Building Levels in Unity

Unity is a top industry choice, perfected for video game development, simulation creation, and environmental design. Its accessibility, flexible tuning, and fair licensing have made it the number one option for independent developers throughout the world.

From the basics to a playable demo, this book will help you build levels in Unity with hands-on practices.

Full of practical examples, it will start by getting you comfortable with the engine so it will enable you to freely navigate and complete tutorials with ease. The book will walk you through the technical requirements of importing your own assets, created with popular 2D and 3D applications, and how to optimize and enhance them with Unity.

By the end of the book, you will get accustomed to Unity editor and will be able to develop a fully-featured game world in Unity.

Who this book is written for

The book is aimed at game artists with no past programming experience who are interested in designing levels in Unity. It does not assume detailed knowledge of similar game platforms.

What you will learn from this book

- Get familiar with Unity editor by learning basic tools and features
- Import and configure custom props and environmental assets
- Use Mecanim and Animate props inside Unity
- Discover details and nuances of character, import and create locomotion with Mecanim
- Work with lighting to greatly enhance the environment
- Create transitions, apply final touches, and journey through levels as a standalone application
- Bring sound and particle effects to your levels

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Using Mecanim for Advanced Animations'
- A synopsis of the book’s content
- More information on Building Levels in Unity
Volodymyr Gerasimov is a game designer, developer, and producer who has worked on multiple titles in companies such as Holymountain Games, Best Way Soft, and Gameloft. Being introduced to Unity in its early versions, he continues to use and explore it to this day as a powerful, flexible, and affordable solution for personal projects and independence start-ups. His previously co-authored Unity 3.x Scripting, Packt Publishing. You can follow Volodymyr on his personal blog at blog.vladgerasimov.com.
Preface

You've just installed Unity and don't know where to start, or simply wish to learn about new features that come with the fifth version of this engine. It doesn't matter if this is your first game engine or you are thinking of making a smooth transition to it, this book has got you covered. Throughout this book, you will be creating an outdoor environment, learning associated tools and features by following practical examples in a step-by-step fashion and solidifying your knowledge by completing practical tasks. Complementary files will allow you to start from any chapter you are interested in and also serve as a catch-up option if you don't wish to complete the tutorials. By the end of this book, you will know how to apply your knowledge of level design, animation, modeling, and much more to the best engines on the market.

What this book covers

Chapter 1, *Meet Unity*, introduces you to the Unity editor and basic tools used throughout the book. We will start from the very beginning by creating a starting project, discussing available windows, parameters, scene navigations, and package imports, and close this chapter with the challenge of recreating a custom window layout.

Chapter 2, *Importing and Configuring Props*, explains the process of exporting assets from 3D modeling apps and importing them into Unity, followed by their configuration, tuning, and troubleshooting common errors. By the end of the chapter, we will look into new materials introduced in Unity 5 and set up LODs for imported assets.

Chapter 3, *Shaping Landscape*, dives into the process of creating a terrain for the outdoor environment, as well as foliage, water, skyboxes, and trees.

Chapter 4, *Dealing with Basic Animations*, gives us the first look at how to handle animations in Unity 5. We will look into the Legacy system, the pros and cons of using it, import animations for the props, and use them to trigger scripts.
Preface

Chapter 5, *Invite Your Characters to Unity*, guides you through the entire process of how to import a humanoid character and get it prepared for the Mecanim animation system, as importing characters into Unity can cause a lot of problems if not done correctly.

Chapter 6, *Using Mecanim for Advanced Animations*, uses mocap animations and creates a basic locomotion state control to demonstrate the power of Mecanim.

Chapter 7, *Lighting Up the World*, explains what built-in Enlighten is capable of doing and talks about some limitations that come with it by lighting up an interior scene. Realtime Global Illumination boosted Unity’s rendering capabilities through the roof. We will also look into other features such as light probing, reflection probes, lightmapping, projectors, light cookies, halos, and lenses.

Chapter 8, *Bringing the Sound*, discusses how the sound works and sets up the ambient sounds and music for our level using Audio Mixer.

