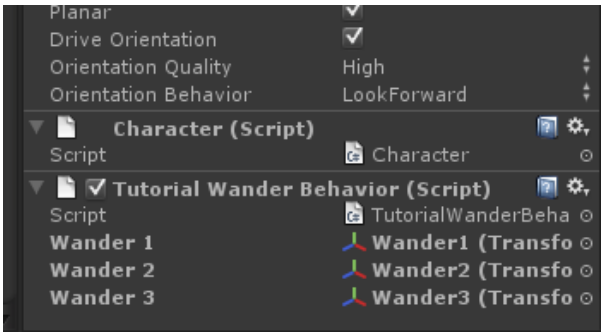


Finally, we’ll build the tree and start it in our Start function. Here’s what the whole class should look like:

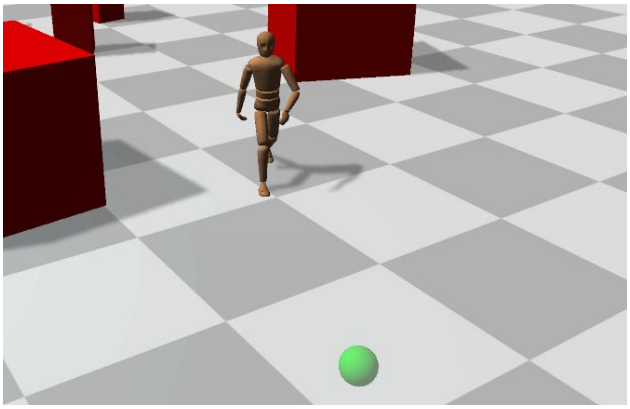
```
public class TutorialWanderBehavior : Behavior
{
    public Transform wander1;
    public Transform wander2;
    public Transform wander3;

    protected Node ST_ApproachAndWait(Transform target)
    {
```

```
        return new Sequence(
            new LeafTrace("Going to: " + target.position),
            this.Node_GoTo(target.position),
            new LeafWait(1000));
    }

    protected Node BuildTreeRoot()
    {
        return
            new DecoratorLoop(
                new SequenceShuffle(
                    ST_ApproachAndWait(this.wander1),
                    ST_ApproachAndWait(this.wander2),
                    ST_ApproachAndWait(this.wander3)));
    }

    // Use this for initialization
    void Start()
    {
        base.StartTree(this.BuildTreeRoot());
    }
}
```

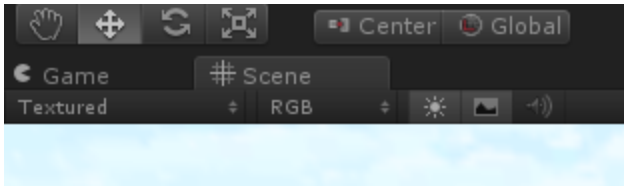
Now attach this script to the wanderer. Delete the original `Behavior` script if it's there and give the new `TutorialWanderBehavior` script references to Wander1, Wander2, and Wander3.

