Lesson Five

Going Questing

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class AdventureGame : MonoBehaviour {
5     public GameObject sword;
6     public NPCManager npcManager;
7
8
9     // Use this for initialization
10    void Start () {
11        npcManager.SpawnEnemies(10, 0, 0, 0, 10);
12        ItemHandler.SpawnItem (sword, 0, 3, 10);
13    }
14
15    // Update is called once per frame
16    void Update () {
17
18    }
19 }
```

Figure 1: The end of AdventureGameGenerator.cs after Lesson Four

1 Introduction to Lesson Five: Going Questing

The "quest" is a typical part of the adventure genre. Typically they are challenging tasks given by a wise person or elder. In adventure games quests are given to the player by some non-player character (NPC). In this lesson we will add NPCs and set up your "game world" to include everything needed for a few simple quests. In the process you'll get to write NPC speech, utilize the power of inheritance, and learn about the use of randomness to procedurally generate content.

2 NPCManager and NPC Appearance

If you closed Unity between lessons, go ahead and open it back up. Open the LessonFour-Six scene. You might want to take a second to play the game again once this scene is open to remind yourself what the world you made looked like. Remember, at any time throughout this lesson you can change how your game world looks in AdventurGameGenerator.cs.

To begin we'll be adding our first NPC to the game. To do that, open up AdventureGame.cs.

Figure 2: All three of the different NPCs in their default appearance.

You can find this file in the Project View under the Codebase folder then the LessonsFour-Six folder. Double click it to open it in MonoDevelop. In the prior lesson you added code to this script to handle spawning enemies and the sword within the Start function. Our code ended up looking like Figure 1, yours probably looks similar (potentially with different arguments).

To this we'll want to add a call to spawn our very first NPC. To do this we'll use a function in NPCManager, which looks like:

```
public FriendlyNPC SpawnNPC(int npcType, float x, float y, float z)
```

This function is called "SpawnNPC" and takes as arguments an int npcType, a float x, a float y, and a float z. It returns a new type of variable called "FriendlyNPC", we'll get into that later.

As you might expect these x, y, and z values are the ones that determine the location of the NPC. The int "npcType" argument determines the type of NPC. There are three types of NPC in the game, so the value can be 0 (baby), 1 (tall), or 2 (squat). So in AdventureGameGenerator we could add the line "npcManager.SpawnNPC(0,5,5,5);" to create a "baby" NPC at position (5,5,5). Alternatively you could add the line "npcManager.SpawnNPC(2,3,4,5)" to spawn a "squat" NPC at position (3,4,5).[1] Let's go ahead and add a single NPC to our game world. Pick whichever one you like! We show an example of each of the three different NPCs in Figure 2, 0, 1, and 2 from left to right. We'd also recommend removing the call that spawns enemies for now, it's a bit time-consuming to have to deal with all the ghosts running around! Doing that your Start function

[1]Using a variable besides 0,1, or 2 for npcType will simply default to the closest allowable number. So using a number greater than 2 would spawn a squat NPC (as if you used 2) using a number less than 0 would spawn a baby NPC (as if you used 0).

should look something like the below code. Your location arguments may have to differ, based on the appearance of your game world.

```
void Start () {
  ItemHandler.SpawnItem (sword, 0, 3, 10);
  npcManager.SpawnNPC (0, 5, 1.5f, 2);
}
```

Play the game and check out your NPC! It should look at you if you're close, and stop watching you otherwise.

This "tree-like" character may not match the game world you made in the prior lesson. Say you had a lava island, or a snow island! So there are actually two other functions you can use to give you the ability to customize the appearance of the character. These would be:

public FriendlyNPC SpawnCustomNPC(int npcType, float x, float y, float z, string npcTexture, string clothingTexture)

AND

public FriendlyNPC SpawnMostCustomNPC(int npcType, float x, float y, float z, string hatTexture, string headTexture, string noseTexture, string bodyTexture, string leftLegTexture, string rightLegTexture)

In the first option, "SpawnCustomNPC" there are two additional arguments. These are string arguments that determine the material to use (like the ones we used in Lesson 4 such as "Diamond" and "Lava"). for the NPC's "skin" (the first one: npcTexture) and the NPC's clothing (the second one: clothingTexture).

The second option, "SpawnMostCustomNPC"there are six additional arguments. These are all string arguments that determine the material for the part of the NPC mentioned in the argument name. So in other words the hat (hatTexture), head (headTexture), nose (noseTexture), body/outfit (bodyTexture), left leg (leftLegTexture), and right leg (rightLegTexture). There's a list of all usable material names at the end of this lesson, but you can also use the string value "None" to cause the part specified of the NPC not to show up at all!

With these two different functions, both defined in NPCManager, you should be able to customize your NPC to match your world. We present several different NPCs and the commands in Start used to create them in Figure 3. Feel free to mess around to find an NPC appearance that matches your world. Once you've found one, move on to the next section.

Definition!

REFERENCE:
In programming, a reference is a way of referring to a variable that allows us to interact with some object (such as an NPC).

2.1 Adding a Quest

Now that you've got an NPC we need to make that NPC give a quest to the player character. To do that, we'll need some **reference** to the newly created NPC. In computer science, a reference is a way of referring to a variable that allows us to interact with some object (such as an NPC). For example, the "sword" variable in "AdventureGame" can be described as a reference, in that it refers to a sword "object" that exists elsewhere. We'll need a reference to the created NPC in order to set up a quest for it.

```
13          npcManager.SpawnCustomNPC (0, 5, 1.5f, 2, "Lava", "Stone");
14          npcManager.SpawnCustomNPC (1, 4, 1.5f, 2, "Lava", "Stone");
15          npcManager.SpawnCustomNPC (2, 3, 1.5f, 2, "Lava", "Stone");
16          npcManager.SpawnMostCustomNPC (0, 2, 1.5f, 2, "Snow", "Pumpkin","Pumpkin", "Snow", "Pumpkin", "Pumpkin");
17          npcManager.SpawnMostCustomNPC (1, 1, 1.5f, 2, "None", "Pumpkin","Pumpkin", "Snow", "Pumpkin", "Pumpkin");
```

Figure 3: The NPCs and the commands that make them.

In order to get such a reference we can make use of the fact that all of the functions that "spawn" an NPC return a FriendlyNPC value. We can store that value into a variable in order to get our reference! The variable will need to be of the type "FriendlyNPC". We used "questGiver" as the variable name, but feel free to use whatever variable name you like. To place the value returned by the function used to spawn your NPC, you just have to use an "=" like we have throughout these lessons. In other words just put the function call to spawn your NPC on the right side of "FriendlyNPC questGiver = ".

Once you have a reference to the NPC, we'll want to call the function "SetQuest" with it. SetQuest takes a single string value argument: the name of the quest we want to use, we'll use "Quest" for now. So the call to SetQuest should look like:

```
questGiver.SetQuest ("Quest");
```

Play the game once you've made these changes! If you need some help figuring out exactly what to do, check out Figure 4. In the game go up to your NPC and see what happens!

If everything worked correctly, you should see a message pop up when you get close to the NPC

```
void Start () {
    npcManager.SpawnEnemies (10, 10, 0, 10, 10);
    ItemHandler.SpawnItem (sword, 0, 3, 10);
    FriendlyNPC questGiver = npcManager.SpawnMostCustomNPC (1, 1, 1.5f, 2, "None", "Pumpkin", "Pumpkin", "Snow", "Pumpkin", "Pumpkin");
    questGiver.SetQuest ("Quest");
}
```

Figure 4: What our Start function looked like in AdventureGame.

