MineCode

Mark Riedl and Matthew Guzdial

Georgia Institute of Technology

March 29, 2017

1 Introduction

MineCode is a set of seven lessons designed to teach basic programming concepts through the process of building and fixing existing computer games. It covers—in abbreviated form—the essentials of computer programming that one might learn in a semester-long university course. It is not a substitute for a university programming course, but will give hands on experience with programming concepts that will bootstrap further self-learning. Our philosophy is to allow the learner to jump in and immediately start fixing and making games. Computer games are necessarily complicated. Whereas many programing courses require a degree of slogging through abstract concepts before putting them to good use in projects that are not trivial and divorced from real things, our goal is to get readers hands dirty with computer game code as quickly as possible.

Initially, this will involve *fixing* computer games. By fixing games, we do not mean taking code that doesn't work at all and making it work. Instead, we provide fully functional, simple games that just aren't that much fun to play and ask the learner to make them more interesting. We will provide simple computer games that are not quite right and introduce concepts necessary to make these games better. For example, one of the first concepts a beginning programmer needs to know about are *variables*, chunks of computer memory that can store values for use later on. These values are used by computer programs in various ways. We present these concepts so that the learner can immediately see the effect of variables on game play. We also strive to provide opportunities for learners to express their creativity. Instead of having one right answer, we will encourage learners to explore programming concepts to make the games uniquely their own.

Later, we will remove the crutches and more and more of the game will be built by the learner. Indeed, as early as Lesson 3, the learner will embark on a series of exercises that build off each other toward a single, complete game that leaves much up to the learner.

In addition to exposing learners to the basic programming principles and concepts, the end of the final lesson results in a set of code, mostly written by the learner, that can form the basis for future new games. This code can be easily modified to make first person shooters, role-playing games, or other types of games. We hope that learners will feel they have something they can continue to build on, experiment with, and express creativity through. Our goal is to not leave the learner with the friction of having to start over after one set of lessons is complete.

Why did we call our lessons *MineCode*? The name is an attempt to capture the fact that we are borrowing the look and feel of *MineCraft* while also allowing capturing the fact that game that you make by the end of the lessons will be *yours*—something you made and potentially unique from the games that other people make while learning to code with these lessons.

2 The Game Engine

A *game engine* is a codebase that supports the creation of new games. *Unity3D* is a popular game engine that strives to allow independent creators make high-quality computer games. It hides the gory details of 3D graphical rendering, physics, etc. under the hood and allows coders to focus on what the game is about.

MineCode provides an additional layer of game engine code on top of Unity3D that emulates the aesthetic and much of the functionality of *Minecraft*. Why *Minecraft*? It is arguably one of the most recognized games in the world. Furthermore, *Minecraft* is recognized as a game that makes it easy to build and create within the game world. But *Minecraft* doesn't make it easy to make new

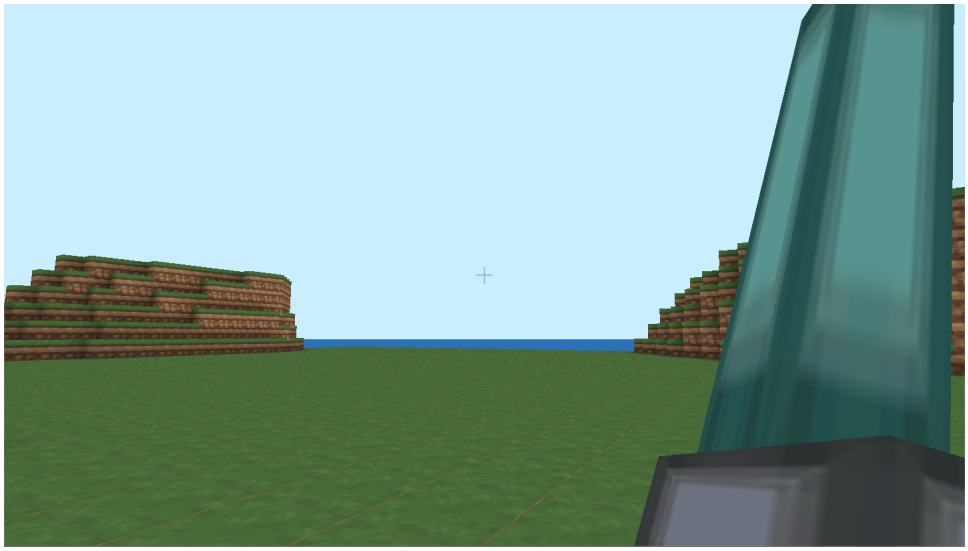

Figure 1: *MineCode* borrows the look and feel of *Minecraft*.