Chapter 9, *Exploring the Particle System*, provides you with a practical example of creating a particle system, recommendations, tips and tricks, and a challenge to create your own particle system with provided resources. Although particles are fun, they can be overwhelmed by a variety of options available in the Particle editor.

Chapter 10, *Final Editing and Build*, enables us to get our character to walk around the level, talk about the project and quality settings, and finish with a playable build of our level.
Using Mecanim for Advanced Animations

The Mecanim animation system is a major feature of the 4th version of Unity. It's flexible and easy to use, with the additional benefits of reducing programmer involvement in building animation systems and shortening the development cycle in general. This is the part where you'll learn why that is the case.

Throughout the chapter, we will look into the following topics:

- The animator tool’s basics and functionalities
- Creation and control of state machines
- Blending, combining, and layering motions
- Creating basic locomotion

By the end of this chapter, you will be able to setup, control, and maintain animation systems in a matter of minutes with minimum reliance on coding. We will look into different case scenarios based on available animations, functionality requirements, and rigs. Without further ado, let's get started!

The animator controller

Animation control is a tedious and complicated process. Mandatory proficiency in programming for all developers who used the Legacy animation system pretty much resulted in animation control being fully delegated to programmers writing state machine codes. Well, that's not the case anymore. A little bit of coding is still required, however, it plays more of a support role since major work is now done in Visual Editor, which allows animators to take full control of how and when animations are played, setup transitions, limitations, and control them with conditions. This new system is known as Mecanim.
Using Mecanim for Advanced Animations

To avoid skepticism, I will stop singing the praises of the system itself and give you a hands-on experience of how it really works.

But before we do that, there are a few things we need to take care of:

1. Create a prefab for the Robot and drag it into the scene.
2. Open Asset Store by navigating to Window | Asset Store in the menu on top of the screen.
3. Type Raw mocap data for Mecanim in the search field.
4. Select Raw mocap data for Mecanim from the search result.
5. Click on the Import button to download the package.

This package is provided for free by Unity Technologies.
If we click on our robot's prefab right now we will see that it has a component attached to it called Animator. That's the Mecanim version of the Animation component for the Legacy system. However, unlike Animation, it doesn't store a list of clips; instead, it holds state machines in an Animator Controller object. In order to work with Mecanim, we need to create an object:

1. Click on Create in the Project window.
2. Select Animator Controller.
3. Name it CharacterController.
4. Assign it to the Controller parameter of the Animator component.
5. Make sure that Apply Root Motion is checked.

**Animation states**

Double-click on the created CharacterController or navigate to Window | Animator in the top menu—it will take us to the Animator window where all the magic is happening.

The Mecanim animation system is based around the idea of separating character behavior into different states with assigned animation clips to play while in any particular state. The transitions between states are defined by the set of conditions and are controlled by the Parameters.
Using Mecanim for Advanced Animations

States
Right click on the empty space of the Animator window and navigate to Create State | Empty.

<table>
<thead>
<tr>
<th>Create State</th>
<th>Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Sub-State Machine</td>
<td>From Selected Clip</td>
</tr>
<tr>
<td>Paste</td>
<td>From New Blend Tree</td>
</tr>
<tr>
<td>Copy current StateMachine</td>
<td></td>
</tr>
</tbody>
</table>

This created our first state, to which we can now assign an animation clip by:

1. Selecting Create New State
2. Renaming the state to Idle (in the Inspector)
3. Assigning the Idle_Neutral_1 animation clip (Raw Mocap | Animations | Idle | Idle_Neutral_1) to the Motion parameter in the Inspector window

Done, if we just hit Play, we will see our character playing the Idle_Neutral_1 animation in the scene window.

Let's take a step back for a second and see why this is happening.

The first created state is automatically assigned by default and is given an orange color. Try to create a new state by selecting Run_Impulse animation clip in the Project window and navigating to Create State | From Selected Clip. This will create a state; assign it to a selected Run_Impulse animation clip and rename it accordingly. Alternatively, we can just drag and drop animation clips from the Project to Animator window.