```
60      private void PlayerEnteredArea(){
61          if (HasQuest()) {
62              if(!myQuest.HasStarted()){
63                  if (myQuest.CanStart ()) {
64                      myQuest.QuestStart ();
65                  } else {
66                      myQuest.CannotStart ();
67                  }
68              }
69              else{
70                  if(myQuest.CanEnd()){
71                      Quest oldQuest = myQuest;
72                      RemoveMyQuest();
73                      oldQuest.QuestEnd();
74
75                  }
76              }
77          }
78      }
```

Figure 5: The code that handles calling the various Quest functions in FriendlyNPC.cs

that says "Quest cannot start!". That's something! But why does it say that?

Given that the only thing you changed before this message showed up was the call to add "Quest" to your NPC, it makes sense that this lead to the change you saw. Let's go ahead and open Quest.cs up and take a look at what's going on.

2.2 Diving into Quest.cs

You can open up Quest.cs from the Unity Project view. It can be found under the following folders: Codebase, NPC, and then Quests. Double click it to open it in MonoDevelop.

Once Quest.cs is opened, take a look at it. Scroll down until you see text that matches the message that popped up, can you find it?

You should see it in the function "CannotStart", but let's make sure.

Change the text in the function "CannotStart" and play the game again then walk up to the NPC. Make sure it shows your new text!

Well that proves that! You should have seen your text pop up when you got near the NPC. If you didn't make sure you save Quest.cs after making your changes and play the game again.

But *why* is the text popping up? Check out Figure 5, this is the chunk of code from FriendlyNPC that handles the various calls to the Quest (held in the variable myQuest). The function is called when the player enters the "area" of the NPC, or stays in the area for a few seconds. Can you find where "CannotStart" is called in this chunk? Does it make sense that its being called?

Tracing the "PlayerEnteredArea" function you should see that to get to the call to "myQuest. CannotStart()" it must be true that the NPC has a Quest (which we know, since we added one to it), and that it *isn't* true that the quest has started (! myQuest.HasStarted()). This should make sense given that in Quest we can see that HasStarted() returns the variable "started" which is false (so with the "!" (not) operator, the condition is true). So "myQuest.CanStart ()" cannot be true, which is the case since in Quest we can see "CanStart" just returns false. So that's why CannotStart is called! If "myQuest.CanStart ()" returned true instead then "QuestStart" in Quest would be called.

So that explains the logic of quests in FriendlyNPC in terms of when functions are called, but how does that help us actually set up a quest for the player? Actually, we won't be using Quest.cs to do that at all! We'll instead be using the "children" of Quest that inherit from it in order to create many different Quests that all use the chunk of code in 5. Can you think why that might be?

Think about it like this, we could go ahead and put all the code into Quest.cs we'd need to make a single quest. But what if we wanted multiple quests? Most adventure games have more than one, so it'd only make sense for our little adventure game to be the same. But Quest.cs can't contain the code to handle multiple different quests, it's only made to hold one! So we'll need to use it as the basis for *other* classes, other capital "Q" Quests, that'll actually contain the quests of the game.

To make these different quests we'll need to dive a bit deeper into the details of Quest.cs.

2.3 Diving deeper into Quest.cs

Quest is a bit different from the scripts we've seen before. While it inherits from MonoBehaviour ("Quest : MonoBehaviour" at the top of the file), it introduces a couple new keywords and has the most functions out of any class we've seen (don't worry though! They're all very simple). The two new keywords are **protected** (used for the second two variables up at the top) and **virtual** (used in the last half of the functions).

Definition!

PROTECTED:
An access modifier, like private or public that allows access to a variable or method only in the class it is defined in and that class' children.

The keyword "protected" is an access modifier like public or private, and it works a bit like private. Except that instead of restricting a variable to only being accessed in a single class, it means a variable can be accessed in the class it is defined in and *any classes that inherit from that class*. For example, since Quest inherits from MonoBehaviour, any variables or functions marked with protected in MonoBehaviour could be used in Quest, but could not be used in a class that did not inherit from MonoBehaviour. We'll go more into depth with this so don't worry if it's still confusing.

Definition!

VIRTUAL:
A keyword that allows for a function in one class to be overwritten in the children of that class.

The keyword "virtual" allows for a function in one class to be overwritten in the "children" of that class (those classes that inherit from it). In this case, overwritten means that the function will have the exact same definition (in terms of name, arguments, return type) in some "child" class, but will have different code inside it. This means that a function in the "child" can be called in the same way as a function in the "parent" class, but to a much different effect.

That's all very abstract, so let's give an example. Remember our example of a class being like a house blueprint, capable of making many actual houses. You may have heard of "cookie cutter"

```
using UnityEngine;

public class FirstQuest : Quest {
    //Called when the quest starts
    public override void QuestStart (){
        base.QuestStart ();

        //Add code here
    }

    public override void QuestUpdate (){
        base.QuestUpdate ();

        //Add code here
    }

    public override void CannotStart (){
        GUIManager.SetDisplayTextColor ("I can't start for some reason!", 3, Color.red);//Replace this line
    }

    public override bool CanStart (){
        return base.CanStart();//Replace this line
    }

    public override bool CanEnd (){
        return base.CanEnd();//Replace this line
    }

    public override void QuestEnd (){
        base.QuestEnd ();

