

8

How It All Comes Together

We've almost arrived at the end of our journey. We learned all the essential tools to implement fun AI in our Unity game. We stressed on this throughout the course of the book, but it's important to drive the point home: the concepts and patterns we learned throughout the book are individual concepts, but they can, and often should, be used in harmony to achieve the desired behavior from our AI. Before we say our goodbyes, we'll look at a simple tank-defense game that implements some of the concepts that we learned to achieve a cohesive "game", and I only say "game" because this is more of a blueprint for you to expand upon and play with. In this chapter, we will:

- Integrate some of the systems we've learned in a single project
- Create an AI tower agent
- Create our `NavMeshAgent` tank
- Set up the environment
- Test our sample scene

Setting up the rules

Our "game" is quite simple. While the actual game logic, such as health, damage, and win conditions, are left completely up to you, our example focuses on setting you up to implement your own tank-defense game.

When deciding on what kind of logic and behavior you'll need from your agent, it's important to have the rules of the game fleshed out beyond a simple idea. Of course, as you implement different features, those rules can change, but having a set of concepts nailed down early on will help you pick the best tools for the job.

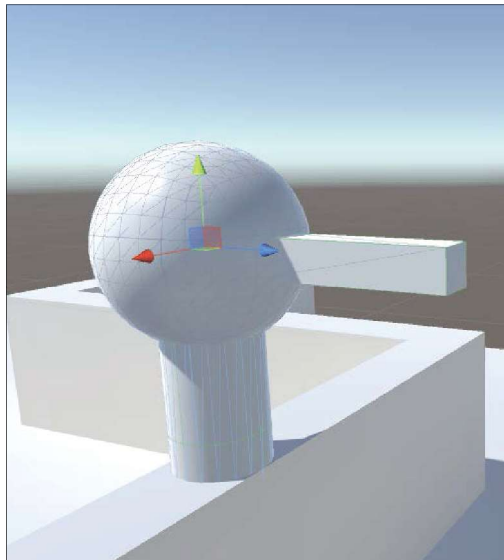
It's a bit of a twist on the traditional tower-defense genre. You don't build towers to stop an oncoming enemy; you rather use your abilities to help your tank get through a gauntlet of towers. As your tank traverses the maze, towers along the path will attempt to destroy your tank by shooting explosive projectiles at it. To help your tank get to the other side, you can use two abilities:

- **Boost:** This ability doubles up your tank's movement speed for a short period of time. This is great for getting away from a projectile in a bind.
- **Shield:** This creates a shield around your tank for a short period of time to block oncoming projectiles.

For our example, we'll implement the towers using a finite state machine since they have a limited number of states and don't require the extra complexity of a behavior tree. The towers will also need to be able to be aware of their surroundings, or more specifically, whether the tank is nearby so that they can shoot at it, so we'll use a sphere trigger to model the towers' field of vision and sensing. The tank needs to be able to navigate the environment on its own, so we use a NavMesh and NavMeshAgent to achieve this.

Creating the towers

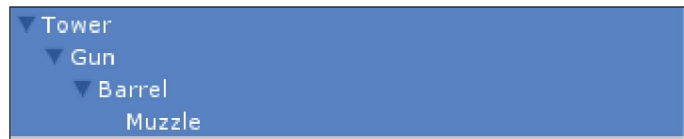
In the sample project for this chapter, you'll find a `Tower` prefab in the `Prefabs` folder. The tower itself is quite simple; it's just a group of primitives arranged to look like a cannon, as you can see in the following screenshot:



Our beautiful primitive shape tower

The barrel of the gun is affixed to the spherical part of the tower. The gun can rotate freely on its axis when tracking the player so that it can fire in the direction of its target, but it is immobile in any other way. Once the tank gets far enough away, the tower cannot chase it or reposition itself.

In the sample scene, there are several towers placed throughout the level. As they are prefabbed, it's very easy to duplicate towers, move them around, and reuse them between the levels. Their setup is not terribly complicated either. Their hierarchy looks similar to the following screenshot:

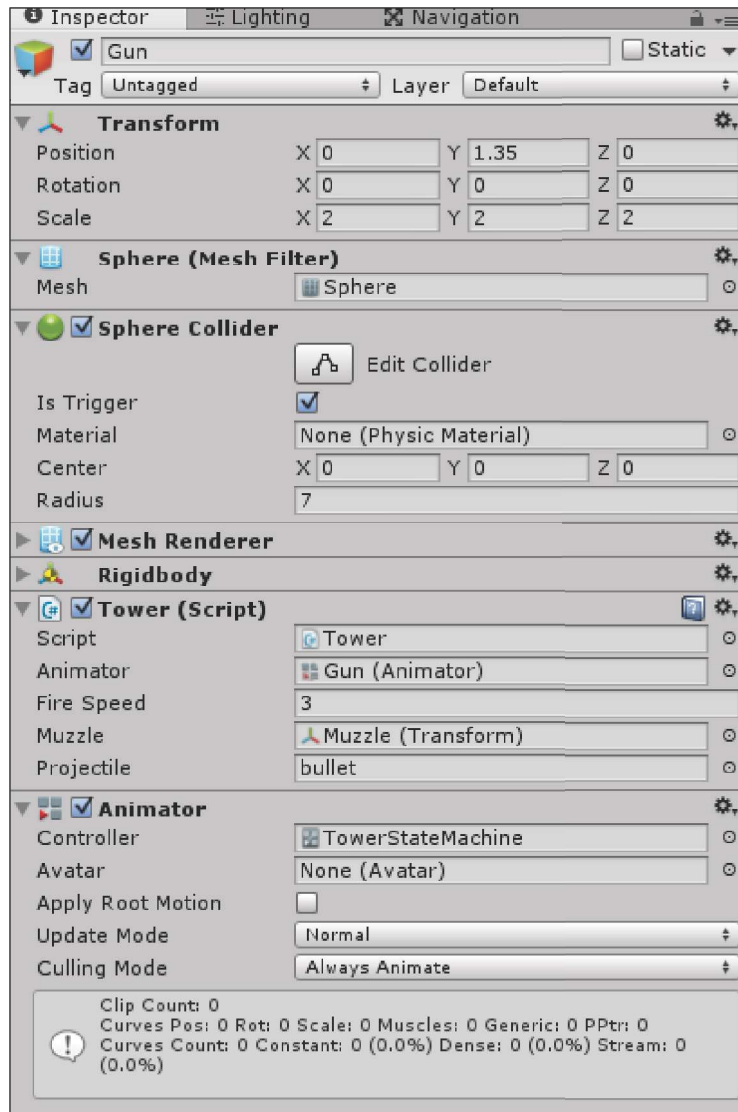


The Tower hierarchy in the inspector

The breakdown of the hierarchy is as follows:

- **Tower:** Technically, this is the base of the tower—the cylinder that holds the rest of it up. This serves no function but to hold the rest of the parts.
- **Gun:** The gun is where most of the magic happens. It is the sphere mounted on the tower with the barrel on it. This is the part of the tower that moves and tracks the player.
- **Barrel and Muzzle:** The muzzle is located at the tip of the barrel. This is used as the spawn point for the bullets that come out of the gun.

We mentioned that the gun is where the business happens for the tower, so let's dig in a bit deeper. The inspector with the gun selected looks similar to the following screenshot:

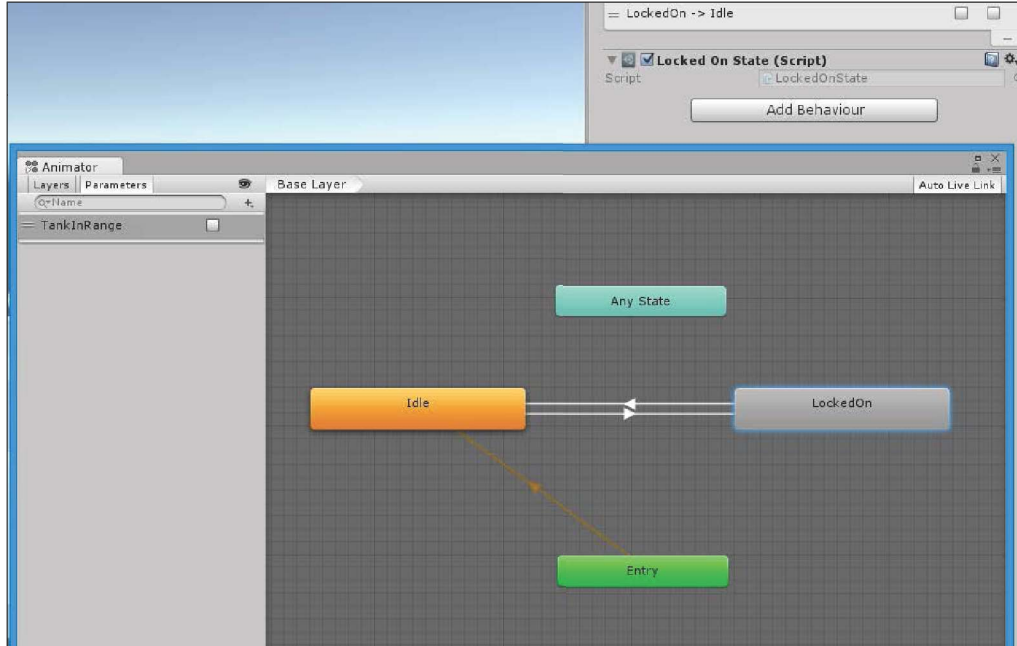


The inspector for the gun

There is quite a bit going on in the inspector here. Let's look at each of the components that affect the logic:

- **Sphere Collider:** This is essentially the tower's range. When the tank enters this sphere, the tower can detect it and will lock on to it to begin shooting at it. This is our implementation of perception for the tower. Notice that the radius is set to 7. The value can be changed to whatever you liked, but 7 seems to be a fair value. Also, note that we set the **Is Trigger** checkbox to true. We don't want this sphere to actually cause collisions, just to fire trigger events.
- **Rigidbody:** This component is required for the collider to actually work properly whether objects are moving or not. This is because Unity does not send collision or trigger events to game objects that are not moving, unless they have a rigid body component.
- **Tower:** This is the logic script for the tower. It works in tandem with the state machine and the state machine behavior, but we'll look at these components more in depth shortly.
- **Animator:** This is our tower's state machine. It doesn't actually handle animation.

Before we look at the code that drives the tower, let's take a brief look at the state machine. It's not terribly complicated, as you can see in the following screenshot:



The state machine for the tower

There are two states that we care about: `Idle` (the default state) and `LockedOn`. The transition from `Idle` to `LockedOn` happens when the `TankInRange` bool is set to true, and the reverse transition happens when the bool is set to false.

The `LockedOn` state has a `StateMachineBehaviour` class attached to it, which we'll look at next:

```
using UnityEngine;
using System.Collections;

public class LockedOnState : StateMachineBehaviour {

    GameObject player;
    Tower tower;

    // 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) {
        player = GameObject.FindWithTag("Player");
        tower = animator.gameObject.GetComponent<Tower>();
        tower.LockedOn = true;
    }

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

    // 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) {
        animator.gameObject.transform.rotation = Quaternion.identity;
        tower.LockedOn = false;
    }
}
```

When we enter the state and `OnStateEnter` is called, we find a reference to our player. In the provided example, the player is tagged as "Player" so that we are able to get a reference to it using `GameObject.FindWithTag`. Next, we fetch a reference to the `Tower` component attached to our tower prefab and set its `LockedOn` bool to true.

As long as we're in the state, `OnStateUpdate` gets called on each frame. Inside this method, we get a reference to the `Gun` `GameObject` (which the `Tower` component is attached to) via the provided `Animator` reference. We use this reference to the gun to have it track the tank using `Transform.LookAt`.



Alternatively, as the `LockedOn` bool of the `Tower` is set to `true`, this logic could be handled in the `Tower.cs` script, instead.

Lastly, as we exit the state, `OnStateExit` gets called. We use this method to do a little cleanup. We reset the rotation of our gun to indicate that it is no longer tracking the player, and we set the `Tower's LockedOn` bool back to `false`.

As we can see, this `StateMachineBehaviour` interacts with the `Tower.cs` script, so let's look at `Tower.cs` next for a bit more context as to what's happening:

```
using UnityEngine;
using System.Collections;

public class Tower : MonoBehaviour {
    [SerializeField]
    private Animator animator;

    [SerializeField]
    private float fireSpeed = 3f;
    private float fireCounter = 0f;
    private bool canFire = true;

    [SerializeField]
    private Transform muzzle;
    [SerializeField]
    private GameObject projectile;

    private bool isLockedOn = false;

    public bool LockedOn {
        get { return isLockedOn; }
        set { isLockedOn = value; }
    }
}
```

First up, we declare our variables and properties.

We need a reference to our state machine; this is where the `Animator` variable comes in. The next three variables, `fireSpeed`, `fireCounter`, and `canFire` all relate to our tower's shooting logic. We'll see how that works up later.

As we mentioned earlier, the muzzle is the location the bullets will spawn from when shooting. The projectile is the prefab we're going to instantiate.