games that aren't *Minecraft*. To turn *Minecraft* into a new game, one must get the source code and modify it. The art of modifying existing games is called *modding*. That is possible; the source code is available, but then one must figure out how it works and how to change it without breaking it. By building our own game engine that looks and feels like *Minecraft* on top of *Unity3D*, we provide source code to the learner that is easier to modify. Further, since the source code is built on top of *Unity3D*, the learner can easily change the look and feel of the game engine so that it no longer resembles *Minecraft*. Therefore, while we are using *Minecraft* to provide a familiar environment for learning to code, we are not locking the learner into that environment.

3 Roadmap

Each lesson is designed to take 1-2 hours, plus give some suggestions on how to go farther and make something unique. The lessons cover the following programming topics.

1. **Variables and mathematic operators.** The variable is the most basic building block in computer programming. We will explore how small changes in variables can have big impacts on an existing game called *Snowman Attack*. The game works out of the box, but is "unwinnable". Learners will tweak the variables in the game to make it winnable and change the feel of the game to be more fun.

    - Introduction to modifying scripts in *Unity3D*.
    - Introduction to variables.
    - Using mathematical operations.
    - Making and using new variables.

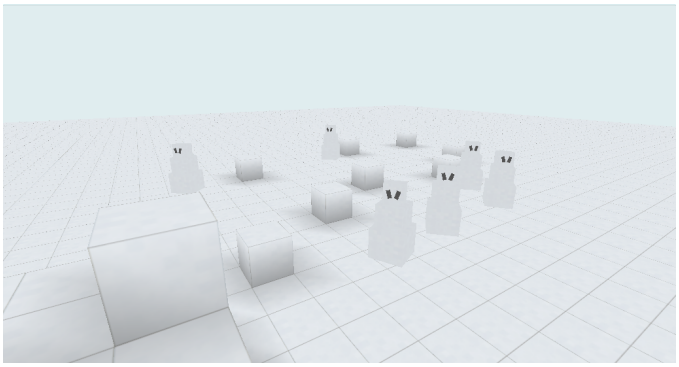

Figure 2: Snowman Attack.

2. **Conditionals.** The logic of any computer program is controlled by the ability to check the value of a variable and make a decision—to execute one chunk of code or another in response to some circumstance or player action. In this lesson, the learner has to implement code to make it possible for a player to escape a cave. The cave has a door, but it does nothing. The learner must make it so that a player can open the door, and then make it interesting to do so.

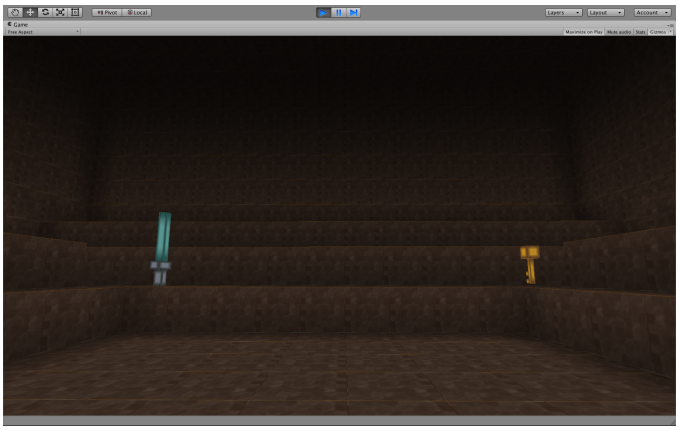

Figure 3: A sword and a key.

3. **Functions.** Functions are blocks of code that can be used over and over again. This lesson introduces learners to writing and using functions. It also gets into some particulars of the *Unity3D* game engine and how scripts work in the engine that we were able to previously gloss over. Once the learner understands how to use functions, a wealth of *Unity3D* functionality opens up to use. By the end of the lesson, the learner will have built a game in which players must fight an army of ghosts. There are numerous opportunities to customize the ghosts.