We can choose to make the Run_Impulse state as the default by right-clicking on it and selecting Set as Layer Default State.
Now we have two animation states, but how do we make a transition from one to the other? The short answer to that is, we have to create one:

1. Right click on the **Idle** state.
2. Select **Make Transition**.
3. Left-click on the **Run_Impulse** state.

Now these states are connected with a one way transition that will switch from playing the **Idle_Neutral_1** animation to the **Run_Impulse** animation as soon as the transition conditions are met. By default, it's set to make transition as soon as one animation is about to finish playing, also known as **Exit Time**, then it will use the remaining time to blend it with the next animation clip. Give it a try; a character should now play the usual **Idle_Neutral_1** animation and then blend into the **Run_Impulse** animation.

### Parameters

The key to controlling animation transitions are custom parameters hidden under a **Parameters** tab at the top left corner of the **Animator** window.
Parameters are used to communicate between the animator and the scripts. Initial parameters can be set in the Animator window by pressing a + sign and selecting a parameter type to add. Then, programmers can reference this parameter through code and change it. The change in the parameter value will be used to trigger the animation transition by conditions set by the animator. So you see, developers only need to use code to change the value of the parameters, the actual management is controlled by the conditions set in the Animator window.

Parameter types
As of Unity V 5.01, we have access to four parameter types, which are as follows:

- **New Int**: These are whole and negative numbers, no fractions
- **New Float**: These are decimal numbers
- **New Bool**: These are the Boolean values of true and false
- **New Trigger**: This is the same as bool, but with an extra benefit of setting itself to false as soon as the transition is completed (we will use the trigger parameter as an example later)

Setting conditions for transitions
So how do we use these parameters to trigger a change in our animation?

Let's start with the most basic example, switching from standing to walking, or, in our case, from the Idle_Neutral_1 animation to the Run_Impulse animation. Our goal is, whenever our character's horizontal velocity is greater than zero, the character should start running; however, he should go back to standing still as soon as the velocity is set back to zero. Here is how we can achieve this:

1. Create a new parameter of a type Float.
2. Name it ZSpeed.

This parameter will be referenced by script, so make sure that you got the naming correct.

We now need to select the transition arrow from Idle to Run_Impulse as follows:

1. With the ZSpeed parameter created we can now set it under Conditions at the bottom of the Inspector window by selecting the transition arrow.
2. Set condition to ZSpeed being Greater than 0.1. This will allow us to have a smooth transition from one animation to another.
3. Uncheck **Has Exit Time** box

![Image of an animated state transitions](image)

But we aren't done yet, this is just a one way transition and we have yet to revert it to the **Idle** state:

1. Create a transition from the **Run_Impulse** state to **Idle**.
2. Set a condition for this transition for **ZSpeed** to be **Less** than 0.1.
3. Uncheck **Has Exit Time** box
4. Take the script **MoveForward** from the **Chapter 6** folder and assign it as a component of the **Robot GameObject** in the Scene window.
Before we proceed, make sure to check the Loop Time box for the **Run_Impulse** animation by selecting the Run_Impulse file (the parent of the animation clip) and going to the Animation tab.

That will do. The **MoveForward** script will take care of the input and changing the **ZSpeed** value; the rest of it will be handled by the transitions that we have just set in the **Animator** window. See it for yourself, hit **Play** and use the **W** key to move forward.

### Blend trees

Having animations transition from one to another is all good and dandy; however, in our day and age, due to increase of motion range, blending multiple animations to create new sets of motions is in great demand. That’s what blend trees allow us to achieve.

### Overview

To better understand animation blending, let's compare it to the animation transition that we've seen in the previous example. With animation transition, we move from one animation to another with a short period of linear interpolation in between; animation blending is that short period of interpolation. Blend trees allow us to set multiple animations and define when and how they are going to affect the character by controlling them with parameters.

