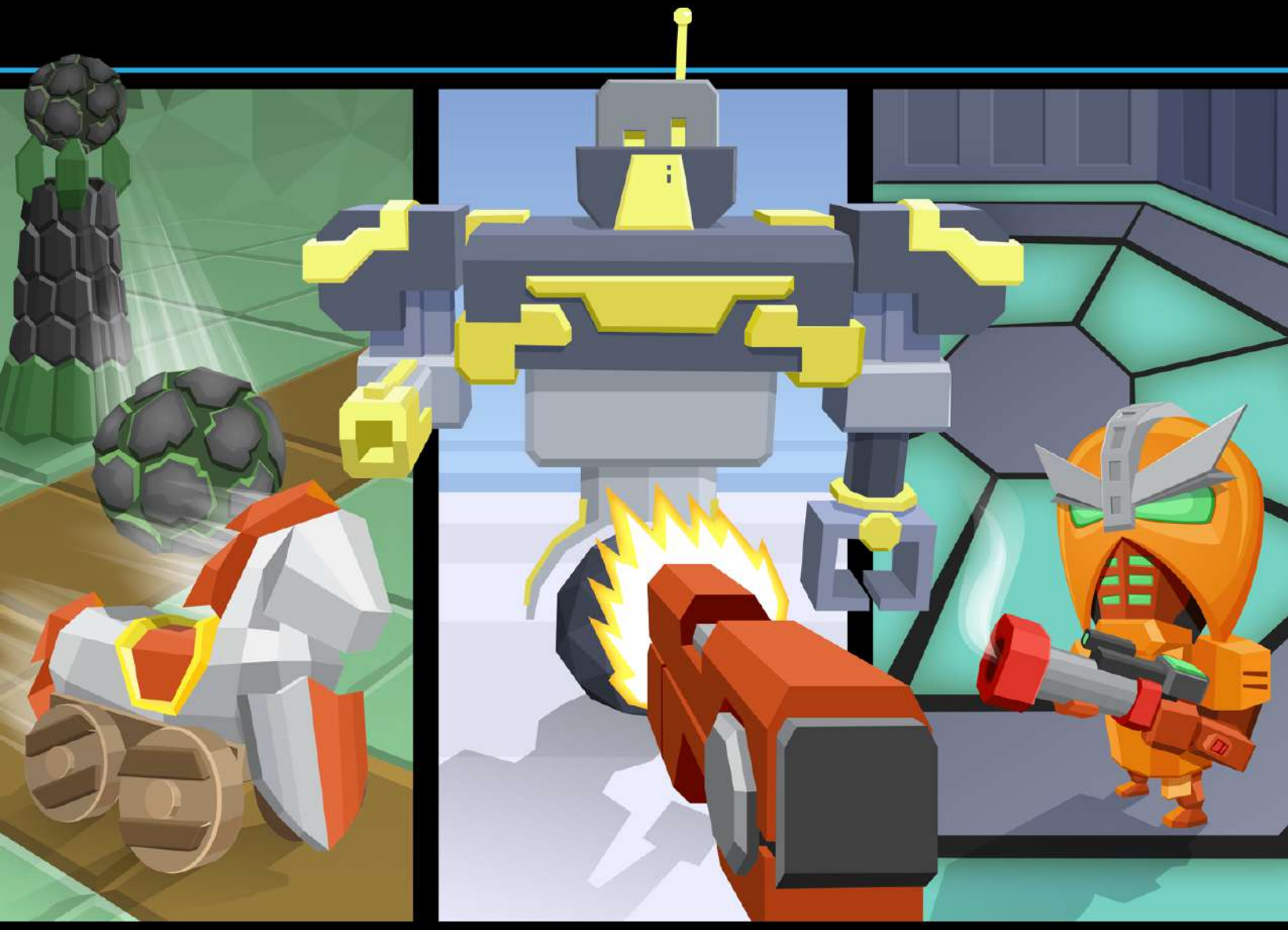
Make 4 Complete Unity Games
from Scratch Using C#



Unity Games
by Tutorials

By the raywenderlich.com Tutorial Team

Brian Moakley, Mike Berg, Sean Duffy,
Eric Van de Kerckhove, Anthony Uccello

Table of Contents: Extended

Section II: First-Person Shooters

In this section, you'll learn how to create one of the most popular types of games: a first-person shooter.

In the process, you'll create a game called **Robot Rampage**, where you'll blast waves of enemy robots using a variety of weapons and power-ups. Get your boom stick ready!

Chapter 9: The Player and Environment

By Anthony Uccello

In the next few chapters, you'll create a first-person shooter, or FPS as it's more commonly known. Some well-known FPS titles include *Halo*, *Call of Duty*, and *Doom*.

The game you'll create is **Robot Rampage**; you'll control a player who has to mow down robots to survive. The robots will spawn in waves and the player must collect health, ammunition, and armor pickups in order to stave off the robots as long as possible. The longer the player survives, and the more robots they kill, the higher their score.

This game is broken down into three parts. In this part you'll set up the weapons, add sounds, and the code to handle all the firing.

In the next chapter, you'll add the robot bad guys, some powerups, and lots of flying projectiles.

In the final chapter, you'll add the all the user interface elements to keep track of the score, ammo, and even a menu to restart the game when it's over.

In short, you'll have a completed first person shooter game.

Getting started

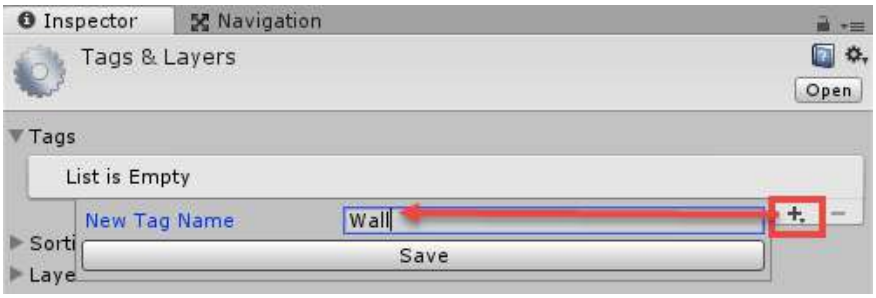
To get started, open up the Starter project in Unity. In the Project Browser, open the **Scenes** folder and double-click the **Battle** scene to open it.

Believe it or not, the map is mostly setup for you. The map itself consists of a floor, surrounded by a series of walls. After you complete the tutorial, feel free to create other configurations for the game by adjusting the wall positions.

To get started, you'll need to setup the wall GameObjects so the robots can avoid them. So far in this book you've been using Layers, but Unity has another tool known as **tags**. A tag is just a bit of text used to identify GameObjects. Each GameObject can have only one tag, and thankfully, they're easy to use. By assigning a tag to each wall GameObject you'll be able to setup some pathfinding rules for the robots to follow so they can move around walls.

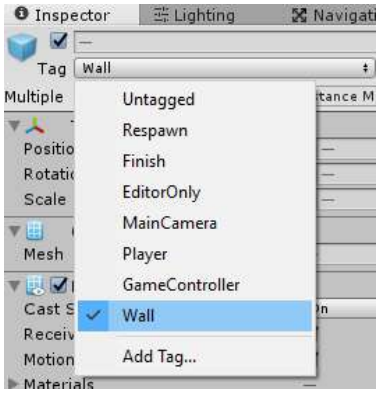
In the Hierarchy, click on the **Inner Walls** GameObject. In the Inspector, click the **Tag** dropdown and click **Add Tag**.

Click the plus button in the **Tag Window** and type **Wall** in the first slot:



Expand the `InnerWalls` GameObject by clicking the arrow next to it in the Hierarchy. Select all the child GameObjects (named **Wall** and **WallPost**); a quick way to do this is select the first GameObject then hold **Shift** and click the last item.

With all the child GameObjects selected, click the the **Tag dropdown** and select the **Wall** tag you just created:

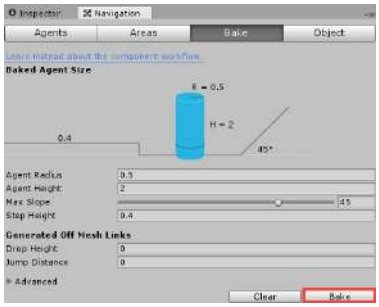


All the walls are properly tagged. Now you can make your NavMesh for your robots.

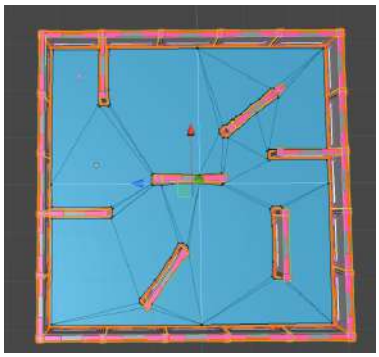
> **Note**: Forgot how to make a NavMesh or you skipped ahead to this chapter? If so, check out Chapter 5, "Managers and Pathfinding" which covers NavMeshes in depth.

Open the **Navigation** window by going to **Window/Navigation**. Ensure the **Wall** and **WallPost** GameObjects are selected, and make sure that the **Navigation Static** checkbox in the **Object tab** is checked.

Select the **Bake** tab and click the **Bake** button.



You should see the ground turn light blue in the scene:



The robots now have a walkable area in the map. Pretty cool, eh? You can now click the mini arrow next to **InnerWalls** to hide the list of Wall and WallPost GameObjects.

With the NavMesh taken care of, it's time to create the player.

Adding the player

The first step for adding the player … is to add the player! :]

In the Hierarchy, create an empty GameObject and name it **Player**. You are going to use Unity's **Character Controller** with a few modifications. Click **Assets\ImportPackage\Characters**.

If you don't see the Character Controller option, then you didn't import the Standard Assets when you installed Unity. To fix this, re-rerun the Unity installer and only check Standard Assets for the install. A standard Unity Asset is like any regular package. It's just a collection of files.

Click **Import** on the menu to import the character assets. When the import is completed, you'll notice a new folder in the Project Browser called Standard Assets. This contains everything that you imported.

The first modification is to remove jumping and a few other actions you won't need such as footsteps and headbobbing. Robot Rampage doesn't have jump obstacles, and by default the Unity **Character Controller** uses the spacebar to jump.

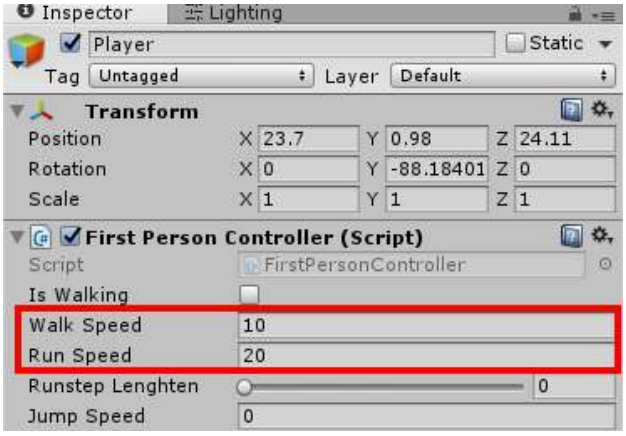
In the Project browser, open the **FirstPersonController** in your code editor located in **Assets/Standard Assets/Characters/FirstPersonCharacter/Scripts**.

Comment out lines **39**, **50**, **112 to 127**, **165-176**, and **182-200**. To comment out a line, simply place // in front of the line.

Save the file and switch back in Unity. Drag the **FirstPersonController** script on top of the **Player** GameObject. Click the **Player** GameObject, and use the Inspector to set **Position** to **(23.7, 2.68, 24.11)**, **Rotation** to **(0, -88.18, 0)**.

This puts the player in the one of the corners and rotates the player facing the open hallway as opposed to staring at a wall.

The `FirstPersonController` script also has values for run speed and walk speed. Set the **Player Walk Speed** to **10**, and **Run** to **20**.



Those run and walk values worked best for this game. When you make your games, play around with these values to see what works for you.

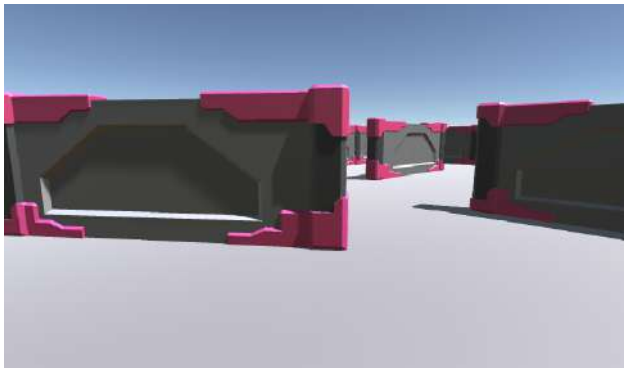
Adding the camera

Currently, the camera is divorced from the player when you want them to be a happy couple, frolicking together through the robot wasteland. Thankfully, you don't have to play matchmaker.

Click the the **Camera** GameObject and drag it onto the **Player** GameObject to make it a child of **Player**. The camera will now move with the player.

Set the **Camera** position as: **(0, 0.8, 0)** and **Rotation** to **(0,0,0)**.

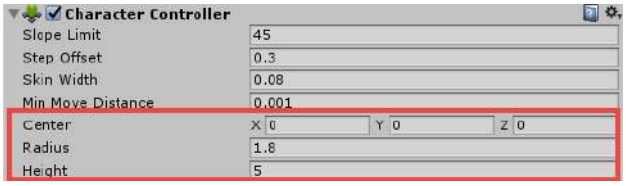
Click Play; use the WASD keys to move around and the mouse to look around as you move. You can also hold the Shift key to run.



Unfortunately, stopping the game can be a little difficult when using the first person character controller since the mouse pointer is not available. Thankfully there are shortcuts. Press **Control-P** (or **Command-P** on the Mac) to exit play mode.

With first person controller added, you just need to configure the capsule collider. A first person character controller comes with a plain old regular character controller which already contains a capsule collider. You can learn more about character controllers in Chapter 4, “Physics”.

With the Player GameObject still selected, find the **Character Controller** in the Inspector and set the **Center** property to (**0**, **0**, **0**), the **Radius** to **1.8**, and the **Height** to **5**.



Next is the fun part where you get to add three weapons and start shooting them!

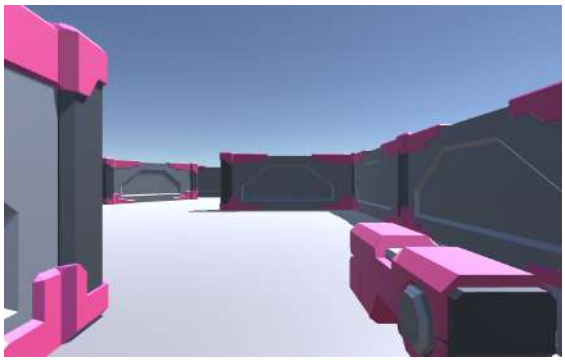
Creating weapons

In Unity, a gun is just another GameObject. The big difference is the perspective. Putting it close to to the camera gives you the perspective of a first person shooter that we all love.

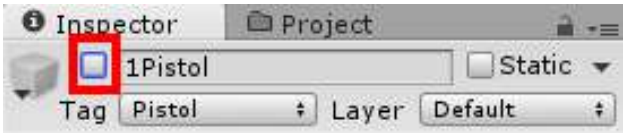
Select the **Camera** GameObject, then right-click on it and select **Create Empty** to create an empty GameObject under the **Camera**. Name your new GameObject **1Pistol**. The number in front of the name refers to the equip key the player will use.

Drag the **RobotRampage_Pistol** model from inside the Models folder in the Project browser onto the **1Pistol** GameObject. By parenting the model to a GameObject, the animations you'll create later will occur in local space.

Set the **Position** of the **1Pistol** GameObject to **(0.69, -0.72, 1.26)** and the **rotation** to (**0**, **0**, **0**).



This gives you the perspective of the gun directly in front of you. In the Inspector, **deactivate** the pistol by **unchecking** the checkbox next to the gun name. This makes it disappear so you can add another gun.



Create another empty GameObject under the **Camera** and name it **2AssaultRifle**. You'll usually need to adjust the scale of imported models to fit your game, and this is no exception. Drag the **RobotRampage_AssaultRifle** model underneath your new GameObject. Set the **2AssaultRifle position** to (**0.54, -0.67, 0.913**), the **rotation** to (**0**, **0**, **0**), and set the **scale** to (**0.01, 0.01, 0.01**).

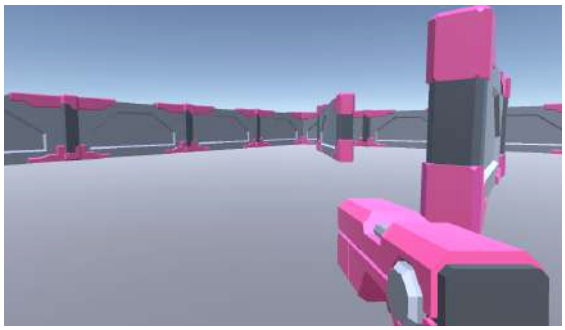
Deactivate the assault rifle like you did with the pistol.

Do the same again for the Shotgun, name it **3Shotgun** and use the **RobotRampage_Shotgun** model. Set the **3Shotgun** position to (**0.3, -0.36, 0.65**), the **rotation** to (**0**, **0**, **0**), and set its **scale** to (**0.01, 0.01, 0.01**).

At this point, you now have your three weapons added to your game. Leave the **1Pistol** GameObject active and deactivate the **2AssaultRifle** and **3Shotgun** GameObjects.

The code will work this way as well: As the player switches guns, you will turn on the active gun and turn off the other ones.

Click Play and walk around with one of the weapons. Feels good, doesn't it? :]



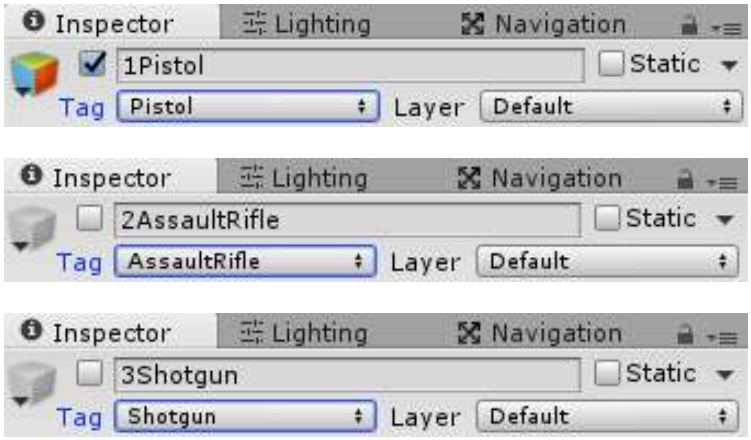
Now is a good time to set the tags on each weapon.

Adding tags to GameObjects

Select the **1Pistol** GameObject, click the **Tag** dropdown and click **Add Tag**. Click the + button under **Tags** and type **Pistol**. Click the + again and type in **AssaultRifle**, then finally click the + one more time and type in **Shotgun**.

Now to assign the appropriate tag to its weapon.

Click the **1Pistol**, then click the **Tags** dropdown and set it to the **Pistol** tag you just created. Do the same for the **2AssaultRifle** and **3Shotgun**, setting their tags appropriately:



With everything tagged and ready to go, it's time to jump into the code editor.

Creating the Constants file

The very first thing you'll do is create a `Constants` file. This makes life easier by having common name references all other scripts can use. Whenever you create a new C# script in this section, you'll do it in a generic **Scripts** folder.

Right-click the **Assets** folder in the Project browser, select **Create\Folder** and name the new folder **Scripts**. In the new **Scripts** folder, create a **C# Script** and name it **Constants**.

