**ADAPT**

We now introduce an **Adapt Tutorial** created by Alexander Shoulson to facilitate the job to the users of this tool.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 it has been tested to work up to Unity 5.2. 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;

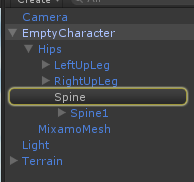
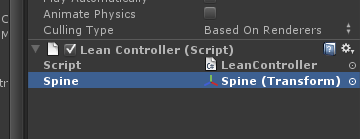
// Apply the new rotation

spine.rotation = Quaternion.Euler(rot);

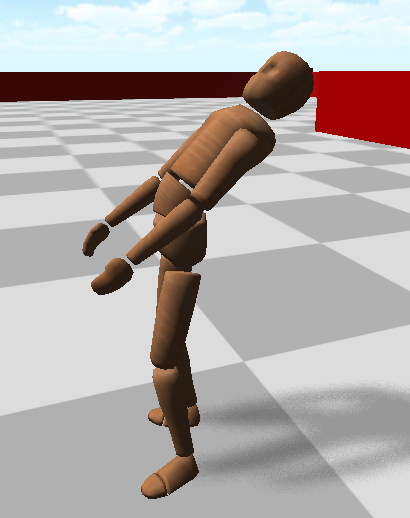
}

}

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;

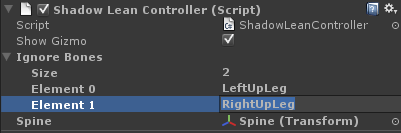
// Apply the new rotation

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,

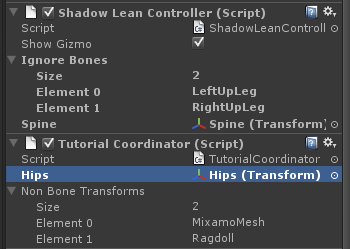
this.transform.GetChild(0),

this);

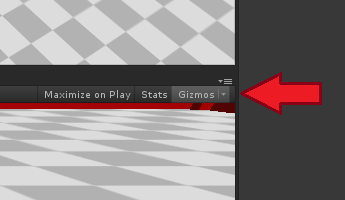
}

}

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.



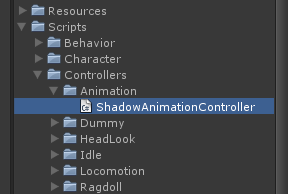
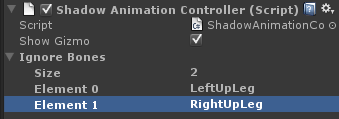
We should be able to lean the character again with R and F. If you enable Gizmos, and press E, you should be able to see the skeleton of the underlying shadow for the lean controller as well.

****

**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));

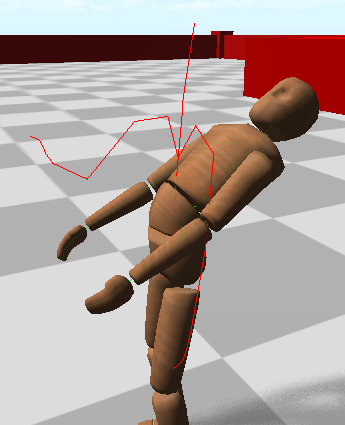
This takes the two buffers, blends them using the weight value and its inverse, and stores the result back in buffer1 (this is why it’s good to reuse buffers rather than creating new ones). Now, one last thing. Let’s make the animation controller actually play an animation when we press the T key. Add 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);