The most basic example of the use of the blend trees will be creating a smooth transition from walking to running. If you've ever played a third person game with a gamepad, you might have noticed how smoothly the character goes from walking to running as you slowly tilt the joystick forward — that's the exact feel that we can achieve using blend trees.
Creating a blend tree

Think of the blend trees as a more advanced version of animation states that we've used recently. Their creation process is very similar as well:

1. Right-click on the empty area of the Animator window
2. Navigate to Create State | From New Blend Tree

This will create a new block that is deceptively similar to the usual animation state.

Blend tree transitions work in the exact same way as the animation state transition, so much that we can simply convert our previously created Run_Impulse state into a Run blend tree by:

1. Right-clicking on the Run_Impulse state.
2. Selecting Create New Blend Tree in State.

Using the latter method will allow us to preserve all transition conditions we've built before.

You might gain an impression that using the blend trees should be the default way to go. That is however, not true, you only need to rely on them for animation blending, otherwise, stick to using states, they are much simpler.
Inside the blend tree

Double-clicking on the Blend Tree will take us inside of it, to where all the blending happens.

As you can see, we already have a slider with a ZSpeed parameter control. That is partially because that is the only parameter we have available at the moment. This parameter will be controlling the blend between the animations. Now, with Blend Tree selected, let's take a quick look at the Inspector view:

The Parameter field specifies the parameter that will be controlling the animation blending (ZSpeed, in our case).

Motion contains the list of animation clips and blend trees that we can blend between:

1. Hit the + sign of the Motion list and add two motion fields.
2. Assign the WalkFWD and Run_Impulse animation clips to them (WalkFWD at the top and Run_Impulse at the bottom).
In the Animator window, you should now get something like the tree diagram on this figure:

![Tree diagram showing blend tree with states WalkFWD and Run_Impulse and parameter ZSpeed](image)

Notice how the WalkFWD state is a little bit brighter than the Run_Impulse state, with the ZSpeed parameter set at 0, and switches as soon as we bring it to a max of 1. That is the sign of blending, with the dominant clip being brighter. Not only that, but even the connecting cord color changes from blue to white based on which animation clip is currently dominant. This is to show how blend trees are going to work; the actual controls to these blends is in the Inspector window, that now contains more parameters with previously added clips.

![Parameter window with ZSpeed and threshold values](image)

Right below the blending Parameter, you will find a pyramid graph that maps added motions to the Parameter value. With two animations, we can clearly see that one of them is fully affecting the character at the ZSpeed value of 0, while linearly blending into a different animation by the time the parameter value reaches 1.
0 to 1 values are given to parameters by default; we can change that by modifying the maximum Threshold value of the motions. All the motions listed, share the animation weight scale, meaning that the threshold is a point at which a single animation gets a 100 percent of the weight while others have no effect at all. This means that there is a value of the animation Parameter at which only a single animation will be played. To change the Threshold field, make sure to uncheck the Automate Thresholds box right below it. Let's change the Threshold value of the Run_Impulse motion to 4 to better represent what's happening in the scene.

The aforementioned Automate Thresholds, is actually a very useful parameter to keep in mind. It's main purpose is to evenly distribute all motions in between their min and max Thresholds. For example, if we were to add an additional motion in between walking and running, set it's Threshold to 1 and check Automate Threshold, it will instantly jump to 2 (0, 2, 4).

But what is actually happening when you add more than two motions? How do they share the graph? The rules are simple, but they aren't very intuitive from the first glance, it goes like this:

- Each motion must have a Threshold at which it becomes the only motion affecting the model
- At any point in time, animations have to share the 100 percent effect on the model
- Each motion down the Motion list, should have a higher Threshold value than the one above

The first one is very easy to work around; if you don't wish a particular animation to be played without blending, just set its threshold at the point that the animation Parameter won't reach.

The second one uses the same principal as, let's say, vertex weight painting; at any point in time, no animation or group of animations can affect the model more than 100 percent; if there are two animations and one of them is affecting the model at 80 percent, the affect of the other animation can't be anything but 20 percent, or there is no blending.