Lastly, `isLockedOn` is get and set via `LockedOn`. While this book, in general, strays away from enforcing any particular coding convention, it's generally a good idea to keep values private unless explicitly required to be public, so instead of making `isLockedOn` public, we provide a property for it to access it remotely (in this case, from the `LockedOnSate` behavior):

```
private void Update() {
    if (LockedOn && canFire) {
        StartCoroutine(Fire());
    }
}

private void OnTriggerEnter(Collider other) {
    if (other.tag == "Player") {
        animator.SetBool("TankInRange", true);
    }
}

private void OnTriggerExit(Collider other) {
    if (other.tag == "Player") {
        animator.SetBool("TankInRange", false);
    }
}

private void FireProjectile() {
    GameObject bullet = Instantiate(projectile, muzzle.position,
    muzzle.rotation) as GameObject;
    bullet.GetComponent<Rigidbody>().AddForce(muzzle.forward *
300);
}

private IEnumerator Fire() {
    canFire = false;
```




```

        FireProjectile();
        while (fireCounter < fireSpeed) {
            fireCounter += Time.deltaTime;
            yield return null;
        }
        canFire = true;
        fireCounter = 0f;
    }
}

```

Next up, we have all our methods, and the meat and potatoes of the tower logic. Inside the `Update` loop, we check for two things: are we locked on and can we fire? If both are true, we fire off our `Fire()` coroutine. We'll look at why `Fire()` is a coroutine before coming back to the `OnTrigger` messages.

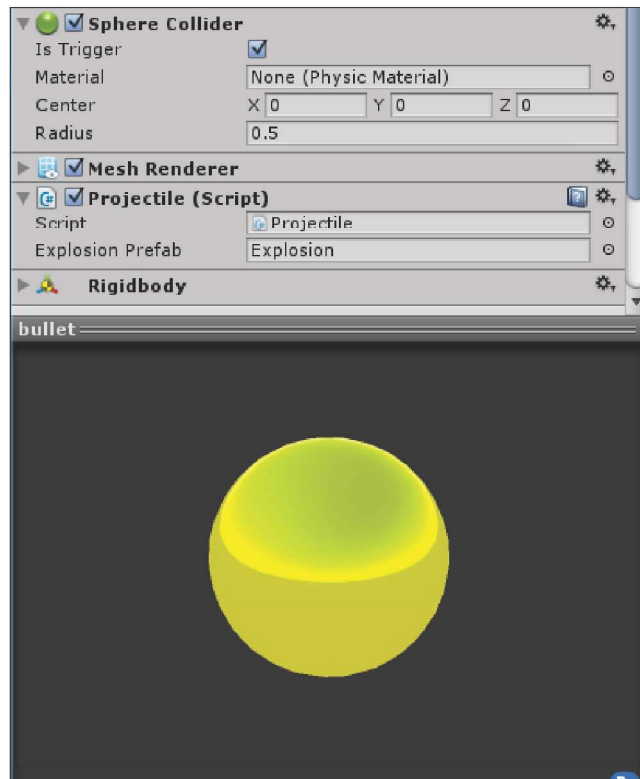
 Coroutines can be a tricky concept to grasp if you're not already familiar with them. For more information on how to use them, check out Unity's documentation at <http://docs.unity3d.com/Manual/Coroutines.html>.

As we don't want our tower to be able to constantly shoot projectiles at the tank like a projectile-crazy death machine, we use the variables that we defined earlier to create a cushion between each shot. After we call `FireProjectile()` and set `canFire` to false, we start a counter from 0 up to `fireSpeed`, before we set `canFire` to true again. The `FireProjectile()` method handles the instantiation of the projectile and shoots it out toward the direction the gun is pointing to using `Rigidbody.AddForce`. The actual bullet logic is handled elsewhere, but we'll look at that later.

Lastly, we have our two `OnTrigger` events—one for when something enters the trigger attached to this component and another for when an object leaves said trigger. Remember the `TankInRange` bool that drives the transitions for our state machine? This variable gets set to true here when we enter the trigger and back to false as we exit. Essentially, when the tank enters the gun's sphere of "vision", it instantly locks on to the tank, and the lock is released when the tank leaves the sphere.