        //Add code here
    }
}
```

Figure 6: FirstQuest.cs initially

housing, which are neighborhoods where all the houses look pretty much the same. Say that you wanted you were a developer who wanted to make a neighborhood like this (as it's a big cost-saving approach). You might create one blueprint, but "pencil in" certain sections of it. Then you could make a copy of the blueprint and alter just those penciled in sections. This copy could be thought of as "inheriting" from the first blueprint, with those pencilled in sections being equivalent to using the "virtual" keyword, seeing as they can be "overwritten". They'd be unique parts that would still fit with the unchanged sections!

2.4 Messing with the First Quest

Let's put this together by looking at our first child of Quest, which we'll make into our first real quest: "FirstQuest.cs". Before we do anything else let's make sure we pass the NPC this new quest. Return to AdventureGame.cs in MonoDevelop (should still be open in a tab along the top). Replace the string value passed into SetQuest with "FirstQuest".

Play the game and walk up to the NPC once more. The text should be different, and not just in the words it used!

Instead of the prior text the text "I can't start for some reason!" in red should have popped up. How did that happen? Well let's open up FirstQuest.cs and see! The file can be found in the same folder as Quest.cs, so double-click it to open it in MonoDevelop.

When you open up FirstQuest.cs you should see something that looks like 6. It should be

Definition!

OVERRIDE:
A keyword that specifies that the function in question overrides the behavior of the "inherited" function.

pretty easy to pick out where the line causing the display text is, as it's in FirstQuest's version of "CannotStart". FirstQuest's function in fact *overrides* Quest's CannotStart function, which should make sense given that it's definition includes the new keyword: **override**. The keyword override specifies that the function that its used in front of overrides the behavior of the inherited function of the same name from the "parent" class. So in this case CannotStart in FirstQuest.cs overrides the behavior of CannotStart in Quest.cs. So it calls the function "SetDisplayTextColor" instead of "SetDisplayText" and uses different text.

Let's give an example to help explain what's going on here. Go ahead and remove the *entire* function CannotStart from FirstQuest.cs. Just delete the entire function.

Play the game and go up to the NPC. See what happens!

Now that's a bit weird, huh? The text you added into Quest.cs showed up instead of the text from FirstQuest! This is the power of inheritance. Since FirstQuest inherits from Quest it by default acts the same as Quest.cs. It's only if we *override* certain functions that they behave differently. Essentially Unity calls the chunk of code in FriendlyNPC seen in Figure 5 and when it hits the line myQuest.CannotStart() in Figure 5 it calls the "closest" CannotStart it can. When FirstQuest.cs had one, it called that, but since it doesn't it calls the one found in Quest.cs. This is similar in the blueprint example to if we didn't change the pencilled in sections, we'd just use those parts from the original.

2.5 Making the First Quest

Hopefully now that you have a better grasp of inheritance we'll walk you through the steps of setting up this first quest to show you how it's done. Then you'll have more freedom for setting up the following quests. To begin with we need to determine what exactly we want this quest to be about. Given that there's a sword in the game world already, why don't we have it be that the NPC wants the player to go fetch it?

To set up the quest we'll need to determine the following things, based on the functions that can be overridden from Quest:

1. What starting condition there is, if any (CanStart)?
2. What should happen when the quest starts (QuestStart)?
3. What should happen every Update call of the quest, if anything (QuestUpdate)?
4. What condition must be reached for the quest to end (CanEnd)?
5. What should happen when the quest ends (QuestEnd)?

So let's walk through each step of this for our fetch quest:

1. CanStart: There doesn?t need to be a condition that must be true for the quest to start, so CanStart just needs to return "true".
2. QuestStart: When the quest starts, we need some way to prompt the player to get the sword. Feel free to use GUIManager.SetDisplayText or GUIManager.SetDisplayTextColor. Either will work! You can check Figure 6 for an example of GUIManager.SetDisplayTextColor in action.

```
3 public class FirstQuest : Quest {
4     //Called when the quest starts
5     public override void QuestStart (){
6         base.QuestStart ();
7         GUIManager.SetDisplayTextColor ("Bob: Hey! You don't look tough, go get a sword!", 3, Color.green);
8     }
9
10    public override void QuestUpdate (){
11        base.QuestUpdate ();
12    }
13
14    public override bool CanStart (){
15        return true;
16    }
17
18    public override bool CanEnd (){
19        return ItemHandler.GetCountCollected ("Sword") > 0;
20    }
21
22    public override void QuestEnd (){
23        base.QuestEnd ();
24        GUIManager.SetDisplayTextColor ("Bob: Whoa much better!", 3, Color.green);
25    }
26 }
```

Figure 7: FirstQuest.cs after the initial changes to make it a full, real quest.

3. QuestUpdate: Nothing needs to happen during QuestUpdate, so we can safely ignore or even delete that.

4. CanEnd: The ending condition should be when the player has the sword. Now if you'll recall LessonTwo, we went through a whole bunch of trouble to determine that. We actually have a much simpler way to figure it out, now that you know relational operators. We can call ItemHandler.GetCountCollected("Sword") to determine the number of swords the player has collected. If that number goes above 0 we know the quest can end. (ItemHandler.GetCountCollected("Sword")>0).

5. QuestEnd: We need some way to tell the user that they did well, so use GUIManager.SetDisplayText or GUIManager.SetDisplayTextColor again. We'll add more to this later, but that'll be enough for now.

Try to make all the changes spelt out above before checking out Figure 7 which has our version of FirstQuest.cs. Worth noting that in calls to GUIManager.SetDisplayTextColor "Color" is a class with "red", "green" and so on being special variables. There are other color options, so feel free to play around with it!

Play the game and go up to the NPC, and then follow the quest to the end. Make sure it all works!

Now that that's working it's useful to consider *how* it's working. Take a look back at Figure 5 and try to trace through it. Can you see where things are being called and in what order? Does that match up to what you saw in the game? Try playing the game again and going up to the NPC a second time without a sword. Does what happen (or doesn't happen) make sense?

Definition!

BASE:
A keyword that refers to the "parent" of the current class.

In Figure 7 you can see one more keyword we haven't yet explained. The keyword **base** refers to the "parent" of the current class. So in this case, base refers to "Quest". You see it in both QuestStart and QuestEnd calling the Quest-version of these functions, as those set the variables started and ended in Quest to true. That's necessary as otherwise the code in FriendlyNPC won't

work! We could set those variables to true ourself, but there's no need to rewrite that code when we can just have the "parent" class Quest do it for us.

2.6 Second Quest

It's a common practice in games to randomize the location of items or enemies in order to add variety to the game. In fact, we've already made use of this fact with the randomly spawning enemies you added in the last lesson (and will re-add). To illustrate this we'll create a second quest that involves collecting some randomly placed items. To begin with though we'll first want to create the quest without randomness, and then add that in.

For this quest, let's have it be that the NPC that spoke to you before about needing a sword wants you to collect some things that it lost (now that you look like a big tough warrior, with your sword and all). You've got two options for what to have the player collect at this point: coins or cats. We'll continue forward using coins, but it should be simple enough for you to figure out how to use cats instead.

To begin with let's go ahead and just make a version of the quest without randomness and with a single coin (or cat) needing to be grabbed. To do that we'll first need to open up SecondQuest.cs. It can be found in the same folder as Quest.cs and FirstQuest.cs and should look very much like the other two when you first open it up. Just as we did above, we'll go through each step of the first draft of this quest:

1. CanStart: We should ensure that before the player can start that they have a sword, using the same technique we did for CanEnd last time.

2. QuestStart: When the quest starts, we'll want a single coin to appear. To do that you can use: ItemHandler.SpawnItemFromString(string itemName, float posX, float posY, float posZ). In this case if we're trying to spawn a Coin use the string value "Coin" and then the x, y, and z coordinates you want it to spawn at. We'll also need to tell the user to go get the coin that just popped up.

3. QuestUpdate: Nothing needs to happen during QuestUpdate, so we can safely ignore or even delete that.

4. CanEnd: The ending condition should be when the user has a single coin (we can change this later when we're spawning more than one coin).

5. QuestEnd: Tell the user thanks for grabbing the coin.

Once again, try to write all the code for the quest yourself, without looking at Figure 8. However, if you need help or want to double-check what you have it's there as our first draft version of SecondQuest. Can you figure out what changes need to be made to get a cat instead of a coin to show up?

You could try to play the game again at this point, but there'd be a problem. The NPC never has SecondQuest set as it's quest! Now we could go back to AdventureGame.cs and replace "questGiver.SetQuest ("FirstQuest");" with "questGiver.SetQuest ("SecondQuest");" but then the NPC would go straight to SecondQuest, and we want to do FirstQuest *then* SecondQuest.

The solution here is to use the power of the "protected" variable from Quest: myNPC. Because myNPC is marked as protected in Quest, it's available to all of Quest's children as if it were defined