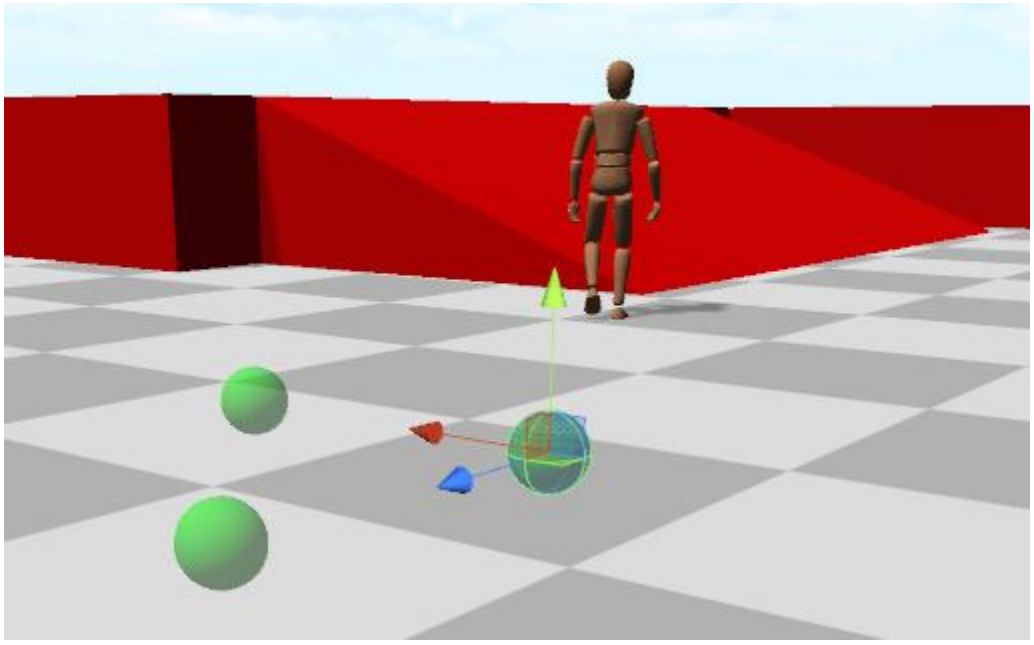


Run the simulation, and you should see the character walking around.



Now, try switching to Scene view and moving some of the waypoints around somewhere else.





We have a problem!
Notice anything? Strange, isn't it, how the character completely ignores them. Let's go back and look at `ST_ApproachAndWait` for a second.

```
protected Node ST_ApproachAndWait(Transform target)
{
    return new Sequence(
        new LeafTrace("Going to: " + target.position),
        this.Node_GoTo(target.position),
        new LeafWait(1000));
}
```

Aren't we fetching the target transform's position? Shouldn't that change as the transform moves around? Normally in a typical Unity C# script, that would be the case. If something were like this in an `Update` function, then every time the `Update` function was called, it would evaluate `target.position` and see where it was at that moment. However, here we're building the tree at the start of the simulation, and using this node without ever calling ST_ApproachAndWait again during runtime. The value of `target.position` is evaluated at the very start of the simulation ("tree build time"), and stored as a constant in the tree rather than being evaluated when the tree is ticked ("tree tick time"). This is fine for a lot of situations, but sometimes we want the data in a behavior tree to be more dynamic. To deal with this, TreeSharpPlus has a special class called `Val`. The `Val` class (short for Value), can either store a constant value, or a closure for fetching a value. I'm assuming you're familiar

with closures and lambda functions by now. If not, check out the resources at the beginning of the tutorial. Here's a quick `Val` example:

```
Val<float> t1 = Val.Val(0.6f);
Val<float> t2 = Val.Val(Time.time);
Val<float> t3 = Val.Val(() => Time.time);

// This will display "0.6f"
Debug.Log(t1.Value);

// This will display the time when t2 was initialized
Debug.Log(t2.Value);

// This will display the time when Debug.Log(t3.Value) was called
Debug.Log(t3.Value);
```

Use `Val.Val()` as the easiest way to construct a `Val` type. Because `Val` can use closures, it can also access local variables from a function after that function has terminated. For example:

```
Val<int> MakeVal()
{
    int x = 6;
    Val<int> foo = Val.Val(() => x);
    Debug.Log(foo.Value); // Will display 6
    x++;
    Debug.Log(foo.Value); // Will display 7
    return foo;
}

void DoStuff()
{
    Val<int> bar = MakeVal();
    Debug.Log(bar.Value); // Will display 7
}
```

