Lesson Six

Who's the Boss?

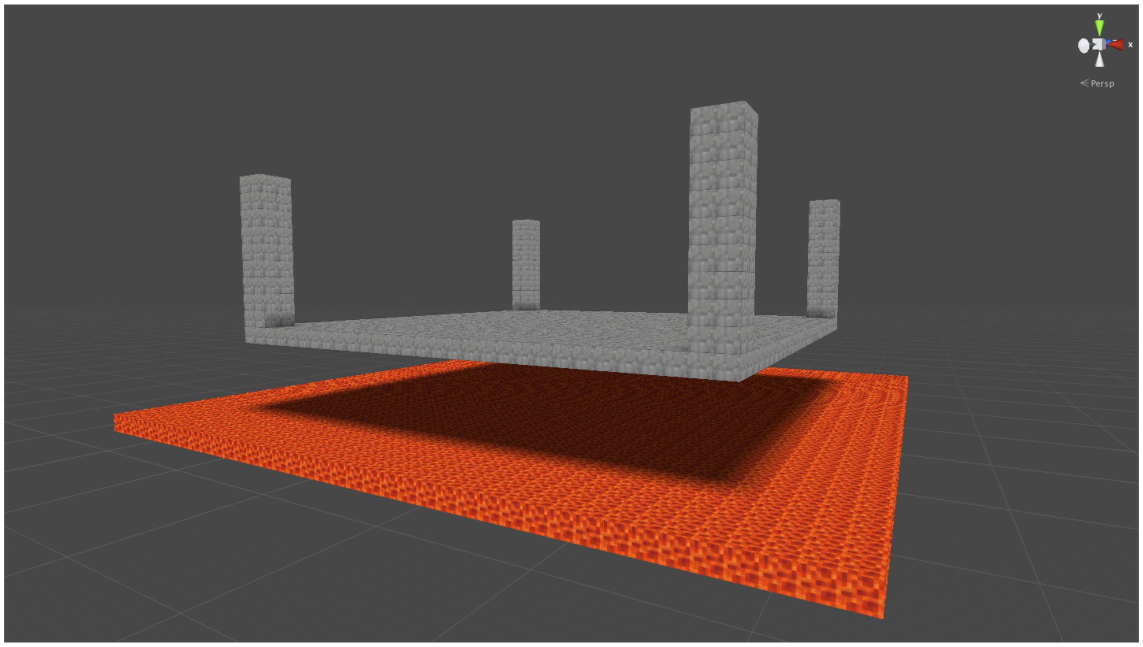


Figure 1: Our boss area as an example

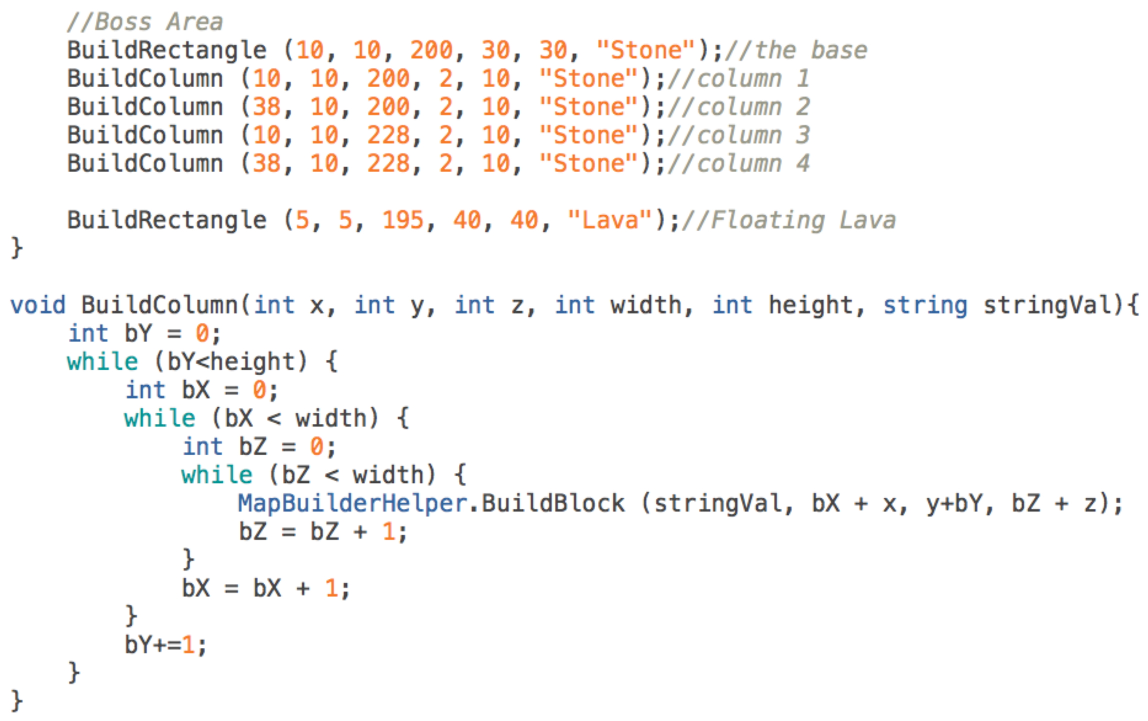
1 Introduction to Lesson Six

In the past two lessons you've built up a tiny adventure game complete with enemies, items, and quests. But we're missing one core element of the adventure game to tie them all together: a boss.

"Bosses" tend to appear at the end of quests, with a big "final boss" showing up at the very end of the game. They are typically similar to the enemies fought throughout the game, but bigger and with different ways to attack the player. To finish out this set of lessons we'll go through the process of creating a boss. This is the final "full" lesson, as the remaining lesson only gives some suggestions for potential jump-off points for your creating your own game. As such, we recommend having completed all of lesson four and five before continuing.

2 Creating a Boss Area

Most video games have special lairs where bosses can be found, whether these are caves, castles, or dungeons. Before creating our boss we'll first need to create a special are like one of these. To do that, go ahead and open "AdventureGameGenerator.cs". We'll want to create a brand new region of the game world via adding calls to BuildRectangle in GenerateMap. We'd recommend making this new location visible to the player, but unaccessible. That way they can play through the whole game seeing the boss area before getting to it at the end. Check out our example in Figure 1 for some inspiration, and the code that generated the area is Figure 2 (including a new function we added to help!). If you have any trouble with this, please check back to lesson four for a step-by-step walkthrough of the process.