```
public class SecondQuest : Quest {
    //Called when the quest starts
    public override void QuestStart (){
        base.QuestStart ();
        GUIManager.SetDisplayTextColor ("Bob: Alright cool you have a sword, can you find my lucky coin?", 3, Color.green);
        ItemHandler.SpawnItemFromString ("Coin", 10, 3, 10);
    }

    public override void CannotStart (){
        GUIManager.SetDisplayTextColor ("You don't have a sword!", 3, Color.blue);//Replace this line
    }

    public override bool CanStart (){
        return ItemHandler.GetCountCollected ("Sword") > 0;
    }

    public override bool CanEnd (){
        return ItemHandler.GetCountCollected ("Coin") > 0;
    }

    public override void QuestEnd (){
        base.QuestEnd ();
        GUIManager.SetDisplayTextColor ("Bob: That's it! That's my lucky coin! Thanks!", 3, Color.green);
    }
}
```

Figure 8: SecondQuest.cs after our first "draft" of the quest is complete.

as a global variable in them. So that means both FirstQuest and SecondQuest can use it! Using that variable will allow us to set SecondQuest to the NPC in FirstQuest's QuestEnd. Simply open up FirstQuest and add the following line to QuestEnd:

```
myNPC.SetQuest ("SecondQuest");
```

Play the game! Play through the entirety of FirstQuest, and then return to the same NPC to begin SecondQuest.

That works! But how? To set FirstQuest to this NPC in AdventureGame we used the line: "questGiver.SetQuest ("FirstQuest");", but to set SecondQuest to this NPC in FirstQuest we used the line: "myNPC.SetQuest("SecondQuest);". questGiver and myNPC are entirely different variables, how did this work?

This worked because even though questGiver and myNPC are different variables they have the same value, which references the NPC that we see in the game. It'd be like if we defined two variables a and b, which both stored the value "5". It wouldn't be weird that multiplying either variable's value by 2 we get 10, would it? It's the same here, with both variables "holding" the same NPC value.

2.7 Adding Randomness

Now let's add in randomness! To start with, let's figure out the effect that we want to achieve. We want to make it so that Coins or Cats appear in random locations across an area. So we can think about this as wanting to be able to pass random values into the x, y, or z arguments of ItemHandler.SpawnItemFromString(string itemName, float posX, float posY, float posZ).

But we don't want "truly" random variables, or else we would get values going from negative infinity to positive infinity, which wouldn't make the coins or cats collectable for the player. Instead we want to make it random within some range of possible values. For example, make it so that the objects only appear above one of the floating rectangular islands, or on top of one of the trees. You may want to refer back to AdventureGameGenerator.cs for reference.

```
public override void QuestStart (){
    base.QuestStart ();
    GUIManager.SetDisplayTextColor ("Bob: Alright cool you have a sword, can you find my lucky coin?", 3, Color.green);
    int numSpawned = 0;

    while (numSpawned < 10) {
        float x = Random.Range (0, 20);
        float z = Random.Range (0, 20);
        ItemHandler.SpawnItemFromString ("Coin", x, 3, z);
        numSpawned = numSpawned +1;
    }
}
```

Figure 9: QuestStart of SecondQuest.cs with randomly placed coins.

Unity has a function to return a random float within a range: "Random.Range(float min, float max);" Random is the class that handles all randomness in Unity, and Range is a function within it that returns a random number between some minimum and some maximum value. We can use this to get a random number in the following way:

```
float x = Random.Range(0, 20)
```

With this line of code the value in the variable "x" would be somewhere between 0 and 20. Before getting too much further let's show this in action. Go ahead and create some variable (we used x) and have it set up to be a random number within some range that'll make the coin possible to get. So if you have a rectangular island of length 20 that starts at the origin "Random.Range(0, 20)" would work or if you had an island of length 10 starting with an x-coordinate of 15 then "Random.Range(10, 25)" and so on.

Replace the current x-argument in your call to "SpawnItemFromString" with this new random variable. Play the game and see where the item spawns! Then play it again and see where it spawns this time.

It works! Now try the same thing with a variable to control the z-coordinate of the item you spawn.

Replace the current z-argument in your call to "SpawnItemFromString" with this new random variable. Play the game again!

So now we have a single coin or cat spawning at a random location. But what if we wanted many different coins or cats at different locations? What we'd essentially want is to rerun the code that comes up with two random numbers and then spawns an item at that location many times. That sounds a bit like a while loop!

To use a while loop to spawn many coins or cats you'll need a variable to determine how many have spawned (we used numSpawned) and an end condition that determines the number to spawn (we went with 10).

Make your changes so that you spawn a large number of items with a while loop and then play the game. If you have trouble figuring out how to get the while loop working, you can check Figure 9.

With that you should have randomly spawning items. Feel free to make use of multiple while loops in order to specify multiple islands for the items to spawn on, or even try to break the chunk of code out into a more general function.

2.8 Reward: Magic Sword

Most of the time an NPC in an adventure game gives some reward for completing a quest. For example, a brand new sword. However we'll have to actually *make* this new sword before we can have the NPC give it to the player. This is going to be pretty complicated as it'll involve messing with more of the Unity interface than just scripts. So we'll break it into multiple sections.

2.9 Making a New Script

First off, we'll make a script to handle the behavior of the magic sword. To make a new script return to the Unity window. Find the "Items" folder inside the "Codebase" folder. Right click it (or Control-Click if you don't have right clicking) mouse over "Create" then click "C# Script" when it is highlighted. Then a script file will appear with the default name "NewBehaviourScript" rename it "MagicSword" then press enter. You've now created the MagicSword.cs file! If you find you can't right-click on the Items folder you can also hit the "Create" button right at the top of the "Project" tab then follow the rest of the steps, after which you'll need to draw MagicSword into the Items folder (this isn't strictly necessary, but helps with organization). Double click your new MagicSword.cs file to open it in MonoDevelop.

If you accidentally create a script named NewBehaviourScript.cs (it happens) either delete it and try again or rename both the script file name (in Unity) and the main class name to MagicSword (in MonoDevelop).

Figure 10 shows the default appearance of MagicSword.cs (and all new Unity scripts, actually). Now we could try to build-up the behavior of MagicSword from scratch, but there's a very similar script that we can have it inherit from. Can you think what it might be?

The answer is to have MagicSword inherit from Sword! To do that we'll just need to change MagicSword to inherit from Sword instead of MonoBehavior. So just replace MonoBehavior with Sword in the class definition "public class MagicSword : MonoBehaviour". We'll do more in MagicSword.cs later to make it go beyond just the behavior of Sword, but this is actually sufficient to have it act exactly as Sword does!

2.10 Making the New Model

Every object in the game we've dealt with so far has had a 3D model associated with it, from the NPCs to the enemies to the cats. In order to have our magic sword in the game, we'll want to have one of those too. Once again the easiest thing to do is to base it off of what's in Sword now. So let's do that!

Go into the project view and find the sword "model" in the folders GameAssets then Item then Prefabs. Highlight the file named "Sword" and then go up to the Edit menu and click "Duplicate". (Alternatively use command-D or control-D to duplicate the sword model). Once it's duplicated you should see a new item called "Sword 1". Click on it, the "Inspector" view will now change as well. We'll want to make a few changes here.

First let's rename the object to do that with "Sword 1" highlighted in the project view, click the box with "Sword 1" in the Inspector view. Click in this box, which will highlight the entire text then write the new name (we went with MagicSword).

Next we still have the Sword.cs script attached to this model instead of the MagicSword.cs script we just made. In the inspector view you should see a box with the words "Sword (Script)"

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class MagicSword : MonoBehaviour {
5
6     // Use this for initialization
7     void Start () {
8
9     }
10
11    // Update is called once per frame
12    void Update () {
13
14    }
15 }
```

Figure 10: The default appearance of MagicSword.cs.

in bold. Click the gear in the top right of this box and select "Remove Component" from the drop down menu.

Next we'll need to Add the MagicSword.cs script to this model. Click the large "Add Component" button. Begin to type in "MagicSword" and Unity should find our MagicSword.cs script. Select it.

Last we need to add some info to MagicSword.cs, we need to add the Item Name (we went with Magic Sword, this will later be used for spawning it) and we need to click the "Auto Replace" toggle. When you're done, the Inspector view should look something like Figure 11.

You might be wondering what this whole bit was about. Well Unity uses what's referred to as a "components-based" system. That means that everything in the game world is made up of "components" these components include scripts (they handle behaviors and actions), and the 3D models that you actually see. Filling in the "Item Name" and "Auto Replace" were actually both public variables from Item! Feel free to check for yourself by opening Item.cs. Because they are public Unity can "see" the variables, and presents them to the user. Item Name is a string value, which is why you inputted text for it, and Auto Replace is a boolean value, which is why you had a check-box for it (true or false).

2.11 Making use of your Magic Sword

First you set up the MagicSword.cs script, then you set up the model and added the script. Let's call this combined entity an "object". Now that we have this combined object let's go ahead and put it into the game world. Select the MagicSword object in the Project View and then just click and drag it into the "Hierarchy" view. You should see a sword appear in the "Scene" view.

Play the game and see what happens!

You should see that the player seems to start with the MagicSword (which at the moment looks exactly like the Sword). That's because the player starts in the same position as where the MagicSword object is located, and because "Auto Replace" is set to true, the player automatically "holds" it.

We could change that! In the Hierarchy view click on "MagicSword" then in the "Inspector" view find the little box labeled "Transform". Find the row labelled "Position" and change the X value from 0 to 10. When you're done the inspector view should look like Figure 12.

Play the game and see the difference. You should now see two swords, the regular sword spawned by AdventureGame.cs and the MagicSword you placed yourself (this one may be sticking out of the ground depending on the placement of your islands).

You can now go over and pick up your MagicSword and swing it just like a normal sword! In fact the MagicSword acts exactly like a Sword as MagicSword.cs inherits from Sword.cs and there are no changes made. The only way it's different in behavior is that the NPC won't act like you have a sword if you pick up the magic sword. That's because the check in FirstQuest uses the item name "Sword" and the MagicSword has an item name "Magic Sword".

We should change the behavior of MagicSword so that it actually behaves more "magically" than a regular Sword. But first let's make the Magic Sword look a bit more unique than just a regular sword.