Open your new **Constants** script in your script editor and replace the contents of the file with the following:

```
public class Constants {
  // Scenes
  public const string SceneBattle = "Battle";
  public const string SceneMenu = "MainMenu";

  // Gun Types
  public const string Pistol = "Pistol";
  public const string Shotgun = "Shotgun";
  public const string AssaultRifle = "AssaultRifle";

  // Robot Types
  public const string RedRobot = "RedRobot";
  public const string BlueRobot = "BlueRobot";
  public const string YellowRobot = "YellowRobot";
```

```
    // Pickup Types
    public const int PickUpPistolAmmo = 1;
    public const int PickUpAssaultRifleAmmo = 2;
    public const int PickUpShotgunAmmo = 3;
    public const int PickUpHealth = 4;
    public const int PickUpArmor = 5;

    // Misc
    public const string Game = "Game";
    public const float CameraDefaultZoom = 60f;
}
```

Each commented section represents the type data you'll use in upcoming scripts. You don't need to know what each constant does right now; you'll learn that as you work through the next few sections.

To start, you'll use the gun types to create the weapon switching logic.

Add the following before the closing brace:

```
public static readonly int[] AllPickupTypes = new int[5] {
  PickUpPistolAmmo,
  PickUpAssaultRifleAmmo,
  PickUpShotgunAmmo,
  PickUpHealth,
  PickUpArmor
};
```

This keeps track of all possible pickup types.

Weapon logic

Save your work and switch back to Unity. Create a C# script in the **Scripts** folder, name it **GunEquipper** and add the following variables to the class:

```
public static string activeWeaponType;

public GameObject pistol;
public GameObject assaultRifle;
public GameObject shotgun;

GameObject activeGun;
```

The GameObject variables reference each gun, and `activeGun` keeps track of the currently equipped gun.

Add the following method below `activeGun`, replacing the original `Start()`:

```
void Start () {
    activeWeaponType = Constants.Pistol;
    activeGun = pistol;
}
```

Here you initialize the starting gun as the pistol.

You now need to load the appropriate gun when the player switches weapons. Add the following method:

```
private void loadWeapon(GameObject weapon) {
  pistol.SetActive(false);
  assaultRifle.SetActive(false);
  shotgun.SetActive(false);

  weapon.SetActive(true);
  activeGun = weapon;
}
```

This method will turn off all gun GameObjects, set the passed-in GameObject as active, then update the `activeGun` reference. You'll call this code from `Update()`. Update that method so it looks like the following:

```
void Update () {
  if (Input.GetKeyDown("1")) {
    loadWeapon(pistol);
    activeWeaponType = Constants.Pistol;
  } else if (Input.GetKeyDown("2")) {
    loadWeapon(assaultRifle);
    activeWeaponType = Constants.AssaultRifle;
  } else if (Input.GetKeyDown("3")) {
    loadWeapon(shotgun);
    activeWeaponType = Constants.Shotgun;
  }
}
```

The code each in each `if` statement checking whether the pressed key is 1, 2, or 3. Checking for input in every frame like this is known as **polling**. If one of those number keys is hit, the appropriate block calls `loadWeapon()` and passes it the appropriate weapon to activate. It then sets **activeWeaponType** as a static. This makes it easy for other scripts to query which gun is the player has equipped.

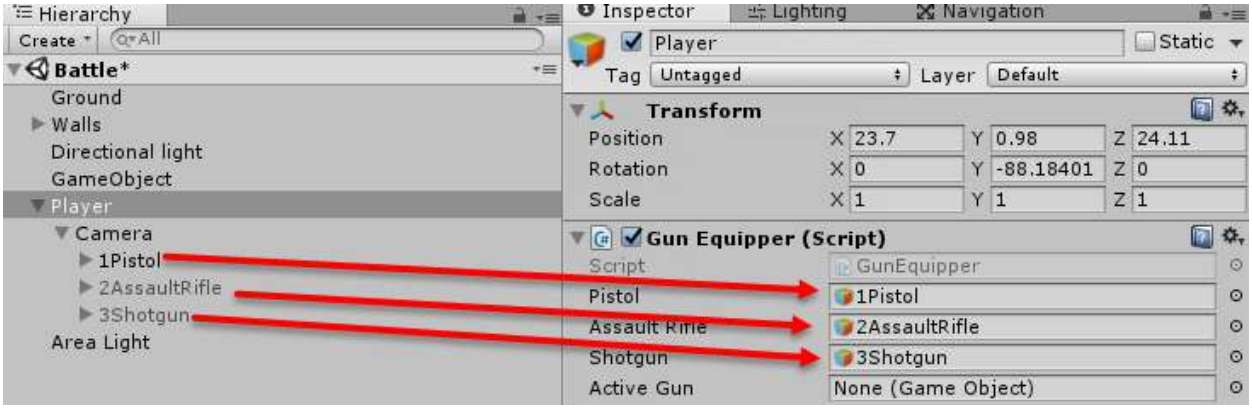
Finally, add the following convenience method:

```
public GameObject GetActiveWeapon() {
  return activeGun;
}
```

This simply returns the `activeGun` so other scripts can read that piece of information.

Save your work, switch back to Unity, and attach the **GunEquipper** script to the **Player** GameObject:

Select the **Player** GameObject and drag the **1Pistol** GameObject from the Hierarchy to the respective field in the **GunEquipper** script component. Repeat this for **2AssaultRifle** and **3Shotgun**.



Click Play and press the 1, 2, or 3 key to change the weapon.

What use is a great-looking weapon without ammunition?

Ammunition logic

To get the gun firing, you have determine a fire rate. When the player fires a gun, you want there to be a pause between each bullet. This means you must keep track of when the last bullet has fired so you can fire the next bullet.

Create a C# script, name it **Gun** and add to it the following variables:

```
public float fireRate;
protected float lastFireTime;
```

`fireRate` is the speed at which the gun will fire, and `lastFireTime` tracks the last time the gun was fired. Update `Start()` in **Gun** to:

```
void Start() {
  lastFireTime = Time.time - 10;
}
```

This sets `lastFireTime` to 10 seconds ago. When the game starts, the player will be able to fire the gun immediately.

Finally, replace `Update()` with the following:

```
protected virtual void Update() {

}

protected void Fire() {

}
```

You will subclass these later on, as each weapon will have its own slight variation on these methods.

Save your work and switch back to Unity. Create a C# script, name it **Pistol** and replace its contents with the following:

```
using UnityEngine;

public class Pistol : Gun {
  override protected void Update() {
    base.Update();
    // Shotgun & Pistol have semi-auto fire rate
    if (Input.GetMouseButtonDown(0) && (Time.time - lastFireTime)
        > fireRate) {
      lastFireTime = Time.time;
      Fire();
    }
  }
}
```

This checks whether enough time has elapsed between shots to allow another one. If so, it will trigger the gun firing animation — you'll be adding that in just a moment.

Save your work, switch back to Unity and attach your **Pistol** script to the **1Pistol** GameObject.

Note that the **fireRate** hasn't been set yet. Click on the **1Pistol** GameObject and set the **fireRate** to **0.6**.

Excellent. Now, it's time for the shotgun. Select the **Pistol** script and press **Control-D** (or **Command-D** on the mac). This will duplicate the script. Change the name to **Shotgun**. Open the script in your code editor. Replace this:

```
public class Pistol : Gun {
```

with this:

```
public class Shotgun : Gun
```

As with the pistol, this will be a semi-automatic weapon, but since it's a shotgun it needs a longer time between shots. Save your work, switch back to Unity, attach the **Shotgun** script to the **3Shotgun** GameObject and set the **fireRate** to **1**.

Finally, you need to do the same with assault rifle. Create a script named **AssaultRifle**, attach it to the **2AssaultRifle** GameObject, and replace its contents with the following:

```
using UnityEngine;

public class AssaultRifle : Gun {

  override protected void Update() {
    base.Update();
    // Automatic Fire
    if (Input.GetMouseButton(0) && Time.time - lastFireTime > fireRate) {
      lastFireTime = Time.time;
      Fire();
    }
  }
}
```

This looks almost the same as the code you added to the **Pistol** script, but there's one key difference: Instead of `GetMouseButtonDown`, this script uses `GetMouseButton` to see if the player is holding down the left mouse button to auto-fire.

Save and switch back Unity. Set the **fireRate** of the **AssaultRifle** script to **0.1**.

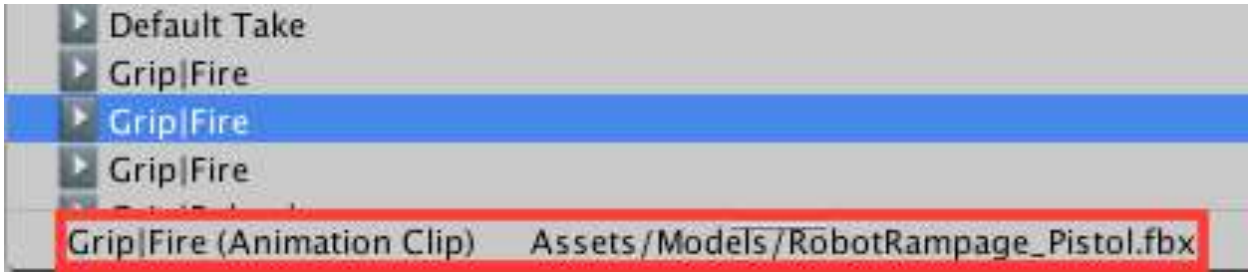
With the firing in place, you now need to apply your animations. You'll write the actual firing code itself soon.

Working with animation states

Now to add the animation. If you've jumped directly to this chapter without reading any of the others or you still feel shaky with Unity's animation in general, then definitely read Chapter 5, “Animation”.

Right-click on the **Animations** folder, select **Create\Animator Controller** and name it **Pistol**. Right-click on your new **AnimatorController** and click **open** to bring up the **Animator** window.

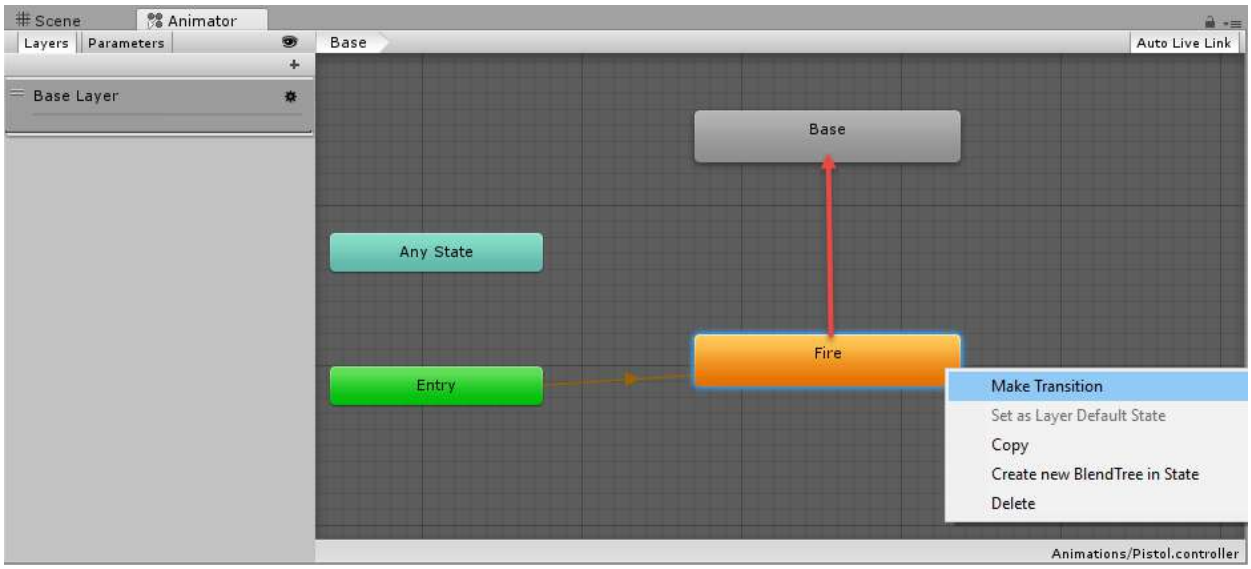
Right-click in the grid area and select **Create State\Empty**. Click on the new state and in the Inspector, rename it to **Fire**. Click on the control to the right of the **Motion** box and enter **Fire** in the Search pop-up. Find **Assets/Models/RobotRampage_Pistol.fbx** in the list and double-click it.



At this point you have a controller with one job: to fire the gun.

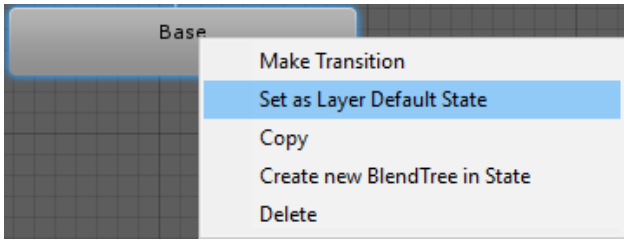
You need a default state for this animation. In the Animator window create another empty state as you did above and name it **Base**.

Right-click **Fire**, then click **Make Transition**. Drag from **Fire** to **Base** like so:



After the `Fire` animation plays, the animation will change to the `Base` state, where the gun is at rest.

Right-click the **Base** state and select **Set as Layer Default State** to make it the base animation:



Theres only one thing left to do: Hook up this animation to the code! Open the **Gun** script and add the following line to **Fire()**:

```
  GetComponentInChildren<Animator>().Play("Fire");
```

This fetches the animator controller and tells it to play the `Fire` animation. But before the animation can play, you'll need to add that component to your GameObjects.

Of course, by creating a animator controller for the pistol, you need one for the shotgun and assault rifle. In the Project Browser, select the **Pistol animator controller** in the Animations folder. **Duplicate it twice** by pressing **Control-D** (or **Command-D** on a Mac). Name one **Shotgun** and the other **AssaultRifle**.

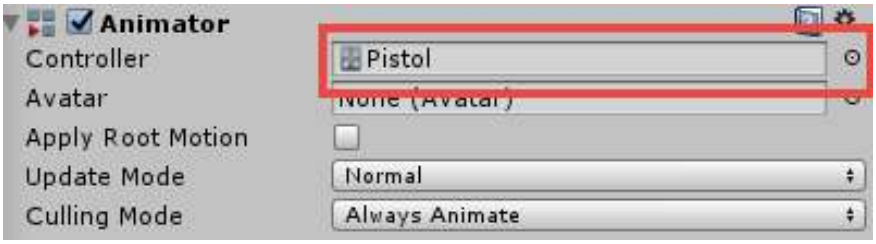
Open the **Shotgun** animator controller, and select the **Fire** state. In the Inspector, set the **Motion** to **Grip|Fire**, making sure it originates from the **Assets/Models/RobotRampage_Shotgun.fbx** model.

Finally, open the **AssaultRifle** animator controller in the animator, and select the **Fire** state. In the Inspector, set the **Motion** to **Grip|Fire**, making sure it originates from the **Assets/Models/RobotRampage_AssaultRifle.fbx** model.

With all your animations in place, you just need to add them to the weapons.

Adding Animator components

Head back to Unity, click on the **1Pistol** GameObject, click **Add Component**, type in **Animator** and select the **Animator** component. Click the **Controller** box on the **Animator** you just added and select the **Pistol** animator controller from the pop-up.



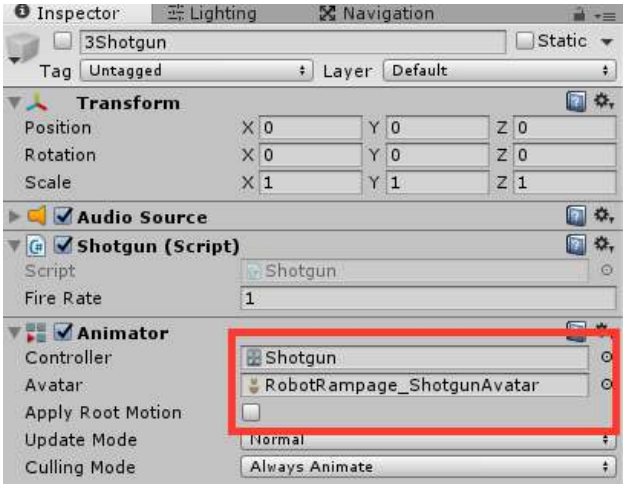
Next click the **Avatar** box on the **Animator** and select the **RobotRampage_PistolAvatar**.

An **Avatar** tells the animation system how to animate the transforms of the model. When you import a model, you need to configure the avatar. For the sake of simplicity, this has been done for you (as such configuration goes well beyond the scope of the book).

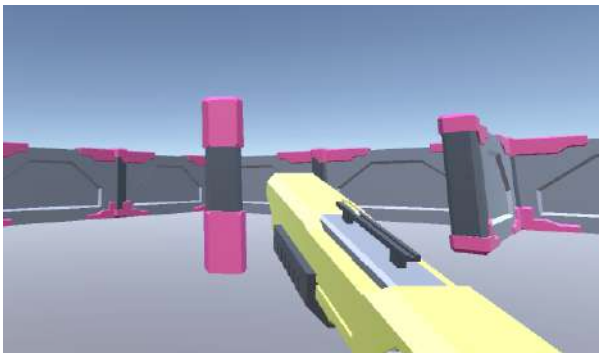
Click Play and click the left mouse button to fire the pistol.

For the assault rifle, select the **2AssaultRifle** GameObject and drag the **AssaultRifle** animator controller on to the Inspector, beneath all the existing components. Unity will automatically add the animator to the GameObjet. Set the **Avatar** to **RobotRampage_AssaultRifleAvatar**.

Do the same with **3Shotgun**, dragging the **Shotgun** animator controller on it and setting the **Avatar** to **RobotRampage_ShotgunAvatar**:



Click Play and test your weapons, switching between them with 1, 2, or 3:



Adding the reticle

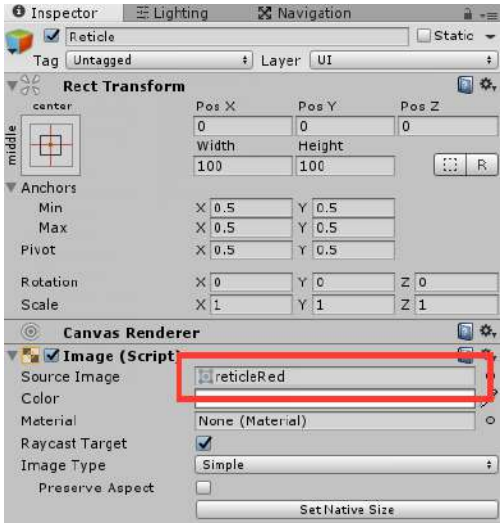
It's time to add the reticle each gun will use. The reticle is just a simple 2D image placed in front of the camera. This is the equivalent of pasting a target over the lens of your cell phone camera! It's not very technical, but it gets the job done.

You'll do this using Unity's user interface tools. In a few more chapters, you'll really learn about them in depth. For this chapter, consider this section a warm-up!

To get started placing user interface elements, you need a canvas. Unity provides several types of canvases. For this game, you'll use an overlay canvas. This works as an overlay on top of the screen. Elements added to the overlay will cover the game behind it, but you can minimize the effect through transparency.

In the Hierarchy, click Create and select **UI\Canvas** and rename the created GameObject to **GameUI**. Right-click on the **GameUI** GameObject, select **UI\Image** and rename the **Image** to **Reticle**.

In the Inspector, click the control to the right of the **Source Image** box and select **reticleRed** from the dialog.



You'll now see the red reticle in the center of the screen.

The shotgun and assault rifle have coordinating colored reticles, so the reticle should update when you switch guns.

To do this, create a new script and name it **GameUI**. Then create a new GameObject in the Hierarchy and name it **GameUI** as well. Attach the **GameUI** script to the **GameUI** GameObject.

