2
Finite State Machines and You

In this chapter, we'll expand our knowledge about the FSM pattern and its uses in games and learn how to implement it in a simple Unity game. We will create a tank game with sample code, which comes with this book. We'll be dissecting the code and the components in this project. The topics we'll cover are as follows:

- Understanding Unity's state machine features
- Creating our own states and transitions
- Creating a sample scene using examples

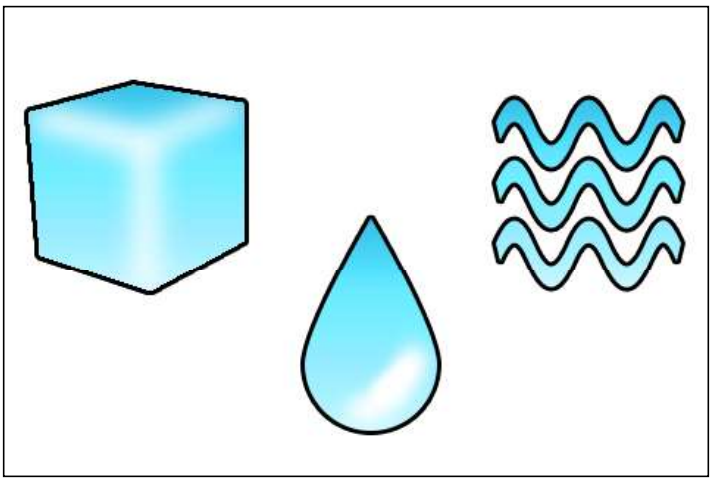
Unity 5 introduced state machine behaviors, which are a generic expansion of the Mecanim animation states that were introduced in the 4.x cycle. These new state machine behaviors, however, are independent of the animation system, and we will learn to leverage these new features to quickly implement a state-based AI system.

In our game, the player will be able to control a tank. The enemy tanks will be moving around in the scene with reference to four waypoints. Once the player tank enters their visible range, they will start chasing us and once they are close enough to attack, they'll start shooting at our tank agent. This simple example will be a fun way to get our feet wet in the world of AI and state FSMs.

Finding uses for FSMs

Though we will primarily focus on using FSMs to implement AI in our game to make it more fun and interesting, it is important to point out that FSMs are widely used throughout game and software design and programming. In fact, the new system in Unity 5 that we'll be using was first used in the Mecanim animation system.

We can categorize many things into states in our daily lives. The most effective patterns in programming are those that mimic the simplicity of real-life designs, and FSMs are no different. Take a look around and you'll most likely notice a number of things in one of any number of possible states. For example, is there a light bulb nearby? A light bulb can be in one of two states: on or off. Let's go back to grade school for a moment and think about the time when we were learning about the different states a matter can be in. Water, for example, can be solid, liquid, or gaseous. Just like in the FSM pattern in programming, where variables can trigger a state change, water's transition from one state to another is caused by heat.



The three distinct states of water

Though there are no hard rules beyond these of our own implementation in programming design patterns, it is a characteristic of FSMs to be in one and only one state at a time. With that said, transitions allow for a "hand-off" of sorts between two states, just like ice slowly melts into water. Additionally, an agent can have multiple FSMs, driving any number of behaviors, and states can even contain state machines of their own. Think Christopher Nolan's *Inception*, but with state machines instead of dreams.

Creating state machine behaviors

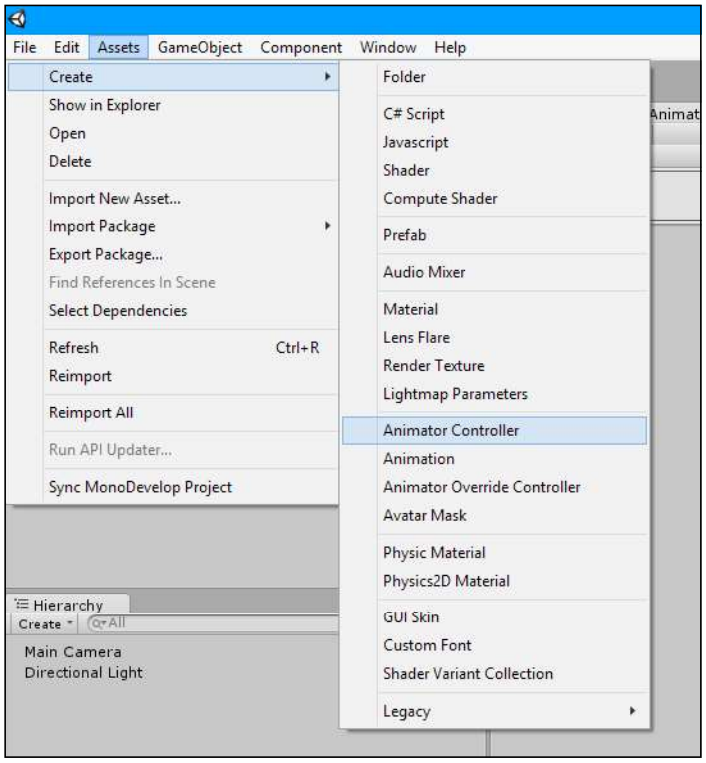
Now that we're familiar with the concept of a state machine, let's get our hands dirty and start implementing our very own.

As of Unity 5.0.0f4, state machines are still part of the animation system, but worry not, they are flexible, and no animations are actually required to implement them. Don't be alarmed or confused if you see code referencing the `Animator` component or the `AnimationController` asset as it's merely a quirk of the current implementation. It's fathomable that Unity will address this in a later version, but the concepts will likely not change.