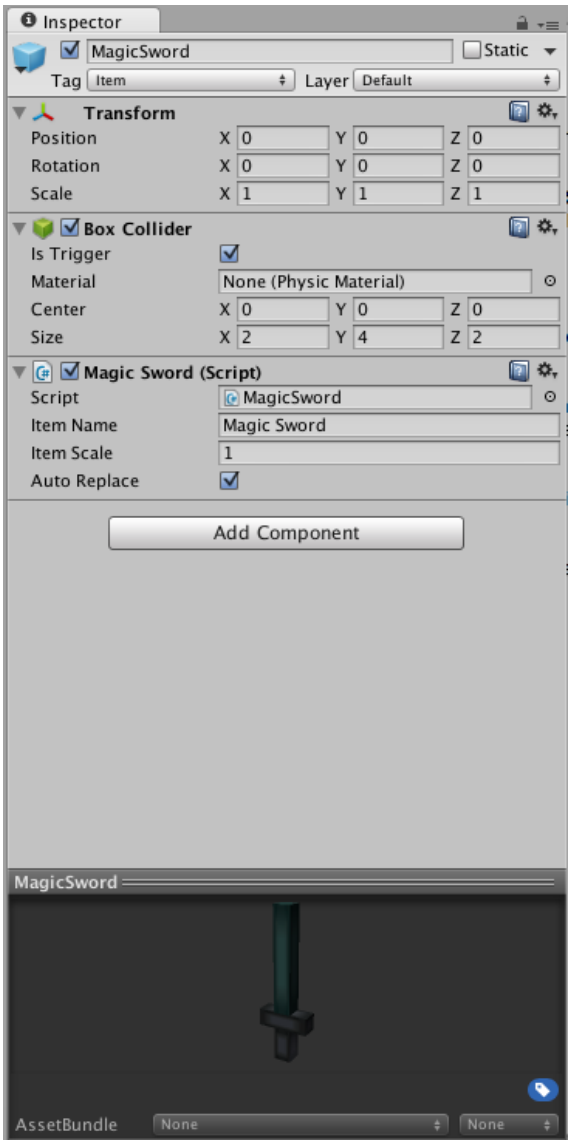

Figure 11: The model MagicSword with the magic sword script (MagicSword.cs) attached.

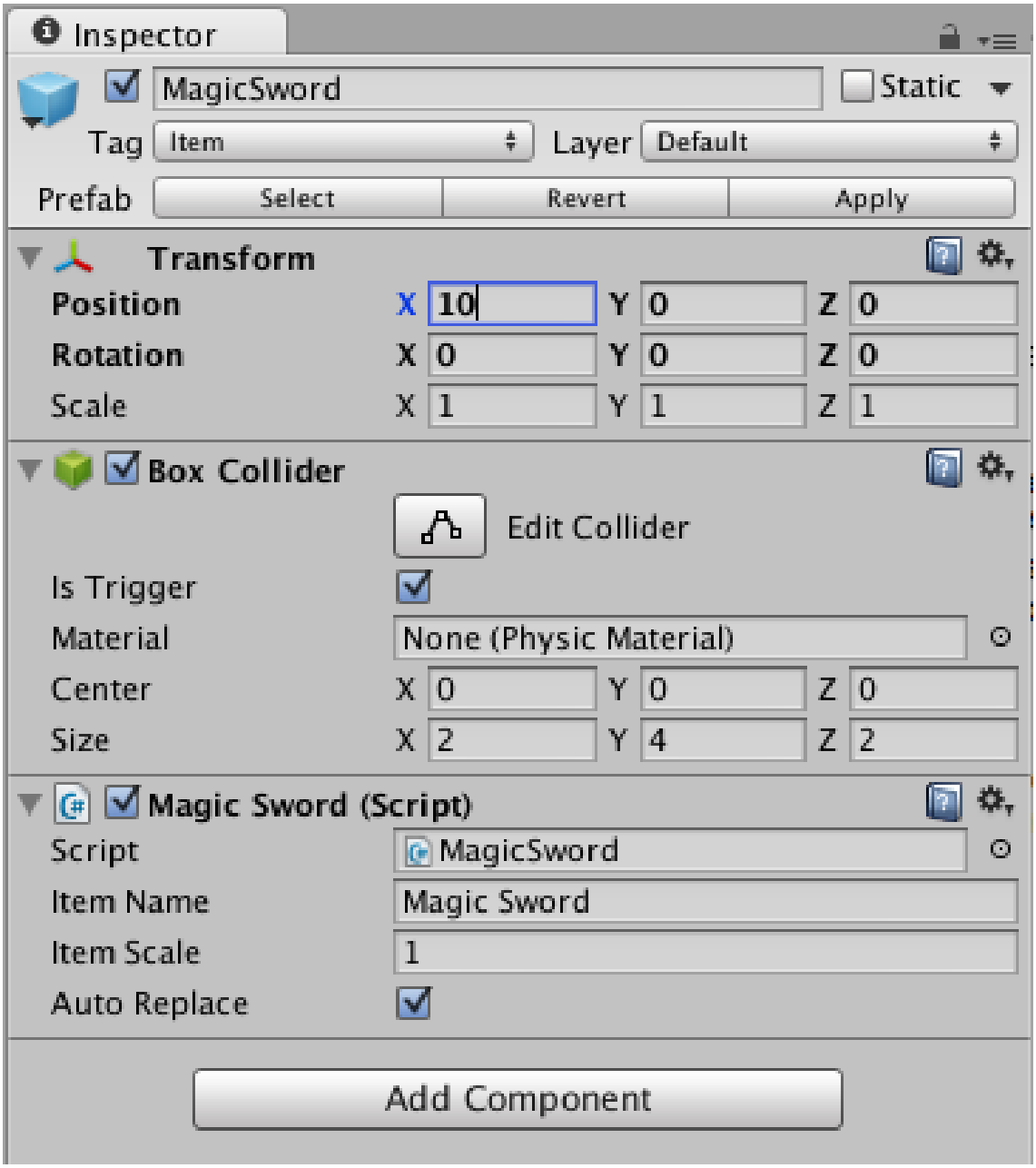

Figure 12: The inspector view of the MagicSword object

2.12 The Magic Sword's Appearance

In 3D game engines 3D models (like the "Sword" model) have what's referred to as a "texture" on them to determine what colors are used on the model. In Unity textures are placed onto a model via the use of "materials". So if we want to make our Magic Sword look different, it'll need to use different materials than what the sword model we duplicated was using.

First let's see what materials the MagicSword model is using right now. Go to the Hierarchy View and click the MagicSword object. Click the tiny triangle next to it, and then the tiny triangle next to the "Hilt" part that appears below it. Click on the "Hilt", "Blade", or "SwordHilt" parts and the Inspector will change showing information on this part that's within the larger MagicSword model. Within the Inspector view you should see that "Hilt" and "SwordHilt" both have the material "Iron" on them and "Blade" has the material "Diamond" on it.

In order to change this we'll need to make use of other materials. Look back at the Project view and in the "Items" folder find the Materials folder and open it up. You should see many materials, all of those you can put onto the different NPC parts and more! Click and drag the materials of your choice onto the various parts of the MagicSword. If you can no longer see it in the "Scene" view due to it's new position feel free to move it back to (0,0,0) instead of (10, 0, 0).

Regardless your best way to see the difference is just to play the game. Feel free to do that and check what your MagicSword looks like up close.

Once you're happy with your sword (ours for example ended up looking like Figure 13) the only thing left is to add some magic to this magic sword.