Open **GameUI** in your code editor and add the following to the top of the script:

```
using UnityEngine.UI;
```

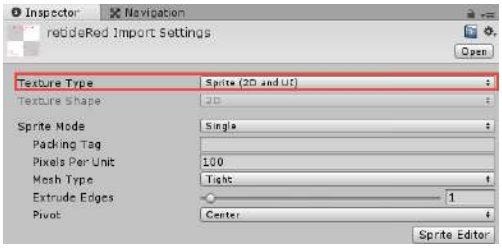
This lets you access the base Unity UI classes. Next, add the following variables inside the class:

```
[SerializeField]
Sprite redReticle;
[SerializeField]
Sprite yellowReticle;
```

```
  [SerializeField]
  Sprite blueReticle;
  [SerializeField]
  Image reticle;
```

There's a few things going on in this code block. First, you'll notice the `SerializeField` attribute. Attributes allow you to convey information to the C# runtime about particular classes. In this case, you are declaring that variables are accessible from the Unity Inspector but not from other scripts.

You'll notice there is both a **Sprite** and an **Image**. A sprite represents an imported texture meant to be used with a 2D game or the user interface (UI). These are your JPG and PNG files that you have imported into Unity. By setting the texture type to a Sprite in the Inspector, you can add them to the user interface.



An Image is what displays the Sprite to the screen. You can think of it like so: the Sprite is a film reel whereas the Image is the film projector.

In this case, you have three Sprites acting as source image data for the reticle. Since only one will be displayed, there is only one Image.

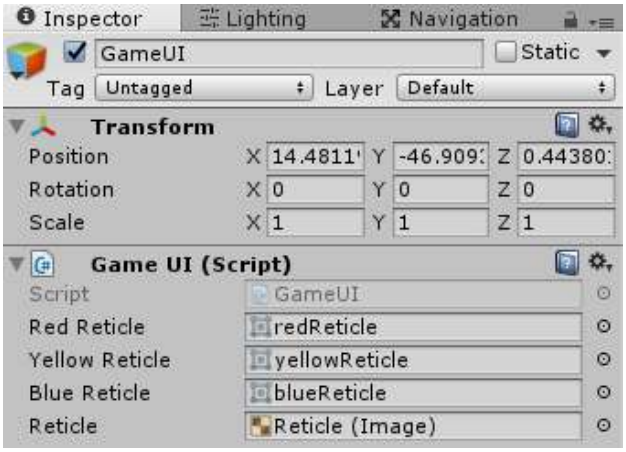
Add the following method below the last variable:

```
public void UpdateReticle() {
  switch (GunEquipper.activeWeaponType) {
    case Constants.Pistol:
      reticle.sprite = redReticle;
      break;
    case Constants.Shotgun:
      reticle.sprite = yellowReticle;
      break;
    case Constants.AssaultRifle:
      reticle.sprite = blueReticle;
      break;
    default:
      return;
  }
}
```

This will change the sprite to reflect the active gun.

Save your work and switch back to Unity.

Click the **GameUI** GameObject, click on each **[ColorName]Reticle** box and add the appropriate Sprite. For example, you'd add the **redReticle** Sprite to **Red Reticle**. When you've added all three, drag the **Reticle** GameObject to the **ReticleField** like so:

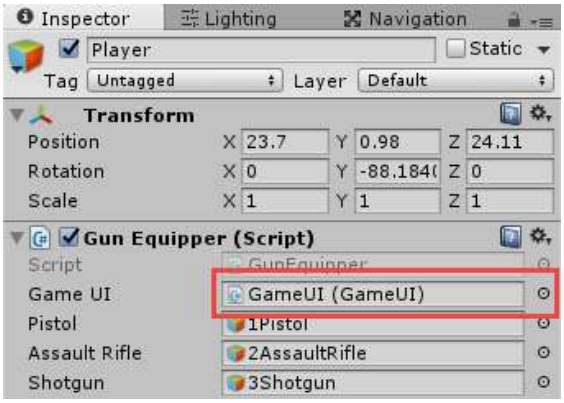


You'll need to call `UpdateReticle()` from somewhere to change the reticles. The **GunEquipper** script looks like the perfect spot.

Open the **GunEquipper** script in your code editor. Create a reference to the **GameUI** by creating a `GameUI` variable at the top of the script, just below the opening brace:

```
  [SerializeField]
GameUI gameUI;
```

Save the script and go back to Unity. Select the **Player** GameObject in the Hierarchy and drag the **GameUI** GameObject to the **Game UI** field in the **Gun Equipper** component:



Now open the **GunEquipper** script in the code editor and add the following line to the end of each `if` statement in **Update()**:

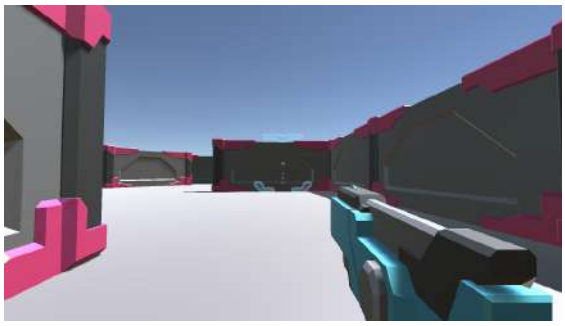
```
gameUI.UpdateReticle();
```

`Update()` should now look like the following:

```
void Update() {
  if (Input.GetKeyDown("1")) {
    loadWeapon(pistol);
    activeWeaponType = Constants.Pistol;
    gameUI.UpdateReticle();
  } else if (Input.GetKeyDown("2")) {
    loadWeapon(assaultRifle);
    activeWeaponType = Constants.AssaultRifle;
    gameUI.UpdateReticle();
  } else if (Input.GetKeyDown("3")) {
    loadWeapon(shotgun);
    activeWeaponType = Constants.Shotgun;
    gameUI.UpdateReticle();
  }
}
```

This will update the reticle every time the player changes weapons.

Click Play and switch between weapons to see the reticle change:



Now the player can see exactly where their shots will go.

Managing ammunition

Since the player has to manage their ammunition in Robot Rampage, you should provide some audible feedback so the player knows when they've run dry. If the player has sufficient ammunition, play a "fire" sound along with the fire animation. If the player is out of ammunition, play an empty chamber "click" sound and don't run the fire animation.

You've probably guessed you simply need a variable to track the amount of ammunition. That would work, but in Robot Rampage you need to worry about the type of ammunition as well. In this case, it makes sense to have an **Ammo** class.

Create a C# script, name it **Ammo**. Open it in your code editor and add the add the following import to the top (if it's not there already):

```
using System.Collections.Generic;
```

This lets you use the `Dictionary` type. In the body of the class at the top, add the following variables:

```
[SerializeField]
GameUI gameUI;

[SerializeField]
private int pistolAmmo = 20;
[SerializeField]
private int shotgunAmmo = 10;
[SerializeField]
private int assaultRifleAmmo = 50;

public Dictionary<string, int> tagToAmmo;
```

The variables `pistolAmmo`, `shotgunAmmo`, `assaultRifleAmmo` track their respective ammunition counts. `tagToAmmo` is a dictionary of type `string` and `int`, which lets you map a gun's type to its ammunition count.

You need to initialize the Dictionary at game start, so add the following method below where you declare the Dictionary:

```
void Awake() {
  tagToAmmo = new Dictionary<string, int> {
    { Constants.Pistol , pistolAmmo},
    { Constants.Shotgun , shotgunAmmo},
    { Constants.AssaultRifle , assaultRifleAmmo},
  };
}
```

`Awake()` is a special method called before `Start()`. In this case, you want the dictionary to initialize before any `Start()` methods to prevent null access errors. This method simply makes each gun type a key in the dictionary and sets that key's value to the appropriate ammunition type. If you ever wanted to add another ammunition type, you can simply extend the dictionary.

Add the following method:

```
public void AddAmmo(string tag, int ammo) {
  if (!tagToAmmo.ContainsKey(tag)) {
```

```
      Debug.LogError("Unrecognized gun type passed: " + tag);
    }

    tagToAmmo[tag] += ammo;
  }
```

This will add ammunition to the appropriate gun type. If you've passed in an unrecognized bad gun type, you log it as an error.

Next add the following method:

```
  // Returns true if gun has ammo
  public bool HasAmmo(string tag) {
    if (!tagToAmmo.ContainsKey(tag)) {
      Debug.LogError("Unrecognized gun type passed: " + tag);
    }

    return tagToAmmo[tag] > 0;
  }
```

This will return `true` if the gun type has at least 1 bullet left, or `false` if it has no more bullets.

Add the following method as well:

```
  public int GetAmmo(string tag) {
    if (!tagToAmmo.ContainsKey(tag)) {
      Debug.LogError("Unrecognized gun type passed:" + tag);
    }

    return tagToAmmo[tag];
  }
```

This simply returns the bullet count for a gun type.

Finally, you need a method to actually consume ammunition. Add the following:

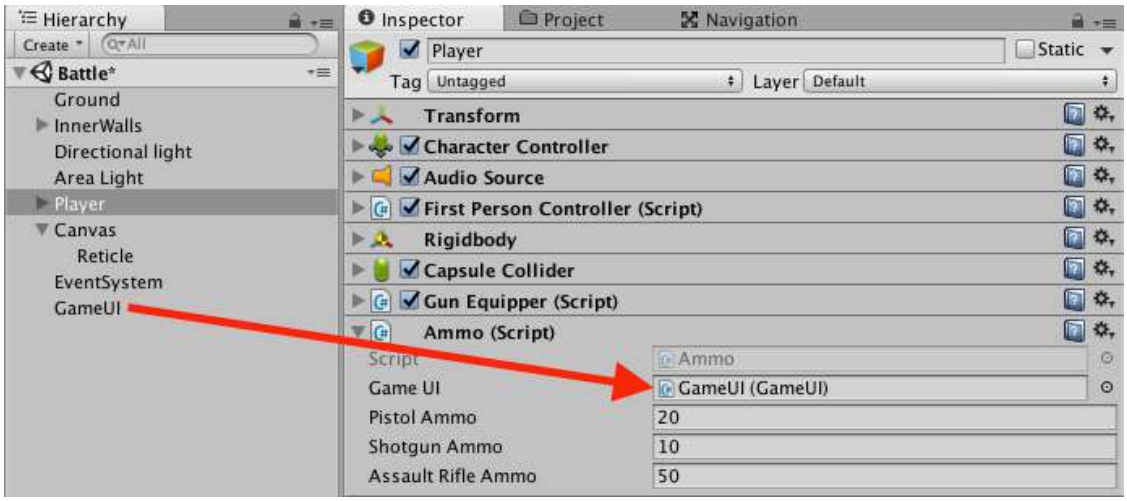
```
  public void ConsumeAmmo(string tag) {
    if (!tagToAmmo.ContainsKey(tag)) {
      Debug.LogError("Unrecognized gun type passed:" + tag);
    }

    tagToAmmo[tag]--;
  }
```

Like all the other methods, this checks for the correct tag. If it finds the appropriate ammunition, it subtracts a bullet.

At first glance, this code might seem like overkill, since you could simply access each ammunition type directly instead of using a dictionary. The problem with that approach is that when you add new ammunition types, you end up with redundant code along with getters and setters for every type of ammunition! With a dictionary, you can

simply map it out once and fetch the corresponding ammunition by its tag, rather than having to worry about calling a specific method.

Save your work, switch back to Unity, and attach the **Ammo** script to the **Player** GameObject. Click on the **Player** GameObject and drag the **GameUI** GameObject to the **GameUI** field in the **Ammo (Script)** component.



All that's missing is sound. Open the **Gun** script in the code editor and add the following just below where you declared `fireRate`:

```
public Ammo ammo;
public AudioClip liveFire;
public AudioClip dryFire;
```

This tracks the gun's ammunition and stores references to the fire and dry-fire sound effects.

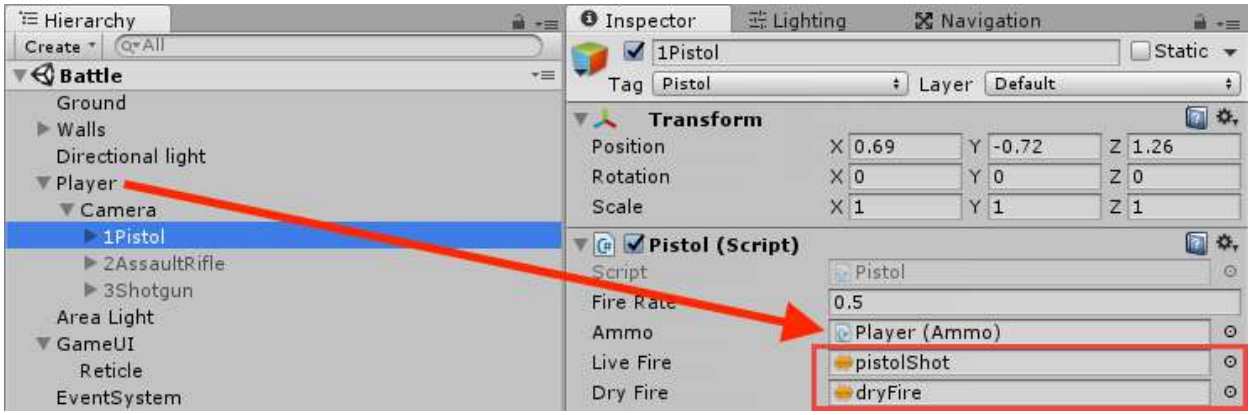
Next change `Fire()` to the following:

```
protected void Fire() {
  if (ammo.HasAmmo(tag)) {
    GetComponent<AudioSource>().PlayOneShot(liveFire);
    ammo.ConsumeAmmo(tag);
  } else {
    GetComponent<AudioSource>().PlayOneShot(dryFire);
  }
  GetComponentInChildren<Animator>().Play("Fire");
}
```

This checks if the player has any remaining ammunition. If so, you play the `liveFire` sound; otherwise, play the `dryFire` sound. Of course, if you fire the gun, then you should expend one bullet.

You now need to set the Ammo and AudioClips for each gun.

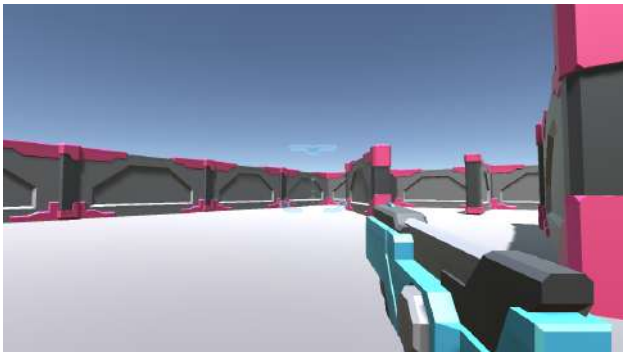
Click the **1Pistol** GameObject and drag the **Player** GameObject to the **Ammo** field to create a reference to the players **Ammo** script. Click the control to the right of the **LiveFire** field and select the **pistolShot** audio file. Next, click the control to the right of the **dryFire** field and add the **dryFire** audio.



Do this again for both the **3Shotgun** and the **2Assaultrifle** using their respective sounds.

All that's left is to add an **AudioSource** component. Select the **1Pistol**, **2AssaultRifle**, and **3Shotgun** GameObjects. In the Inspector click **AddComponent**, type **AudioSource** and select the AudioSource component.

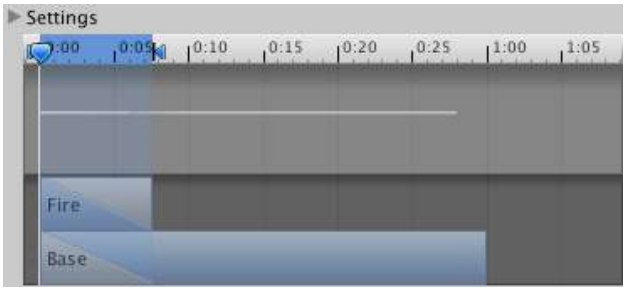
Click Play and fire away with each gun in turn; as long as you have ammunition, you should hear the fire sound. Once you run out of ammunition, you should hear the dry-fire sound instead and the fire animation should not play.



There is one issue with the assault rifle. The gun fires faster than the animations. This makes the motion out of sync with the action. To fix this, you need to return to the animator.

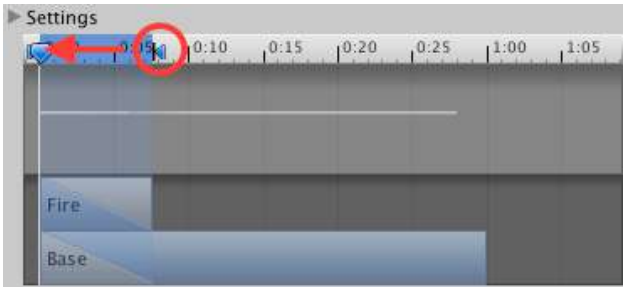
If your animator window is closed, make sure to reopen it by clicking **Window\Animator**. In the Hierarchy, select the **2AssaultRifle** GameObject and you'll see its animation states in the animator.

By default, animations blend into each other during a transition. This blending is referred to as exit time. **Select** the **transition** from **Fire** to **Base**. In the Inspector, you'll the visual depiction of the exit time under the **Settings** category.

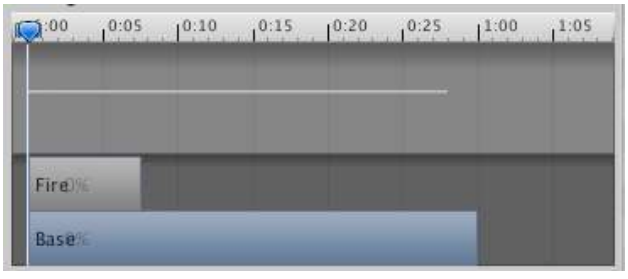


There are two ways to remove exit time. You can uncheck the `Has Exit Time` property, but you can only do this when you have a condition. Since you don't have a condition, you can set the duration of the transition to zero. This does the same thing.

To do this, drag the right blue arrow in the timeline all the way to the left and hit enter to save it.



When done, it will like like this:



This causes new animations to interrupt any playing animations. Play your game and switch to the assault rifle. Hold down the fire button and channel your inner Rambo!

Where to go from here?

That does it for this chapter. In the next chapter, you'll add enemies to feel the wrath of your weapons — and some pickups so you can replenish your ammunition and health, among other things. You'll also get to use the NavMesh you've created and add the damage mechanics for the player and the robots.

Chapter 10: Adding Enemies

By Anthony Uccello

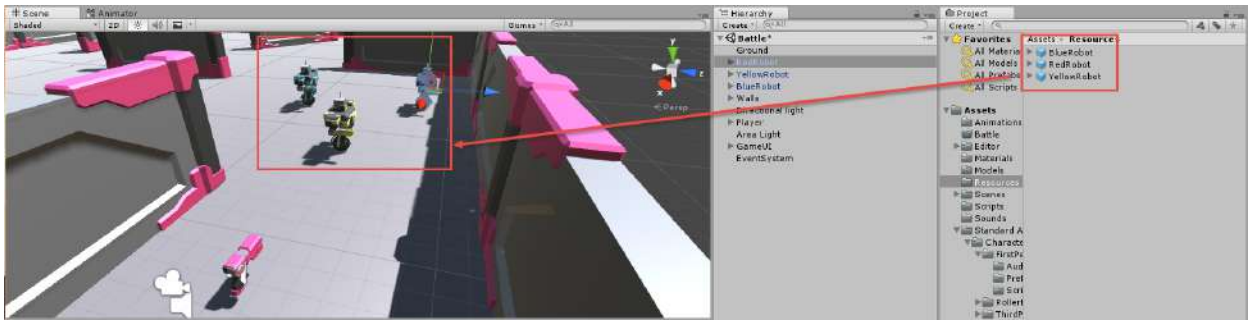
In the last chapter you added the first person controls, let the player switch guns, and added visual and audio effects. But what good is firing a gun if you don't have something to shoot at — and what fun is it if they can't shoot back? :]

In this chapter, you'll add some robots to the mix, but to make things interesting, these will be evil robots. Evil robots that will try to kill you. You'll add lots of them, give them the ability to fire, but to make things fair, you'll also add some pickups as well. In short, by the end of this chapter, you'll have yourself a first person shooter.