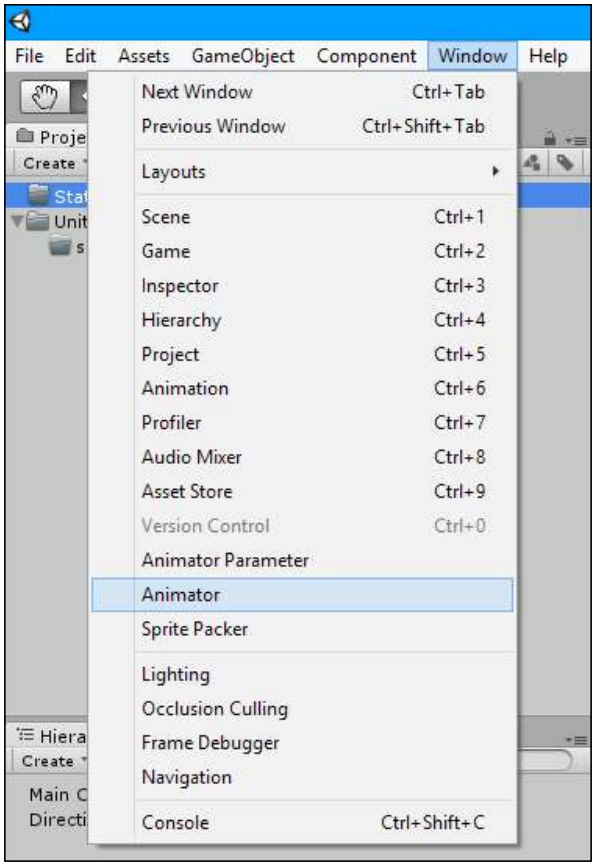
Let's fire up Unity, create a new project, and get to it.

Creating the AnimationController asset

The `AnimationController` asset is a type of asset within Unity that handles states and transitions. It is, in essence, an FSM, but it also does much more. We'll focus on the FSM portion of its functionality. An animator controller can be created from the **Assets** menu, as shown in the following image:



Once you create the animator controller, it will pop up in your project assets folder, ready to be named. We'll name it `TankFsm`. When you select the animator controller, unlike most other asset types, the hierarchy is blank. That is because animation controllers use their own window. You can simply click on **Open** in the hierarchy to open up the **Animator** window, or open it in the **Window** menu, as you can see in the following screenshot:

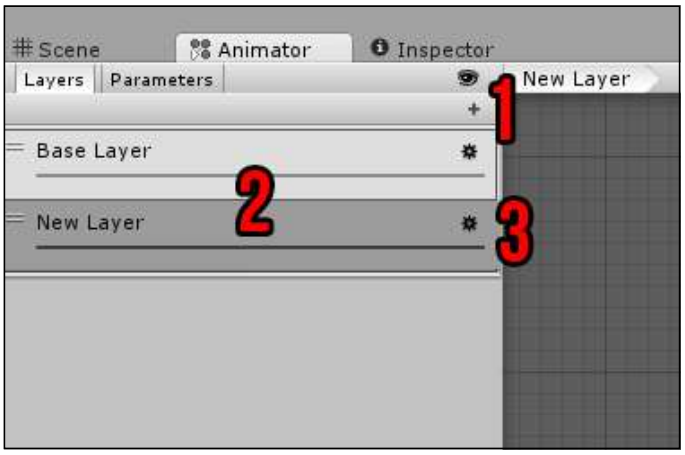


Be sure to select **Animator** and not **Animation** as these are two different windows and features entirely.

Let's familiarize ourselves with this window before moving forward.

Layers and Parameters

Layers, as the name implies, allow us to stack different state machine levels on top of each other. This panel allows us to organize the layers easily and have a visual representation. We will not be doing much in this panel for now as it primarily relates to animation, but it's good to be familiar with it. Refer to the following screenshot of the window to find your way around the layers:

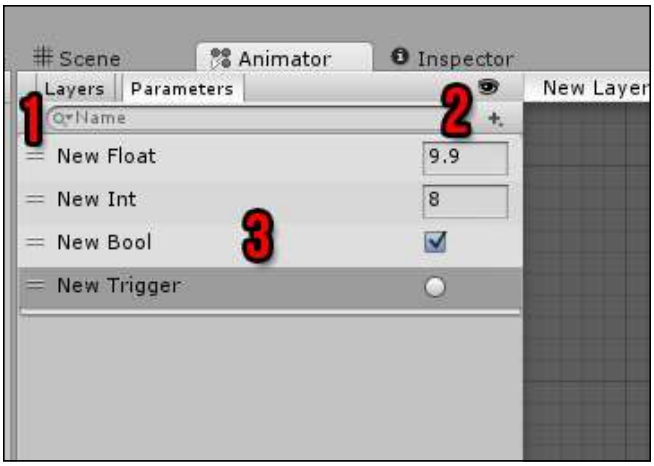


Here is a summary of the items shown in the previous screenshot:

- **Add layer**: This button creates a new layer at the bottom of the list.
- **Layer list**: These are the layers currently inside the animator controller. You can click to select a layer and drag-and-drop layers to rearrange them.
- **Layer settings**: These are animation-specific settings for the layer.