2.13 Adding Magic to the Magic Sword

By magic we mean behavior beyond what a normal sword can do. There are a variety of options here, and we'll walk you through a couple of them. Feel free to pick and choose whatever elements you like. First though, you might want to re-add your line in AdventureGame to npcManager.SpawnEnemies so that you have enemies to actually test the magic sword out on.

After doing that let's open Sword.cs back up, and see what we're working with so far. Remember, because MagicSword.cs inherits from Sword.cs and has nothing in it Unity is just calling the functions of Sword.cs.

Depending on what you chose to do in Lesson 3, your Sword.cs will look more or less like Figure 14. But regardless it'll be the case that Sword.cs has two functions: "Use" (a function that it inherits and overrides from it's parent Item.cs) and "HitEnemy" (a function that is defined in Sword and is virtual meaning it's set up to be overridden).

The first thing you want to do is probably get rid of the chunk of code in "HitEnemy" that causes the game to end. We probably don't want that to be the case any more! We'll want a Quest to end the game.

Next, we need to determine how the contents of Sword.cs impact what we do in MagicSword. Essentially, we have to decide which of the two functions we want to override. Remember when each is called. Use is called anytime the player clicks while "wielding" the item. HitEnemy is called anytime the item use "hit" an enemy. Because of that, if we want to make MagicSword have a special effect, we probably want to override HitEnemy.

Let's go ahead and return to MagicSword.cs. Begin typing the following and MonoDevelop should auto-finish it for you (when you see HitEnemy come up as an option to override, click enter/return). If not just type the whole thing:

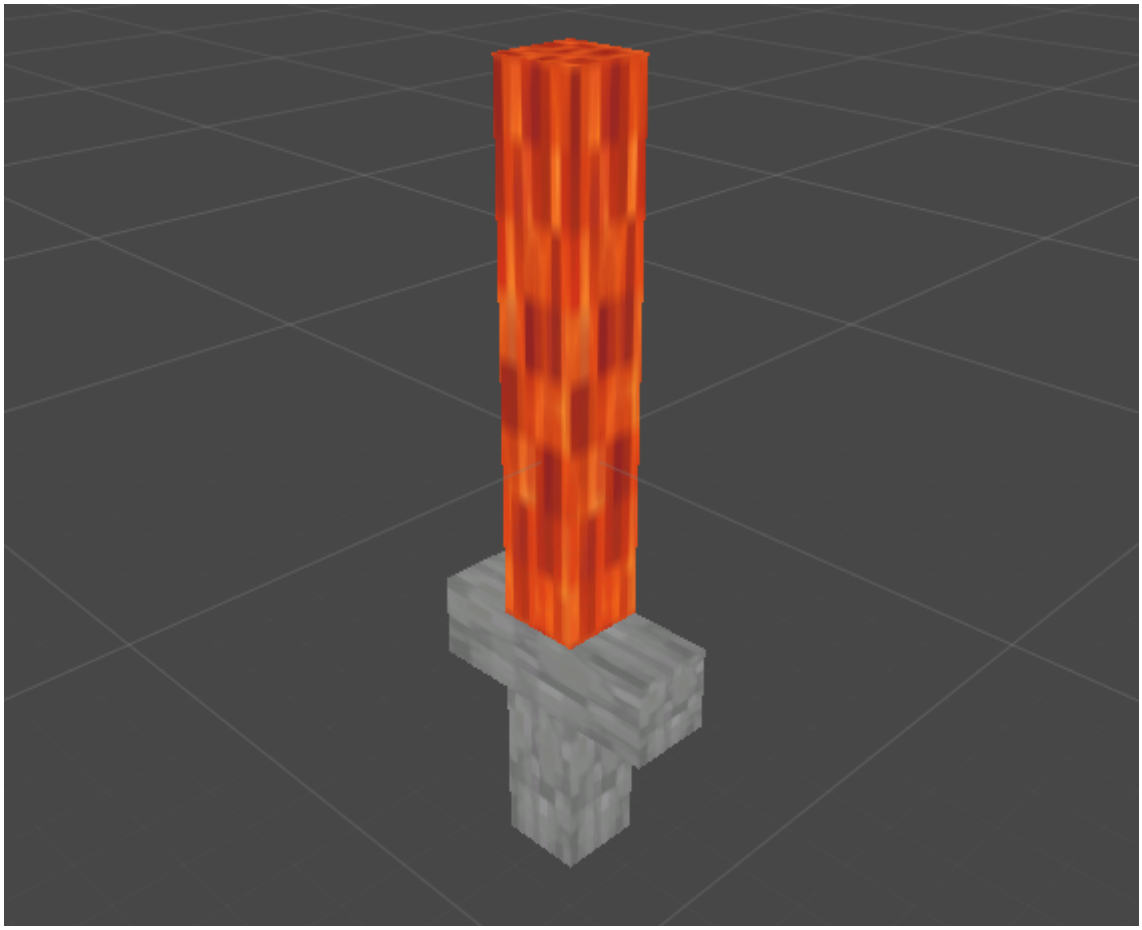

Figure 13: Our final MagicSword.

```
1 using UnityEngine;
2
3 public class Sword : Item {
4     int enemyCount = 0;
5
6     //This method is called when the sword hits an enemy
7     public virtual void HitEnemy(Enemy enemy){
8         Destroy (enemy.gameObject);
9
10        //TODO; Add code here for handling the win condition
11        if (enemyCount > 9) {
12            GameManager.EndGame("You Win!");
13        }
14    }
15
16    //This function is called when the sword is held and the "destroy" button is hit.
17    public override bool Use (GameObject hitObject){
18        Enemy enemy = hitObject.GetComponent<Enemy> ();
19
20        //Is the thing we hit an enemy?
21        if (enemy != null) {
22            //If it is, call HitEnemy and return true
23            HitEnemy(enemy);
24            return true;
25        }
26
27        //return false if we did not hit an enemy, as the sword was not "used"
28        return false;
29    }
30 }
```

Figure 14: Sword.cs, which holds the behavior of the MagicSword object at the moment

```
public override void HitEnemy (Enemy enemy){
  base.HitEnemy (enemy);
}
```

Recall that "base" is a reference back to the parent of the current class, so in effect "base.HitEnemy(enemy)" is calling the function "HitEnemy" in Sword.cs. For now, let's keep it. This will have the effect of auto-killing (if your code looks like ours as in Figure 14) any hit enemy. But we probably want to add something *more* than that given that this is a *magic* sword.

You've got some options here, and they both have to do with using new functions from GameManager. Namely:

1. GameManager.GetClosestEnemy: Takes in a GameObject and returns the closest enemy to the passed in GameObject.

2. GameManager.DestroyAllEnemies: Takes in no arguments and simply destroys all enemies.

Let's try that second one out first! Just add "GameManager.DestroyAllEnemies();" underneath the call to base.HitEnemy().