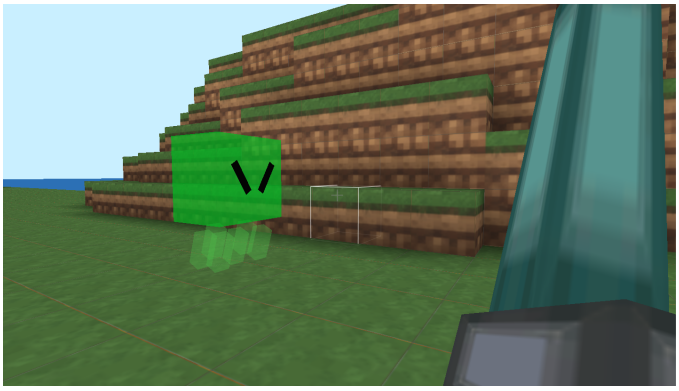

Figure 4: Fighting ghosts.

4. **Loops.** One of the things computers are really good at is doing the same thing over and over, maybe with small variations each time. This is true for games too. Loops are chunks of code that repeat over and over again. This lesson explores *procedural content generation*, the writing of code that automatically creates part of the game. In this lesson, the learner will write code that creates the landscape in the game world. If this were normal *Minecraft*, the player would build block by block. Instead, the learner will write the code that builds hills and trees.

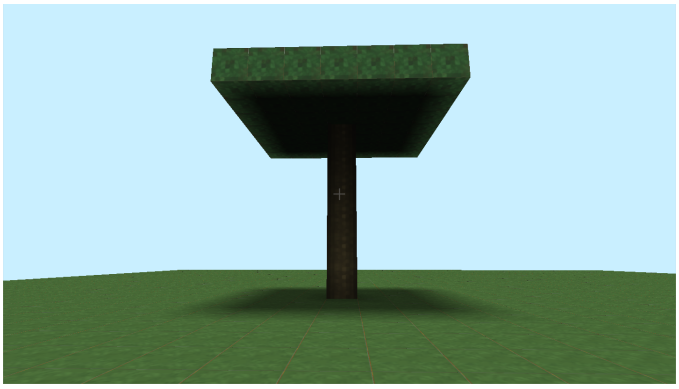

Figure 5: A procedurally generated tree.

5. **Object orientation.** If functions allow a programmer to use chunks of code over and over again, the *objects* allow programmers to use groups of functions over and over again. In this lesson, learners will learn about objects and inheritance, where new types of objects borrow functionality from old types of objects. Learners will make new types of non-player characters (NPCs) for their game and write the code that sends the player on quests. The player is going to need a magic sword; the learner will see how to take an existing sword object and make a new type of sword object with special, custom properties.

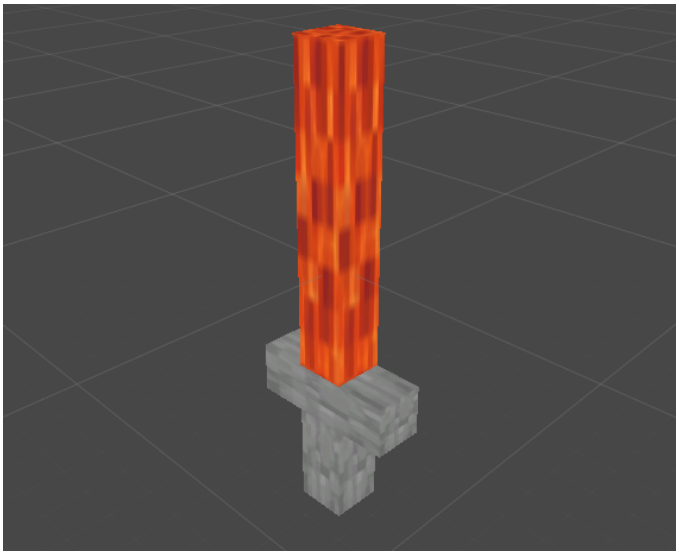

Figure 6: A magic sword.

6. **Practice!** Lesson 6 challenges the learner to pull a lot of the programming concepts together to make their game even better. Learners will program a Boss monster for the player to fight. The lesson also gets into the tools available as part of *Unity3D* to make game art.