The third is more of a recommendation, if you try to force a lower threshold, it will overlap with over motions and the results will be unpleasant to look at; this also makes sure that you don't have more than two active motions at a time.
But what if there is an actual need to blend more than two motions? Is there a way to do that? Certainly there is, but it might become a little bit hard to manage. The way to do that is by adding another Blend Tree to the Motion list instead of the Motion field.

Other blend tree options
There are two other parameters in the motion list that are worth mentioning, next to the threshold:

- A clock sign \( \mathbb{C} \) is an animation speed parameter and is particularly useful if you are trying to create a reverse motion, such as creating a landing motion from a jumping motion
- Mirror animation \( \mathbb{M} \) is available only for the humanoid rigs and allows you to mirror animation from one side of the body to another using an avatar

There are also two functions available at the very bottom:

Compute Thresholds is an interesting one. It allows you to automatically compute the threshold values based on the motion at the root of the animation. This basically means that if you have a walking and running animation that moves the character, you can use the Speed parameter from the drop-down menu to control them and set up thresholds based on the speed at which the character moves. That is, however, completely useless if there is no root motion. The parameters available are:

1. Speed (the magnitude of velocity)
2. Velocity X
3. Velocity Y
4. Velocity Z
5. Angular Speed (Rad)
6. Angular Speed (Deg)

**Adjust Time Scale**
If you've added a Blend Tree to the motion list, delete it for this parameter to show up. Choosing Homogeneous Speed will allow you to rescale the speed of animations based on the minimum and maximum value set in the motion list. Naturally, some animations can have different speed and length; however, blending has to find a compromise and readjust them so that all motions are played in a single blending. Reset Time Scale will return the value of the animation speed to 1.

**Layers**
Unlike animation blending, which is a transition from one animation to another, there is a demand for combining different animations to create new ones. What if we have an attack animation and a running animation, but we don't have the attack while running animation? Thankfully, we don't have to animate every single case scenario, and can choose to combine them instead by using animation layers.

**Creation**
The Layer tab is located at the top right corner of the Animator window; it allows us to add and remove layers by hitting the + sign or navigating to Right Click | Delete as well as modify them.

As you can see, there are three parameters to control layers:

- The Weight slider controls the effect of each layer on the model.
- Mask is the key component of making the combination of layers and animation combining work. By creating an AvatarMask with upper body parts enabled, we can make them follow animation instructions given in this particular layer.
• **Blending** is a type of combining that we are planning to use; there are two to choose from:
  
  - **Override** will completely ignore any previous layers and overwrite motions with those from this layer for body parts specified in the Avatar Mask
  - **Additive** will add the motion on top of previous layers and their motions

The **Weight** parameter values are important to properly control **Blending**.

**Hand waving on move motion**

To combine the hand waving and walking animations, we need to create an AvatarMask first with only arms and head being affected, as follows:

1. Create a new AvatarMask via the Create drop-down menu of the Project window.
2. Call it WaveMask.
3. Under the Humanoid property of the WaveMask in the Inspector window, uncheck everything but left hand, arm, and arm IKs.
4. Return to the Animator window and create a new layer called Wave under the Layer tab.
5. Set **Weight** parameter of the Wave layer to 1.
Assign the newly created WaveMask to the Mask property of the Wave layer.

We can now enter the layer by clicking on it and setting up the state for it.

Let's talk a little bit about how we communicate on it and setting up the state for it. There is no transition from one layer to another and every layer has to have its own state system that will work independently from other layers. As such, our newly created Wave layer will theoretically be in two possible states, the waving state and the passive state, which is all we require of it. Following this logic, we will need to create two states:

- The first state will be called Passive and should contain no animation clip and be set to default
- The second state will be called Waving and will contain Robot Armature | Robot_ArmatureAction Animation Clip that was imported with our robot
To control the state transition, we will add a new parameter of a trigger type and call it \textit{Waved}. \textit{Waved} is controlled by code and will be enabled by a click; which transits us to the \textit{Waving} state and uncheck itself immediately (unlike \textit{bool}, we don’t need to bother disabling trigger via code). We also require two transitions from and to the \textbf{Passive} state:

- From \textbf{Passive} to \textbf{Waving} will be controlled by the \textit{Waved} trigger, for that we need to list it under \textbf{Conditions}

- From \textbf{Waving} to \textbf{Passive} should be done as soon as the animation has finished playing, for that, we leave the default \textbf{Has Exit Time} under the transition \textbf{Conditions}

That’s about it. With everything set correctly, we should be able to walk/run forward and wave with our hand.