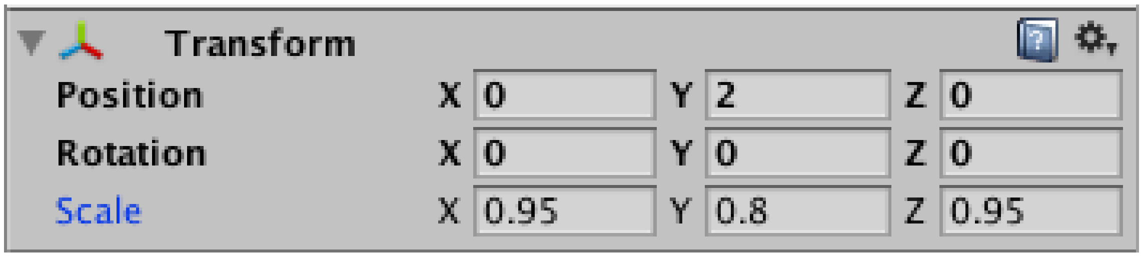


```
    //Boss Area
    BuildRectangle (10, 10, 200, 30, 30, "Stone");//the base
    BuildColumn (10, 10, 200, 2, 10, "Stone");//column 1
    BuildColumn (38, 10, 200, 2, 10, "Stone");//column 2
    BuildColumn (10, 10, 228, 2, 10, "Stone");//column 3
    BuildColumn (38, 10, 228, 2, 10, "Stone");//column 4

    BuildRectangle (5, 5, 195, 40, 40, "Lava");//Floating Lava
}

void BuildColumn(int x, int y, int z, int width, int height, string stringVal){
    int bY = 0;
    while (bY<height) {
        int bX = 0;
        while (bX < width) {
            int bZ = 0;
            while (bZ < width) {
                MapBuilderHelper.BuildBlock (stringVal, bX + x, y+bY, bZ + z);
                bZ = bZ + 1;
            }
            bX = bX + 1;
        }
        bY+=1;
    }
}
```

Figure 2: The code to generate 1



Transform						
Position	X	0	Y	2	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	0.95	Y	0.8	Z	0.95

Figure 3: The player's initial transform in Unity's window.

Play the game as needed to test how your boss area is looking as you work. Don't worry about getting it perfect now though, just get down something the player can stand on.

Of course at this point there's no way to get to this new boss area, floating lava or not. To do that we'll want to "teleport" the player once they have finished all of the other quests, but to do *that* we'll first need to talk about "transform" and "Vector3".

2.1 Transforms and Vectors

At some point while creating your own boss area and typing out values for x, y, and z positions you might have thought to yourself: "Wow, these three variables seem to go together a lot, if one there was some way to store them all together!" Well you're not alone!

Definition!

VECTOR3:
A class that only exists to hold three values, which can then be used to define a position.

Unity, the game engine these lessons are written in, has a special class to hold these three values called a **Vector3**. Vector3 class variables have many uses, but the primary one is to hold position information of objects. So instead of having to create individual x, y, and z variables, all three values to define a position can be stored in a Vector3.

If you click back to Unity and then click on the "Player" object in the hierarchy you should see something that looks like Figure 3. The "Transform" class holds an object's position, rotation, and scale in Unity. That means that the player's initial position is (0,2,0).

This may be difficult to understand so let's try an experiment. Change the Y-value of the Position variable of Transform from 2 to 100.

Play the game! It should be immediately obvious what's different. If it isn't, try looking down.

The player started way up in the air! That's because you changed the player's starting position from (0, 2, 0) to (0, 100, 0). You could use this to start the player in the boss area, by setting the player's starting position to (15, 12, 210) for example for our boss area. But that's not what we wanted! We want the player to be transported to the Boss Area after completing the initial quests. Thankfully there's a way around this, as we can also access the player's Transform and it's Position variable from code. The Position variable is actually a Vector3, so we'll need a new Vector3 variable to change it. But first we'll need a place in the code to access the player's Transform. To do that we'll need a reference to the player's object, which each of the quests have.

2.2 Teleporting the Player

While eventually we'll want to teleport the player only after they pick up the final Magic Sword, just for testing purposes, let's go ahead and add the teleportation to FirstQuest. Go ahead and open it up. Then to QuestStart add the following line:

```
player.transform.position = new Vector3 (0, 100, 0);
```

Save your changes and play the game! Go up and talk to the first NPC. You might want to change the player's initial position back to (0,2,0) first.

Speaking to the NPC should now send send the player flying into the air. If this didn't work, make sure that your "CanStart" function returns true. But how exactly does this happen? Well you might recall from Lesson Five that the "Quest" class has a protected "player" variable that "pointed" to the player object. So that explains that portion.

The variable "player" contains within it a variable "transform" that is lower-cased as it is a class variable of the class Transform. This allowed us to then access the player's transform's position variable.

Definition!

NEW:
The new keyword creates a class variable from a class in a process called "instantiation".

On the other side of the equals sign there's a new keyword: **new**. The "new" keyword tells Unity to create a new class variable from a class. So in this case, that would be a class variable for the Vector3 class. Recall that previously you had class variables already created for you, with the "new" keyword you can make new class variables all on your own.

The values in the parentheses after the "Vector3" class name are the default starting values for the class variable that's created. So in this case that gives this Vector3 class variable starting values of (0, 100, 0). In other words it's x-value is 0, it's y-value is 100, and it's z-value is 0. We can use this fact to teleport the player over to the boss area, instead of into the air.

Our boss area's center is located at around (25, 10, 215), so for us the command to send the player to the boss area looks like:

```
player.transform.position = new Vector3 (25, 12, 215);
```

Note that we place the player slightly above the center of the boss area, as we don't want the player falling through the boss area. To determine the values you should use find the the function call for the base of the boss area, for us that line looks like:

```
BuildRectangle(10, 10, 200, 30, 30, "Stone");
```

You can then add half of the x size and z size values to the starting point of the rectangle (e.g. 30/2 = 15; 10+15 = 25; 200+15 = 215).

Once you've changed the teleport position for the player, save and play the game. Go up to the NPC and you should see the player instantly transported to the boss area.

If your player doesn't end up in the center of your boss area, try the beginning of your boss area ((10, 10, 200) in our case). Once your player is transporting there, we'll need to next deal with the lack of boss in the boss area.

2.3 Creating the Boss: Material Customization

The boss area presently lacks any sort of boss. We could put a large number of enemies there, but that would be be fairly boring given that the player has already seen the normal "ghost" enemies several times. It would like be more interesting for players if we created a brand new enemy type.