Play the game and hit a single enemy with the Magic Sword. See what happens!

That's pretty effective! However, it pretty much destroys any challenge in the game. You could use the first function mentioned, calling it with "GameManager.GetClosestEnemy(gameObject)" to return the closest Enemy to the Sword and then look at the code in Sword.cs to figure out how to destroy *just* that one. But we leave that for you! We recommend removing the line to wipe out all enemies though, as it'll ruin the "final fight" of lesson 6.

As an alternative to mass extinction, or getting "GameManager.GetClosestEnemy" working. We could also change the *aesthetic* impact of using the sword. In games, how an item feels to use is just as important as what it does. What if when the sword hit an enemy it created an explosion? Or sparks? You can make this happen with the function call "ParticleManager.CreateEffect("effect name", enemy.gameObject);" in MagicSword's HitEnemy function. Replace "effect name" with "Explosion", "Sparks", or "Smoke" to see what we mean! You can even mix and match the effects.

Play with the code, both its actual effect and it's aesthetic effect. What feels better? What leads to a better experience?

2.14 Hooking in the Sword

Recall that we originally wanted the MagicSword to be the reward for SecondQuest.cs. But there's a problem there as right now the MagicSword is just hanging out in space from the start so the player doesn't have to do the quest to get it.

We could fix that, remember that you created coins or cats with "ItemHandler.SpawnItemFromString", but there's just one problem. ItemHandler doesn't have a reference to the MagicSword object since we just finished making it. You can try to go ahead and use this function, but it won't work. Go to AdventureGame.cs, and try "ItemHandler.SpawnItemFromString("Magic Sword", 0, 1, 0);". The game will error out.

The problem is scripts can't automatically see all the objects in the Unity project. To be able to spawn versions of them (like all the coins/cats) they need to have a variable that stores that value, a reference. Happily, that's pretty easy to do. First, head back to the Unity window.

We'll want to get rid of the MagicSword object currently hanging out in the game world at the start. But first let's save the changes we've made to the MagicSword object! In the Hierarchy view

select MagicSword, now in the Inspector View hit the "Apply" button in the top right. That'll make sure all your changes are saved for this item. Check that worked by finding the MagicSword object in the Project view. The preview on the bottom of the Inspector View when it is selected should now look like the Sword you made.

If that matches up, delete the MagicSword object in the Hierarchy. That'll get rid of the MagicSword just hanging around in the game on start. You can delete it by right clicking the MagicSword in the Hierarchy View and selecting "Delete". Or, while it is selected, hitting Edit¿Delete. NOTE: Make *sure* not to delete the MagicSword in the Project View, as that'll get rid of the MagicSword in your project completely and you'll have to start from scratch. A good way to note is if you see a "Are you sure?" Dialogue Box, that's the *wrong* one.

Play the game real quick just to make sure there's not a Magic Sword just floating there.

Now that, that's taken care of find the "Player" object in the Project View and select it. Scroll down in the Inspector View when that changes and you should see the "ItemHandler" box. That's the ItemHandler.cs you've been using! In fact all the scripts used in the game have to be attached to objects. Now, you should see a list called "Spawnable Items" that has Coin, Cat, Key, and Sword in it. That's where we want to put our MagicSword. From the Project View click and drag the MagicSword object onto the Spawnable Items texts. If it turns bold and you see MagicSword join the list, you did it!

MagicSword is now spawnable! Just go to SecondQuest.cs and to the "QuestEnd" function add a call to: "ItemHandler.SpawnItemFromString("Magic Sword", 0, 1, 0);". You may also want to change the text to acknowledge the gift. Up to you!

Play the game, collect the coins/cats, and receive your reward!

2.15 OPTIONAL: Using your Reward

Of course it's a bit boring that there night be any way to test out your new reward, assuming you've already taken out all the ghosts. You can actually use a static function in GameManager.cs to spawn more ghosts to take out when you get the sword. The function is called "CreateEnemies" and it takes a number of enemies to spawn, an x value, a y value, a z value, and a range value. Just like "SpawnEnemies" in AdventureGame.cs! You'll call it with the line "GameManager.CreateEnemies(..." and then filled in with all of your correct values. Can you figure out where to place it and what values to use? Play around with it! Maybe even have the NPC say some final text that corresponds to the appearance of more ghosts as well.

3 End of Lesson 5

That's all we have for Lesson Five! Feel free to play around with the Quest dialogue, maybe even add another Quest with coins/cats, or add a few more NPCs. It's up to you! When you're done, why not show it to someone else and see if they like your tiny adventure game? In this lesson you learned how to:

1. Use inheritance to your advantage with Quest and MagicSword
2. Make new scripts
3. Make npcs

4. Make a sword

In the next and final lesson you'll have options for various different games of your own to make, using all the skills you've learned so far. You can think of it as the "Final Boss". Plus you'll actually be able to make use of your reward!

3.1 Lesson Five Glossary

Vocabulary Word	Definition	Examples
reference	n programming, a reference is a way of referring to a variable that allows us to interact with some object (such as an NPC).	myNPC, myQuest
protected	A simple kind of loop that repeats a certain chunk of code *while* some condition is true.	[Not Appropriate]
virtual	A keyword that allows for a function in one class to be overwritten in the children of that class.	[Not Appropriate]
override	A keyword that specifies that the function in question overrides the behavior of the "inherited" function.	[Not Appropriate]
base	A keyword that refers to the "parent" of the current class.	base.QuestStart();

3.2 List of NPC Materials

String Value	Description
"Grass"	The original block used in AdventureGameGenerator. Appears to have grass on top with a dirt bottom.
"Stone"	A grayish block made of stone.
"Leaves"	A green block that appears to be entirely made of leaves.
"Trunk1"	A dark brown material resembling a tree-trunk. Its top and bottom look like the inside of a tree, while its outside looks like bark.
"Trunk2"	A darker brown version of the "Trunk" material.
"Trunk3"	A darker brown version of the "Trunk2" material.
"Pumpkin"	A material that looks like an uncarved pumpkin.
"Lava"	A material that appears to be made of unmoving lava.
"Sand"	A material that appears to be constructed from sand.
"Gold"	A material that appears to be made of pure gold.
"Diamond"	A material that appears to be made of a blue gem.
"Iron"	A material that appears to be made of iron.
"Water"	A material that appears to be made of water.
"Wood"	A material that appears to be made of processed timber.
"Snow"	A material like that used in Lesson 1 that appears to be made of snow.