\section*{2D blending}

Let’s get back to blending for just a second, or two. So far, we’ve looked into a blending mode controlled by a single parameter to make our walk to run transition possible; however, apart from walking in a straight line, we could also benefit from smoothly turning right and left instead of rapidly switching directions. To control this, we require an input from both the vertical and horizontal axes, there are twice as many parameters. Thankfully, Unity has got us covered here by providing 2D blending with various variations to suit our needs.
Blending variations

Looking at the blend tree in the Inspector window, you probably have noticed that we’ve omitted speaking about a parameter called Blend Type. This is done on purpose, as most blend types work in a similar fashion, as default 1D blending that we’ve looked at, with a layer of their own complexity deriving from working with twice as many parameters. So if, by any chance, you didn't fully grasp how 1D blending works, I would strongly recommend you go back and look at the examples provided with this chapter, along with the official documentation that comes with Unity (by clicking on the bookmark next to the BlendTree name in the Inspector window), as the topic we are about to discuss will heavily rely on the previous material.

Feel confident? Good, let's move on!

The drop down menu of the Blend Type parameter contains four options, as follows:

1D: The blending is controlled by one parameter, the one we've already looked at.

2D Simple Directional: This is the motion that is controlled by two parameters that represent direction. Taking walking as the example, we can use vertical and horizontal speeds as two directional parameters to control the single motion of walking. With this particular 2D blending type, we are limited to a single motion for single direction; therefore, no transition to running is possible.

2D Freeform Directional: This is just like the previous one, but with no single motion restriction. Walking | Jogging | Running | Sprinting: You name it and it can be incorporated into this blending type.

2D Freeform Cartesian: This type is better used with controlling parameters that don't represent directions, for something like angles.

Direct: This allows you to directly control the blend parameter. It also allows you to assign a unique parameter to each motion for it to be controlled manually, which is perfect for facial expressions.
It’s all about the dots

In the following example, we will use 2D Freeform Directional Blend Type. We will require a second parameter, which we will call XSpeed of a type Float and a 3rd motion in a Motion list left empty. Your Inspector window should now look similar to the one on the following figure:

Don’t worry about getting the exact color grading, it will change based on the selected element

At first glance, this looks nothing like the simple graph from 1D blending; however, it is actually quite similar to, if not better than, a representation of motion transition.

Take a look at the Motion list. Instead of the familiar threshold, we find Pos X and Pos Y, which are in fact, the thresholds of the first and the second controlled parameters (ZSpeed is represented by the X axis and XSpeed by the Y axis of the diagram). X and Y motion positions are projected on the 2D plane and represented by the blue icons.

Here is how the mapping works. The origin is located in the center of the diagram with X values increasing from left to right, while Y values do the same from the bottom to the top.
By clicking on any blue icon, you will see a motion that it corresponds to, highlighted in the Motion list or vice versa; selecting the motion from the list will highlight it on the diagram.

By selecting different motions, you will notice that the color of the diagram changes. The color indicates the weight of the specific motion in certain ranges of X and Y values, with saturation of the blue color indicating the weight influence. With the Run_Impulse Motion selected, you can see, on the diagram, that from its threshold position (2,0), it has full control over the animation; however, it gains less influence when closer to the origin, at which the WalkFWD motion threshold is set.

The grey icon represents the motion list items that don't have animation clips selected for them (which is the empty clip that we've just created).