To do that, we'll take much the same approach as we did in creating the magic sword, taking an already existing game object and modifying it. Except that this time we'll be going further with the modifications.

To start we'll need to duplicate one of the default enemies so that we can modify it. In the project view find the "Ghost" object under GameAssets and then under NPC Assets. Duplicate

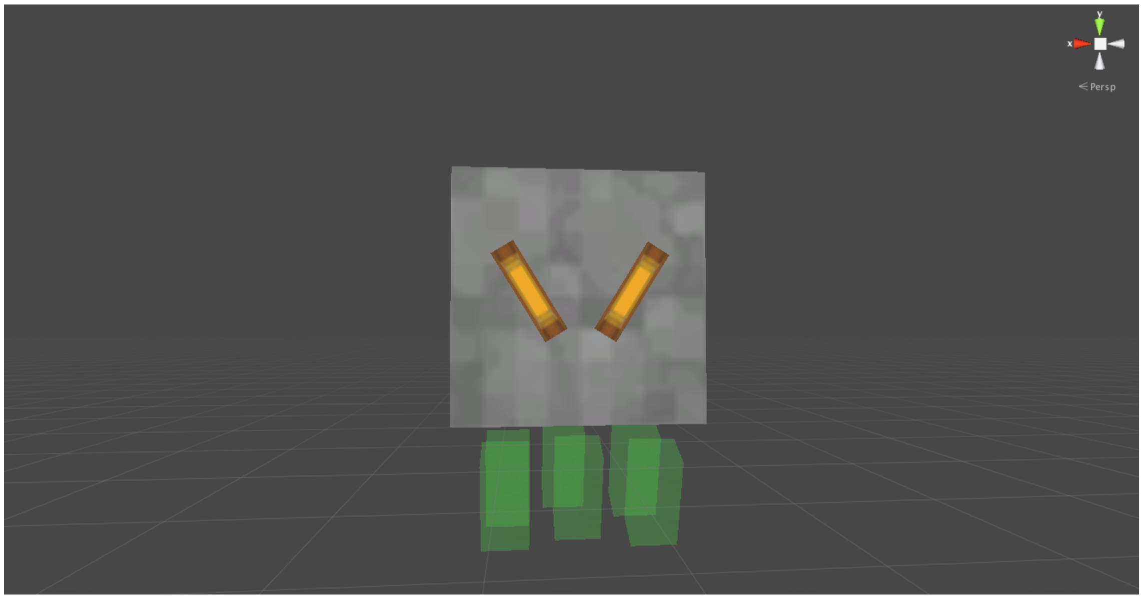


Figure 4: An example of our own version of the boss ghost.

this object using either Edit>Duplicate or control/command D. Then rename this new object from Ghost 1 to “Boss”. Drag this boss onto the scene view. You should see a green ghost in your scene! To get a closer view select the “Boss” object in the Hierarchy and click the “F” button on your keyboard. This tells Unity to focus the Scene view onto a particular object.

Recall from the previous lesson that we altered the sword into a magic sword visually with the use of materials. We’ll start with the same here. You can find the materials in the GameAssets folder under Item then Materials. As a reminder, click and drag your material onto the part of the boss you want to have that material in the Scene view. Alternatively click the arrow next to the “Boss” line in the Hierarchy view to see each of the boss’ different parts. If you’re having a hard time seeing your ghost in the Scene, you can hold “alt” or “option” and the left mouse button to rotate the scene view. Feel free to play around with the materials until you find something you like!

For our boss we made use of the “Stone” material for the primary “Boss” body, and the “Gold” material for the eyes (EyeLeft and EyeRight in the hierarchy view). You may notice that we didn’t change the “GhostBit” sections at all. We’ll get to why in the next section.

2.4 Creating the Boss: New Visualization Customization

In many games it’s customary for bosses to be a good deal larger than normal enemies. As you might recall earlier we looked at the “transform” component on the player. While you altered the “position” variable in order to change where the player was located, you can also use the transform variable to affect a game object’s size via the “Scale” variable.

Go ahead and click on the “Boss” object in the Hierarchy view. in the Inspector view you should now see the boss’ different components, including it’s “transform”. Find the “Scale” variable, which

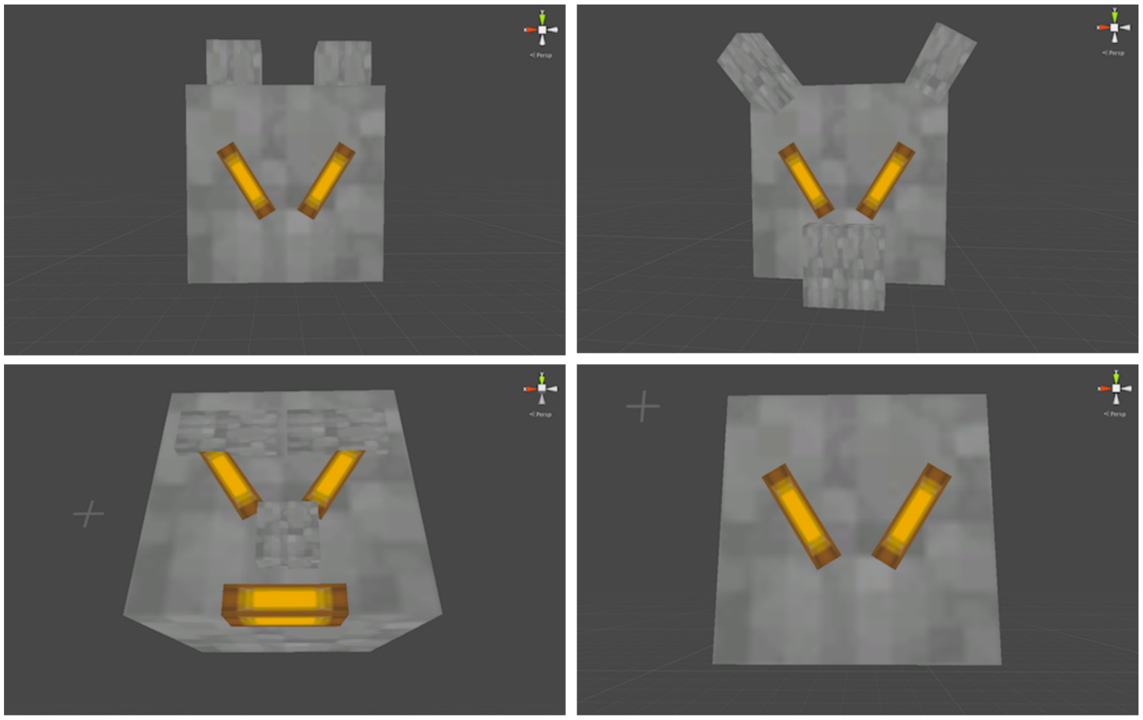


Figure 5: Four possible boss ghosts.

should currently be set to (1, 1, 1). Let's start by doubling that in size, so change all the values to (2, 2, 2). With that you should see the entire boss grow to be two times larger!