(You can experiment in the `TutorialWanderBehavior` class using the `Start()` function, if you'd like to play with the `Val` type.) If you're designing your own function to be used in a behavior tree, it's a good idea to use `Val<T>` rather than just `T` for your parameters. For example:

```
void Function1(float time)
{
    Debug.Log("The time is: " + time);
}

void Function2(Val<float> time)
{
    Debug.Log("The time is: " + time.Value);
}

void DoStuff()
{
    Function1(0.6f);
    Function1(Time.time);
```

```
        Function2(0.6f); // Still works (implicit conversion)
        Function2(Time.time); // Still works (implicit conversion)
        Function2(Val.Val(0.6f));
        Function2(Val.Val(Time.time));
        Function2(Val.Val(() => Time.time));
    }
```

Because we can implicitly cast to `Val` (but not from it), we gain a lot of flexibility from using it in our parameters.

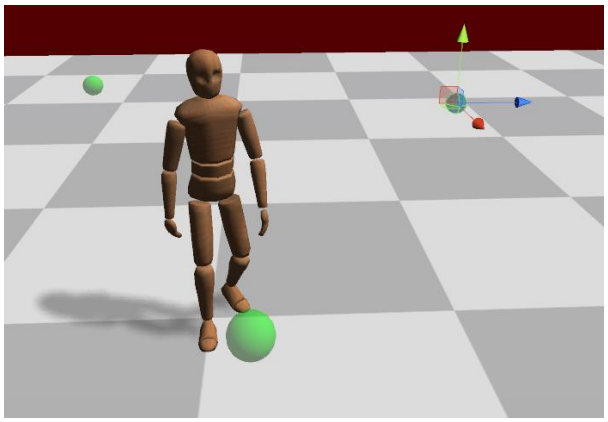
Using Some Vals

Now let's incorporate the `Val` type into our behavior tree. Go back to the `ST_ApproachAndWait` function and change it to look like this:

```
    protected Node ST_ApproachAndWait(Transform target)
    {
        Val<Vector3> position = Val.Val(() => target.position);

        return new Sequence(
            //new LeafTrace("Going to: " + position.Value),
            this.Node_GoTo(position),
            new LeafWait(1000));
    }
```

(Notice that this is missing the LeafTrace node, as I had to take it out. Can you figure out why there's a problem with it? Answer: The string concatenation at the "+" operator would be performed at *tree construction time*, rather than at each *tree tick time*, so even though position is a `Val<Vector3>`, the string given to the `LeafTrace` would be constant from the beginning of the simulation. You can get around this, but I'm not going to bother for this tutorial. Always be aware of whether or not your tree data values are fixed at tree construction time.)

Save the file, run the simulation again, and start moving the waypoints around. The wanderer should start going to the new positions instead of the old ones.



Notice that if you move any of the waypoints somewhere the character can't reach, you start getting errors. This is because the `Nav_GoTo` nodes will fail, that failure will break the loop (DecoratorLoop only loops on success), and the character will keep trying to tick its tree even after the tree has

terminated with failure. You can fix this by forcing the shuffle sequence to always succeed by using a ForceStatus decorator, like so:

```
    protected Node BuildTreeRoot()
    {
        return
            new DecoratorLoop(
                new DecoratorForceStatus(RunStatus.Success,
                    new SequenceShuffle(
                        ST_ApproachAndWait(this.wander1),
                        ST_ApproachAndWait(this.wander2),
                        ST_ApproachAndWait(this.wander3))));
    }
```

This way, even if a waypoint is unreachable, the tree will keep running and will select the next possible node (or nothing, if no node is available). (Note that you could also replace the Sequence with a Selector, but that could cause the character to go to the same waypoint multiple times in a row. Using a shuffling sequence ensures that the character will always try different waypoints.)