The red icon is our current value of two controlled parameters. You can change the values of ZSpeed and XSpeed on the Blend Tree block and see it changing its position accordingly; or you can simply drag it within the diagram yourself and see the character react to the changes in the Preview window. You can change the positions of blue and grey icons that way as well.
Last, but not least, there is a circle around the blue icons which is a visual representation of the weight of the particular motion that it is circled around; the bigger the influence, the bigger the radius of the circle. You can try to drag the red icon from one motion to another and see the radius of circles around blue icons change.

Hopefully, this overview will make this diagram a little bit easier to read and manage as we are about to take a look at how to use it to create a full walk/run cycle for our character.

Creating locomotion
Sounds like a little bit too much to take in, well, let's take it in a more digestible form, and take a step-by-step approach to create a working character locomotion:

1. Create a new animation controller and call it Robot_Locomotion.
2. Assign it to the Animator controller parameter of the robot.
3. Add the Locomotion script to the robot components (located in the Chapter 6 folder) and remove (or deactivate) MoveForward.

Setup
Here is a list of motions that we will aim to create:

- Walking (forward and backward)
- Strafing (right and left)
- Turning (while walking and on the spot)

This will allow us to make full use of animations that we have and will require minimum code support.

Regarding parameters, there are three that we will require:

- Float ZSpeed and XSpeed to control movement and rotation
- Bool Strafing to know when our character needs to strafe and to avoid conflict with the walking cycle

Add these parameters to the previously created Robot_Locomotion animation controller, the script will take care of the input and will change parameters accordingly, so we don't have to worry about them.
Using Mecanim for Advanced Animations

**Walking**

We will utilize a blend tree to help us create a walking cycle.

Create a new **Blend Tree** and rename it; **Locomotion**.

To control this **Blend Tree**, we will require two parameters, **ZSpeed** and **XSpeed**. One of the parameters is directional (**ZSpeed**), but the other one is controlling the rotation of the character (**XSpeed**) making **2D Freeform Cartesian Blend Type** a logical choice.

As for **Motion** fields, there are 5 that we will require with the following animations clips:

1. Idle_Neutral_1
2. WalkFWD
3. WalkFWD
4. SprintForwardTurnRight_NtrlWide
5. SprintForwardTurnRight_NtrlWide

Having two walking animation clips is not a mistake as it might seem, we will use the same clip twice; for forward and backward motions, by changing the animation speed. To utilize the same animation of turning while moving forward, we will have to mirror one of them by checking the mirror checkbox.
For the parameter values, refer to the following figure:

Walking is mapped to the W, A, S, and D buttons; try testing it right now.

There are some modifications that need to be applied to the animations in order to avoid some of the issues.

For **Idle_Neutral_1** and **WalkFWD**, check the following boxes:

1. **Loop Time**
2. **Loop Pose**
3. **Bake Into Pose** under **Root Transform Rotation**

The first two are expected; we want our idle animation to loop over and over again. The latter checkbox makes sure that the rotation done by the animation is not affecting the GameObjects transform, to avoid the camera shaking while the character is playing this animation.
Using Mecanim for Advanced Animations

For \texttt{WalkForwardTurnRight\_NtrlWide}:

1. \textbf{Loop Time} and \textbf{Loop Pose} need to be checked
2. \textbf{Cycle Offset} needs to be set to 0.5 for smooth looping
3. \textbf{Start} and \textbf{End} need to be set to 16 and 362, respectively

Since we are using mocap data there are plenty of motions on animations that we don't need to create proper looping. The \textbf{Start} and \textbf{End} parameters allow us to isolate that particular region that will give the best results for looping.

\textbf{Turning}

To turn animations, we will take a fraction of the \texttt{Idle\_NeutralTO45IdleTONeutralIdle} clip and create a new clip from it called \texttt{Idle\_To\_Right} in the \textbf{Clips} list of the animation settings. The new clip will start from 280 and end at 315 with \textbf{Loop Time} and \textbf{Loop Pose} checked.