Making the towers shoot

If we look back at our `Tower` component in the inspector, you'll notice that a prefab named `bullet` is assigned to the `projectile` variable. This prefab can be found in the `Prefabs` folder of the sample project. The prefab looks similar to the following screenshot:



The bullet prefab

The `bullet` game object is nothing fancy; it's just a bright yellow orb. There is a sphere collider attached to it, and once again, we must make sure that `IsTrigger` is set to `true` and it has a `Rigidbody` (with `gravity` turned off) attached to it. We also have a `Projectile` component attached to the `bullet` prefab. This handles the collision logic. Let's take a look at the code:

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {

    [SerializeField]
```

```

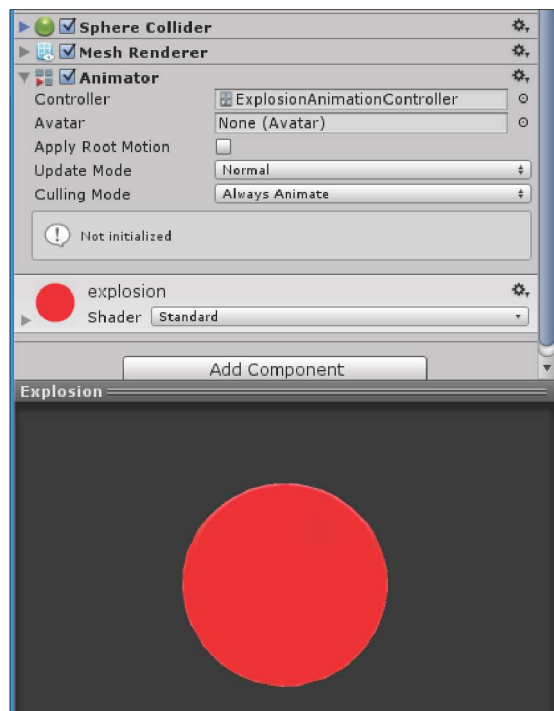
private GameObject explosionPrefab;

void Start () { }

private void OnTriggerEnter(Collider other) {
    if (other.tag == "Player" || other.tag == "Environment") {
        if (explosionPrefab == null) {
            return;
        }
        GameObject explosion = Instantiate(explosionPrefab,
transform.position, Quaternion.identity) as GameObject;
        Destroy(this.gameObject);
    }
}
}

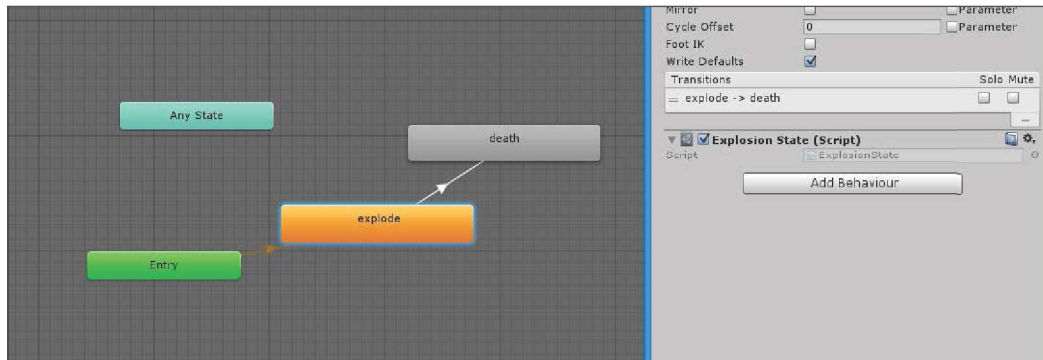
```

We have a fairly straightforward script here. In our level, we have all of the floor and walls tagged as "Environment", so in our `OnTriggerEnter` method, we check that the trigger this projectile is colliding with is either the player or the environment. If it is, we instantiate an explosion prefab and destroy the projectile. Let's take a look at the explosion prefab, which looks similar to this:



Inspector with the explosion prefab selected

As we can see, there is a very similar game object here; we have a sphere collider with `IsTrigger` set to `true`. The main difference is an animator component. When this explosion is instantiated, it expands as an explosion would, then we use the state machine to destroy the instance when it transitions out of its explosion state. The animation controller looks similar to the following screenshot:




The animation controller driving the explosion prefab

You'll notice the `explode` state has a behavior attached to it. The code inside this behavior is fairly simple:

```
// 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) {
    Destroy(animator.gameObject, 0.1f);
}
```

All we're doing here is destroying the instance of the object when we exit the state, which occurs when the animation ends.

 If you want to flesh out the game with your own game logic, this may be a good place to trigger any secondary effects such as damage, environment particles, or anything you can think of!

Setting up the tank

The example project also includes a prefab for the tank, which is simply called (you guessed it) `Tank`, inside the `Prefabs` folder.

The tank itself is a simple agent with one goal—reach the end of the maze. As mentioned earlier, the player has to help the tank out along the way by activating its abilities to keep it safe from oncoming fire from the towers.

By now you should be fairly familiar with the components you'll encounter along the way, except for the `Tank.cs` component attached to the prefab. Let's take a look at the code to figure out what's going on behind the scenes:

```
using UnityEngine;
using System.Collections;

public class Tank : MonoBehaviour {
    [SerializeField]
    private Transform goal;
    private NavMeshAgent agent;
    [SerializeField]
    private float speedBoostDuration = 3;
    [SerializeField]
    private ParticleSystem boostParticleSystem;
    [SerializeField]
    private float shieldDuration = 3f;
    [SerializeField]
    private GameObject shield;

    private float regularSpeed = 3.5f;
    private float boostedSpeed = 7.0f;
    private bool canBoost = true;
    private bool canShield = true;
```

There are a number of values that we want to be able to tweak easily, so we declare the corresponding variables first. Everything from the duration of our abilities to the effects associated with them is set here first:

```
    private bool hasShield = false;
    private void Start() {
        agent = GetComponent<NavMeshAgent>();
        agent.SetDestination(goal.position);
    }

    private void Update() {
        if (Input.GetKeyDown(KeyCode.B)) {
            if (canBoost) {
                StartCoroutine(Boost());
            }
        }
        if (Input.GetKeyDown(KeyCode.S)) {
            if (canShield) {
                StartCoroutine(Shield());
            }
        }
    }
}
```

Our `Start` method simply does some setup for our tank; it grabs the `NavMeshAgent` component and sets its destination to be equal to our goal variable. We will discuss more on that soon.

We use the `Update` method to catch the input for our abilities. We've mapped `B` to `boost` and `S` to `shield`. As these are timed abilities, much like the towers' ability to shoot, we implement these via coroutines:

```
private IEnumerator Shield() {
    canShield = false;
    shield.SetActive(true);
    float shieldCounter = 0f;
    while (shieldCounter < shieldDuration) {
        shieldCounter += Time.deltaTime;
        yield return null;
    }
    canShield = true;
    shield.SetActive(false);
}

private IEnumerator Boost() {
    canBoost = false;
    agent.speed = boostedSpeed;
    boostParticleSystem.Play();
    float boostCounter = 0f;
    while (boostCounter < speedBoostDuration) {
        boostCounter += Time.deltaTime;
        yield return null;
    }
    canBoost = true;
    boostParticleSystem.Pause();
    agent.speed = regularSpeed;
}
```

The two abilities' logic is very similar. The `shield` enables and disables the `shield` game object, which we define in a variable in the inspector, and after an amount of time equal to `shieldDuration` has passed, we turn it off, and allow the player to use the `shield` again.

The main difference in the `Boost` code is that rather than enabling and disabling a game object, the `boost` calls `Play` on a particle system we assign via the inspector and also sets the speed of our `NavMeshAgent` to double the original value, before resetting it at the end of the ability's duration.



Can you think of other abilities you'd give the tank? This is a very straightforward pattern that you can use to implement new abilities in your own variant of the project. You can also add additional logic to customize the shield and boost abilities here.

The sample scene already has an instance of the tank in it with all the variables properly set up. The inspector for the tank in the sample scene looks similar to the following screenshot:



Inspector with the tank instance selected

As you can see in the preceding screenshot, we've assigned the `Goal` variable to a transform with the same name, which is located in the scene at the end of the maze we've set up. We can also tweak the duration of our abilities here, which is set to **3** by default. You can also swap out the art for the abilities, be it the particle system used in the boost or the game object used for the shield.

The last bit of code to look at is the code driving the camera. We want the camera to follow the player, but only along its `z` value, horizontally down the track. The code to achieve this looks similar to this:

```
using UnityEngine;
using System.Collections;

public class HorizontalCam : MonoBehaviour {
    [SerializeField]
    private Transform target;