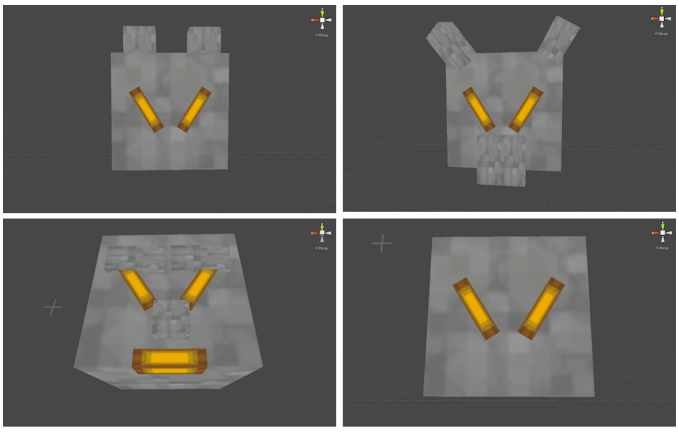

Figure 7: Boss monsters.

7. **Make your own game.** The final lesson provides some pointers on how to take the next steps and make a game that is unique. With the pointers in this lesson, one can make a tower defense game, a shoot-em-up game, a murder mystery game, or an adventure game.

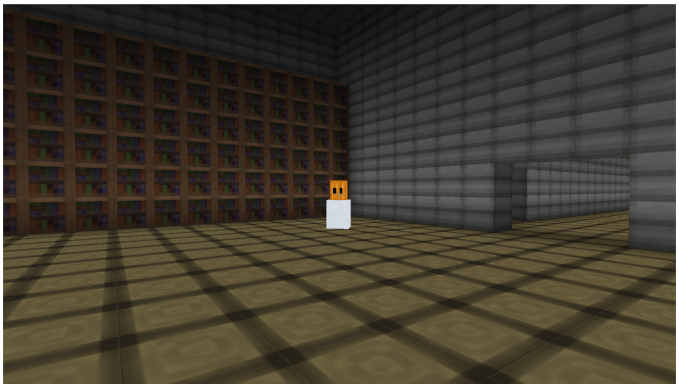

Figure 8: A non-player character in a murder mystery game.

4 Final Note

The following series of lessons introduce basic programming skills in a manner meant to engage and educate. We hope to dispel the potentially daunting appearance of programming via an interactive environment in which students can immediately see the impact of their work. While these lessons are in no way meant to replace an introductory course in computer science, they should benefit any student interested in computer science both in terms of hands-on programming knowledge and the confidence to approach new programming problems. We encourage any student to approach these lessons ready to learn and willing to flex their creativity, as we feel that is the way to get the most from them.

Lesson One

Snowmen and Variables

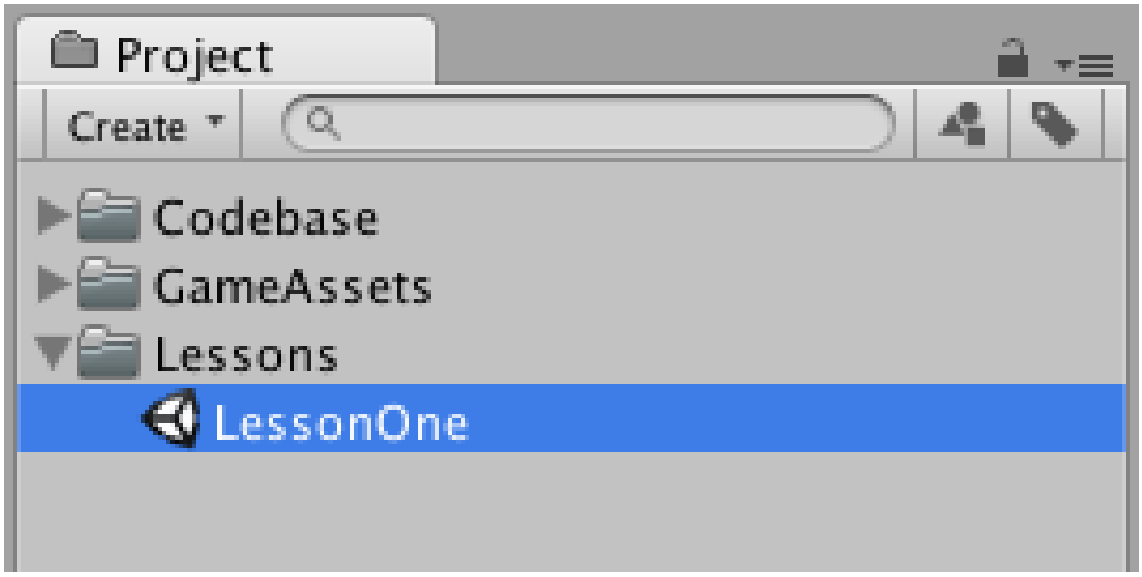

Figure 1: The lesson one "scene" file.