}

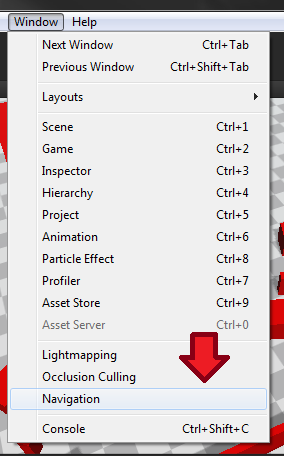
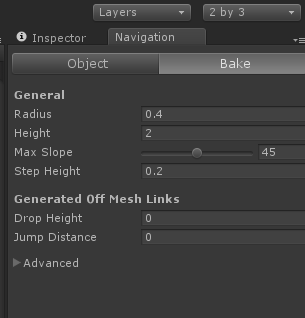
Now you should be able to press T and have the animation controller blend in and out automatically. This should look much better, without any stuttering artifacts or awkward poses. Congratulations, you’ve successfully created your own character controller and integrated it with a built-in ADAPT controller. 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:

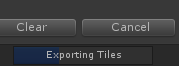
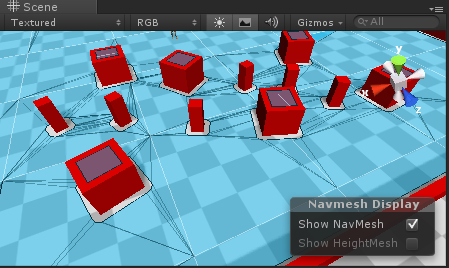
**Radius:** The maximum radius of any agent we’ll have in the world. For the purpose of navigation, each agent is treated like a cylinder.

**Height:** The minimum height of any agent.

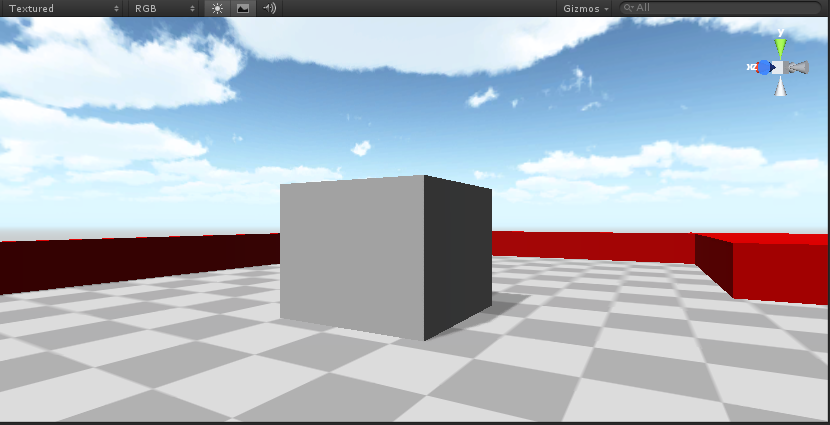
**Max Slope:** The maximum incline (an angle, in degrees) that any agent can climb

**Max Slope:** The maximum step height that any agent can climb

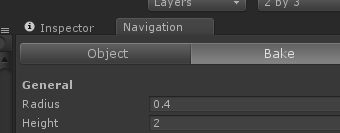
Ignore the others for now. Off-mesh links deal with generating gaps between areas in the navmesh, and the advanced options deal with the detail and fidelity of the navmesh when it’s being generated (sacrificing generation speed for quality). Click Clear, and then Bake to create a new navmesh from scratch (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.

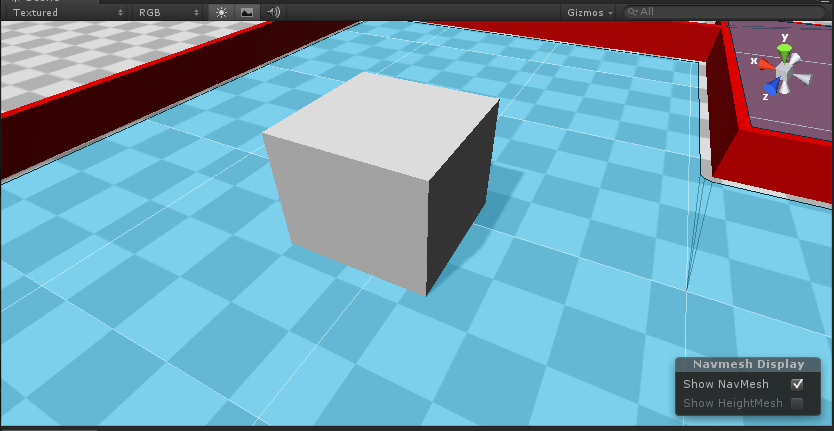
 

Let’s quickly add one more piece of geometry to the environment. Create a new cube gameobject, then scale it and place it somewhere in the scene.