But hold on, that doesn't quite make sense does it? You only changed the "Boss" part, which is just the main cube of the boss' body. Why would the entire boss get larger, from the eyes to the ghost "bits" at the bottom?

See how in the hierarchy view the eyes and "bits" are all "within" the main boss body? This means that any changes to the Transform of the top object will impact all the objects within it. Let's confirm this. If we move the top boss body by changing the position value of it's transform, all the other bits within it should move as well. Change your boss' position values to the same center values that worked for your player.

The boss will disappear from your Scene view. To refocus the scene view on the boss click "f" again while selecting the boss in the Hierarchy view. You can see that the entire boss moved as one, including all the things "inside" the boss. This may not seem particularly useful, but it's what allows us to have objects made of multiple parts in Unity stick together. We can change the Transform values of objects "inside" the main object and they'll still stick to the main object!

That might seem a bit confusing, so let's test this out. Try changing the scale, position, or rotation values of the "bits" inside the main "Boss" body. You can either change these values individually or use the arrows that pop up in the scene view to do it. Also feel free to use different materials on them. See what you can make! You can see examples of things we came up with in Figure 5.

You might notice in moving around the bits that the position values do not seem quite right. For example, one would think the bits should be near the point that the body is at, but the values are much smaller. This is because when an object is inside another it's Transform's values are relative. That means (0, 0, 0) values for a bit or an eye will put it right at the center of the Boss object. That's how we were able to hide some of the bits in Figure 5

When you're happy with how your boss looks, try to play the game! As the player, warp over to the boss area and check it out.

You might notice that the boss acts exactly like a regular enemy. This should make sense given that we haven't changed any of the scripts attached to the Boss, only the visual appearance of the Boss. In the next section we'll cover giving the boss more interesting behaviors.

2.5 Creating the Boss: Behavior Script

Recall that to create the magic sword behavior we made a new script that *inherited* from Sword.cs. It makes sense to do something similar here. The primary script you've interacted with for the enemy is Enemy.cs. We could consider creating a new Boss.cs that inherited from this, but there's a problem. Recall that with MagicSword we overrode the function HitEnemy because it was defined as "virtual". Take a look at Enemy.cs (located in the folder NPC inside the Codebase folder), do you see any virtual functions?

Unfortunately there aren't any virtual functions in Enemy.cs that could be overridden. We could still override from Enemy.cs, but it would mean having to rewrite all of the behavior in Enemy.cs. As you might recall there's quite a bit of code you wrote in Enemy.cs to control movement and seeing the player and it doesn't really make sense to duplicate that. What other script could we inherit from, if not this one, to change the behavior of the boss?

RangeAttackController.cs, which handles the range of the enemy's attack and the actual attack behavior. You can find it in the same folder as Enemy.cs. Opening it up you can see that both GetProjectileRange() and Fire() are virtual, meaning we can override them!

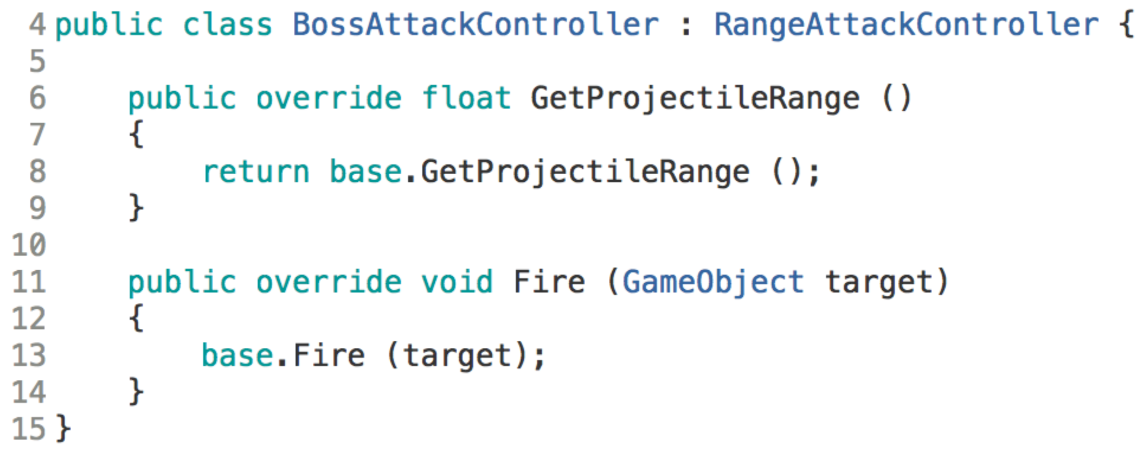
Now to create the new script! In the Project view you can either right click the NPC folder and select Create > C# Script or click the "Create" button with the NPC folder selected and click C# Script. Name this new script BossAttackController and then go ahead and open it up in MonoDevelop by double-clicking it.

Once it's opened, change the script to inherit from RangeAttackController instead of MonoBehavior. Then override Fire and GetProjectileRange. You should only have to begin writing "override" for the auto-complete to fill. You can also delete the Start and Update functions that the new script fills in automatically. When you're done your script should look like Figure 6.

2.6 Creating the Boss: Adding Behavior

Next we'll need to change the code in BossAttackController.cs to grant the boss new behaviors! As a starting point let's look at "Fire" in RangeAttackController.cs. You can find RangeAttackController in the NPC folder. You can also check Figure 7. We'll step through each line inside the function and see what it's doing.