Second, we have the **Parameters** panel, which is far more relevant to our use of the animator controller. Parameters are variables that determine when to transition between states, and we can access them via scripts to drive our states. There are four types of parameters: `float`, `int`, `bool`, and `trigger`. You should already be familiar with the first three as they are primitive types in C#, but `trigger` is specific to the animator controller, not to be confused with physics triggers, which do not apply here. Triggers are just a means to trigger a transition between states explicitly.

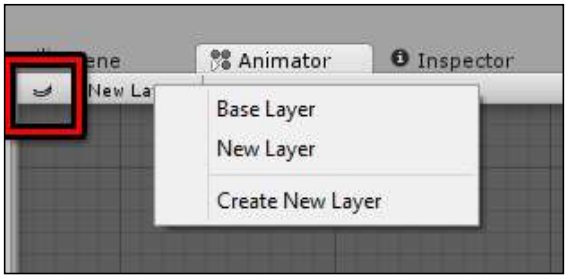
The following screenshot shows the elements in the **Parameters** panel:



Here is a summary of the items depicted in the previous screenshot:

- **Search**: We can quickly search through our parameters here. Simply type in the name and the list will populate with the search results.
- **Add parameter**: This button lets you add new parameters. When you click on it, you must select the parameter type.
- **Parameter list**: This is the list of parameters you've created. You can assign and view their values here. You can also reorder the parameters to your liking by dragging-and-dropping them in the correct order. This is merely for organization and does not affect functionality at all.

Lastly, there is an eyeball icon, which you can click to hide the **Layers** and **Parameters** panels altogether. When the panels are closed, you can still create new layers by clicking on the **Layers** dropdown and selecting **Create New Layer**:



The animation controller inspector

The animation controller inspector is slightly different from the regular inspector found throughout Unity. While the regular inspector allows you to add components to the game objects, the animation controller inspector has a button labeled **Add Behaviour**, which allows you to add a `StateMachineBehaviour` to it. This is the main distinction between the two types of inspectors, but apart from this, it will display the serialized information for any selected state, substate, transition, or blend tree, just as the regular inspector displays the data for the selected game object and its components.

Bringing behaviors into the picture

State machine behaviors are a unique new concept in Unity 5. While states existed, conceptually, in the original implementation of Mecanim, transitions were handled behind the scenes, and you did not have much control over what happened upon entering, transitioning, or exiting a state. Unity 5 addressed this issue by introducing behaviors; they provide a built-in functionality to handle typical FSM logic.

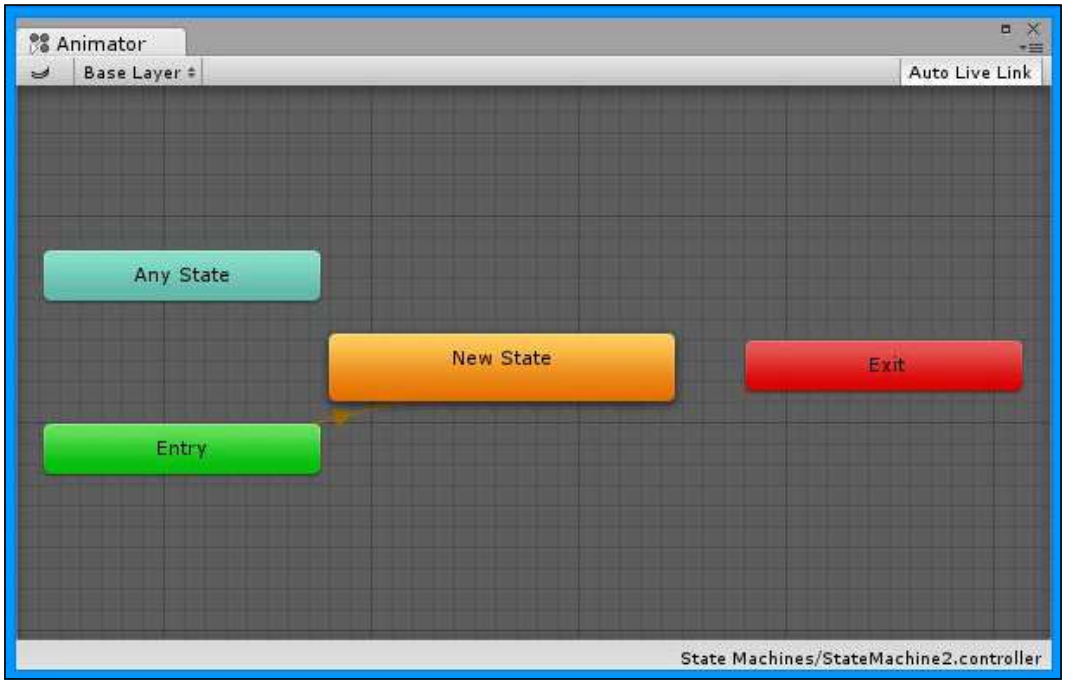
Behaviors are sly and tricky. Though their name might lead you to believe they are related to `MonoBehaviour`, do not fall for it; if anything, these two are distant cousins at best. In fact, behaviors derive from `ScriptableObject`, not `MonoBehaviour`, so they exist only as assets, which cannot be placed in a scene or added as a component to a GameObject.

Creating our very first state

OK, so that's not entirely true since Unity creates a few default states for us in our animator controller: **New State**, **Any State**, **Entry**, and **Exit**, but let's just agree that those don't count for now, OK?

- You can select states in this window by clicking on them, and you can move them by dragging-and-dropping them anywhere in the canvas.
- Select the state named **New State** and delete it by either right-clicking and then clicking on **Delete** or simply hitting the *Delete* key on your keyboard.