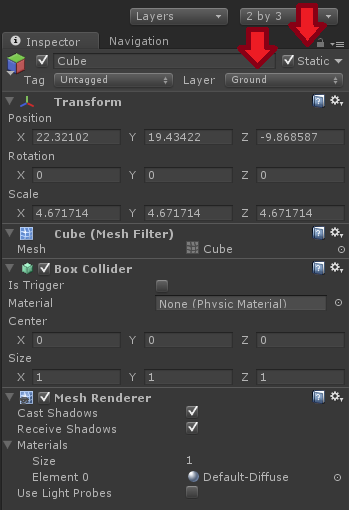


Notice that our navmesh seems to ignore it (if your navmesh isn’t visible, go back to the Navigation tab and make sure that that’s active):

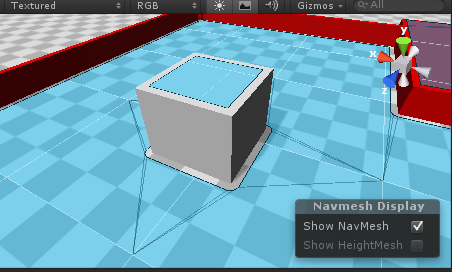




We’ll need to generate a new navmesh, but we need to make sure of one thing first. In order for an object to be recognized by the navmesh generator, it needs to be marked as static in Unity. Also, be sure to set its layer to “Ground” (this doesn’t affect the navmesh, but other systems in ADAPT will behave 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

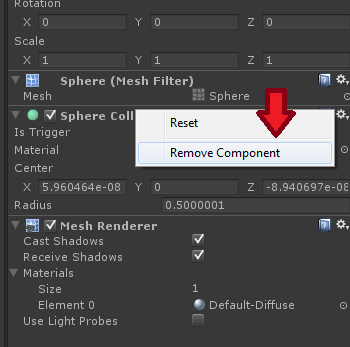
if (Input.GetKeyDown(this.code) == true)

this.controller.Target = transform.position;

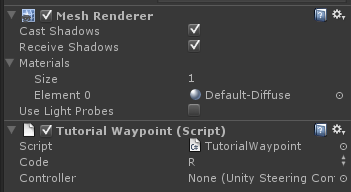
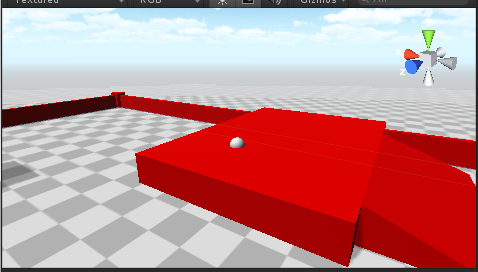
}

}

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

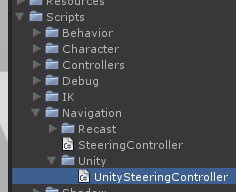
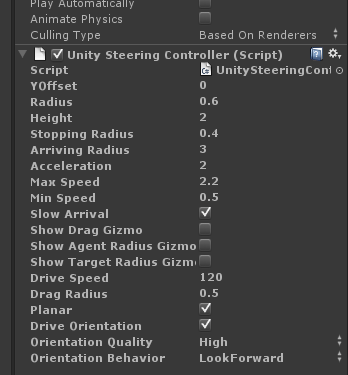
****

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.