Making Friends

Our Wanderer looks pretty lonely, so let's make him someone to talk to. Drag in the TutorialFriend prefab and place it somewhere within the red walls. Delete TutorialFriend's generic `Behavior` component if it has one (and make sure you've done the same for the Wanderer). We'll give him a very simple behavior routine. Create a script `TutorialIdleBehavior`, like this:

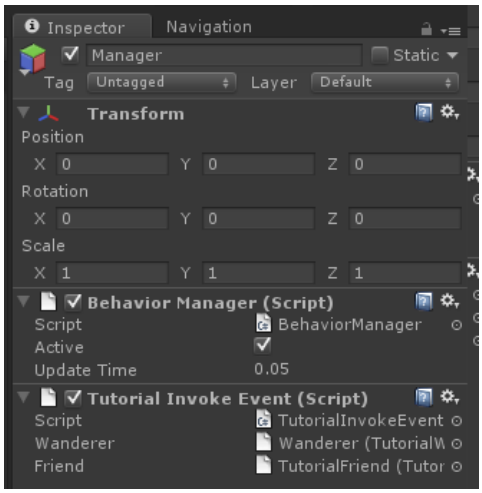
```
using System;
using UnityEngine;
using TreeSharpPlus;
using System.Collections;

public class TutorialIdleBehavior : Behavior
{
    protected Node BuildTreeRoot()
    {
        return
            new DecoratorLoop(
                new Sequence(
                    new LeafWait(6000),
                    this.Node_Gesture("relieved_sigh")));
    }

    void Start()
    {
        base.StartTree(this.BuildTreeRoot());
    }
}
```

The Friend will stand idle and play a sighing animation every six seconds. Attach `TutorialIdleBehavior` to TutorialFriend. Now we're going to make a conversation event between the Wanderer and TutorialFriend. Create a new script called `TutorialInvokeEvent`, like so:

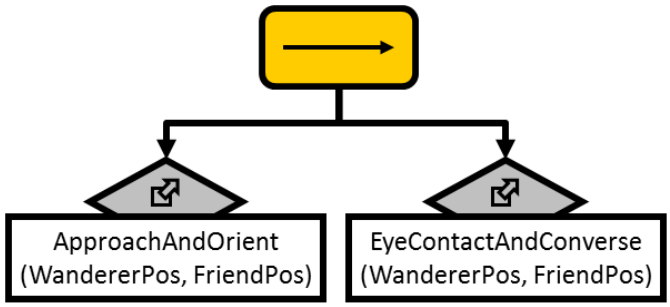
```
using UnityEngine;
using TreeSharpPlus;
using System.Collections;
```

```
public class TutorialInvokeEvent : MonoBehaviour
{
    public Behavior Wanderer;
    public Behavior Friend;


    void Update ()
    {
    }
}
```

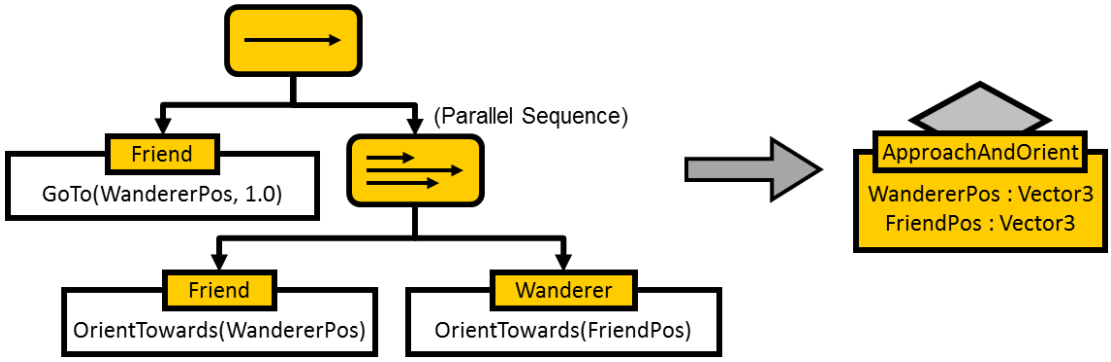
Attach this new script to the global Manager object in the scene and give it a reference to both the Wanderer and TutorialFriend.