- If you select the **Any State**, you'll notice that you do not have the option to delete it. The same is true for the **Entry** state. These are required states in an animator controller and have unique uses, which we'll cover up ahead.



To create our (true) first state, right-click anywhere on the canvas and then select **Create State**, which opens up a few options from which we'll select **Empty**. The other two options, **From Selected Clip** and **From New Blend Tree**, are not immediately applicable to our project, so we'll skip these. Now we've officially created our first state.

Transitioning between states

You'll notice that upon creating our state, an arrow is created connecting the **Entry** state to it, and that its node is orange. Unity will automatically set default states to look orange to differentiate them from other states. When you only have one state, it is automatically selected as the default state, and as such, it is automatically connected to the entry state. You can manually select which state is the default state by right-clicking on it and then clicking on **Set as Layer Default State**. It will then become orange, and the entry state will automatically connect itself to it. The connecting arrow is a **transition connector**. Transition connectors allow us some control over how and when the transition occurs, but the connector from the entry state to the default state is unique in that it does not provide us any options since this transition happens automatically.

You can manually assign transitions between states by right-clicking on a state node and then selecting **Make Transition**. This will create a transition arrow from the state you selected to your mouse cursor. To select the destination of the transition, simply click on the destination node and that's it. Note that you cannot redirect the transitions though. We can only hope that the kind folks behind Unity add that functionality at a later point, but for now, you must remove a transition by selecting it and deleting it and then assigning an all-new transition manually.

Setting up our player tank

Open up the sample project included with this book for this chapter.

It is a good idea to group like assets together in your project folder to keep it organized. For example, you can group your state machines in a folder called `StateMachines`. The assets provided for this chapter are grouped for you already, so you can drop the assets and scripts you create during this chapter into the corresponding folder.

Creating the enemy tank

Let's go ahead and create an animator controller in your assets folder. This will be your enemy tank's state machine. Call it `EnemyFsm`.

This state machine will drive the tank's basic actions. As described earlier, in our example, the enemy can patrol, chase, and shoot the player. Let's go ahead and set up our state machine. Select the `EnemyFsm` asset and open up the **Animator** window.

Now, we'll go ahead and create three empty states that will conceptually and functionally represent our enemy tank's states. Name them `Patrol`, `Chase`, and `Shoot`. Once they are created and named, we'll want to make sure we have the correct default state assigned. At the moment, this will vary depending on the order in which you created and named the states, but we want the **Patrol** state to be the default state, so right-click on it and select **Set as Layer Default State**. Now it is colored orange and the **Entry** state is connected to it.

Choosing transitions

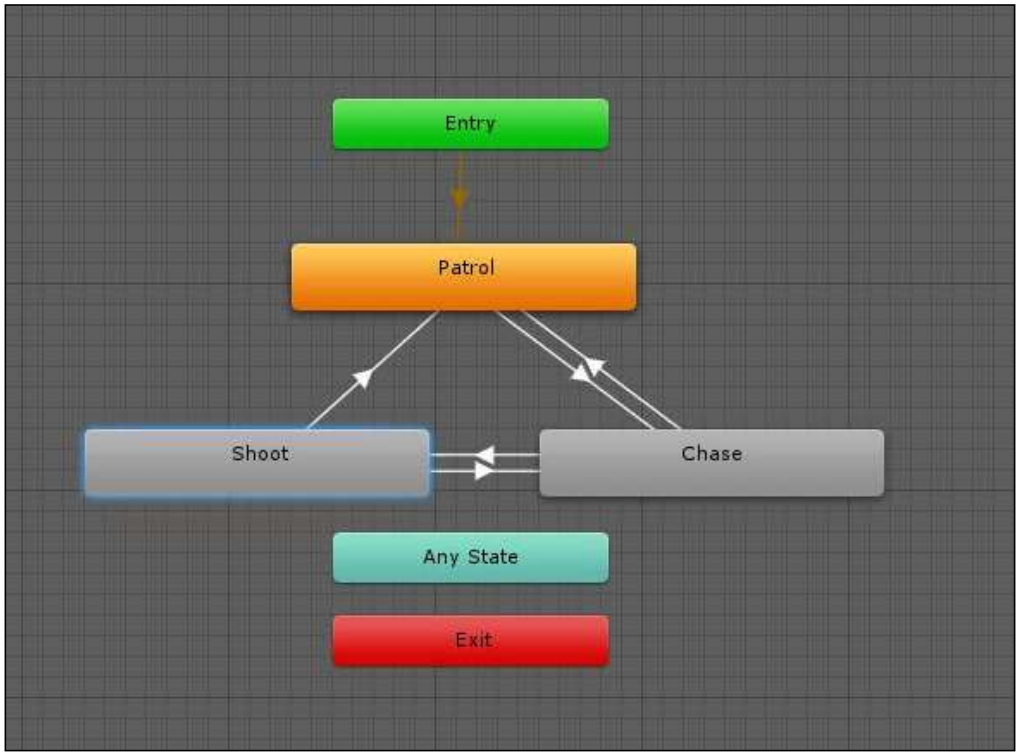
At this point, we have to make some design and logic decisions regarding the way our states will flow into each other. When we map out these transitions, we also want to keep in mind the conditions that trigger the transitions to make sure they are logical and work from a design-standpoint. Out in the wild, when you're applying these techniques on your own, different factors will play into how these transitions are handled. In order to best illustrate the topic at hand, we'll keep our transitions simple and logical:

- **Patrol**: From patrol, we can transition into chasing. We will use a chain of conditions to choose which state we'll transition into, if any.