Creating the robots

Open the current project in progress, or you can start fresh by opening the starter project for this chapter included with the resources folder. Open the **Battle** scene to pick up where you left off.

Click on the **Resources** folder and drag **RedRobot, YellowRobot, and BlueRobot** onto the scene in front of the player, somewhere above ground level:



You'll now configure them all together, instead of separately as you've done.

Click on the **RedRobot** GameObject, and hold Control, or Command on a Mac, while selecting the **YellowRobot** and the **BlueRobot** GameObjects.

> **Note**: Whenever you're asked to select the robots in future, select all three of these robot GameObjects in this way.

Select all the Click **Add Component** and select the **New Script** option. Name the script **Robot**, and ensure **C Sharp** is the chosen Language. This creates a new script in the **Assets** folder.

To keep things organized, move the **Robot** script from the **Assets** folder to the **Scripts** folder. Open the **Robot** script in your code editor.

Add the following variables to your new **Robot** script inside the class:

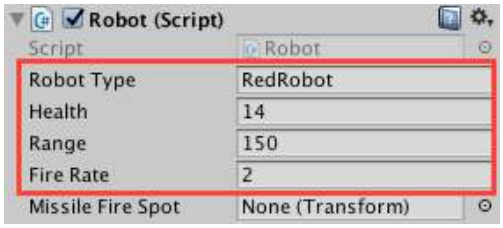
```
[SerializeField]
private string robotType;

public int health;
public int range;
public float fireRate;

public Transform missileFireSpot;
UnityEngine.AI.NavMeshAgent agent;

private Transform player;
private float timeLastFired;

private bool isDead;
```

`robotType` is the type of robot: `RedRobot`, `BlueRobot`, or `YellowRobot`. Notice that you are serializing the variable. Again, this is so you can access it in the Inspector but not from other scripts. The value of this field matches a constant defined in the `Constants` script.

`health` is how much damage this robot can take before dying, `range` is the distance the gun can fire, and `fireRate` is how fast it can fire.

`agent` is a reference to the Nav Mesh Agent component, `player` is what the robot should track, and `isDead` tracks whether the robot is alive or dead.

Save your work and switch back to Unity. Click the **RedRobot** GameObject and enter **RedRobot** in the **robotType** field. Set **Health** to **14**, the **Range** value to **150**, and the **Fire Rate** to **2**. Click **Apply** when done to apply those changes to the prefab.

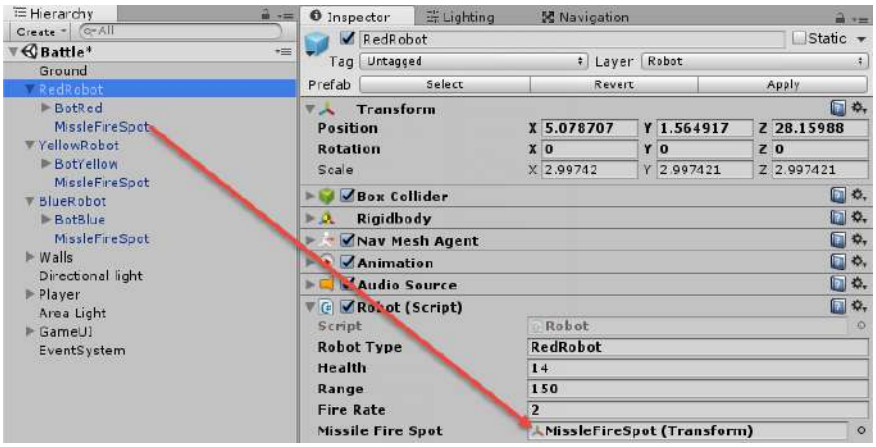


Do the same for the **YellowBot** with the following values: **Robot Type YellowRobot**, **Health 20**, **Range 300**, **Fire Rate 3**. Click **Apply** when done.

And of course, do the same for the **BlueRobot** with the following values: **Robot Type BlueRobot**, **Health 10**, **Range 200**, **Fire Rate 1**. Click **Apply** when done.

The **Missile Fire Spot** parameter is a coordinate on the robot model from which the missiles fire.

Click the arrow next to robot GameObject to display its children. Select each robot GameObject, one at a time, and drag its child GameObject **MissileFireSpot** to the **Missile Fire Spot** field on the **Robot** Component. Make sure to **apply** the change.



Now open the **Robot** script in your code editor and add and update the following methods:

```
void Start() {
  // 1
  isDead = false;
  agent = GetComponent<UnityEngine.AI.NavMeshAgent>();
  player = GameObject.FindGameObjectWithTag("Player").transform;
}

// Update is called once per frame
void Update() {
  // 2
  if (isDead) {
    return;
  }
```

```
    // 3
    transform.LookAt(player);

    // 4
    agent.SetDestination(player.position);

    // 5
    if (Vector3.Distance(transform.position, player.position) < range
        && Time.time - timeLastFired > fireRate) {
      // 6
      timeLastFired = Time.time;
      fire();
    }
  }

  private void fire() {
    Debug.Log("Fire");
  }
```

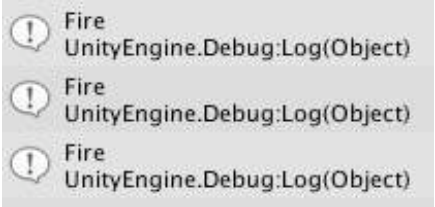
Looking at the numbered comments, here's what this code is doing:

1. By default, all robots are alive. You then set the `agent` and `player` values to the NavMesh Agent and Player components respectively.
2. Check if the robot is dead before continuing. There are no zombie robots in this game!
3. Make the robot face the player.
4. Tell the robot to use the NavMesh to find the player.
5. Check to see if the robot is within firing range and there's been enough time between shots to fire again.
6. Update `timeLastFired` to the current time and call `Fire()`, which simply logs a message to the console for the time being.

You need to assign an appropriate tag to the Player GameObject so the robots can find it.

Save your script and switch back to Unity. Click the **Player** GameObject, and as you did before with the guns, click the **Tag** dropdown and select the **Player** tag.

Click Play; you'll see the robots move towards the player and the "Fire" log should appear in the Console.

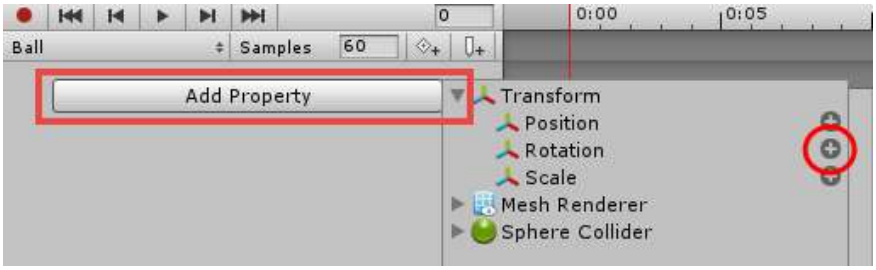


Animating robots

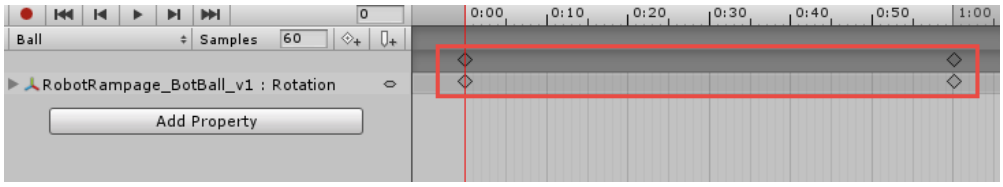
Right now the robots move statically towards the player; it would look much better if they actually rolled on their ball foot. As well, the robots should make a firing motion when they fire at the player.

Click a **RobotRampage_BotBall_v1** GameObject on any Robot (expand the arrows on its children to find it), then go to **Window\Animation** to open the **Animation** window. Click the **Create** button:

Enter **Ball** for the name, set the path to the **Animations** folder, then click Save. Click **AddProperty**, click the arrow beside **Transform** then click the + next to **Rotation**:

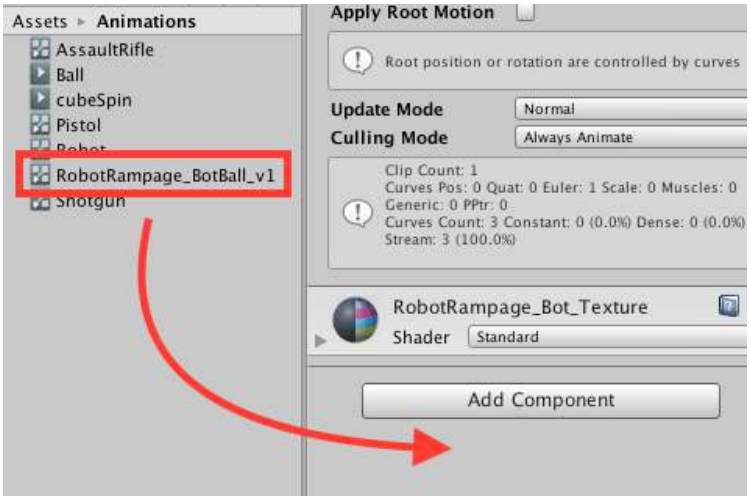


A pair of keyframes (grey diamonds) will appear 1 second apart on the timeline



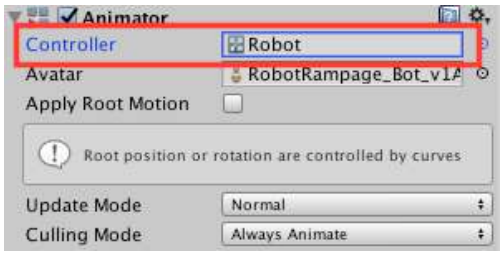
Click the last keyframe and notice that the **Rotation** fields for the GameObject have turned red in the Transform component in the Inspector. Set the **Rotation X** value to **-360**. This will cause the ball to rotate 360 degrees.

Select the other two **RobotRampage_BotBall_v1** GameObjects at the same time. You can find these by expanding the child GameObjects **BotRed**, **BotYellow**, and **BotBlue**. With all the **RobotRampage_BotBall_v1** GameObjects selected, drag the **RobotRampage_BotBall_v1** into the Inspector.



Click Play and you'll see the robot's ball spin:

Now to add the firing animation. Select all three **RobotRampage_Bot** GameObjects and set their **Animator**'s **Controller** to **Robot** by clicking the circle beside the **Controller** field and selecting **Robot**:



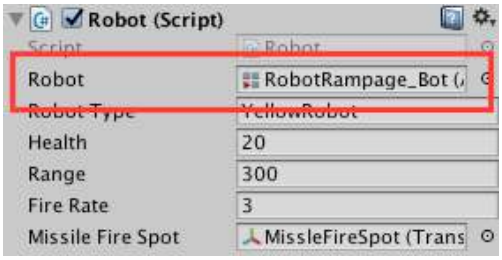
Select each **RobotRampage_Bot** individually and hit **Apply** on each one to apply the changes to the prefabs. Now open the **Robot** script. Add the following instance variable:

```
public Animator robot;
```

Then update `Fire()` to the following:

```
private void fire() {
  robot.Play("Fire");
}
```

Save and switch back to Unity. Select the **YellowRobot** and drag the child **RobotRampage_Bot** into the **Robot** property. **Apply** the change to the prefab.



This plays the **Fire** animation when the robot fires a missile — this is one of the states inside the **Robot** animator controller. Save your work and switch back to Unity.

Do this for the **RedRobot** and the **BlueRobot**, making sure to **apply** your changes when finished.

Click Play and you'll see the fire animation run when the robot fires.

That looks good, but it's missing something. Oh, right — an actual missile! :]

Firing robot missiles

Create a C# script and name it **Missile**. Drag a copy of each of the **RobotMissile[Color]** prefabs from the **Resources** folder onto the Hierarchy. It doesn't matter where they end up on screen because you'll be removing them soon.

Select each of the **RobotMissile[Color]** GameObjects you just dragged into the Hierarchy in turn, and drag the **Missile** script on to each of them.

Open the **Missile** script in the code editor add the following variables:

```
public float speed = 30f;
public int damage = 10;
```

`speed` is how fast the missile should travel. `damage` is how much damage this missile will cause when it hits the player.

Now add the following methods just below the `damage` variable:

```
//1
void Start() {
  StartCoroutine("deathTimer");
}

// 2
void Update() {
  transform.Translate(Vector3.forward * speed * Time.deltaTime);
}

// 3
IEnumerator deathTimer() {
  yield return new WaitForSeconds(10);
  Destroy(gameObject);
}
```

Welcome to coroutines! In computer jargon, you'll often hear the word `threading` tossed around a bit. This is just a way of having the computer do multiple things at once. In a game, this would mean calculating your artificial intelligence in one thread while playing sound in another.

Threads are a gnarly topic and while you can use them in Unity, you can simulate threading without all the pain and suffering through the use of coroutines. A coroutine will run code at a certain designated time. They run on the main thread and work pretty much like regular code, except they run in intervals. Because they run in the main thread, you don't get any of the nasty concurrency headaches from regular threading.

Coroutines take methods that return `IEnumerator`. These determine the duration of the coroutine. Here's the breakdown of what you just wrote:

1. When you instantiate a Missile, you start a coroutine called **deathTimer()**. This is name of the method that the coroutine will call.
2. Move the missile forward at `speed` multiplied by the time between frames.
3. You'll notice that the method immediately returns a `WaitForSeconds`, set to 10. Once those ten seconds have passed, the method will resume after the `yield` statement. If the missile doesn't hit the player, it should auto-destruct.

See? Coroutines aren't so bad at all! :]

Save your work and switch back to Unity. Click each **RobotMissile[Color]** one at a time and click **Apply** to set the changes on the prefabs.

Open the **Robot** script. At the top of the file, add the following just below the opening brace of the class:

```
  [SerializeField]
  GameObject missileprefab;
```

`missilePrefab` is the prefab for the missile.

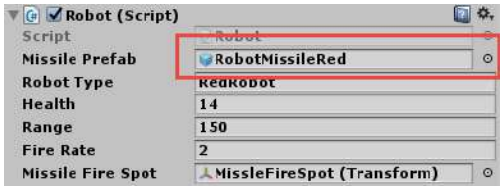
Update the `fire()` method to the following:

```
  private void fire() {
    GameObject missile = Instantiate(missileprefab);
    missile.transform.position = missileFireSpot.transform.position;
    missile.transform.rotation = missileFireSpot.transform.rotation;
    robot.Play("Fire");
  }
```

This creates a new `missilePrefab` and sets its position and rotation to the robot's firing spot. Save your work and switch back to Unity.

Delete the three **RobotMissile[Color]** GameObjects from the Hierarchy.

Click the **RedRobot** GameObject and drag the **RobotMissileRed** prefab from the **Resources** folder to the **Missile prefab** field on the **Robot** Component and click Apply.



Do the same for the **YellowRobot** and **BlueRobot** GameObjects using their respective colored missiles, making sure to **apply** your changes.

Click Play and — duck! Watch those missiles fly!

Adding damage effects

Now is the perfect time to add damage logic to the game. The missiles should damage the player, and the player's gun shots should damage the robots.

Create a new C# script, name it **Player** and add the following variables:

```
public int health;
public int armor;
public GameUI gameUI;
private GunEquipper gunEquipper;
private Ammo ammo;
```

`health` is the remaining life of the player. When this hits zero, it's time to grab another quarter! :]

`armor` is a special health layer for the player that results in 50% reduced damage. Once `armor` reaches 0, then player receives full damage.

`gameUI` and `gunEquipper` are simply references to their respective script types. `ammo` is a reference to the **Ammo** class you created earlier to track weapon ammo.

Add the following to `Start()`:

```
void Start () {
    ammo = GetComponent<Ammo>();
    gunEquipper = GetComponent<GunEquipper>();
}
```

This just gets the `Ammo` and `GunEquipper` component attached to the player GameObject.

Add the following method:

```
public void TakeDamage(int amount) {
  int healthDamage = amount;

  if (armor > 0) {
    int effectiveArmor = armor * 2;
    effectiveArmor -= healthDamage;

    // If there is still armor, don't need to process
    // health damage
    if (effectiveArmor > 0) {
      armor = effectiveArmor / 2;
      return;
    }

    armor = 0;
  }

  health -= healthDamage;
  Debug.Log("Health is " + health);
```

```
    if (health <= 0) {
      Debug.Log("GameOver");
    }
  }
```

`TakeDamage()` takes the incoming damage and reduces its amount based on how much armor the player has remaining. If the player has no armor, then you apply the total damage to the player's health.

If `health` reaches 0, it's game over for the player; for now, you just log this to the console.

Save your work and switch back to Unity. Add the **Player** script to the **Player** GameObject. Set **Health** to **100**, **Armor** to **20**, and add the drag the **GameUI** GameObject over the **GameUI** field. You'll be incorporating armor later in the game.

Next you need to update the **Missile** script to actually tell the player to take damage.

When the **Missile** GameObject collides with the **Player Capsule Collider**, it will damage the player via **OnCollisionEnter()**. Open the **Missile** script in the code editor and add the following method:

```
  void OnCollisionEnter(Collision collider) {
    if (collider.gameObject.GetComponent<Player>() != null
        && collider.gameObject.tag == "Player") {
      collider.gameObject.GetComponent<Player>().TakeDamage(damage);
    }

    Destroy(gameObject);
  }
```

The missile checks if it collided with the **Player** GameObject based on its tag. It also checks to see if if the player is still active, since the **Player** Component will be disabled later in the game over state. If so, it tells the `Player` script to take damage and passes along its damage value. Once the missile hits the player it destroys itself.

Save your work and switch back to Unity.

Click Play and stand in the line of missile fire. Watch as the console logs out the health of the player. In the next chapter, you'll add a UI so the health will be clearly visible on screen, but the console will do just fine for now.

Your game logs `GameOver` when the player's health hits 0. Later on, you'll code a proper game over screen.

Those pesky robots are still invincible. It's time for them to take some damage, too.

Open the **Robot** script and add the following methods:

```
// 1
public void TakeDamage(int amount) {
  if (isDead) {
    return;
  }

  health -= amount;

  if (health <= 0) {
    isDead = true;
    robot.Play("Die");
    StartCoroutine("DestroyRobot");
  }
}

// 2
IEnumerator DestroyRobot() {
  yield return new WaitForSeconds(1.5f);
  Destroy(gameObject);
}
```

1. This has roughly the same logic as the player `TakeDamage()` method, except when `health` hits 0 it plays a death animation before calling `DestroyRobot()`.

2. This adds a delay before destroying the robot. This lets the **Die** animation finish.

Save your work.

Now comes the fun part — actually shooting back. There's two things you need to do: deal damage to the robots but also implement zooming (otherwise known as iron sights).

Open **Gun** and add the following variables:

```
public float zoomFactor;
public int range;
public int damage;

private float zoomFOV;
private float zoomSpeed = 6;
```

`zoomFactor` controls the zoom level when the player hits the right mouse button. `zoomFOV` is the field of view based on the zoom factor. `range` is how far the gun can effectively hit its target. The shotgun has the shortest range, and the pistol has the longest. `damage` is how much damage the gun causes.

Save your work and switch to Unity.

Click the **1Pistol** GameObject and set its **Zoom Factor** to **1.3**, its **Range** to **60** and its **Damage** to **3**.

Click the **2AssaultRifle** GameObject and its **Zoom Factor** to **1.4**, its **Range** to **30** and its **Damage** to **1**.

Click the **3Shotgun** GameObject and set its **Zoom Factor** to **1.1**, its **Range** to **10** and its **Damage** to **10**.

Open the **Gun** script and update `Start()` to the following:

```
void Start() {
  zoomFOV = Constants.CameraDefaultZoom / zoomFactor;
  lastFireTime = Time.time - 10;
}
```

This simply initializes the zoom factor.