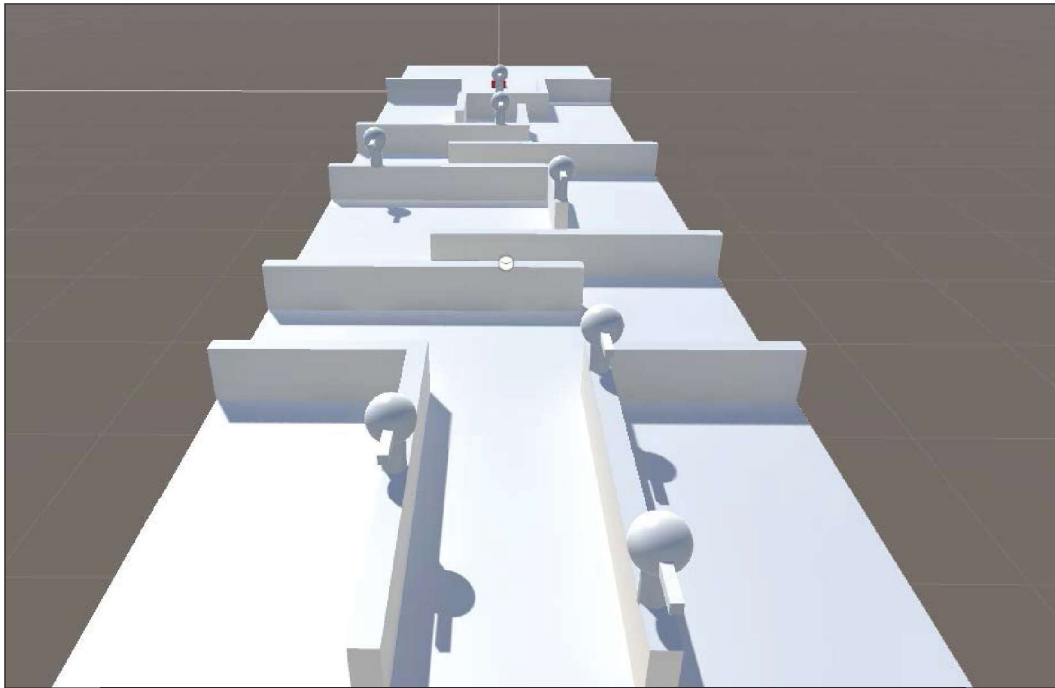
    private Vector3 targetPositon;