![Clips and settings figure]

Two \texttt{Idle\_To\_Right} clips will be added to the \textbf{Locomotion} blend tree with the parameters distribution displayed in the following figure. To eliminate small inconsistencies between the \texttt{WalkFWD} and \texttt{WalkForwardTurnRight\_NtrlWide} animations, we are going to speed up the latter to 1.1 (up from 1)
With this our Locomotion blend tree will be complete and we can move to adding strafing.

**Strafing**

We have two animations for strafing right and left; they will be controlled by XSpeed and Strafing bool, which will be enabled whenever we press the Q or E buttons on a keyboard. This might also be a perfect opportunity to introduce the Sub-State Machine.

**A sub-state machine**

Sub-State is essentially a container for other states and a perfect opportunity to group multiple state machines to preserve space and improve readability.

A sub-state machine is created in the same way as the empty state:

1. Right-click on the empty space of the Animator window.
2. Select Create Sub-State Machine from the drop-down menu.
3. Call the sub-state machine Strafe.
With the sub-state machine created, you may now enter it with a double-click. The environment inside the sub-state machine works exactly as it does on the inside; we can safely create new states, blend trees, and even more sub-states inside. In fact, let's do just that by creating a couple of states to represent strafing right and left—right_strafe_walking and left_strafe_walking.

Modifying a strafing clip
Just like with the turning animation, we will require a fraction of the existing animation, and then need to create a new clip from it. For that we will use Strafe_90HipsLeftFaceFwd:

1. Create a new clip called Strafe_Right.
2. Set Start and End to 90 and 120.
3. Check Loop Time and Loop Pose.
4. Check Bake Into Pose of the Root Transform Rotation.
5. Set Offset of the Root Transform Rotation to -20.8.

The reason for changing the offset is to make this animation more or less loopable. Right now, the animation will move the robot making the start and the end of the animation different; offset will help us to align the start and the end of the clip in a slightly better way.

Transition from and to sub-state
In case you are wondering, the sub-states don't interfere with our ability to translate from one state to another. The (Up) Base Layer block that exists in all sub-states will have all transition lines distributed to selected states and vice versa; you may create transitions to any state or blend tree outside of the sub-state by simply creating a transition like we normally did and selecting the one you are trying to connect to, from a drop-down menu.
With that in mind, let's create transitions from and to both Strafe states:

1. Create the transition from Locomotion to right_strafe_walking, that should be controlled by the Strafing bool being true, and ZSpeed being greater than 0.1.
2. Create a transition from right_strafe_walking to Locomotion that will be controlled by the Strafe bool being false.
3. Do the same thing for left_strafe_walking with the only difference being, that ZSpeed should be less than -0.1.
4. Uncheck Has Exit Time on all four transitions
5. Assign the Strafe_Right animation clip to both strafing states
6. Make sure that Mirror is checked in the left_strafe_walking state.

Lastly we are going to add a script to the camera, that will make it follow our character from behind:

1. Find CameraFollow.js inside the Chapter 6 folder
2. Add it to the MainCamera GameObject in the Hierarchy window
3. Drag and drop the **Robot** GameObject from the scene to the **Char Obj** parameter of the script

![Inspector window showing the Camera Follow (Script) component](image)

Other parameters can be adjusted to your liking.

That's about it, you now should have an animator controller managing your character while walking, turning, and strafing with minimum scripting support.

**Summary**

This concludes the introduction to the Mecanim animation system for Unity 5. We've looked into the basic functionality, constructed a motion cycle from scratch and tried out different approaches based on the available animation clips and character rigs. But there is a lot more to be learned if you are really interested in this tool and want to get the best out of it. Programming, despite delegating most of its work to Mecanim, still makes a vital contribution, by setting parameters and supporting more advanced features such as IKs, so make sure to communicate to your programmers constantly if you, yourself, aren't in charge of the code.

In the next chapter, we will continue to look into other features available in Unity that will make our level shine (literally) and relay best practices via tutorials and discussions.
Where to buy this book

You can buy Building Levels in Unity from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.