Modify `Update()` to the following:

```
protected virtual void Update() {
  // Right Click (Zoom)
  if (Input.GetMouseButton(1)) {
    Camera.main.fieldOfView = Mathf.Lerp(Camera.main.fieldOfView,
zoomFOV, zoomSpeed * Time.deltaTime);
  } else {
    Camera.main.fieldOfView = Constants.CameraDefaultZoom;
  }
}
```

If the player hits the right mouse button, this smoothly animates the zoom effect via `Mathf.Lerp()`.

To determine if the robot was hit, you'll use raycasting. Raycasting is when you fire an invisible ray to check for collisions. First, you must define the damage method. Add the following:

```
private void processHit(GameObject hitObject) {
  if (hitObject.GetComponent<Player>() != null) {
    hitObject.GetComponent<Player>().TakeDamage(damage);
  }

  if (hitObject.GetComponent<Robot>() != null) {
    hitObject.GetComponent<Robot>().TakeDamage(damage);
  }
}
```

This passes the damage to the correct GameObject. Now for the actual raycasting. You can review raycasting by reading Chapter 4, "Physics". To implement the raycasting, add the following to the bottom of `Fire()`:

```
Ray ray = Camera.main.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
RaycastHit hit;
if (Physics.Raycast(ray, out hit, range)) {
  processHit(hit.collider.gameObject);
}
```

This creates a ray and checks what that ray hits. It's that simple — a ray fires out at the range of the gun, and if it collides with a GameObject it calls `processHit()`. This determines if the GameObject hit was a robot; if so, it passes on the damage.

Save your work and switch back to Unity. Press Play and press the right mouse button to zoom, then blast those bad robots until they die.

Creating pickups

The player needs a way to gain back health, armor, and ammo. You'll do this by adding floating pickups the player can grab along the way.

Open the **Player** script in the code editor and add the following:

```
// 1
private void pickupHealth() {
  health += 50;
  if (health > 200) {
    health = 200;
  }
}

private void pickupArmor() {
  armor += 15;
}

// 2
private void pickupAssaultRifleAmmo() {
  ammo.AddAmmo(Constants.AssaultRifle, 50);
}

private void pickupPisolAmmo() {
  ammo.AddAmmo(Constants.Pistol, 20);
}

private void pickupShotgunAmmo() {
  ammo.AddAmmo(Constants.Shotgun, 10);
}
```

1. This adds to the players health and armor respectively.
2. This adds ammunition for that gun type.

Now add the following:

```
public void PickUpItem(int pickupType) {
  switch (pickupType) {
    case Constants.PickUpArmor:
      pickupArmor();
      break;
    case Constants.PickUpHealth:
      pickupHealth();
      break;
    case Constants.PickUpAssaultRifleAmmo:
      pickupAssaultRifleAmmo();
      break;
    case Constants.PickUpPistolAmmo:
      pickupPisolAmmo();
      break;
    case Constants.PickUpShotgunAmmo:
      pickupShotgunAmmo();
      break;
    default:
      Debug.LogError("Bad pickup type passed" + pickupType);
      break;
  }
}
```

`PickUpItem()` takes an `int` that represents the type of item being picked up. Now that the **Player** script can process picking up an item, it's time to create the pickup prefabs.

The `Constants` file references the IDs of all the pickups. These IDs correspond to the five types of pickups. The `int` passed into the `PickUpItem()` will be one of these values.

Save you work and switch back to Unity. Create a C# script, name it **Pickup** and add the following inside the class:

```
public int type;
```

This represents the type of the pickup. Now add the following:

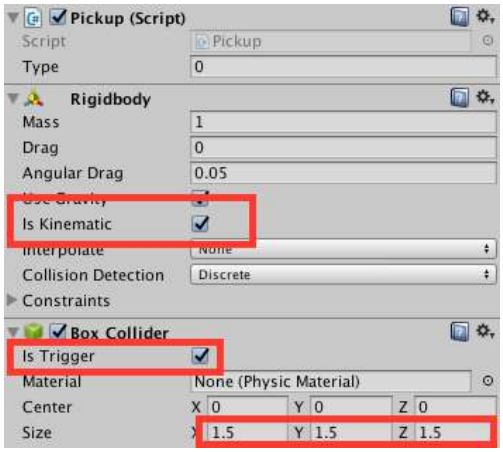
```
void OnTriggerEnter(Collider collider) {
  if (collider.gameObject.GetComponent<Player>() != null
    && collider.gameObject.tag == "Player") {

    collider.gameObject.GetComponent<Player>().PickUpItem(type);
    Destroy(gameObject);
  }
}
```

This listens for a collision with the **Player** GameObject, calls `PickUpItem()` on it, passes along its item type, and then destroys itself. You will add the respawn mechanics shortly.

Save your work and switch back to Unity.

In the **Resources** folder, select the **PickupAmmoAssaultRifle**, **PickupAmmoPistol**, **PickupAmmoShotgun**, **PickupHealth**, and **PickupArmor** prefabs. With all of those selected, click the **Add Component** button from the scripts category, select the **Pickup** script.

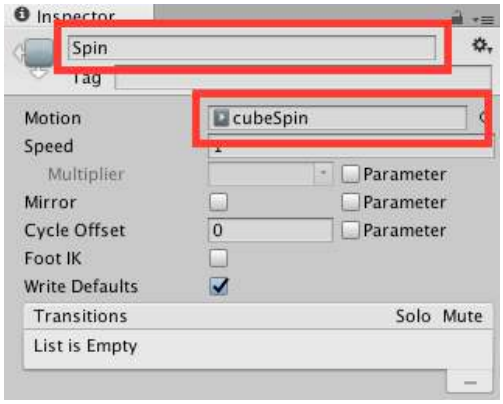
Next, add a **Rigidbody**. Check the **Is Kinematic** box. Finally, add a **Box Collider**. **Check** the **Is Trigger** checkbox and set the **Size** to (**1.5**, **1.5**, **1.5**).



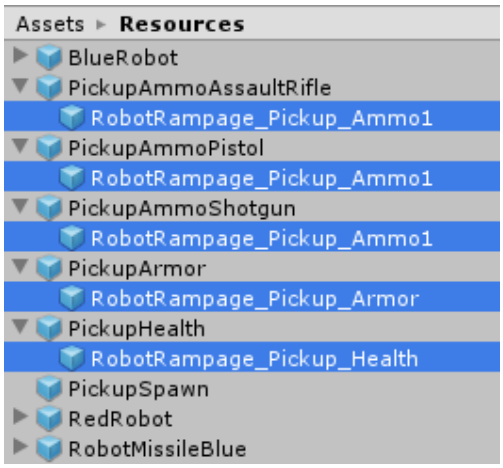
Now to configure each of the pickups. Select the **PickupAmmoPistol** prefab in the **Resources** folder and set its **Type** to **1**, set the **PickupAmmoAssaultRifle** to **2**, the **PickupAmmoShotgun** to **3**, the **PickupHealth** to **4**, and finally, the **PickupArmor** to **5**.

The last thing to do is add the spinning and bobbing animation to the pickup. The animation has been created for you, just need to create the animator. In the Project Browser, select the **Animations** folder and click the **Create** button. Select **Animator Controller** and name it **Pickup**.

Double click the **Pickup** animator controller to open the animator. Right click in the empty grid and select **Create State\Empty**. In the Inspector, name it **Spin** and set the **Motion** to **cubeSpin**.



Excellent! Your pickups are ready to spin! Next, return back to the Project Browser and select the **Resources** folder. Click the arrow next to each of the the **Pickup[Type]** prefabs and select the only child prefab in each:

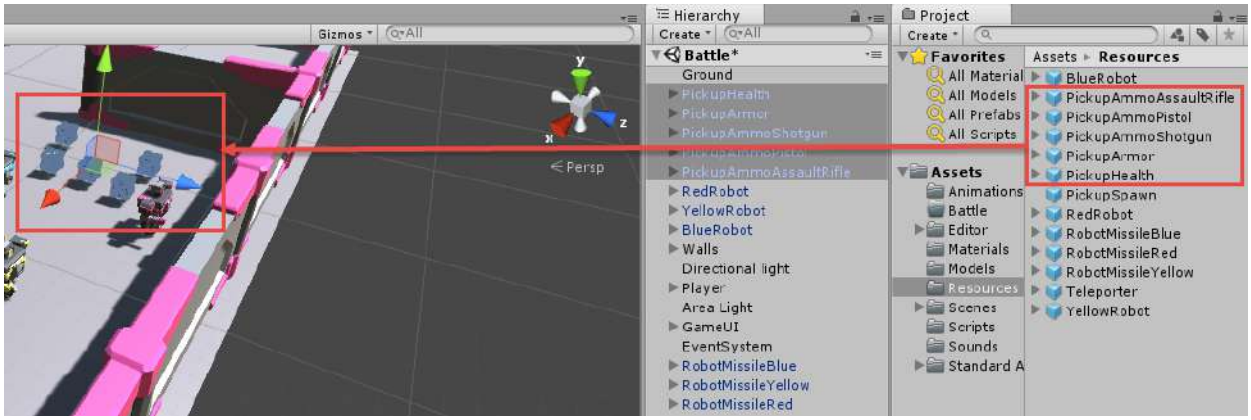


Lock the Inspector and in the Project Browser, select the **Animations** folder. Drag the **Pickup** Animator to the Inspector.

Once added, make sure to **unlock** the Inspector.

By adding the animation here, instead of the parent, the animation plays in local space.

Now drag a copy of each **Pickup[Type]** onto the scene, just slightly above the ground:



Click Play and move the player over each pickup; it should destroy itself when you make contact. You'll add a UI shortly to see the effects of these pickups on screen, but for now if the pickup GameObjects are floating and spinning, and disappear when you walk into them, you've got it all right so far!

More spawning logic

Instead of having the robots and pickups exist in the scene at the start of the game, you'll have them spawn at specific locations.

Select all the **Pickup[Type]** GameObjects and **[Color]Robot** GameObjects in the Hierarchy and press Delete.

You will now code the the pickups and robots to spawn at specific locations around the map.

Create a new C# script, name it **PickupSpawn**, and replace the contents of the file with the following:

```
  [SerializeField]
  private GameObject[] pickups;
```

`pickups` will store all the possible pickup types.

Now add the following methods:

```
  // 1
  void spawnPickup() {
    // Instantiate a random pickup
    GameObject pickup = Instantiate(pickups[Random.Range(0,
  pickups.Length)]);
    pickup.transform.position = transform.position;
    pickup.transform.parent = transform;
  }

  // 2
  IEnumerator respawnPickup() {
    yield return new WaitForSeconds(20);
    spawnPickup();
  }

  // 3
  void Start() {
    spawnPickup();
  }

  // 4
  public void PickupWasPickedUp() {
    StartCoroutine("respawnPickup");
  }
```

Here's what each method does:

1. Instantiates a random pickup and sets its position to that of the **PickupSpawn** GameObject, which you will make in a moment.
2. Waits 20 seconds before calling `spawnPickup()`.
3. Spawns a pickup as soon as the game beings.
4. Starts the Coroutine to respawn when the player has picked up.

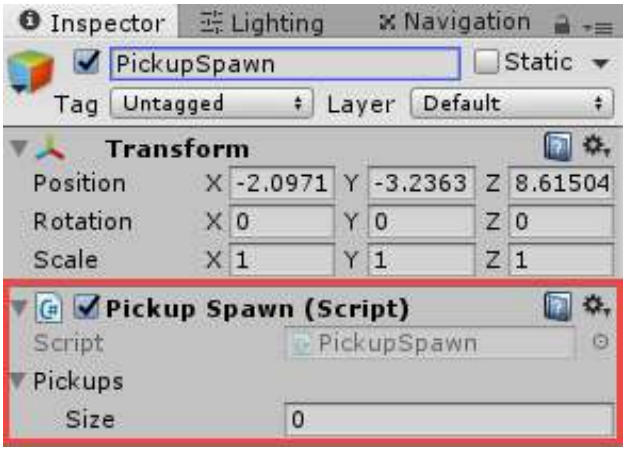
Save your work. Open the **Pickup** script and add the following line just above `Destroy(gameObject)`:

```
  GetComponentInParent<PickupSpawn>().PickupWasPickedUp();
```

Now when the pickup collides with the player, it will signal the **PickupSpawn** script to start the spawn timer.

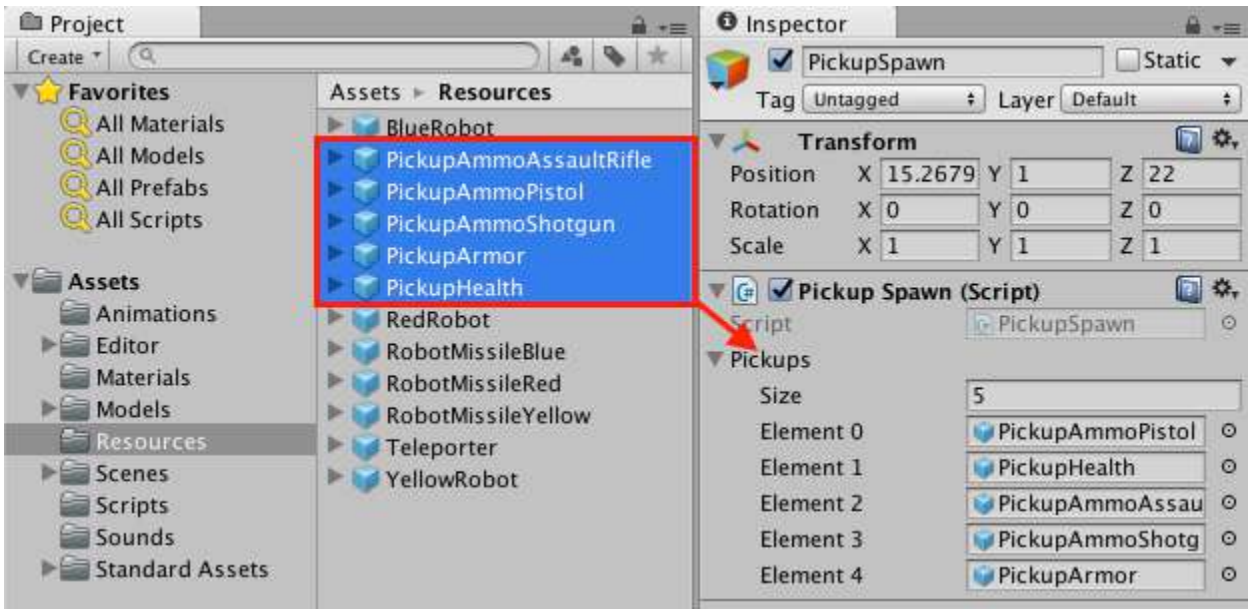
Save your work and switch back to Unity.

Create an empty GameObject, name it **PickupSpawn** and add the **PickupSpawn** script.



With the **PickupSpawn** GameObject selected, **lock** the Inspector.

In the Resources folder, select the **PickupAmmoAssaultRifle**, **PickupAmmoPistol**, **PickupAmmoShotgun**, **PickupHealth**, and **PickupArmor** prefabs and drag them to the **Pickups** property to add them to the array.

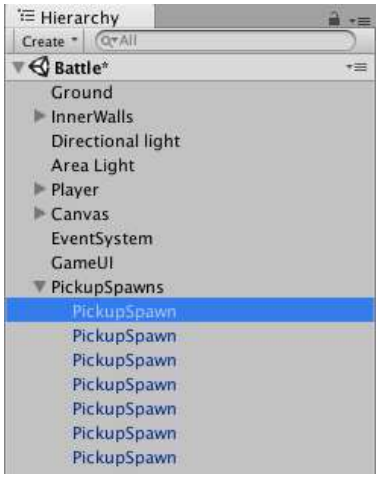


Unlock the Inspector.

Now drag the **PickupSpawn** GameObject into to the **Resources** folder to create a prefab of it.

Create an empty GameObject and name it **PickupSpawns**. Set its **position** all to (**0**, **0**, **0**). Drag the **PickupSpawn** GameObject underneath the **PickupSpawns** and **duplicate** it **six** times by press Control-D (or Command D, on the Mac).

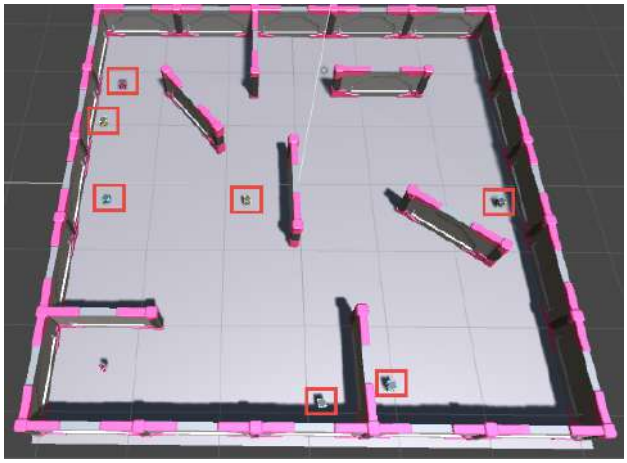
To stay organized, change the each of the names to: **PickupSpawn**.



Starting from the top most **PickupSpawn** GameObject and moving down, set the following positions:

- (**-26.0**, **0.5**, **3**)
- (**26.0**, **0.5**, **-17**)
- (**7.0**, **0.5**, **3.0**)
- (**-27.0**, **0.5**, **3.0**)
- (**28**, **0.5**, **-10.0**)
- (**-9.5**, **0.5**, **27.0**)
- (**-1.5**, **0.5**, **29.0**)

Press Play and you'll see seven pickups spawn in the scene. Feel free to walk around and pick them up.



Robot spawning

Now that the pickups are spawning, you can work on the robots. Drag the **Teleporter** prefab from the **Resources** folder into the scene at ground level.

Create a C# script, name it **RobotSpawn**. Open it in your code editor and add the following variables after the opening brace of the class:

```
[SerializeField]
GameObject[] robots;

private int timesSpawned;
private int healthBonus = 0;
```

`robots` is an array of robot GameObjects to instantiate; these will be the red, yellow, and blue robots. `healthBonus` is how much health each robot gains each wave to make the game get harder as you go along. `timesSpawned` tracks the spawn cycle of the robots.

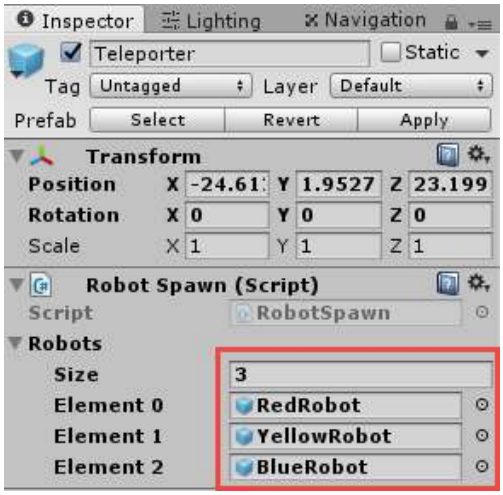
Next add the following method:

```
public void SpawnRobot() {
  timesSpawned++;
  healthBonus += 1 * timesSpawned;
  GameObject robot = Instantiate(robots[Random.Range(0, robots.Length)]);
  robot.transform.position = transform.position;
  robot.GetComponent<Robot>().health += healthBonus;
}
```

`SpawnRobot()` spawns a robot, sets its health, then sets the robot's position. Save your work and switch back to Unity.

Drag the **RobotSpawn** script on to the **Teleporter** GameObject. Select the **Teleporter** GameObject in the Hierarchy, **lock** the Inspector, and drag the **RedRobot**,

YellowRobot, and **BlueRobot** prefabs from the **Resource** folder onto the **Robots** field in the Inspector.

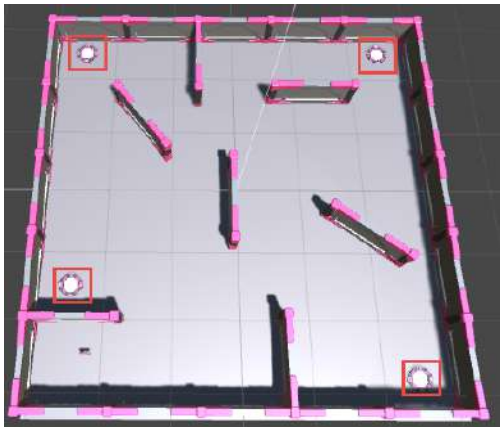


Hit **Apply** and **unlock** the inspector.