  Can the enemy tank see the player? If yes, we go to the next step; if not, we continue with patrolling.

- **Chase**: From this state, we'll want to continue to check whether the player is within sight to continue chasing, close enough to shoot, or completely out of sight that would send us back into the patrol state.
- **Shoot**: Same as earlier, we'll want to check our range for shooting and then the line of sight to determine whether or not we can chase to get within the range.

This particular example has a simple and clean set of transition rules. If we connect our states accordingly, we'll end up with a graph looking more or less similar to this one:



Keep in mind that the placement of the nodes is entirely up to you, and it does not affect the functionality of the state machine in any way. You try to place your nodes in a way that keeps them organized so that you can track your transitions visually.

Now that we have our states mapped out, let's assign some behaviors to them.

Making the cogs turn

This is the part I'm sure you've been waiting for. I know, I've kept you waiting, but for good reason—as we now get ready to dive into coding, we do so with a good understanding of the logical connection between the states in our FSM. Without further ado, select our **Patrol** state. In the hierarchy, you'll see a button labeled **Add Behaviour**. Clicking this gives you a context menu very similar to the **Add Component** button on regular game objects, but as we mentioned before, this button creates the oh-so-unique state machine behaviors.

Go ahead and name this behavior `TankPatrolState`. Doing so creates a script of the same name in your project and attaches it to the state we created it from. You can open this script via the project window, or by double-clicking on the name of the script in the inspector. What you'll find inside will look similar to this:

```
using UnityEngine;
using System.Collections;

public class TankPatrolState : StateMachineBehaviour {

   // OnStateEnter is called when a transition starts and the state
machine starts to evaluate this state
   //override public void OnStateEnter(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
   //
   //}

   // OnStateUpdate is called on each Update frame between
OnStateEnter and OnStateExit callbacks
   //override public void OnStateUpdate(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
   //
   //}

   // OnStateExit is called when a transition ends and the state
machine finishes evaluating this state
   //override public void OnStateExit(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
   //
   //}

   // OnStateMove is called right after Animator.OnAnimatorMove().
Code that processes and affects root motion should be implemented here
   //override public void OnStateMove(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
   //
```

```
    //}

    // OnStateIK is called right after Animator.OnAnimatorIK(). Code
that sets up animation IK (inverse kinematics) should be implemented
here.
    //override public void OnStateIK(Animator animator,
AnimatorStateInfo stateInfo, int layerIndex) {
    //
    //}
}
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Before we begin, uncomment each method. Let's break it down step by step. Unity creates this file for you, but all the methods are commented out. Essentially, the commented code acts as a guide. Much like the methods provided for you in a `MonoBehaviour`, these methods are called for you by the underlying logic. You don't need to know what's going on behind the scenes to use them; you simply have to know when they are called to leverage them. Luckily, the commented code provides a brief description of when each method is called, and the names are fairly descriptive themselves. There are two methods here we don't need to worry about: `OnStateIK` and `OnStateMove`, which are animation messages, so go ahead and delete them and save the file.

To reiterate what's stated in the code's comments, the following things happen:

- `OnStateEnter` is called when you enter the state, as soon as the transition starts
- `OnStateUpdate` is called on each frame, after MonoBehaviors update
- `OnStateExit` is called after the transition out of the state is finished

The following two states, as we mentioned, are animation-specific, so we do not use those for our purposes:

- `OnStateIK` is called just before the IK system gets updated. This is an animation and rig-specific concept.
- `OnStateMove` is used on avatars that are set up to use root motion.

Another important piece of information to note is the parameters passed into these methods:

- The animator parameter is a reference to the animator that contains this animator controller, and therefore, this state machine. By extension of that, you can fetch a reference to the game object that the animator controller is on, and from there, you can grab any other components attached to it. Remember, the state machine behavior exists only as an asset, and does not exist in the class, meaning this is the best way to get references to runtime classes, such as mono behaviors.
- The animator state info provides information about the state you're currently in, however, the uses for this are primarily focus on animation state info, so it's not as useful for our application.
- Lastly, we have the layer index, which is an integer telling us which layer within the state machine our state is in. The base layer is index 0, and each layer above that is a number higher.

Now that we understand the basics of a state machine behavior, let's get the rest of our components in order. Before we can actually see these behaviors in action, we have to go back to our state machine and add some parameters that will drive the states.