- 36: An if-statement that determines if a variable named "shotTimer" has a value of 0. Basically makes sure the enemy can't just constantly shoot.
- 37: Uses the Unity function "Instantiate" that makes a copy of the projectile GameObject (the enemy bullet)

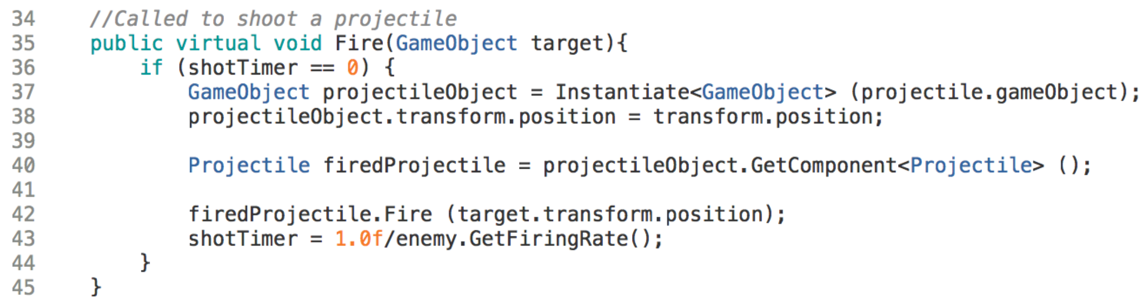


```
public class BossAttackController : RangeAttackController {

    public override float GetProjectileRange ()
    {
        return base.GetProjectileRange ();
    }

    public override void Fire (GameObject target)
    {
        base.Fire (target);
    }
}
```

Figure 6: BossAttackController.cs with the initial setup.

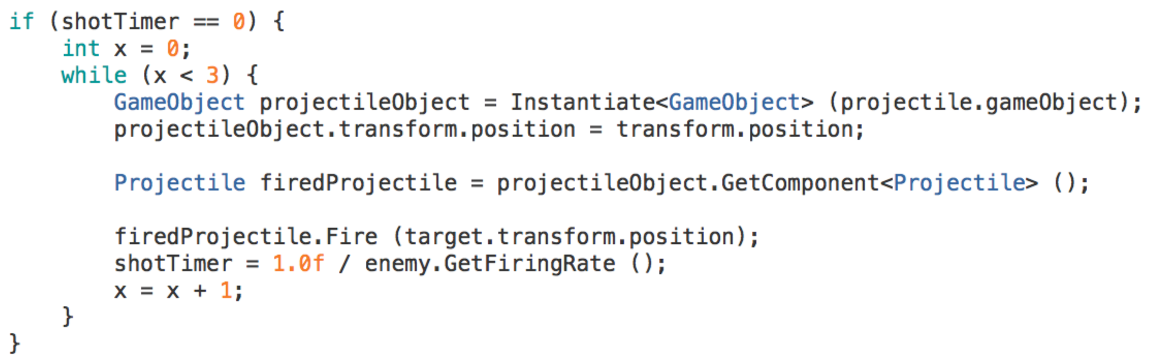


```
    //Called to shoot a projectile
    public virtual void Fire(GameObject target){
        if (shotTimer == 0) {
            GameObject projectileObject = Instantiate<GameObject> (projectile.gameObject);
            projectileObject.transform.position = transform.position;

            Projectile firedProjectile = projectileObject.GetComponent<Projectile> ();

            firedProjectile.Fire (target.transform.position);
            shotTimer = 1.0f/enemy.GetFiringRate();
        }
    }
```

Figure 7: RangeAttackController's Fire



```
if (shotTimer == 0) {
    int x = 0;
    while (x < 3) {
        GameObject projectileObject = Instantiate<GameObject> (projectile.gameObject);
        projectileObject.transform.position = transform.position;

        Projectile firedProjectile = projectileObject.GetComponent<Projectile> ();

        firedProjectile.Fire (target.transform.position);
        shotTimer = 1.0f / enemy.GetFiringRate ();
        x = x + 1;
    }
}
```

Figure 8: BossAttackController's Fire with a while loop

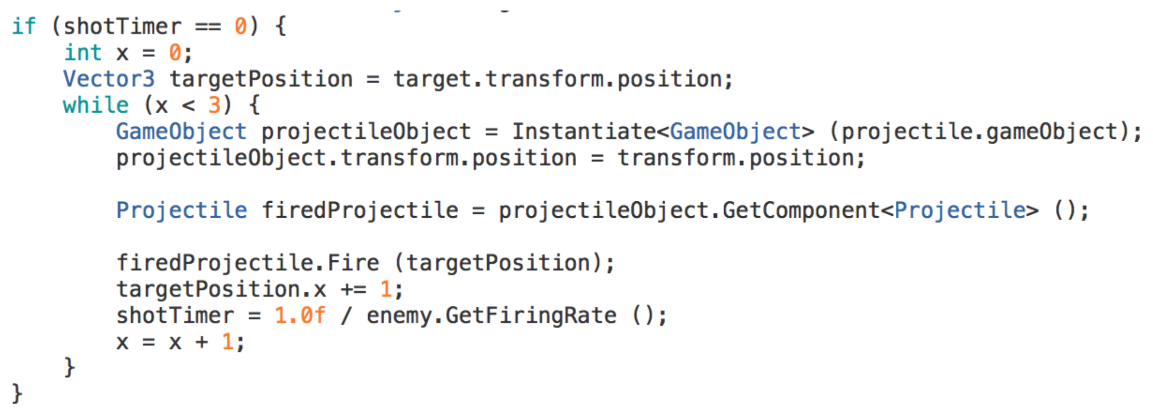
- 38: Set's the current position of the enemy's bullet to the enemy's position.
- 40: Grabs a reference to the Projectile value on the project GameObject
- 42: Fires the projectile at the target's position (the player in this case)
- 43: Resets the shotTimer variable so the enemy cannot immediately fire again.

Using code very much like, we could get very different behavior with some minor changes. Let's copy and paste the current code from RangeAttackController's Fire into BossAttackController's Fire as a start, then come up with different ways to modify it.

A common approach in games is to have a more difficult version of an attack simply be multiple versions of a simpler attack.We could use that same approach here by having the boss shoot several different projectiles at once. To run the same code over and over again we can make use of a while loop. Let's try it out! Define a new integer variable (we used x) with an initial value of 0, then make a while loop around the inside of the if-statement in Fire that will be called 3 times. When you're done the code in Fire might look something like Figure 8. Unfortunately, there's a problem with this code which will cause it to *look* identical to the normal enemy's Fire. Can you see why?

The issue is that all three of the projectiles will appear in the same position and go to the exact same position, along the same path. Meaning that they'll look like a single projectile! How can we fix this?