In the Hierarchy, create an **empty GameObject**, name it **RobotSpawns** and set its **position** to (**0**, **0**, **0**). Drag the **Teleporter** into the **RobotSpawns** and **duplicate** it three times by pressing Control-D (or Command-D on the Mac). Rename each of them **Teleporter**.

Starting from the top most **Teleporter** GameObject, set their positions to the following:

- (**-25.0**, **0**, **-25.0**)
- (**25.0**, **0**, **-25.0**)
- (**-25.0**, **0**, **28.0**)
- (**25.0**, **0**, **15.0**)



Want to make things challenging? Then add four more spawners!

In the next chapter you will add the logic to continually change waves. For now, you will simply spawn robots when you hit Play.

Create a C# script, name it **Game** and replace the contents of the file with the following:

```
private static Game singleton;

[SerializeField]
RobotSpawn[] spawns;

public int enemiesLeft;
```

This uses the **Singleton** pattern to have one and only one item; hence, a `singleton` GameObject reference. You only want one `Game` to track everything such as the score, the robots remaining and the current wave.

You'll add all that in the next chapter; for now, you'll spawn the robots just once.

`spawns` is the array of teleporters that spawn robots each wave, and `enemiesLeft` is a counter which tracks how many robots are still alive.

Now add the following methods below the final variable:

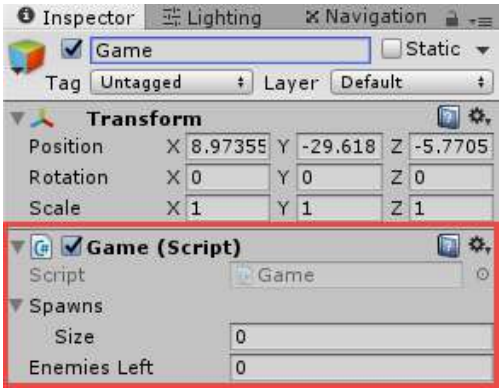
```
// 1
void Start() {
  singleton = this;
  SpawnRobots();
}

// 2
private void SpawnRobots() {
  foreach (RobotSpawn spawn in spawns) {
    spawn.SpawnRobot();
    enemiesLeft++;
  }
}
```

Here, you're doing the following:

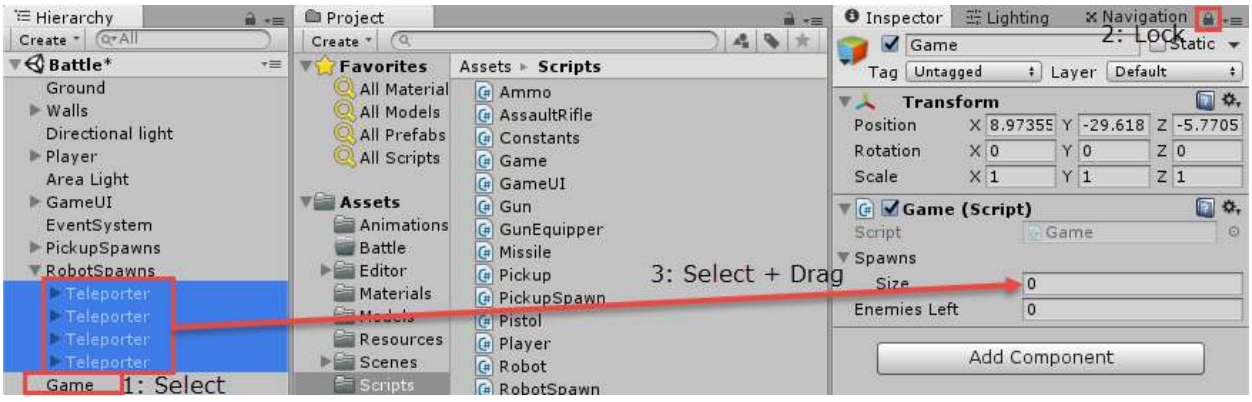
1. Initialize the `singleton` and call `SpawnRobots()`.
2. Go through each **RobotSpawn** in the array and call `SpawnRobot()` to actually spawn a robot. You will see a warning that the singleton is 'initialized but not used'. Ignore this for now; it will be used shortly in the next chapter.

Save your work and switch back to Unity.

Create an **empty GameObject** and call it **Game**. Drag the **Game** script onto this GameObject.



Select the **Game** GameObject and **lock** the Inspector. Drag the **Teleporter** GameObjects on to the **Spawns** field. When done, **unlock** the Inspector.



Click Play and you'll see four robots spawn. Feel free to blast them out of existence and grab some pickups while you're at it!

Adding sound

With the game mostly set up, it just needs a little sound.

Open the **Robot** script and add the following at the top:

```
[SerializeField]
private AudioClip deathSound;
[SerializeField]
private AudioClip fireSound;
[SerializeField]
private AudioClip weakHitSound;
```

Save your changes and head back to Unity. Select the **YellowRobot**, **RedRobot**, and **BlueRobot** Prefabs in the **Resources** folder by Shift-clicking (or Command-clicking on a Mac) and set the **Fire Sound** to **missile**, the **Death Sound** to **deadRobot**, and the **Weak Hit Sound** to **weakHitSound**.

Open **Robot** and add the following before the closing brace of **Fire()** to play the missile firing sound:

```
GetComponent<AudioSource>().PlayOneShot(fireSound);
```

Now, update `TakeDamage()` to play some sounds. Replace `if (health <= 0)` with the following:

```
if (health <= 0) {
    isDead = true;
    robot.Play("Die");
    StartCoroutine("DestroyRobot");
    GetComponent<AudioSource>().PlayOneShot(deathSound);
} else {
```

```
        GetComponent<AudioSource>().PlayOneShot(weakHitSound);
    }
```

Save the file and return back to Unity. Fire away and you'll hear lots of sounds!

Where to go from here?

That's it for this chapter; great job getting this far! If you ran into issues along the way you can always use the starter project in the next chapter, but it's always instructive to try and solve problems yourself first.

In the next chapter, you'll add the final layer of polish to the game with UI, more sounds, more animations, and some tweaked game mechanics like wave spawning.

Chapter 11: Introducing the UI

By Anthony Uccello

In the last chapter you got the pickups, robots spawning, and even some bullets flying. Now it's time to add the UI, wave spawning mechanics, enhanced sound and music, and doing extra damage by using the right gun for the right robot.

More importantly, you'll get an introduction into Unity's user interface system or otherwise known as the UI. This isn't the actual program itself, but rather, the tools used to create user interfaces in your games.

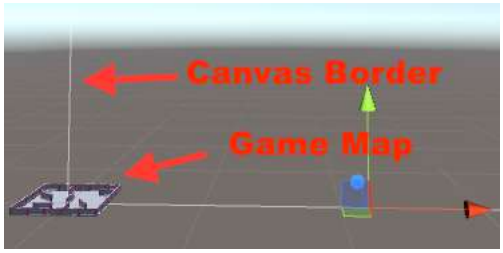
Pre-Unity 4.0, the UI tools were a disaster. For the most part, you had to write everything inside of `OnGui()` and as your UI grew in complexity, so did your code. Thankfully, Unity listened to developers and after many years, they released their new UI in Unity 4.6.

Getting started

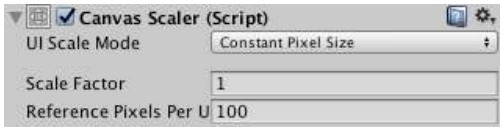
In Chapter 9, "Making a First Person Shooter" you created a **Canvas** named **GameUI**. A Canvas is the root object for all UI elements. You used an overlay canvas, but there other options. For instance, a **world space** canvas allows you to create interfaces that can be accessed in your own game! Think of the computer terminals from Doom 3.

For a canvas to be useful, you needs control and Unity comes with plenty of them. The most common UI elements are **Image**, **Button**, and **Text**.

The canvas is indicated by a giant white box — and you'll notice it's huge!



This is because the canvas is assuming 1 pixel of screen space to be 1 unit inside of Unity. The canvas pixel density can be changed using the **canvas scaler**.



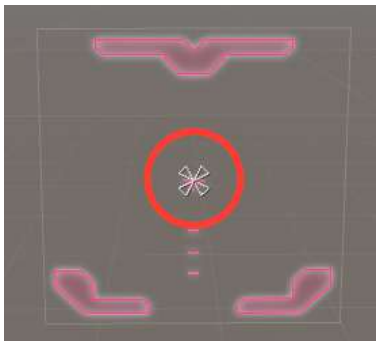
The scalar has a property called **UI Scale Mode** with some interesting options.

- **Constant Pixel Size** means the UI elements are the same size, regardless of screen size.
- **Scale with Screen Size** means UI elements grow bigger with the screen.
- **Constant Physical Size** means the UI elements will always be the same size, regardless of screen size or resolution.

This one component determines how your UI will look on different screen sizes and resolutions. For the purposes of this chapter, you'll be using the default values. Although the canvas shares the same space, you can effectively ignore it. Like any other GameObject, you can always deactivate it to maintain a clear "desk space" so to speak.

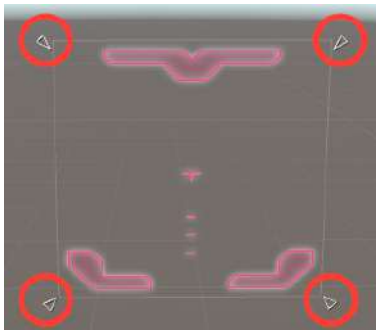
An Image is a UI class of objects that lets you render an image on the screen via a **Sprite**. **Text**, quite logically, lets you render text on the screen. **Button** lets you configure hover and click events. You'll be using these throughout this tutorial.

So you may be wondering, how are controls placed? Unity uses **anchors** and they look like arrows pointing at each other.



An anchor acts like an anchor in real life: it keeps something in place. If wanted to position a UI element in the top left corner and always stay relative to that top left corner, then you would simply set its Anchor to **top left**.

Anchors also can stretch an control. This is useful if you want a control expand or contract based on the screen size. The anchor breaks into individual arrows to form a rectangle. This allows you to define the size of the stretching.

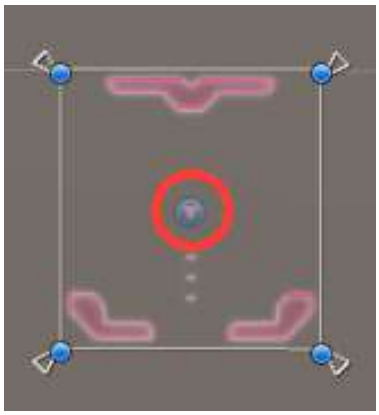


When talking about anchors it's important to understand what a **pivot** is. The pivot is the point on the Image that position is applied to, and it's what the Anchors use for keeping it locked in the right place. The pivot point is also the rotation point.

Having the pivot at the top left corner is the most common, but it can be useful to have the pivot set in various positions. For example, if you want something to be positioned in the top middle of the screen, setting the pivot to the top middle can make things a lot easier.

The pivot is set in normalized coordinates. This means that it goes from 0 to 1 for both height and width where (0,0) is the bottom left corner and (1,1) is the top right corner.

The pivot point is represented by a blue circle:



Through this chapter, you'll be setting the position and pivot points of various UI controls through the Inspector.

If you ever want to manually position controls, you can use the **Rect Tool**.

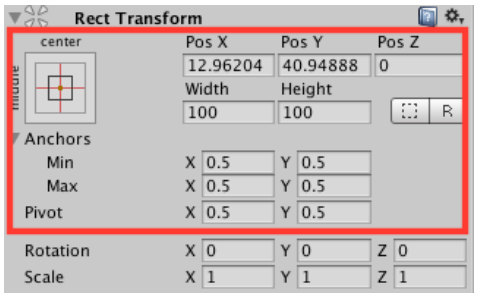
The Rect Tool is the Swiss Army knife of working with 2D. It allows you to scale, rotate, and even reset points. It even works with 3D objects as well although the tool was designed with 2D in mind.

You're about to create the UI, so the above theory will make perfect sense after you complete the section below. Load the **Battle** scene to start. If you need to start fresh, open the starter project in the resources.

Adding UI elements

You are going to create a bunch of Image and TextView UIGameObjects to go under the GameUI you created earlier.

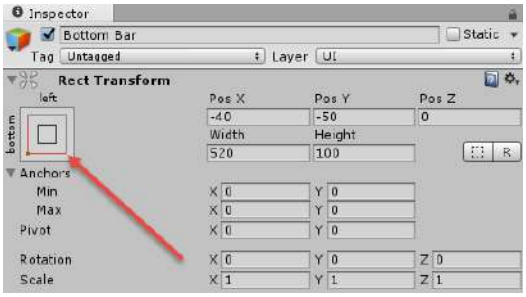
Select the **GameUI** GameObject and in the Hierarchy, right-click, select **UI\Image** and rename it **BottomBar**. Select the image and look at the Inspector. You'll notice it has changed.



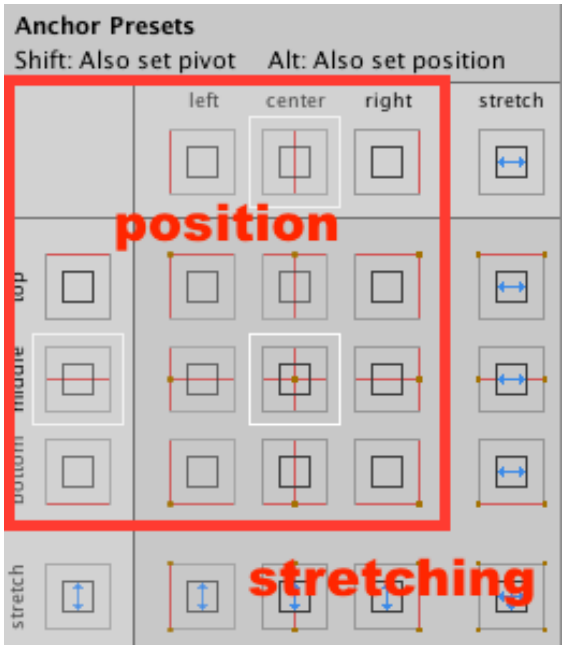
The position property has been replaced by a whole new set of properties that are specifically designed for the UI. This is called the **Rect Transform**.

Whenever you want to set the position of a control in code, you use the Rect Transfrom instead of the regular Transform.

To set this image at the bottom left of the screen, click the **anchor presets** box.



You'll get a whole bunch of options that corresponds to positions on the screen. Along the bottom and right hand side, you'll see stretching presets as well.

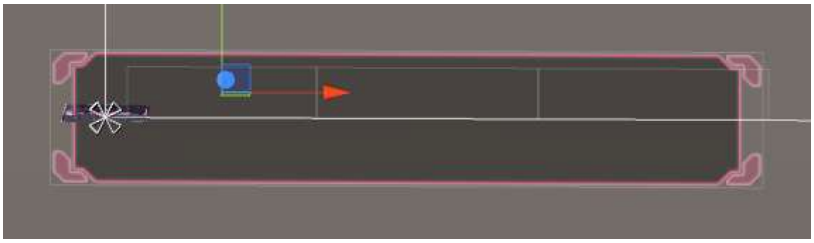


The keyword being: presets. You can do everything here manually by moving the anchor.

Time to get started. To set the position, click the **bottom left** preset.

Set **Pivot** to (**0**, **0**), **Pos** to (**-40**, **-50**, **0**), **Width** to **520**, **Height** to **100**, and **Source Image** to **Pink_Grey_Text**.

You'll end up with an image in the lower left hand side of the Canvas like so:



> **Note**: Since this is an overlay UI, it's helpful to switch the Scene view to isometric mode. In the Scene view, simply click the center cube in the gizmo to change modes, then click it again to return back to perspective mode.

The `Pos` fields work in relation to the anchor. In your case, you set the `PosY` to `-50`. This has the affect of lowering the image from the lower left hand corner of the canvas by `-50`.

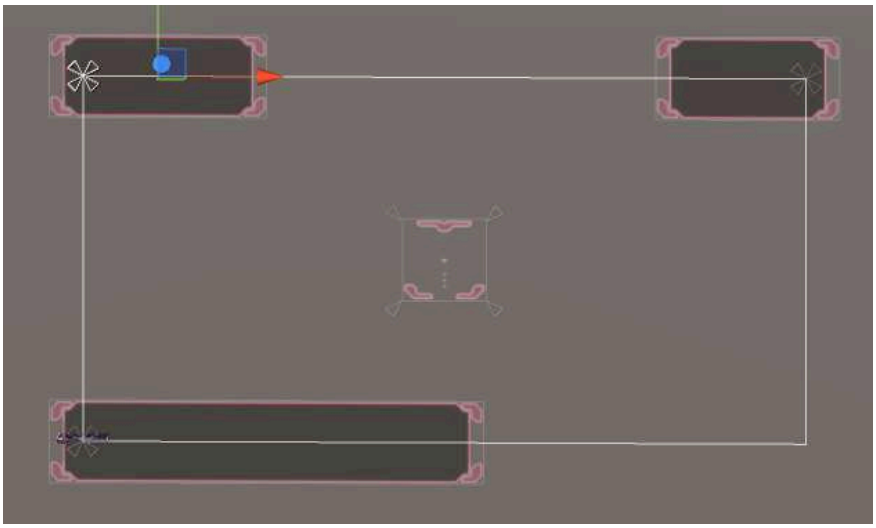
At a later point you may switch presets. You can do this, but the control won't change its on-screen placement although the position numbers change. This is due to Unity calculating the control's new position relative to the new anchor point.

At this point, you'll have a lot of new elements to place to create the UI. First start with the images.

Here's the breakdown:

Type	Name	Anchor Preset	Pivot	Pos	Width	Height	Source Image
Image	TopRightBar	top right	(1, 1)	(40, 50, 0)	220	100	Pink_Grey_Texture
Image	TopLeftBar	top left	(0, 0)	(-40, -50, 0)	260	100	Pink_Grey_Texture

Once done, your UI should look like the following:



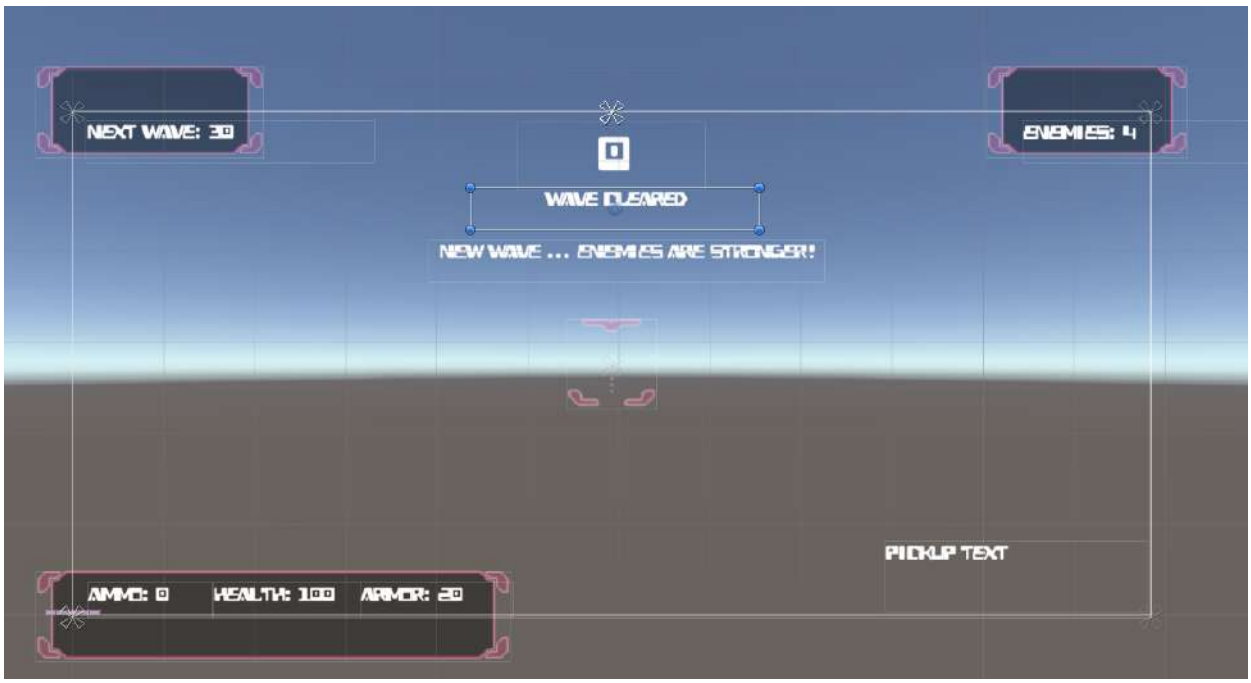
Now create following text items. To speed things up, use the duplicate key (Control-D or Command-D on the Mac).