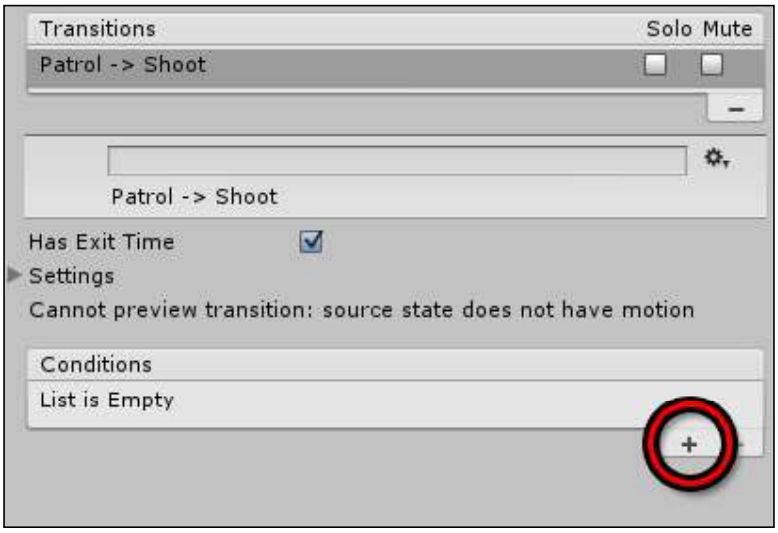
Setting conditions

We will need to provide our enemy tank with a few conditions to transitions states. These are the actual parameters that will drive the functionality.

Let's begin with the **Patrol** state. In order for our enemy tank to go from **Patrol** to **Shoot**, we need to be in range of the player, in other words, we'll be checking the distance between the enemy and the player, which is best represented by a float value. So, in your **Parameters** panel, add a float and name it `distanceFromPlayer`. We can also use this parameter to determine whether or not to go into the **Chase** state.

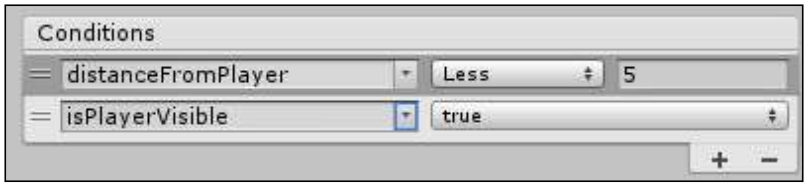
The **Shoot** state and the **Chase** state will share a common condition, which is whether or not the player is visible. We'll determine this via a simple raycast, which will in turn, tell us whether the player was in line-of-sight or not. The best parameter for this is a Boolean, so create a Boolean and call it `isPlayerVisible`. Leave the parameter unchecked, which means false.

Now we'll assign the conditions via the transition connectors' inspector. To do this, simply select a connector. When selected, the inspector will display some information about the current transition, and most importantly, the conditions, which show up as a list. To add a condition, simply click on the **+** (plus) sign:



Let's tackle each transition one by one.

- Patrol to Chase
    - `distanceFromPlayer < 5`
    - `isPlayerVisible == true`



The patrol to chase transition conditions

Chase to patrol gets a bit more interesting as we have two *separate* conditions that can trigger a transition. If we were to simply add two conditions to that transition, both would have to be evaluated to true in order for the transition to occur, but we want to check whether the player is out of range or they are out of sight. Luckily, we can have multiple transitions between the same two states. Simply add another transition connection as you normally would. Right-click on the **Chase** state and then make a transition to the **Patrol** state. You'll notice that you now have two transitions listed at the top of the inspector. In addition, your transition connection indicator shows multiple arrows instead of just one to indicate that there are multiple transitions between these two states. Selecting each transition in the inspector will allow you to give each one separate condition:

- Chase to Patrol (A)
    - `distanceFromPlayer > 5`
- Chase to Patrol (B)
    - `isPlayerVisible == false`
- Chase to Shoot
    - `distanceFromPlayer < 3`
    - `isPlayerVisible == true`
- Shoot to Chase
    - `distanceFromPlayer > 3`
    - `distanceFromPlayer < 5`
    - `isPlayerVisible == true`
- Shoot to Patrol (A)
    - `distanceFromPlayer > 6`
- Shoot to Patrol (B)
    - `isPlayerVisible == false`

We now have our states and transitions set. Next, we need to create the script that will drive these values. All we need to do is set the values, and the state machine will handle the rest.

Driving parameters via code

Before going any farther, we'll need a few things from the assets we imported earlier in the chapter. For starters, go ahead and open the DemoScene folder of this chapter. You'll notice the scene is fairly stripped down and only contains an environment prefab and some waypoint transforms. Go ahead and drop in the `EnemyTankPlaceholder` prefab into the scene.

You may notice a few components that you may or may not be familiar with on the EnemyTank. We'll get a chance to thoroughly explore NavMesh and NavMeshAgent in *Chapter 4*, *Finding Your Way*, but for now, these are necessary components to make the whole thing work. What you will want to focus on is the **Animator** component which will house the state machine (animator controller) we created earlier. Go ahead and drop in the state machine into the empty slot before continuing.

We will also need a placeholder for the player. Go ahead and drop in the `PlayerTankPlaceholder` prefab as well. We won't be doing much with this for now. As with the enemy tank placeholder prefab, the player tank placeholder prefab has a few components that we can ignore for now. Simply place it in the scene and continue.

Next, you'll want to add a new component to the `EnemyTankPlaceholder` game object—the `TankAi.cs` script, which is located in the `Chapter 2` folder. If we open up the script, we'll find this inside it:

```
using UnityEngine;
using System.Collections;

public class TankAi : MonoBehaviour {
    // General state machine variables
    private GameObject player;
    private Animator animator;
    private Ray ray;
    private RaycastHit hit;
    private float maxDistanceToCheck = 6.0f;
    private float currentDistance;
    private Vector3 checkDirection;

    // Patrol state variables
    public Transform pointA;
    public Transform pointB;
    public NavMeshAgent navMeshAgent;

    private int currentTarget;
```

```
    private float distanceFromTarget;
    private Transform[] waypoints = null;

    private void Awake() {
        player = GameObject.FindWithTag("Player");
        animator = gameObject.GetComponent<Animator>();
        pointA = GameObject.Find("p1").transform;
        pointB = GameObject.Find("p2").transform;
        navMeshAgent = gameObject.GetComponent<NavMeshAgent>();
        waypoints = new Transform[2] {
            pointA,
            pointB
        };
        currentTarget = 0;
        navMeshAgent.SetDestination(waypoints[currentTarget].
position);
    }

    private void FixedUpdate() {
        //First we check distance from the player
        currentDistance = Vector3.Distance(player.transform.position,
transform.position);
        animator.SetFloat("distanceFromPlayer", currentDistance);