We want to design an event tree where the TutorialFriend will approach the Wanderer, the two characters will orient towards each other and make eye contact, and then they'll each play a few animations to simulate a conversation. Let's start building the tree.

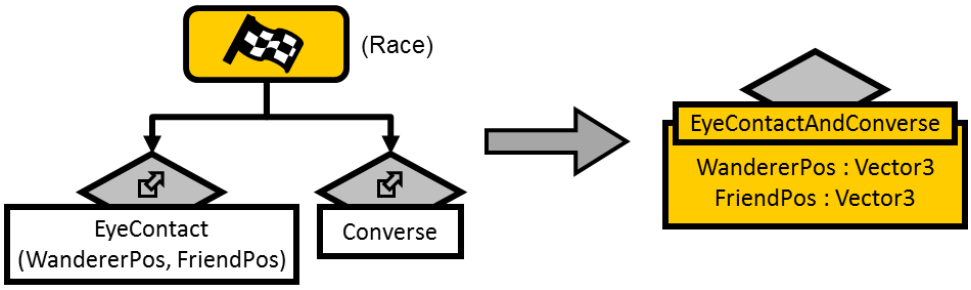


The root of the tree is straightforward. We want to split the conversation into two phases. The first phase involves the Friend walking up to the Wanderer, and the two characters orienting their bodies towards one another. The second phase begins immediately after the first, and involves the characters conversing while maintaining eye contact. Now here's the `ApproachAndOrient` subtree:

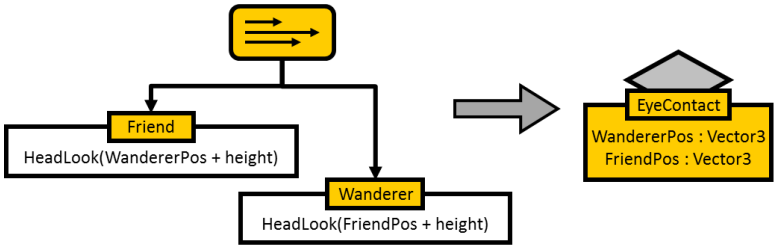


Taking in two Vector3 values, we tell the Friend to go to the Wanderer's position (with a distance of 1.0). The `NavGoTo` node will block until the Friend arrives at the right spot. Once that happens, then in parallel the two characters will turn so that they're oriented towards one another's position. Once this is done, this sub-tree will return success.

The EyeContactAndConverse sub-tree is a little complicated, so we're going to split it up into smaller sub-trees, like so:



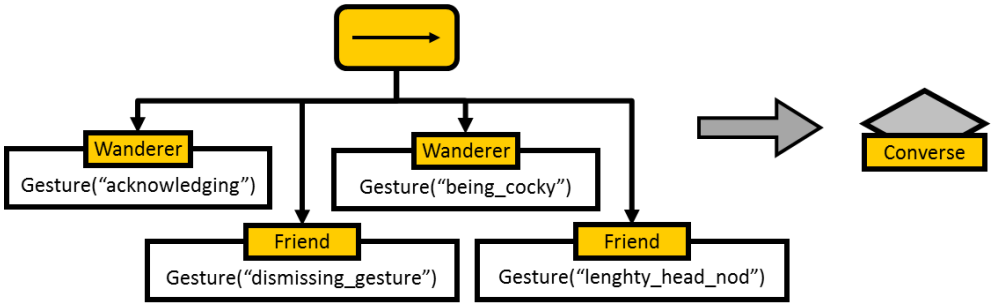
In this sub-tree, we use Race nodes. Race nodes run all of their children in parallel, returning the result (Success or Failure) of their first child to finish. They act similarly to parallel sequence and selector nodes, except they take whichever one of their children finishes first, and terminate all of the others once that happens. In this case, as you'll see, our `EyeContact` sub-tree will never finish on its own. Instead, since the Converse tree *will* eventually finish, the Race node will detect that, and terminate the `EyeContact` sub-tree, which will tell the characters to stop looking at each other. Race nodes are incredibly useful whenever you want one sub-tree to be running continuously for the duration of another. Here's the EyeContact sub-tree:



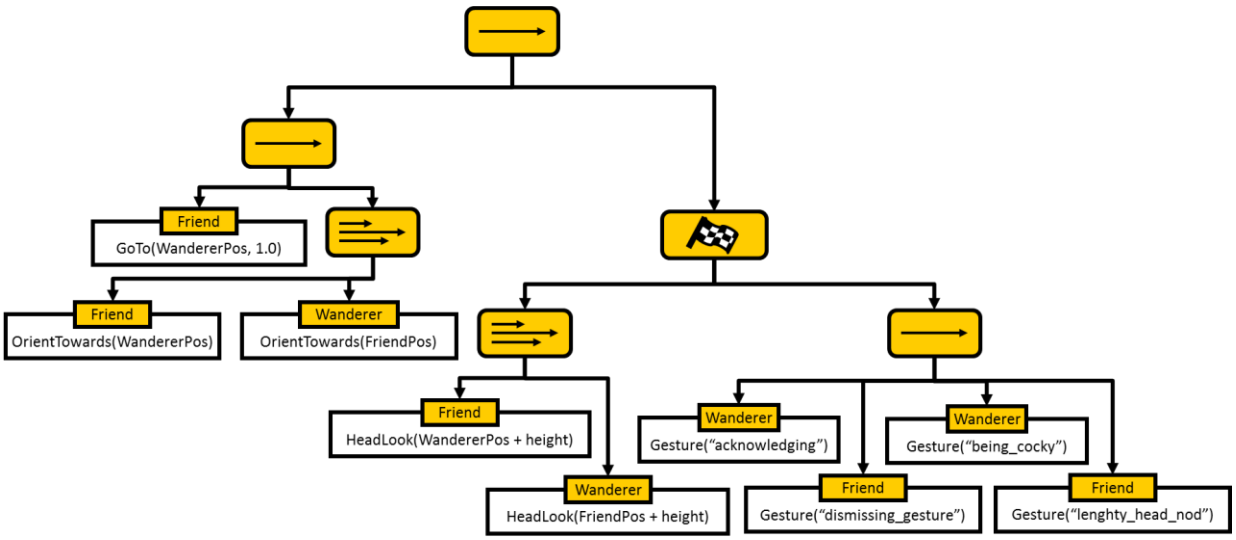
In short, we just use a parallel sequence to make the two characters look at one another with a height offset. There are other ways to get the position of a character's head, but we can estimate using the character's height (in this case, 1.85). Remember that `HeadLookAt` never finishes once it's called. It

will always receive ticks and always report running. The only way to end it is to terminate the node, either by terminating the entire tree, or by using something like a Race node. When `HeadLookAt` is terminated, it will clean up after itself and the character will stop looking at a specific point.

Finally, let's look at the conversation sub-tree:



Probably the most straightforward, this sub-tree just plays a sequence of animations where the two characters take turns. That should flesh out the whole tree, and we've broken it down into nice little pieces as opposed to one gigantic mess, like this:



There's a little redundancy involved when dividing the tree into smaller lookup nodes and sub-trees, but ultimately the improvement in readability is worth the extra node or two. Now that we have the structure of the tree, let's start implementing it, working from the bottom up. Let's start with the Converse subtree, as it's the easiest:

```
using UnityEngine;
using TreeSharpPlus;
using System.Collections;

public class TutorialInvokeEvent : MonoBehaviour
{
    public Behavior Wanderer;
    public Behavior Friend;
```

```
    protected Node Converse()
    {
        return new Sequence(
            Wanderer.Node_Gesture("acknowledging"),
            Friend.Node_Gesture("dismissing_gesture"),
            Wanderer.Node_Gesture("being_cocky"),
            Friend.Node_Gesture("lenghty_head_nod"));
    }

    void Update ()
    {

    }
}
```