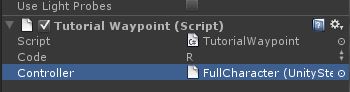
**Arriving Radius:** The radius at which the character will begin to slow down before stopping (this is added to the stopping radius). This feature is toggled with the **Slow Arrival** parameter.

**Drive Speed:** The speed at which we will turn to reorient while walking/running.

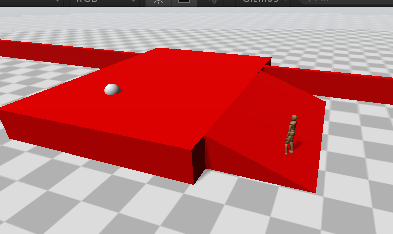
**Drive Orientation:** Turns on or off the ability to turn to face a desired orientation.

**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,

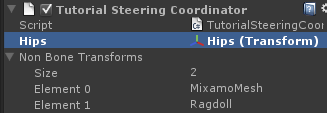
this.transform.GetChild(0),

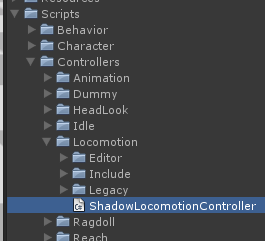
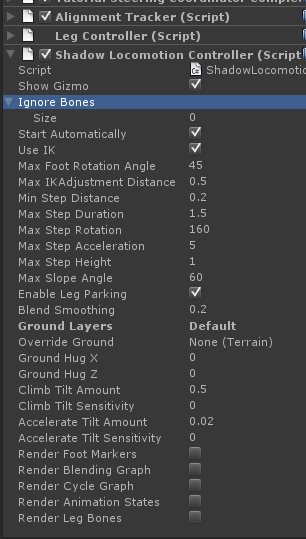
this);

}

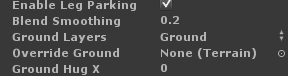
}

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.

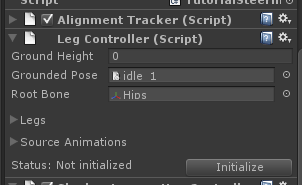
 

** **

We’ll need to configure these components. First, change “Ground Layers” to be “Ground”, rather than “Default”.

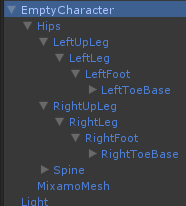
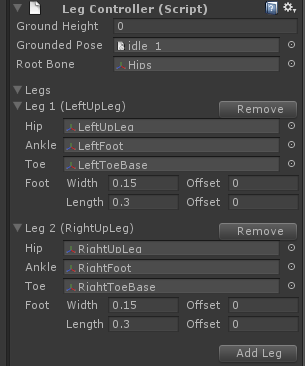
****

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.

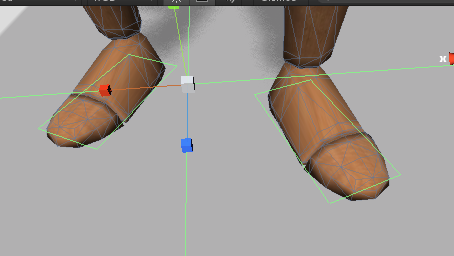
** **

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.

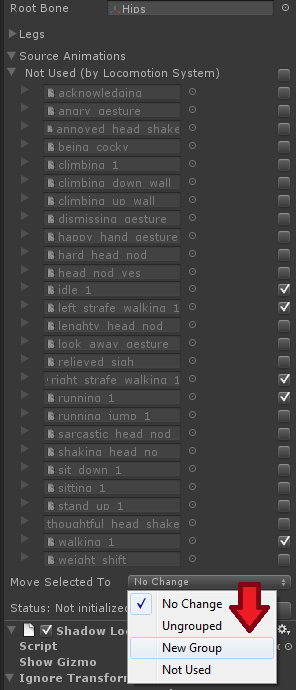
Now, let’s tell it what bones refer to the legs, and where the feet are. Expand the “Legs” dropdown, and add two legs. Configure them like so:

** **

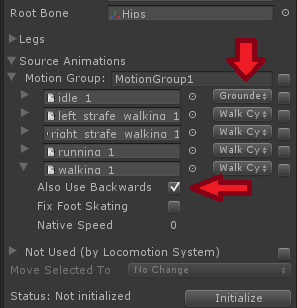
You should be able to see the bottoms of the character’s feet in the scene view:



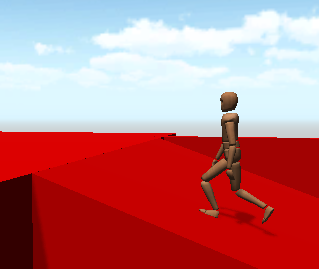
Now, we need to tell the leg controller what animations to use. The locomotion controller works by analyzing these animations and picking a blend of the animations depending on the velocity of the character. Higher speeds will cause the character to play the running animation, or lateral movement will 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).

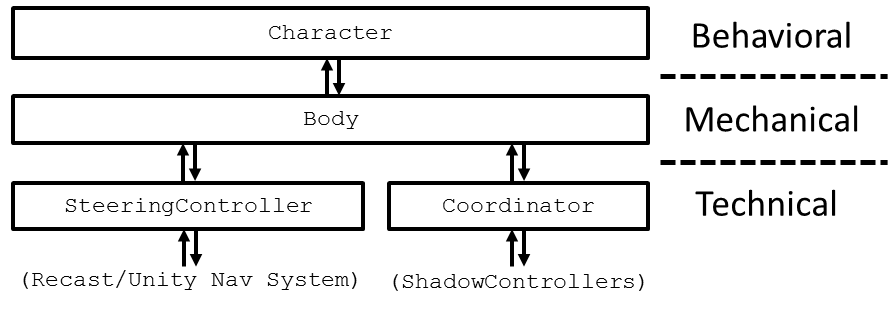
**Tutorial 2b: The Navigation System (Recast)**

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);

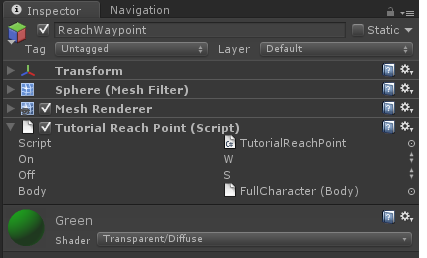
if (Input.GetKeyDown(this.off) == true)

this.body.ReachStop();

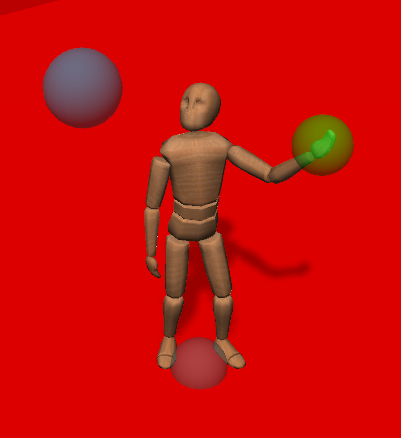
}

}

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>();

}

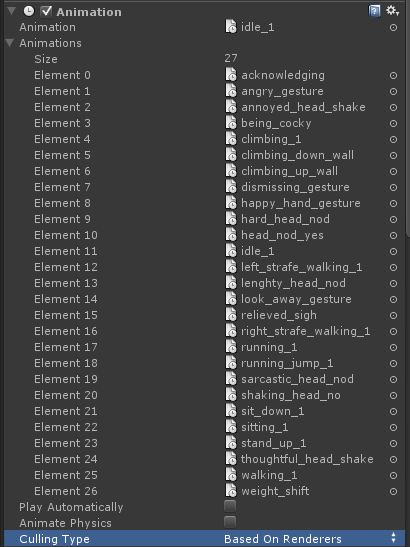
void Update()

{

}

}

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.