        //Then we check for visibility
        checkDirection = player.transform.position - transform.
position;
        ray = new Ray(transform.position, checkDirection);
        if (Physics.Raycast(ray, out hit, maxDistanceToCheck)) {
            if(hit.collider.gameObject == player){
                animator.SetBool("isPlayerVisible", true);
            } else {
                animator.SetBool("isPlayerVisible", false);
            }
        } else {
            animator.SetBool("isPlayerVisible", false);
        }

        //Lastly, we get the distance to the next waypoint target
        distanceFromTarget = Vector3.Distance(waypoints[currentTarget
].position, transform.position);
        animator.SetFloat("distanceFromWaypoint", distanceFromTarget);
    }

    public void SetNextPoint() {
```

```
            switch (currentTarget) {
                case 0:
                    currentTarget = 1;
                    break;
                case 1:
                    currentTarget = 0;
                    break;
            }
            navMeshAgent.SetDestination(waypoints[currentTarget].
    position);
        }
    }
```

We have a series of variables that are required to run this script, so we'll run through what they're for in order:

- `GameObject player`: This is a reference to the player placeholder prefab we dropped in earlier.
- `Animator animator`: This is the animator for our enemy tank, which contains the state machine we created.
- `Ray ray`: This is simply a declaration for a ray that we'll use in a raycast test on our `FixedUpdate` loop.
- `RaycastHit hit`: This is a declaration for the hit information we'll receive from our raycast test.
- `Float maxDistanceToCheck`: This number coincides with the value we set in our transitions inside the state machine earlier. Essentially, we are saying that we're only checking as far as this distance for the player. Beyond that, we can assume that the player is out of range.
- `Float currentDistance`: This is the current distance between the player and the enemy tanks.

You'll notice we skipped a few variables. Don't worry, we'll come back to cover these later. These are the variables we'll be using for our patrol state.

Our `Awake` method handles fetching the references to our player and animator variables. You can also declare the preceding variables as public or prefix them with the `[SerializeField]` attribute and set them via the inspector.

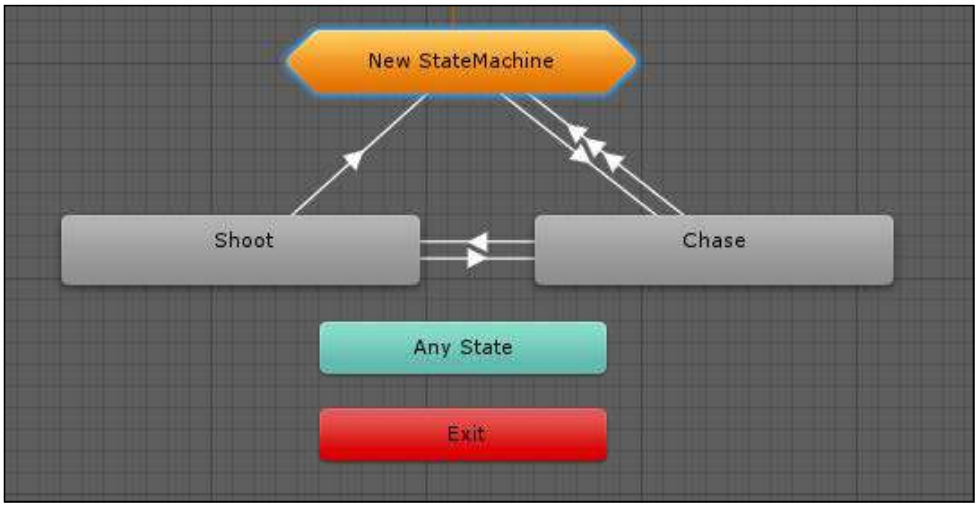
The `FixedUpdate` method is fairly straightforward; the first part gets the distance between the position of the player and the enemy tank. The part to pay special attention to is `animator.SetFloat("distanceFromPlayer", currentDistance)`, which passes in the information from this script into the parameter we defined earlier in our state machine. The same is true for the following section of the code, which passes in the hit result of the raycast as a Boolean. Lastly, it sets the `distanceFromTarget` variable, which we'll be using for the patrol state in the next section.

As you can see, none of the code concerns itself with how or why the state machine will handle transitions; it merely passes in the information the state machine needs, and the state machine handles the rest. Pretty cool, right?

Making our enemy tank move

You may have noticed, in addition to the variables we didn't cover yet, that our tank has no logic in place for moving. This can be easily handled with a substate machine, which is a state machine within a state. This may sound confusing at first, but we can easily break down the patrol state into substates. For our example, the Patrol state will be in one of the two substates: moving to the current waypoint and finding the next waypoint. A waypoint is essentially a destination for our agent to move toward. In order to make these changes, we'll need to go into our state machine again.

First, create a substate by clicking on an empty area on the canvas and then selecting **Create Sub-State Machine**. Since we already have our original Patrol state and all the connections that go with it, we can just drag-and-drop our **Patrol** state into our newly-created substate to merge the two. As you drag the Patrol state over the substate, you'll notice a plus sign appears by your cursor; this means you're adding one state to the other. When you drop the **Patrol** state in, the new substate will absorb it. Substates have a unique look; they are six-sided rather than rectangular. Go ahead and rename the substate to `Patrol`.



To enter a substate, simply double-click on it. Think of it as going in a level lower into the substate. The window will look fairly similar, but you will notice a few things: your **Patrol** state is connected to a node called (**Up) Base Layer**, which essentially is the connection out from this level to the upper level that the substate machine sits on, and the **Entry** state connects directly to the **Patrol** state.

Unfortunately, this is not the functionality we want as it's a closed loop that doesn't allow us to get in and out of the state into the individual waypoint states we need to create, so let's make some changes. First, we'll change the name of the substate to `PatrolEntry`. Next, we need to assign some transitions. When we enter this **Entry** state, we want to decide whether to continue moving to the current waypoint, or to find a new one. We'll represent each of the outcomes as a state, so create two states: `MovingToTarget` and `FindingNewTarget`, then create transitions from the **PatrolEntry** state to each one of the new states. Likewise, you'll want to create a transition between the two new states, meaning a transition from the MovingToTarget state to the FindingNewTarget state and vice versa. Now, add a new float parameter called `distanceFromWaypoint` and set up your conditions like this:

- PatrolEntry to MovingToTarget:
    - `distanceFromWaypoint > 1`
- PatrolEntry to FindingNewTarget:
    - `distanceFromWaypoint < 1`
- MovingToTarget to FindingNewTarget:
    - `distanceFromWaypoint < 1`

You're probably wondering why we didn't assign transition rule from the finding new target state to the MovingToTarget state. This is because we'll be executing some code via a state machine behavior and then automatically going into the MovingToTarget state without requiring any conditions. Go ahead and select the FindingNewTarget state and add a behavior and call it `SelectWaypointState`.

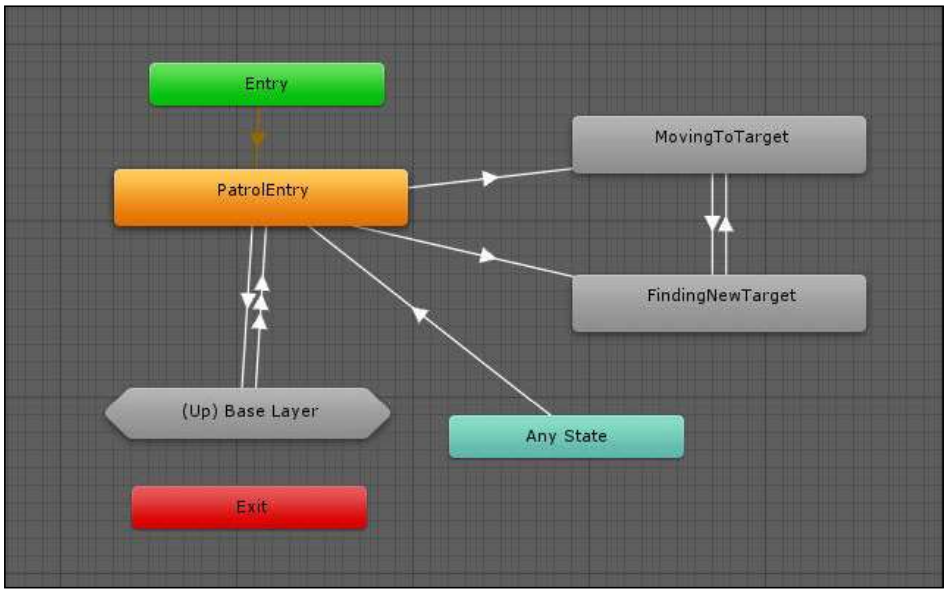
Open up the new script and remove all the methods, except for `OnStateEnter`. Add the following functionality to it:

```
TankAi tankAi = animator.gameObject.GetComponent<TankAi>();
tankAi.SetNextPoint();
```

What we're doing here is getting a reference to our `TankAi` script and calling its `SetNextPoint()` method. Simple enough, right?

Lastly, we need to redo our outgoing connections. Our new states don't have transitions out of this level, so we need to add one using the exact same conditions that our **PatrolEntry** state has to the **(Up) Base Layer** state. This is where **Any State** comes in handy—it allows us to transit from any state to another state, regardless of individual transition connections, so that we don't have to add transitions from each state to the **(Up) Base Layer** state; we simply add it once to the **Any State**, and we're set! Add a transition from the **Any State** to the **PatrolEntry** state and use the same conditions as the **Entry** state has to the **(Up) Base Layer** state. This is a work-around to not being able to connect directly from the **Any State** to the (**Up) Base Layer** state.

When you're done, your substate machine should look similar to this:



Testing

Now, all we have to do is hit play and watch our enemy tank patrol back and forth between the two provided waypoints. If we place the player in the editor in the enemy tank's path, we'll see the transition happen in the animator, out of the Patrol state, into the Chase state, and when we move the player out of range, back into the Patrol state. You'll notice our Chase and Shoot states are not fully fleshed out yet. This is because we'll be implementing these states via concepts we'll cover in *Chapter 3*, *Implementing Sensors*, and *Chapter 4*, *Finding Your Way*.

Summary

In this chapter, we learned how to implement state machines in Unity 5 using animator controller-based state machines for what will be our tank game. We learned about state machine behaviors and transitions between states. With all of these concepts covered, we then applied the simple state machine to an agent, thus creating our first artificially intelligent entity!

In the next chapter, we'll continue to build our tank game and give our agent more complex methods of sensing the world around it.