Well we can change either the position the projectiles begin at, or the position the projectiles end at (their "target position"). Let's go with changing the target position, to make it like the three projectiles spread out. To do that we'll make use of a Vector3 variable. Recall that variables can have the value they store changed. We can make use of this fact to first store the initial target position into a Vector3 variable, then update that variable every time the while loop code is called, passing the variable into the "firedProjectile.Fire" call. To update a Vector3 variable, you can update the Vector3's x, y, or z values individually. For now let's update the "x" value. Try doing this without looking at Figure 9. However, simply making these changes won't actually change the Boss object's behavior, can you think why?



```
if (shotTimer == 0) {
    int x = 0;
    Vector3 targetPosition = target.transform.position;
    while (x < 3) {
        GameObject projectileObject = Instantiate<GameObject> (projectile.gameObject);
        projectileObject.transform.position = transform.position;

        Projectile firedProjectile = projectileObject.GetComponent<Projectile> ();

        firedProjectile.Fire (targetPosition);
        targetPosition.x += 1;
        shotTimer = 1.0f / enemy.GetFiringRate ();
        x = x + 1;
    }
}
```

Figure 9: BossAttackController's Fire set up to create a "spread" of bullets.

2.7 Creating the Boss: Adding BossAttackController to Boss

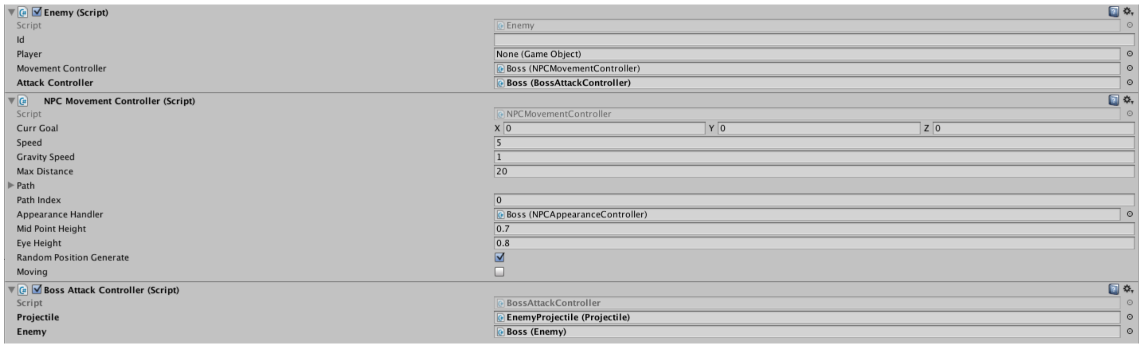
The issue is that we didn't attach our new BossAttackController.cs to the Boss object! It still has RangeAttackController.cs attached to it. Recall that we had to switch out the "MagicSword.cs" script for the "Sword.cs" script for our MagicSword object last lesson. Let's go ahead and make this change! Head back to the Unity window and select the Boss object in the Hierarchy view. Scroll down in the Inspector view till you see the RangeAttackController component. Once you see it, either right click on it and select "Remove Component" or click the cog in the top-right corner of it and select "Remove Component". With the old RangeAttackController.cs removed we'll need to add the BossAttackController.cs. You can add the BossAttackController component in two ways: (1)Press the large "Add Component" button at the bottom of the Inspector View with the Boss object selected then begin typing BossAttackController till you see the script pop up and then select it. (2) Click and drag BossAttackController in the Project View onto the Boss object in the Hierarchy.

Recall that when you attached the MagicSword.cs script component onto the Magic Sword object in the prior lesson you still had to fill in the values for it's public variables. We have to do the same here! Select the Boss object again and scroll down to the BossAttackController component. You will see it has two public variables without values (marked as **None**). We'll need to fill those in!

The first of these needs to be filled in with the Projectile prefab for the Fire function to use. We'll just use the default enemy Projectile for now. You can find it under Game Assets > Prefabs > EnemyProjectile. Find it in the Project View, open Boss back up in the Inspector view and drag the EnemyProject component into the slot on the BossAttackController component.

The second variable to fill in is a link to an Enemy component, in fact the Enemy component on this Boss object! Just click and drag the component onto the Enemy variable in BossAttackController to fill it in. While dragging you may notice the Enemy component has an Attack Controller variable that's labeled as "Missing" can you think why this might be?

The AttackController variable has a "Missing" value because we removed the RangeAttackController component it used to point to! Let's fill it in with the new BossAttackController component. Click and drag the BossAttackController component into the AttackController slot on the Enemy component. You've done it! Your boss is all set up! Your Inspector View for the Boss should look



Enemy (Script) ✓	
Script	Enemy
Id	
Player	None (Game Object)
Movement Controller	Boss (NPCMovementController)
Attack Controller	**Boss (BossAttackController)**
NPC Movement Controller (Script)	
Script	NPCMovementController
Curr Goal	X 0 Y 0 Z 0
Speed	5
Gravity Speed	1
Max Distance	20
Path	
Path Index	0
Appearance Handler	Boss (NPCAppearanceController)
Mid Point Height	0.7
Eye Height	0.8
Random Position Generate	✓
Moving	☐
Boss Attack Controller (Script) ✓	
Script	BossAttackController
Projectile	**EnemyProjectile (Projectile)**
Enemy	**Boss (Enemy)**

Figure 10: Inspector View of Boss components after set up

```
public class Projectile : MonoBehaviour {
    //The speed this projectile will move at
    public float projectileSpeed = 10;
    //The current life (counts up till maxLifeTime)
    private float lifeTime = 0;
    //The max possible amount of time (in seconds) the projectile can be alive
    private const float maxLifeTime = 1f;
    //Destination of this projectile
    private Vector3 destination;
```

Figure 11: The top of the Projectile.cs class

something like Figure 10.[1]

Play the game and see how your new boss attack looks!