As advertised, we create a sequence of alternating gesture nodes. Next, we'll create one of the more complicated subtrees, the EyeContact subtree. Add the following function to the class:

```
    protected Node EyeContact(
        Val<Vector3> WandererPos, Val<Vector3> FriendPos)
    {
        // Estimate the head position based on height
        Vector3 height = new Vector3(0.0f, 1.85f, 0.0f);

        Val<Vector3> WandererHead = Val.Val(() => WandererPos.Value + height);
        Val<Vector3> FriendHead = Val.Val(() => FriendPos.Value + height);

        return new SequenceParallel(
            Friend.Node_HeadLook(WandererHead),
            Wanderer.Node_HeadLook(FriendHead));
    }
```

Given two `Val`s for the positions of the characters, we create new `Val`s to dynamically calculate the head positions of the characters, and then return a parallel sequence that instructs the two characters to look at one another. Once we have these two functions, we can build the `EyeContactAndConverse` sub-tree using the Race node:

```
    protected Node EyeContactAndConverse(
        Val<Vector3> WandererPos, Val<Vector3> FriendPos)
    {
        return new Race(
            this.EyeContact(WandererPos, FriendPos),
            this.Converse());
    }
```

Next, the `ApproachAndOrient` sub-tree using the `Node_GoTo` function and a distance value of 1.0f:

```
    protected Node ApproachAndOrient(
        Val<Vector3> WandererPos, Val<Vector3> FriendPos)
    {
        return new Sequence(
            // Approach at distance 1.0f
            Friend.Node_GoTo(WandererPos, 1.0f),
            new SequenceParallel(
```

```
                Friend.Node_OrientTowards(WandererPos),
                Wanderer.Node_OrientTowards(FriendPos)));
    }
```

And finally, the tree root:

```
    public Node ConversationTree()
    {
        Val<Vector3> WandererPos = Val.Val(() => Wanderer.transform.position);
        Val<Vector3> FriendPos = Val.Val(() => Friend.transform.position);

        return new Sequence(
            this.ApproachAndOrient(WandererPos, FriendPos),
            this.EyeContactAndConverse(WandererPos, FriendPos));
    }
```

Again, we construct `Val`s for the positions of the Wanderer and the `TutorialFriend`, using the member variables of the `TutorialInvokeEvent` class. These are then propagated through the tree and used where needed. Finally, we need to execute this tree! We'll use the "R" key to activate it whenever we want to use it in the scene. Fill your `Update()` function in with this:

```
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.R) == true)
            BehaviorEvent.Run(
                this.ConversationTree(), Wanderer, Friend);
    }
```