    private void Update() {
        targetPositon = transform.position;
        targetPositon.z = target.transform.position.z;
        transform.position = Vector3.Lerp(transform.position,
        targetPositon, Time.deltaTime);
    }
}
```

As you can see, we simply set the target position of the camera equal to its current position on all axes, but we then assign the z axis of the target position to be the same as our target's, which if you look in the inspector, has been set to the transform of the tank. We then use linear interpolation (`Vector3.Lerp`) to smoothly translate the camera from its current position to its target position every frame.

Setting up the environment

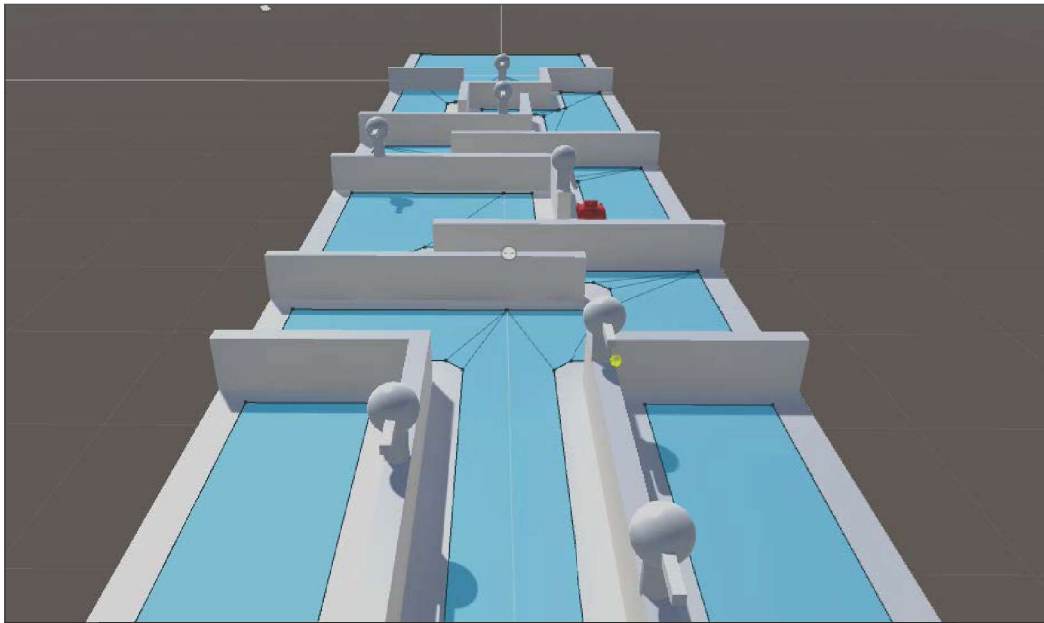
As our tank uses a `NavMeshAgent` component to traverse the environment, we need to set up our scene using static game objects for the bake process to work properly, as we learned in *Chapter 4, Finding Your Way*. The maze is set up in a way so that towers are spread out fairly reasonably and that the tank has plenty of space to maneuver around easily. The following screenshot shows the general layout of the maze:



The gauntlet our tank must run through

As you can see, there are seven towers spread out through the maze and a few twists and turn for our tank to break line of sight. In order to avoid having our tank graze the walls, we adjust the settings in the navigation window to our liking. By default, the example scene has the agent radius set to 1.46 and the step height to 1.6. There are no hard rules for how we arrived at these numbers; it is just trial and error.

After baking the NavMesh, we'll end up with something similar to what's shown in the following screenshot:



The scene after we've baked our NavMesh

Feel free to rearrange the walls and towers to your liking. Just remember that any blocking objects you add to the scene must be marked as static, and you have to rebake the navigation for the scene after you've set everything up just the way you like it.

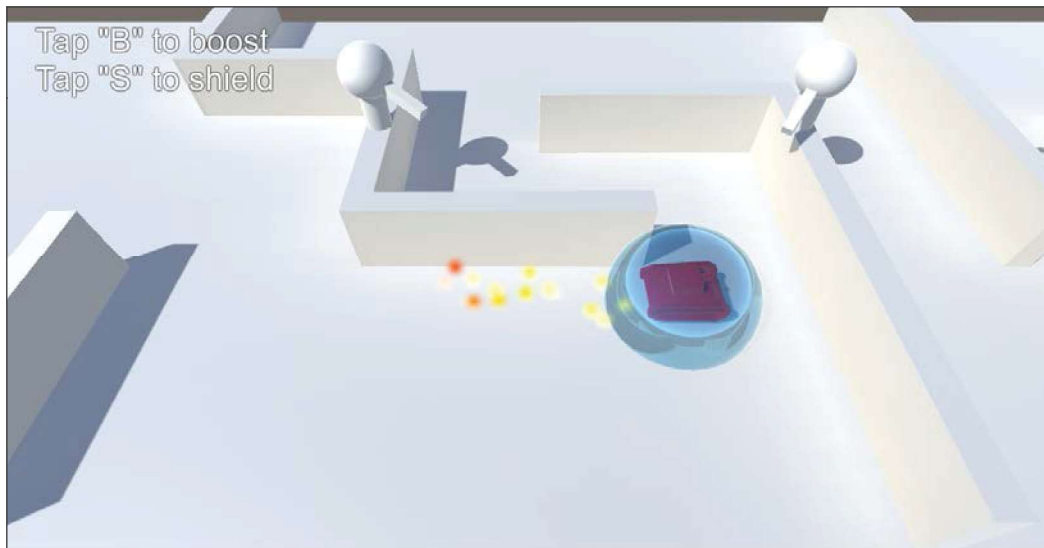
Testing the example

The example scene is ready to play right out of the box, so if you didn't get the itch to modify any of the default settings, you can just hit the **Play** button and watch your tank go. You'll notice we've added a canvas with a label explaining the controls to the player. There is nothing fancy going on here; it's just a simple "press this button to do that" kind of instruction:



Simple instructions to guide the player

The example project is a great example to expand upon and to have fun with. With the concepts learned throughout this book, you can expand on the types of towers, the tank's abilities, the rules, or even give the tank a more complex, nuanced behavior. For now, we can see that the concepts of state machines, navigation, perception and sensing, and steering, all come together in a simple, yet amusing example. The following screenshot shows the game in action:



The tank-defense game in action

Summary

So, we've reached the end. In this chapter, we took a few of the concepts covered in the book and applied them to create a small tank-defense game. We built upon the concept of finite state machines, which we originally covered in *Chapter 2, Finite State Machines and You*, and created an artificial intelligence to drive our enemy towers' behavior. We then enhanced the behavior by combining it with sensing and perception, and finally, we implemented navigation via Unity's NavMesh feature to help our tank AI navigate through our maze-like level, through a gauntlet of autonomous AI towers with one thing on their simple AI minds—destroy!