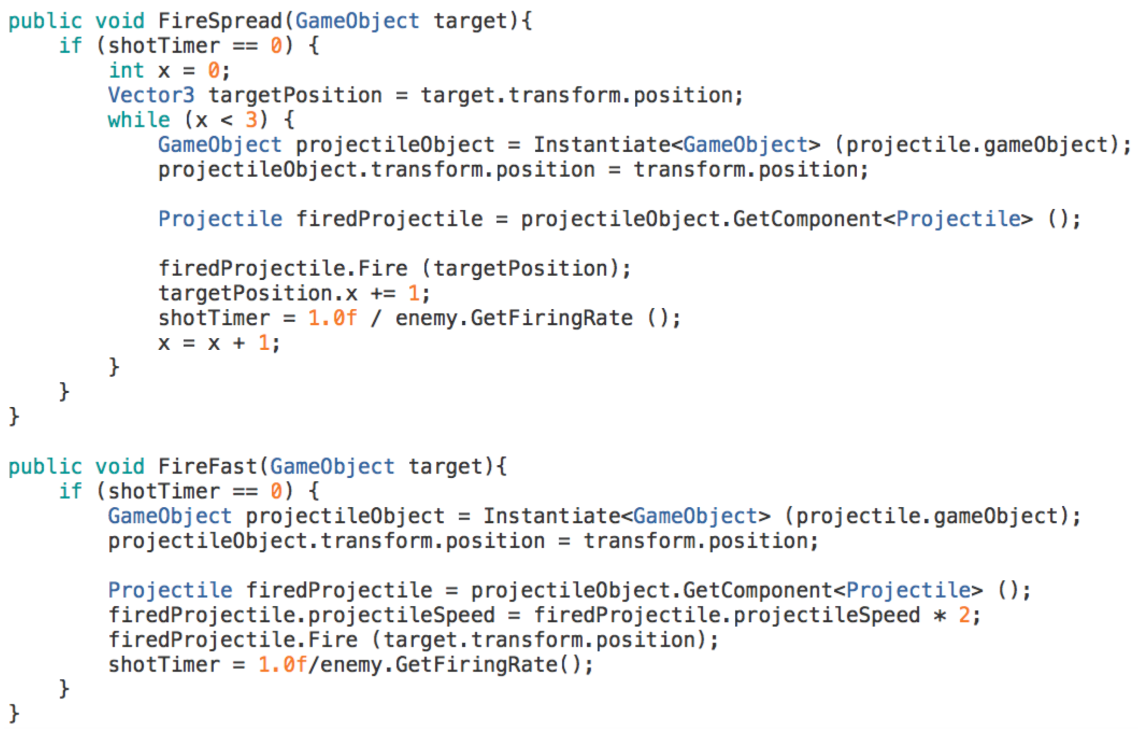
2.8 Additional behaviors

The spread of projectiles looks pretty good, but that's not really enough for a boss, is it? In adventure games a boss tends to have at least two different attacks. Generally these attacks are based on certain conditions, such as distance to the player or the boss' health. We'll walk through adding one more behavior in this section, and leave any behaviors beyond that up to you.

The projectiles that the Boss shoots are kind of slow, don't you think? What if we had a version of the Boss' attack that shot a single very fast projectile if the player was a certain distance away? That might make the fight a bit more challenging! If you take a look at the top of Projectile.cs as seen in Figure 11, you'll see a public variable (remember public means we can modify it outside the class) called projectileSpeed that will allow us to do just that!

Unfortunately before we go creating a new attack we need to figure out *where* to make it. The Fire function is currently full of the code to shoot the "spread" of projectiles. But if we want to be able to shoot the spread *or* the fast shot, we'll need to have access to the code to do both. Can you figure out a good way to do that?

[1]You may notice the Enemy component also has a "Player" variable that's currently set to None. Don't worry! This gets filled in automatically by the GhostController.cs script component. Feel free to open it up to see how it works!



```
public void FireSpread(GameObject target){
    if (shotTimer == 0) {
        int x = 0;
        Vector3 targetPosition = target.transform.position;
        while (x < 3) {
            GameObject projectileObject = Instantiate<GameObject> (projectile.gameObject);
            projectileObject.transform.position = transform.position;

            Projectile firedProjectile = projectileObject.GetComponent<Projectile> ();

            firedProjectile.Fire (targetPosition);
            targetPosition.x += 1;
            shotTimer = 1.0f / enemy.GetFiringRate ();
            x = x + 1;
        }
    }
}

public void FireFast(GameObject target){
    if (shotTimer == 0) {
        GameObject projectileObject = Instantiate<GameObject> (projectile.gameObject);
        projectileObject.transform.position = transform.position;

        Projectile firedProjectile = projectileObject.GetComponent<Projectile> ();
        firedProjectile.projectileSpeed = firedProjectile.projectileSpeed * 2;
        firedProjectile.Fire (target.transform.position);
        shotTimer = 1.0f/enemy.GetFiringRate();
    }
}
```

Figure 12: BossAttackController with the two functions

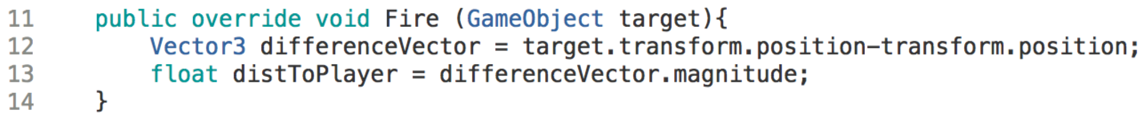
The solution we came up with to do this is to put the code for FireSpread or FireFast in two different functions and then call one or another in the main Fire function! Recall from Lesson Four creating your own functions, essentially you're creating a chunk of code that can be called by simply calling the function name. We named our two functions FireSpread and FireFast. FireSpread should have code inside it identical to the original Fire function in RangeAttackController, but with one addition after the line getting the Projectile reference:

```
firedProjectile.projectileSpeed = firedProjectile.projectileSpeed*2;
```

This will cause the newly firedProjectile to move at twice the speed, as it will hold a value of projectileSpeed two times faster than any other projectiles. Try setting up these functions without taking a look at Figure 12.

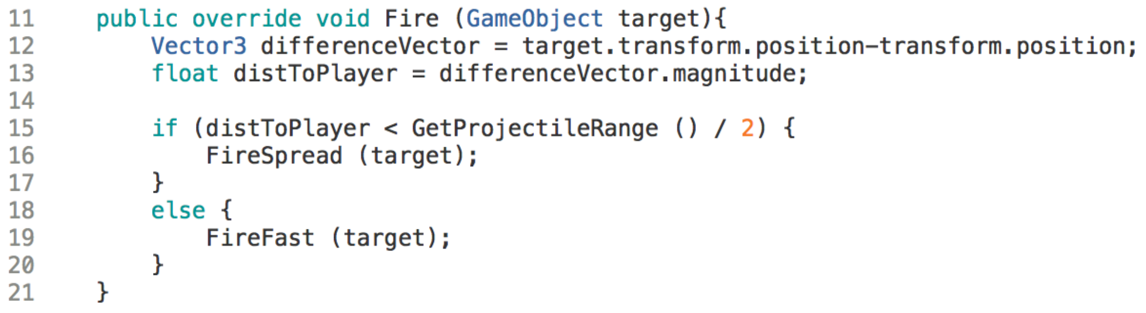
Of course now we have the problem that there's nothing in the Fire function! Say we want the enemy to use the FireSpread attack when the player is closer than half of the max range and to use FireFast otherwise. That sounds a bit like an if statement might be useful, don't you think? But we'd need some way to get the distance between the target and the Boss. We can actually use Vector3s again for this! Vector3 variables can store not just position, rotation, and scale information but also *differences* between these values.