Type	Name	Anchor Preset	Pivot	Pos	Width	Height	Text	Font	Font Style	Font Size	Color
Text	Ammo	bottom left	(0, 0.5)	(16.2, 19, 0)	138	37	Ammo: 0	heavy_data	Bold	25	White
Text	Health	bottom left	(0, 0.5)	(153.9, 19, 0)	162	37	Health: 100	heavy_data	Bold	25	White
Text	Armor	bottom left	(0, 0.5)	(315.9, 19, 0)	168	37	Armor: 20	heavy_data	Bold	25	White
Text	Score	top center	(0,0.5)	(-105, -48, 0)	208	75.3	0	heavy_data	Bold	62	White
Text	PickupText	bottom right	(0, 0.5)	(-292.1, 44.9,0	288.4	77.5	Pickup Text	heavy_data	Bold	25	White
Text	WaveText	top left	(0, 0.5)	(15, -32, 0)	317	47	Next Wave: 30	heavy_data	Bold	25	White
Text	EnemyText	top right	(0, 0)	(-141, -55.5, 0)	317	47	Enemies: 4	heavy_data	Bold	25	White
Text	WaveClearText	top center	(0, 0.5)	(-155.3, -106, 0)	317	47	Wave Cleared	heavy_data	Bold	25	White
Text	NewWaveText	top center	(0,0.5)	(-200,-164,0)	434	47	New Wave … Enemies are stronger!	heavy_data	Bold	25	White

After putting in all those values, you need to make a few more adjustments.

First, **center** align the following text GameObjects: **Score**, **WaveClearText**, and **NewWaveText**.

Next, **left justify** the **PickupText** GameObject.



Lastly, disable the Text component on the following GameObjects: **PickupText**, **WaveClearText**, and **NewWaveText**.

Now play your game. Congrats! You have a brand new user interface!

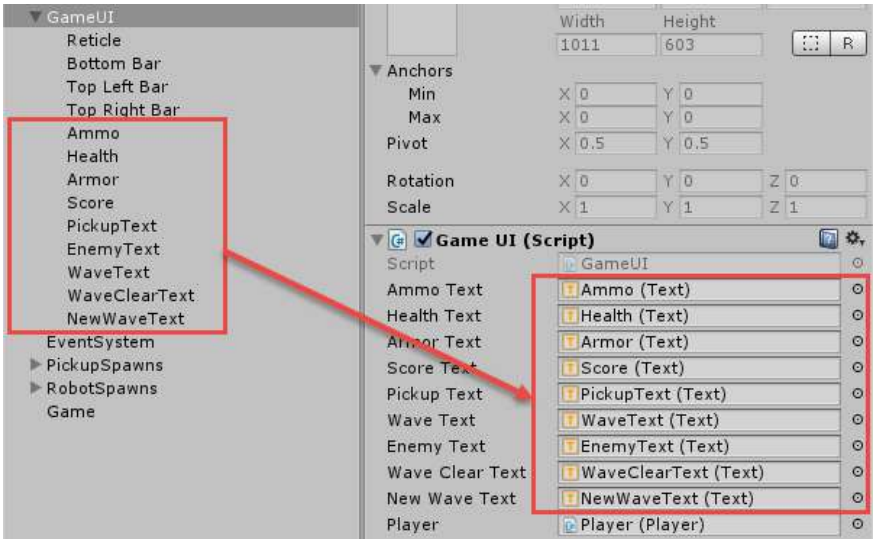
Coding the UI

Open the **GameUI** script in the code editor and add the following variables right below the opening brace of the class:

```
[SerializeField]
private Text ammoText;
[SerializeField]
private Text healthText;
[SerializeField]
private Text armorText;
[SerializeField]
private Text scoreText;
[SerializeField]
private Text pickupText;
[SerializeField]
private Text waveText;
[SerializeField]
private Text enemyText;
[SerializeField]
private Text waveClearText;
[SerializeField]
private Text newWaveText;
[SerializeField]
Player player;
```

These are simply references that correspond to the the UI elements you created above and to the **Player** as well. Save your work and go back to Unity.

Select the **GameUI** and in the Inspector, you'll see all your new fields. Drag all the controls you created to these empty fields. Also, drag the **Player** GameObject to the **Player** field.



Open the **GameUI** script in the code editor and add the following to the bottom of the class:

```
// 1
void Start() {
  SetArmorText(player.armor);
  SetHealthText(player.health);
}

// 2
public void SetArmorText(int armor) {
  armorText.text = "Armor: " + armor;
}

public void SetHealthText(int health) {
  healthText.text = "Health: " + health;
}

public void SetAmmoText(int ammo) {
  ammoText.text = "Ammo: " + ammo;
}

public void SetScoreText(int score) {
  scoreText.text = "" + score;
}

public void SetWaveText(int time) {
  waveText.text = "Next Wave: " + time;
}

public void SetEnemyText(int enemies) {
  enemyText.text = "Enemies: " + enemies;
}
```

Here's what's going on in the code:

1. Initializes the player health and ammo text.

2. This and the rest of the methods are simply setters that set the related text values.

Now to actually update the UI. To do this, you'll use coroutines. Coroutines were covered in the previous chapter so give it a read if you feel unsure of them.

Coroutines are perfect for the UI because you want to show some text for a few seconds and then hide them.

With this in mind, add the following methods to your **GameUI** script:

```
// 1
public void ShowWaveClearBonus() {
  waveClearText.GetComponent<Text>().enabled = true;
  StartCoroutine("hideWaveClearBonus");
}

// 2
IEnumerator hideWaveClearBonus() {
  yield return new WaitForSeconds(4);
  waveClearText.GetComponent<Text>().enabled = false;
}

// 3
public void SetPickUpText(string text) {
  pickupText.GetComponent<Text>().enabled = true;
  pickupText.text = text;
  // Restart the Coroutine so it doesn't end early
  StopCoroutine("hidePickupText");
  StartCoroutine("hidePickupText");
}

// 4
IEnumerator hidePickupText() {
  yield return new WaitForSeconds(4);
  pickupText.GetComponent<Text>().enabled = false;
}

// 5
public void ShowNewWaveText() {
  StartCoroutine("hideNewWaveText");
  newWaveText.GetComponent<Text>().enabled = true;
}

// 6
IEnumerator hideNewWaveText() {
  yield return new WaitForSeconds(4);
  newWaveText.GetComponent<Text>().enabled = false;
}
```

Taking each commented section in turn:

1. Show the wave clear bonus text by setting its `enabled` state to `true`, then immediately call a coroutine that will hide the text again. You do it this way because you can use the coroutine to pause itself before it actually hides the text.
2. Wait for 4 seconds before setting the `enabled` state to `false` — therefore hiding the text.
3. Enable and set the text for the pickup alert and restart the `hidePickup()` coroutine. This lets the player to pick up two or more pickups in quick succession, without the second pickup's text label displaying before the first pickup's text times out.
4. Wait for 4 seconds before removing the pickup text.
5. Show the new wave text.
6. Wait for 4 seconds before removing the new wave text.

Now that the **GameUI** script has the code necessary to update changes to the UI, you need to add in the code that calls this at the appropriate time.

Save your work and open the **Player** script. In `TakeDamage()`, under the line `armor = effectiveArmor / 2;` add the following:

```
gameUI.SetArmorText(armor);
```

Under the line `armor = 0`, add the line:

```
gameUI.SetArmorText(armor);
```

These lines will update the armor as it takes damage.

Replace the line `Debug.Log("Health is " + health);` with:

```
gameUI.SetHealthText(health);
```

This will update the health as it changes in the UI.

In `pickupHealth()` add the following just above the closing brace:

```
gameUI.SetPickUpText("Health picked up + 50 Health");
gameUI.SetHealthText(health);
```

This shows the pickup text alert and updates the health UI.

In `pickupArmor()`, add the following just above the closing brace:

```
gameUI.SetPickUpText("Armor picked up + 15 armor");
gameUI.SetArmorText(armor);
```

This shows the pickup text alert and updates the armor UI.

In `pickupAssaultRifleAmmo()` add the following just before the closing brace:

```
gameUI.SetPickUpText("Assault rifle ammo picked up + 50 ammo");
if (gunEquipper.GetActiveWeapon().tag == Constants.AssaultRifle) {
  gameUI.SetAmmoText(ammo.GetAmmo(Constants.AssaultRifle));
}
```

First this alerts the player about the ammo pickup in the UI. Then the code checks to see if the active gun matches the assault rifle before setting the ammo count. If you didn't do this check, then every time you picked up ammo for a different gun it would change the ammo displayed to the wrong type.

Now add the code for the other ammo types.

Before the closing brace of `pickupPistolAmmo()` add:

```
gameUI.SetPickUpText("Pistol ammo picked up + 20 ammo");
if (gunEquipper.GetActiveWeapon().tag == Constants.Pistol) {
  gameUI.SetAmmoText(ammo.GetAmmo(Constants.Pistol));
}
```

Before the closing brace of `pickupShotgunAmmo()` add:

```
gameUI.SetPickUpText("Shotgun ammo picked up + 10 ammo");
if (gunEquipper.GetActiveWeapon().tag == Constants.Shotgun) {
  gameUI.SetAmmoText(ammo.GetAmmo(Constants.Shotgun));
}
```

Save your work. Open the **Ammo** script in the code editor and in `ConsumeAmmo()`, add the following before the closing brace:

```
gameUI.SetAmmoText(tagToAmmo[tag]);
```

This updates the UI each times the user fires his weapon.

Open the **GunEquipper** script in the code editor. Add the following below the other variable declarations:

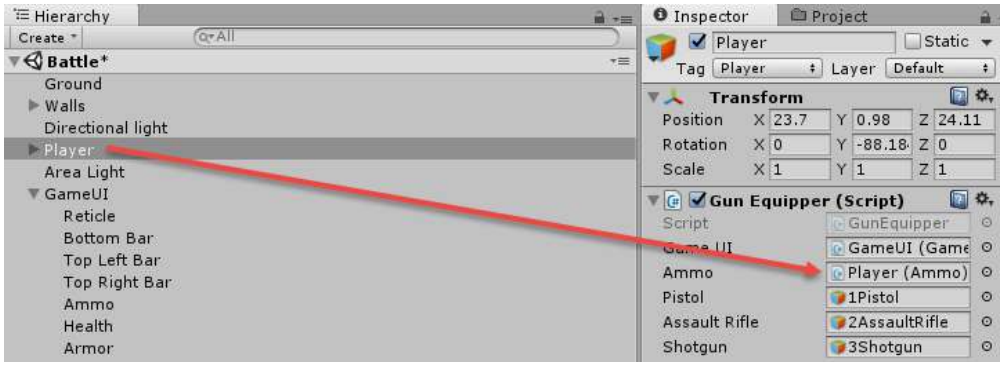
```
[SerializeField]
Ammo ammo;
```

Now to use this variable. Add the following just before the closing brace in `loadWeapon();`:

```
gameUI.SetAmmoText(ammo.GetAmmo(activeGun.tag));
```

This will update the ammunition count when the player switches guns. However, `ammo` hasn't been assigned yet in the editor. Save your work and switch back to Unity.

Select the **Player** GameObject and drag it on top of the **Ammo** field on the **GunEquipper** Component:



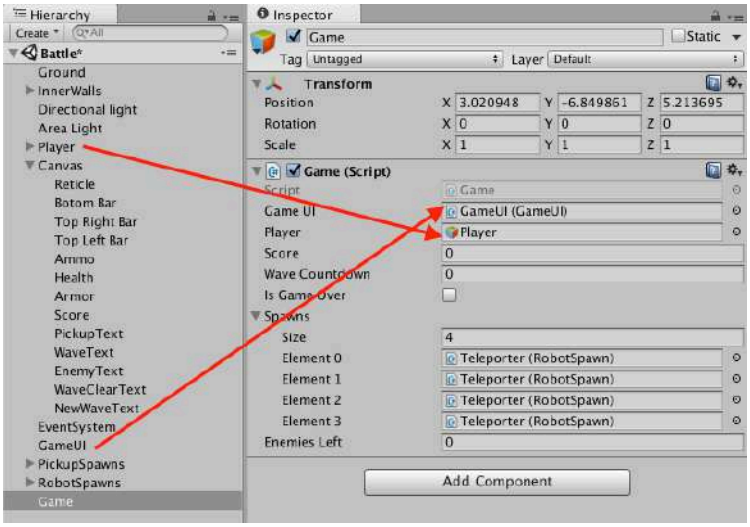
Open the **Game** script in the code editor and add the following variables just after the opening brace of the class:

```
public GameUI gameUI;
public GameObject player;
public int score;
public int waveCountdown;
public bool isGameOver;
```

These are just some references and counter values you'll be using soon. One variable of note is `isGameOver` which tracks whether the game has ended. You'll be coding the Game Over screen a little later, but it's helpful to add this flag now.

Before editing the script further, save your work, go back to Unity, and click the **Game** GameObject in the inspector.

Drag the **GameUI** GameObject over the **Game UI** field and drag the **Player** GameObject over the **Player** field.



Open the **Game** script in the code editor. Add this import at the top under `using UnityEngine;`:

```
using System.Collections;
```

This import will let you use coroutines. Add this method below the closing brace of `SpawnRobots()`:

```
private IEnumerator updateWaveTimer() {
  while (!isGameOver) {
    yield return new WaitForSeconds(1f);
    waveCountdown--;
    gameUI.SetWaveText(waveCountdown);

    // Spawn next wave and restart count down
    if (waveCountdown == 0) {
      SpawnRobots();
      waveCountdown = 30;
      gameUI.ShowNewWaveText();
    }
  }
}
```

You first check to see if the `isGameOver` flag is set. If it isn't, the script will pause for 1 second before decrementing `waveCountdown` and updating the UI. If the countdown hits 0, you call the spawn robots method, reset the countdown then show the message telling the player that a new wave of robots is on its way!

Now add the following immediately below the method you just added:

```
public static void RemoveEnemy() {
  singleton.enemiesLeft--;
  singleton.gameUI.SetEnemyText(singleton.enemiesLeft);

  // Give player bonus for clearing the wave before timer is done
```

```
    if (singleton.enemiesLeft == 0) {
      singleton.score += 50;
      singleton.gameUI.ShowWaveClearBonus();
    }
  }
```

This will be called when robots are killed. It removes 1 from the enemy count and updates the UI. If there are no enemies left, it alerts the player about the bonus score they earned and adds 50 to their score.

The player should also get points every time they kill a robot. Add the following method just below the last one:

```
  public void AddRobotKillToScore() {
    score += 10;
    gameUI.SetScoreText(score);
  }
```

This will give the player points when they kill a robot.

The player should also get 1 point for every second they survive. Add the following method below the last one:

```
  IEnumerator increaseScoreEachSecond() {
    while (!isGameOver) {
      yield return new WaitForSeconds(1);
      score += 1;
      gameUI.SetScoreText(score);
    }
  }
```

This is just a coroutine that updates the score every second.

`Start()` now needs to be updated to set your new variables up. Add the following after the line containing `singleton = this;`:

```
  StartCoroutine("increaseScoreEachSecond");
  isGameOver = false;
  Time.timeScale = 1;
  waveCountdown = 30;
  enemiesLeft = 0;
  StartCoroutine("updateWaveTimer");
```

First you start `increaseScoreEachSecond`, then initialize all the variables to their starting values, before starting `updateWaveTimer`.

In `SpawnRobots()` add the following before the closing brace.

```
  gameUI.SetEnemyText(enemiesLeft);
```

This sets the enemy count to the latest value after spawning new robots. Save your work and open **Robot.cs** in your code editor:

Here, you need to call the `RemoveEnemy()` method you just created. Add the following immediately after the call to `StartCoroutine("DestroyRobot")`:

```
Game.RemoveEnemy();
```

This will reduce the enemy count by 1 and update the UI. Save your change and switch back to Unity.

Great job getting here! Now you can reap the rewards of your hard labor and Hit Play.

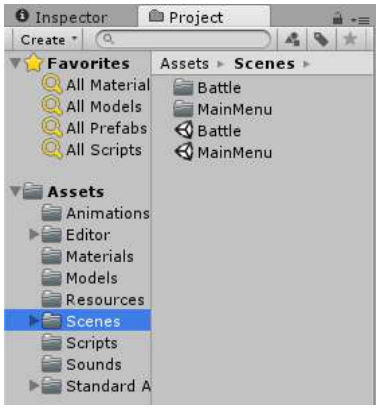
Run around and kill robots and watch the enemies decrease. Notice your ammunition decreases when you fire and updates when you change guns or get a pickup. Also check out the text that notifies the player of the pickup they grabbed. Your health and armor also goes down when you get hit by robots.

You'll also see the new wave text alert, and the bonus you get from killing all the enemies before the wave timer ends. The wave timer will click down every second and spawn new enemies (and reset) when it hits 0.

Adding a Main Menu

You've almost finished the game; this next layer of polish will really make it shine.

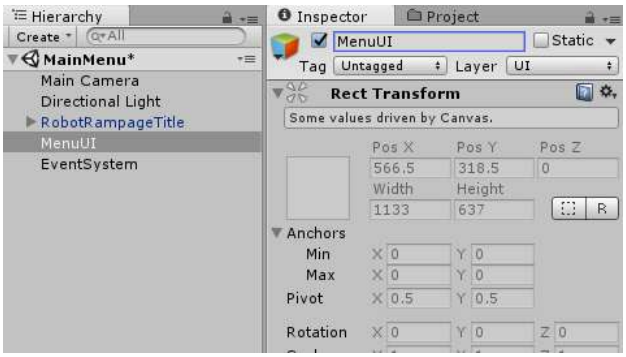
First, you are going to add a main menu. In the **Scenes** folder, right-click, select **Create\Scene** and call it **MainMenu**.



Double-click the scene you just created to open it. Drag over the **RobotRampageTitle** Prefab from the **Resources** folder onto the **Hierarchy**.

Set its **Position** to (**15**, **-61**, **538**), its **Rotation** to (**18.07**, **180**, **0**). This is just a good starting point that was picked when the game was being developed.

Create a **Canvas** like you've done before and name it **MenuUI** — remember how you handled **UI\Canvas** in the Hierarchy.



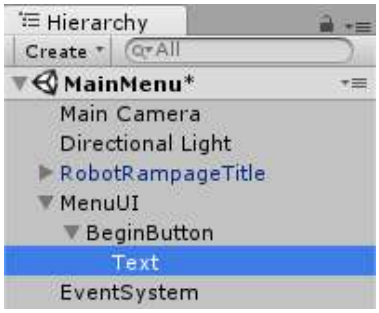
First, create you need to create a button. Right-click in the Hierarchy, select **Create\UI\Button** and rename it to **BeginButton**. Set the **anchor preset** to **bottom center**, **pivot** to (**0.5**, **0.5**), **position** to (**0**, **165.8**, **0**), **width** to **260**, and **height** to **75**.