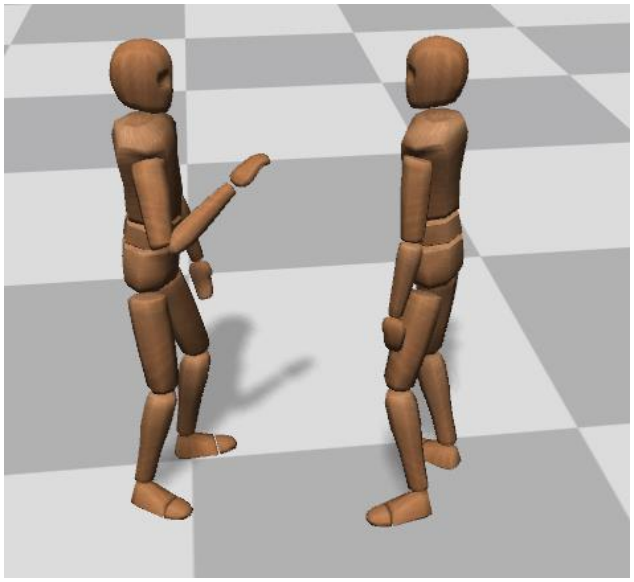
The `BehaviorEvent.Run()` function takes in a behavior tree (which you can build beforehand, or just initialize on the fly like we do here), and a list of all of the agents involved (using the `Behavior` class, or something that inherits from it). The `BehaviorEvent.Run()` function can do some other useful things like setting a priority for an event (events with higher priority will take agents from events with lower priority), a name (for debugging purposes), and a callback function for when the event changes status (loaded, running, terminating, finished, etc.). Here's our final `TutorialInvokeEvent` class:

```
public class TutorialInvokeEvent : MonoBehaviour
{
    public Behavior Wanderer;
    public Behavior Friend;

    /// <summary>
    /// This subtree will cause the two characters to look at each other's
    /// heads. It will run indefinitely until terminated, at which point
    /// the characters will stop gaze tracking
    /// </summary>
    protected Node EyeContact(
        Val<Vector3> WandererPos, Val<Vector3> FriendPos)
    {
        // Estimate the head position based on height
        Vector3 height = new Vector3(0.0f, 1.85f, 0.0f);

        Val<Vector3> WandererHead = Val.Val(() => WandererPos.Value + height);
        Val<Vector3> FriendHead = Val.Val(() => FriendPos.Value + height);
```

```
        return new SequenceParallel(
            Friend.Node_HeadLook(WandererHead),
            Wanderer.Node_HeadLook(FriendHead));
    }

    protected Node Converse()
    {
        return new Sequence(
            Wanderer.Node_Gesture("acknowledging"),
            Friend.Node_Gesture("dismissing_gesture"),
            Wanderer.Node_Gesture("being_cocky"),
            Friend.Node_Gesture("lenghty_head_nod"));
    }

    protected Node EyeContactAndConverse(
        Val<Vector3> WandererPos, Val<Vector3> FriendPos)
    {
        return new Race(
            this.EyeContact(WandererPos, FriendPos),
            this.Converse());
    }

    protected Node ApproachAndOrient(
        Val<Vector3> WandererPos, Val<Vector3> FriendPos)
    {
        return new Sequence(
            // Approach at distance 1.0f
            Friend.Node_GoTo(WandererPos, 1.0f),
            new SequenceParallel(
                Friend.Node_OrientTowards(WandererPos),
                Wanderer.Node_OrientTowards(FriendPos)));
    }

    public Node ConversationTree()
    {
        Val<Vector3> WandererPos = Val.Val(() => Wanderer.transform.position);
        Val<Vector3> FriendPos = Val.Val(() => Friend.transform.position);

        return new Sequence(
            this.ApproachAndOrient(WandererPos, FriendPos),
            this.EyeContactAndConverse(WandererPos, FriendPos));
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.R) == true)
            BehaviorEvent.Run(
                this.ConversationTree(), Wanderer, Friend);
    }
}
```

Make sure this is attached to the Manager with the TutorialFriend and Wanderer assigned, and that the TutorialFriend and Wanderer both have `TutorialIdleBehavior` and `TutorialWanderBehavior`, respectively, with no default `Behavior` component attached. Then run the simulation and press R. The TutorialFriend should approach the Wanderer and engage in a conversation, and then the two characters will return to their normal behavior. You should be able to

call this event as many times as you want and at any point in time, thanks to the BehaviorManager scheduler. Because events operate this way, it is important to always design trees as if they can be terminated at any given time.



Now you've got some single-character and multi-character behavior working! Try adding some more characters and more sophisticated behaviors to scene. You can call more advanced functions using the `LeafInvoke` node, for example (see how the `Character` class uses it). A character can only be in a single event at a time, but events can pre-empt one another using the priority value if they both want to use the same character. Try other behavior tasks too, such as an automated scheduler to make some crowds of characters do more interesting things. You can use static points in the environment (like the waypoints) to direct characters to different places in the map.