Let's consider an example. Say that the player was located at (10, 10, 10) and the Boss was located at (1,9,1). Then the "difference" between them would be (9, 1, 9) as 10-1 = 9, 10-9 = 1, and 10-1 =9. We can get the length of this Vector as a single number using the "magnitude" of



```
public override void Fire (GameObject target){
    Vector3 differenceVector = target.transform.position-transform.position;
    float distToPlayer = differenceVector.magnitude;
}
```

Figure 13: BossAttackController Fire function showing how to get the distance to the player.



```
public override void Fire (GameObject target){
    Vector3 differenceVector = target.transform.position-transform.position;
    float distToPlayer = differenceVector.magnitude;

    if (distToPlayer < GetProjectileRange () / 2) {
        FireSpread (target);
    }
    else {
        FireFast (target);
    }
}
```

Figure 14: BossAttackController Fire function calling both functions.

a Vector3. That likely sounds a bit confusing, so check out Figure 13 for an example. Once you have this, can you figure out how to use an if statement and an else statement to call FireSpread and FireFast as we've suggested? Try to do that before checking Figure 14.

Once you've got the Fire function working, play the game! Make sure you can get both kinds of attacks!

That's all we'll be walking you through, but you could imagine a number of different attacks you could add. If the player is out of range, maybe the boss "teleports" to the player's position (via using transform.position). Or maybe you have a different kind of attack that increases the scale of the projectile? Or maybe an attack that launches the player into the air if they get too close? It's up to you!

2.9 Full Quest Set Up

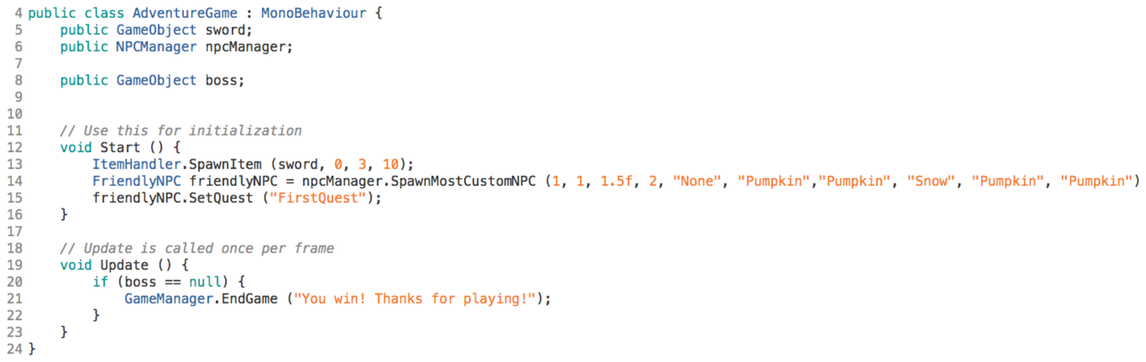
Once you're happy with your Final Boss, let's see about finishing this Quest up! Before we change the player transportation from the first quest to the final quest let's make it so that beating the Final Boss leads to winning the game. We'll make use of this by adding our own reference variable.

First, open up AdventureGame.cs again. We'll then want to make a new public variable to reference the Boss gameobject. Save the script and then select "Admin" in the Hierarchy View. Scroll down till you see the AdventureGame component on it. You should see a new variable slot named Boss that's currently empty! That's what creating the public variable does. If you'd like to see it again, go back to AdventureGame.cs, delete it, and you'll see that it has disappeared in the Unity view. For now though let's drop the Boss gameobject in the Inspector view into the Boss variable on the AdventureGame component.

Recall that when we deleted the RangeAttackController from the Boss gameobject it's Enemy had a variable with the value "Missing". So if the Boss gets deleted (because the hero beats it), then the Boss variable in AdventureGame.cs will also have a "Missing" value. In C# we represent Missing or Nothing with the keyword: **null**. You can then create a condition to represent if the

Definition!

NULL:
The keyword used to represent missing or nothing.



```
public class AdventureGame : MonoBehaviour {
    public GameObject sword;
    public NPCManager npcManager;

    public GameObject boss;


    // Use this for initialization
    void Start () {
        ItemHandler.SpawnItem (sword, 0, 3, 10);
        FriendlyNPC friendlyNPC = npcManager.SpawnMostCustomNPC (1, 1, 1.5f, 2, "None", "Pumpkin","Pumpkin", "Snow", "Pumpkin", "Pumpkin")
        friendlyNPC.SetQuest ("FirstQuest");
    }

    // Update is called once per frame
    void Update () {
        if (boss == null) {
            GameManager.EndGame ("You win! Thanks for playing!");
        }
    }
}
```

Figure 15: AdventureGame after the win game condition is set up.

boss variable has a missing value or no value as:

```
boss == null
```

When you're done your AdventureGame.cs script should look like 15, feel free to change the text for the win screen!

Test out winning! Make sure to pick up the sword before getting transported to the Boss Area so you can fight the boss!

Now all that's left is to remove the teleporting the player from the FirstQuest and place it in either a ThirdQuest spawned with a new NPC along with the MagicSword, or to teleport the player and the MagicSword to the boss location at the end of SecondQuest. We leave that up to you! Once you've got it working, have a friend test it out. What do they think? Is there anything you could change to make it better? One simple thing might be that the Boss is too easy if you didn't add hitpoints to enemies in Lesson3. You could add those hit points or duplicate the boss to create lots of distractions!

3 End of Lesson Six

And that's the end of the lesson! In this lesson we went over:

1. The use of Vector3 to teleport objects
2. Creating new complex game objects and scripts
3. Using null to determine when an object is destroyed
4. Reminders on the use of while loops, if statements, and new functions.

This marks the ends of the lessons. While there is a Lesson 7, it's just a launching point to work on your own game and will have much less descriptions. We hope you enjoyed these lessons!

3.1 Lesson Six Glossary

Vocabulary Word	Definition	Examples
Vector3	A complex variable type to store x,y, and z values	transform.position
new	A keyword that creates a class variable from a class	new Vector3(0,0,0);
null	A keyword that represents Missing or Nothing	null