In the **Image** component, set the **Source Image** to **Pink_Grey_Texture**. In the **Button** component, set the **Transition** to **Sprite Swap**, set the **Highlighted Sprite** to **Pink_Black_Texture** and the **Pressed Sprite** to **Pink_Texture**.

The **Sprite Swap** transition means that different sprite images are used for the button, depending on its state.

The button should look like so:

In the Hierarchy, expand the arrow beside the **BeginButton** to reveal its child **Text** UIGameObject and select it.



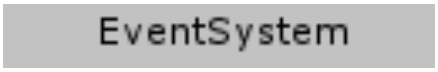
Change the **Text** (in the **Text** Component) to **Begin**, set the color to **White**, the **Font Size** to **46**, and the **Font** to **heavy_data**.

Duplicate the previous button by pressing Control-D (or Command-D on the Mac). Call it **ExitButton** and set the **position** to (**0**, **82**, **0**).

Expand the arrow beside the **ExitButton** to reveal its child **Text**. Change the **Text** to **Exit**. Your UI should like so:

This is a great looking scene, but it doesn't do anything yet! You will be bringing it to life by adding some callbacks to the buttons so that when they are clicked it either begins or exits the game.

Did you notice that a GameObject was created called **EventSystem** in the Hierarchy after you created the first **Button**?



This is how your UI buttons can be clicked to trigger events. The EventSystem GameObject listens for these click (or hover) events and dispatches them to something that needs to know that they've happened. You don't need to worry about how it dispatches its events, all that matters is how you intercept them.

Create a C# script, name it **Menu** and open it in the code editor. First, add this import line at the top of the script:

```
using UnityEngine.SceneManagement;
```

This allows you to use the **SceneManager** type. This is a class that allows you to manage your scenes such as loading, unloading, or even searching for a scene.

Now add the following methods to the body of the class:

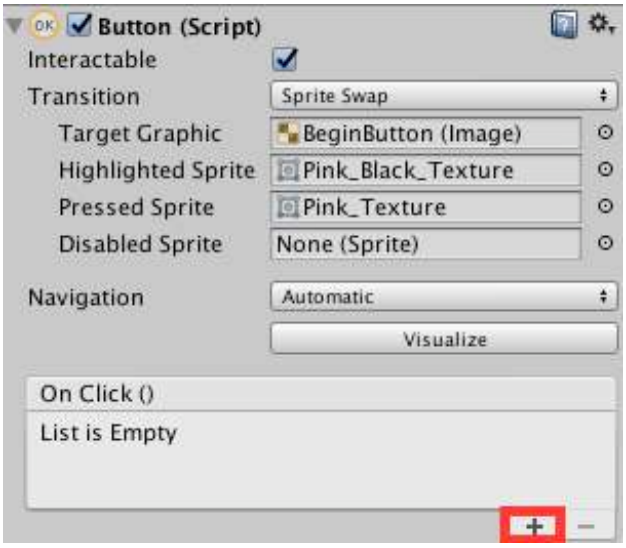
```
// 1
public void StartGame() {
  SceneManager.LoadScene("Battle");
}

// 2
public void Quit() {
  Application.Quit();
}
```

1. This method will be used when the user clicks the Begin button; it simply loads the **Battle** scene.
2. This method will be used to exit the application. However, note that if you are running the game in Unity (instead of within a standalone build) this won't do anything.

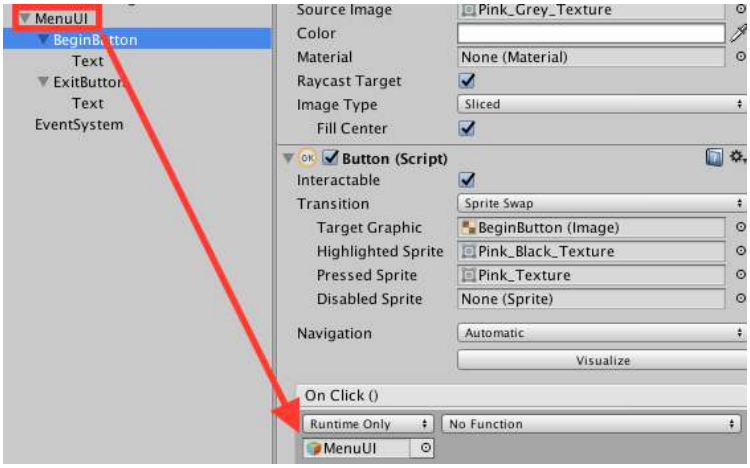
With these methods set up, you can now configure the buttons to actually call them. Save your work and switch back to Unity.

Select the **MenuUI** GameObject and drag over the **Menu** script to it. Now, you must hook up the methods to the buttons.

Select the **BeginButton** GameObject and click the + in the **Button** Component.

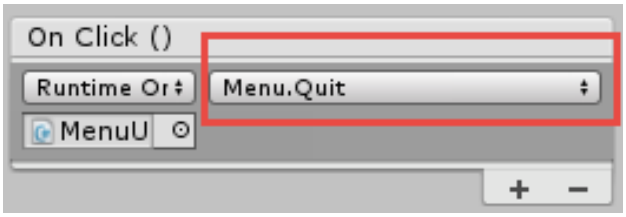


This creates a UnityEvent which lets you call another function when this button is clicked. Drag the **MenuUI** UIGameObject over to the field just below the dropdown menu.



Now the right side dropdown menu will say **No Function**. Click this dropdown and select **Menu\StartGame()**. Now when this button is clicked it will call `StartGame()` in the **Menu** script which will load the battle scene.

Repeat this process for the **ExitButton** except set its callback function to the `Quit()` method.



Hit Play and hit the **Begin** button to load the game. However you might notice the game looks a lot darker. If the lighting looks fine, skip ahead to the Music section.

Music

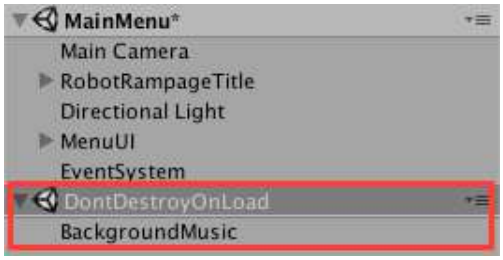
It would be awesome if this game had a soundtrack — that's what you'll do next.

In the **MainMenu** scene, create an empty GameObject and name it **BackgroundMusic**. Create a C# script, name it **BackgroundMusic** and open it in the code editor. Delete the boilerplate methods and replace them with the following method:

```
void Start() {
  DontDestroyOnLoad(gameObject);
}
```

This simply ensures the GameObject that the script is attached to isn't deleted when changing scenes so the music will keep playing when you switch to the Battle! You'll see this reflected in the Hierarchy when you play your game:

Save your changes. Back in Unity, drag this script over the **BackgroundMusic** GameObject.



Select the **BackgroundMusic** GameObject and add an **Audio Source** component. Set the **Loop** checkbox to **Selected** and set the **AudioClip** to **music**. Now hit Play and hit the Begin button and start killing robots with some epic tunes.

Game over

This game is almost done but it needs a game over screen. Open up the **Battle** scene (by double clicking it in the **Scenes** folder).

First, create a Panel by right-clicking in the **Hierarchy** and selecting **UI\Panel**. Rename it to **GameOverScreen**. Set its **Anchor Preset** to **middle center**, **Pos Y** to **0**, its **Width** to **400**, and its **Height** to **300**. Set the **SourceImage** in the **Image** Component to **Box** and set its **Alpha** to **1** (just click the **Color** box and slide the **A** over to the right.

A panel is just another UI controls which you can use to organize your controls. All subsequent UI controls will be **children** of the **GameOverScreen**.

Create a **Text** UIGameObject, rename it to **GameOverText** and put it under the **GameOverScreen** UIGameObject (by dragging to under it or by having it selected when you create the text). Set the **position** to (**0**, **0**, **0**), the **Width** to **400**, the **Height** to **290**.

In the **Text** Component, set the **Text** to **Game Over**, the **Alignment** to **Center**, the **Font** to **heavy_data**, the **Font Size** to **70** and the **Color** to **White**.

The UI should look as follows:

Now create the following buttons.

> **Note**: Once you create your first button, configure the text as explained underneath the table. Then duplicate the button to save yourself some time.

Type	Name	Position	Width	Height	Source Image	Transition	Highlighted Sprite	Pressed Sprite
Button	Restart	(0, 12, 0)	260	60	Pink_Grey_Texture	Sprite swap	Pink_Black_Texture	Pink_Texture
Button	Exit	(0, -34.5, 0)	260	60	Pink_Grey_Texture	Sprite swap	Pink_Black_Texture	Pink_Texture
Button	Menu	(0, -67, 0)	260	60	Pink_Grey_Texture	Sprite swap	Pink_Black_Texture	Pink_Texture

Next, you need to configure the text on all the buttons.

Expand the **Restart** GameObject and select its **Text** child. Change the **Text** to **Restart**, the **Font Size** to **22**, the **Font** to **heavy_data** and the **Color** to **White**.

Do this for both the **Exit** and the **Menu**, changing the actual text to **Exit** and **Menu** respectively.

Your UI will look as follows:

With the layout complete, you just need to hook it up in code.

Open the **Game** script, and before the class declaration add the following:

```
using UnityStandardAssets.Characters.FirstPerson;
using UnityEngine.SceneManagement;
```

You'll need these resources in just a moment. Add the following variable at the top of the script after the opening brace of the class;

```
public GameObject gameOverPanel;
```

Now add these methods to the body of the class:

```
// 1
public void OnGUI() {
  if (isGameOver && Cursor.visible == false) {
    Cursor.visible = true;
```

```
      Cursor.lockState = CursorLockMode.None;
    }
  }

  // 2
  public void GameOver() {
    isGameOver = true;
    Time.timeScale = 0;
    player.GetComponent<FirstPersonController>().enabled = false;
    player.GetComponent<CharacterController>().enabled = false;
    gameOverPanel.SetActive(true);
  }

  // 3
  public void RestartGame() {
    SceneManager.LoadScene(Constants.SceneBattle);
    gameOverPanel.SetActive(true);
  }

  // 4
  public void Exit() {
    Application.Quit();
  }

  // 5
  public void MainMenu() {
    SceneManager.LoadScene(Constants.SceneMenu);
  }
```

Here's the play-by-play:

1. This method frees the mouse cursor when the game is over so the user can select something from the menu.

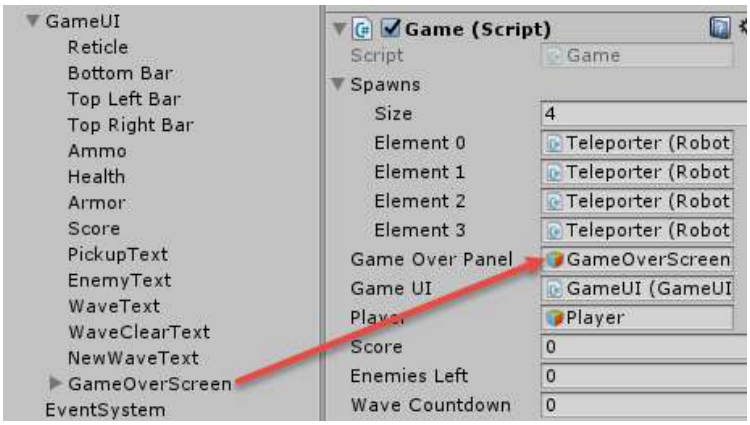
2. This will be called when the game is over. It sets the `timeScale` to 0 so the robots stop moving. It also disables the controls and displays the Game Over panel you just created.

3. When the user wants to restart the game, this method will be used to reload the scene to start the game again.

4. This will be called when the user selects the Exit button. It quits the application, but only if its being run from a build.

5. You also need a method to go back to the main menu when the user selects the Menu button. This just loads the **Menu** scene.

Save your changes and switch back to Unity. Click the **Game** GameObject and drag over the **GameOverScreen** GameObject to the **Game Over Panel**.



Now deactivate the **GameOverScreen** GameObject. This way, it only appears when the game is over.

Open the **Player** script, and add these variables after the opening brace of the class:

```
public Game game;
public AudioClip playerDead;
```

These are references to the **Game** script, and an **AudioClip** that you'll play when the player dies.

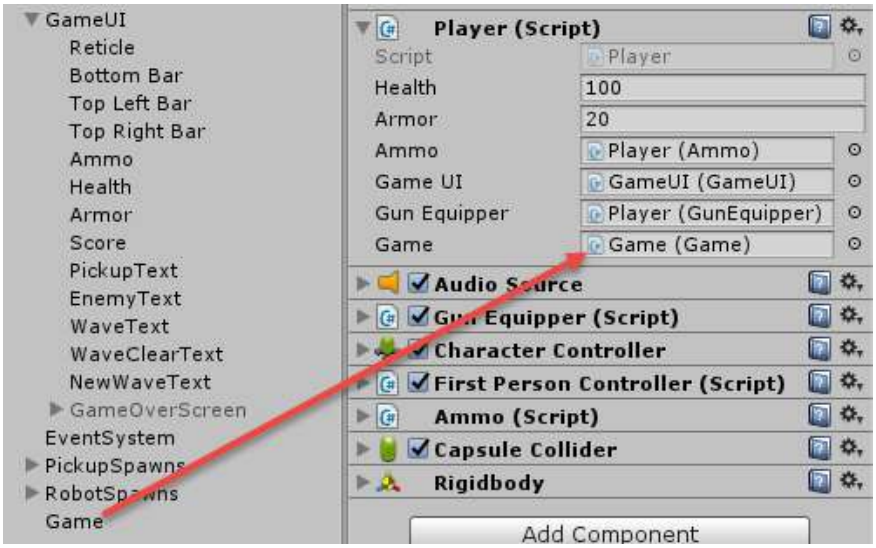
Replace `Debug.Log("GameOver")` with:

```
GetComponent<AudioSource>().PlayOneShot(playerDead);
game.GameOver();
```

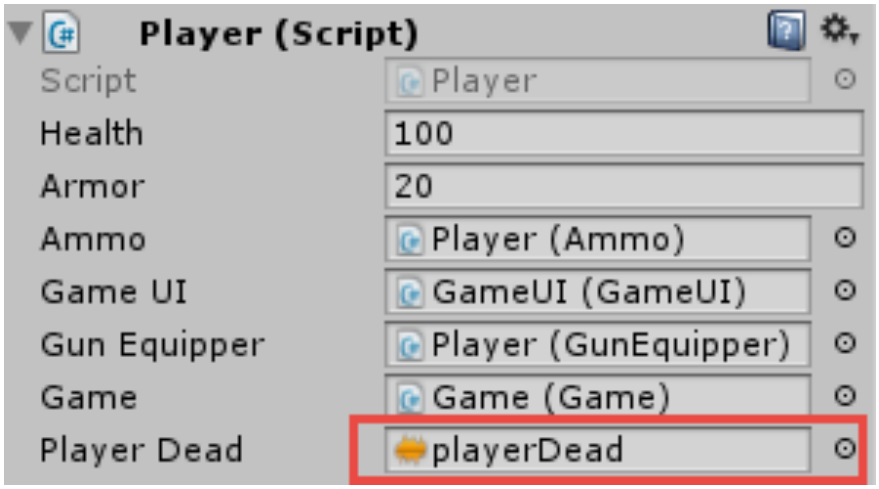
This will play the **playerDead** audio and then call **GameOver** on the **Game** script.

Save your work and switch back to Unity.

Click the **Player** GameObject and drag over the **Game** to the **Game** field.



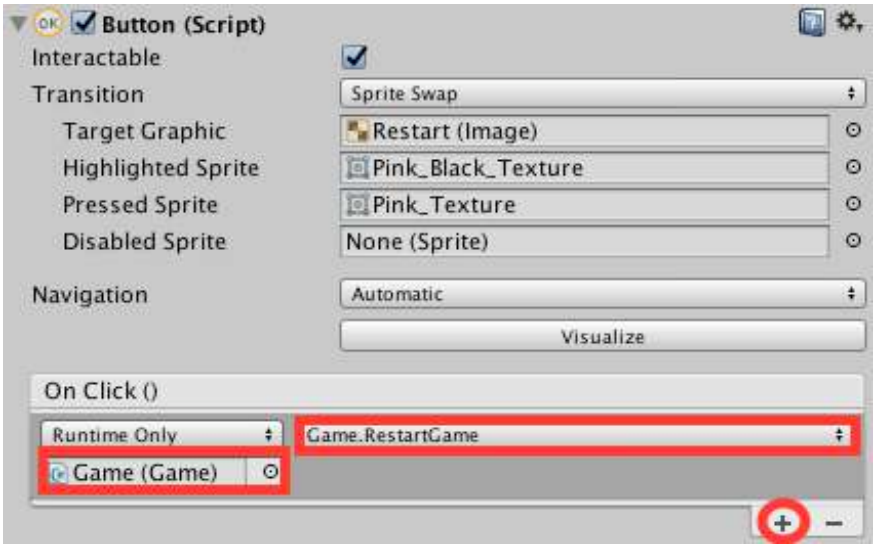
Next set **Player Dead** to the **playerDead** audio file.



Now when the player's health reaches 0 it calls the game over screen and it plays a death sound.

The last thing to do is configure the button callbacks.

Click the **Restart** GameObject. Add an event by clicking the +. Next, drag the **Game** GameObject onto the field, and select **Game\RestartGame()** in the dropdown menu.

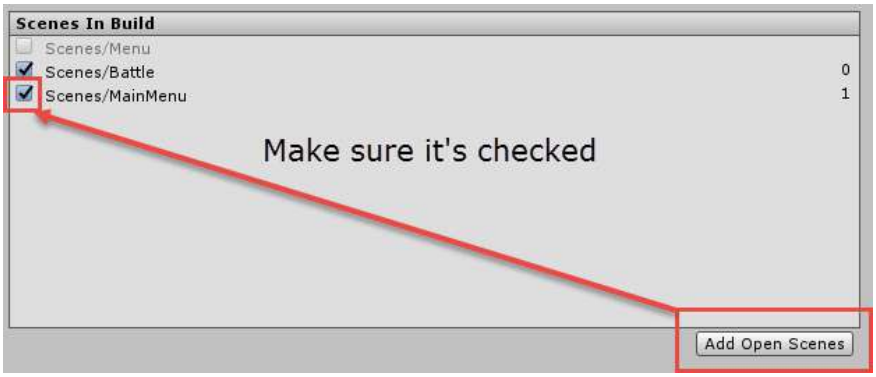


Do this two more times for both the **Exit** and **Menu** GameObjects. When selecting the methods, select **Game\Exit()** for the Exit GameObject, and **Game\MainMenu()** for the Menu GameObject.

Click Play, and blast away until your health reaches 0 to see the game over screen. You should hear the player scream as she dies.

Hit the **Menu** button while the game over screen is up — it's not working! Don't worry, it's simply a matter of adding the MainMenu scene to the build.

Open the **MainMenu** scene, select **File/Build Settings** and click **Add Open Scenes**.



Now go back to the **Battle** scene by double clicking it in the **Scenes** folder, hit **Play**, and die again. With the game over screen open, click **Menu** and you should be taken to the **MainMenu** scene. Clicking **Restart** also should work just fine and reload the battle.

Where to go from here?

Congratulations! You've learned a lot in these last few chapters. From adding weapons and firing logic; to waves of enemies, spawning powerups and flying missiles; to scorekeeping, game over conditions and a full UI, you've created a fantastic and challenging FPS.

You've undoubtedly seen some adjustments you can make to the game. Play around with the map layout; add or change spawn positions, or even give powerups a limited lifespan on the map — you snooze, you lose!

There's a real art to creating FPS games that are intuitive enough to pick up easily, but take some time to master; games that are easy enough in the first few levels to hook players, but take repeated plays to learn the subtle strategy to conquer the game. The most important part of any game is to test-play it with a range of players to find the sweet spot that creates an addictive experience.

Happy blasting!