1 Introduction to Lesson One

Video games run on code. They share this trait with all applications on laptops, smartphones, and tablets. They all had to be written by a computer programmer (or more often several working together on a team). Learning to read and write code means that you can not only understand how these applications work, but can even make your own!

The following lessons will walk you through reading, understanding, and writing code. In the process, you'll also fix a series of broken video games and get the chance to make your own.

2 Lesson One: Snowmen and Variables

Definition!

VARIABLE:
The part of code that stores information for use.

Variables are the primary building blocks of all code. You may have run into variables in math class expressed as "x" or "y" and had to solve for them to determine their value. Variables take a far different form in code. They store information and due to the fact that they are "variable" (they can change), they can be used to make changes to the player, enemies, or world of a game.

To begin with open up the project in Unity. If you haven't yet downloaded Unity or the project onto your computer, take a moment to find the "instructions.txt" file that'll walk you through this process. Once Unity is opened, you'll want to click the "Open Other" button, and open the top-most folder of the downloaded project.

Once the project is open, In the "Project" window of Unity find the folder labelled "Lessons". Open the folder and then double click on the "LessonOne" file.

In this lesson we'll focus on fixing a game in which the player needs to stop angry snowmen from reaching a tower. The snowmen cannot be hurt, only stopped by building a wall to protect the tower.

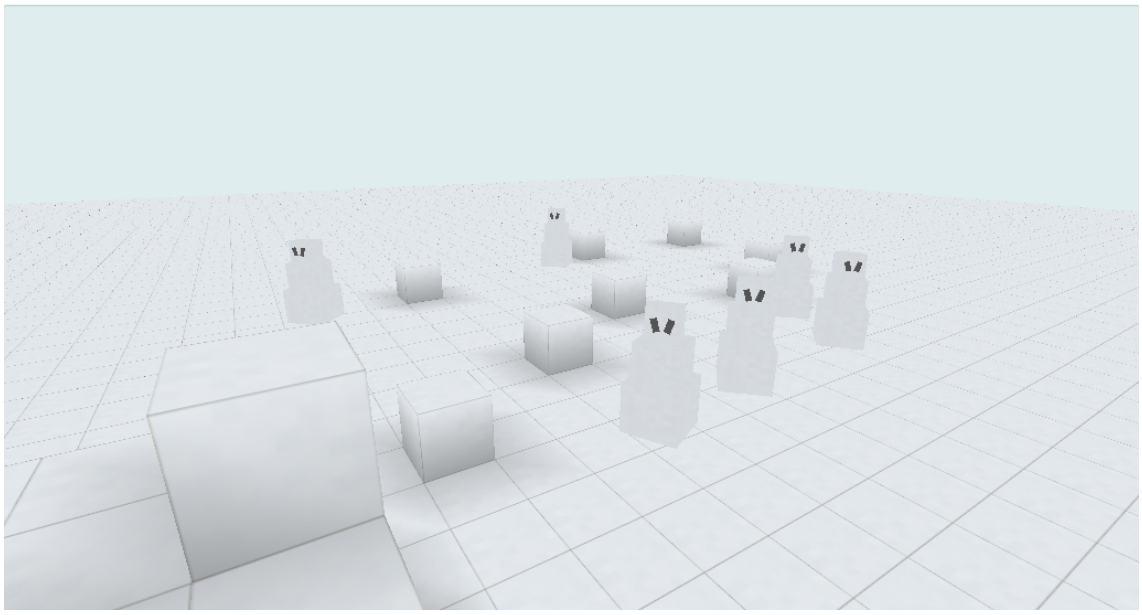

Figure 2: Attack of the Killer Snowmen.

First we'll need to determine how exactly the game is broken. You can start the game by pressing the "play" button near the top of the screen. The controls for the game are to use wasd to move, with mouse to look around. Left click with the mouse to remove blocks and right clock with the mouse to add a new "snow" block. Press space to jump. When the game is done, press escape or backspace to make your mouse cursor visible and click the "play" button again to stop it. (Note: If you can still see your mouse after pressing play you may have to click on the screen once after the game is loaded.)

Play the game, and try to build up a wall to stop the snowmen from reaching the tower as seen in Figure 3.

The problem with the game should be clear from just a single playthrough. The player cannot move fast enough to make a wall in time. The game is not winnable!

In order to resolve this problem, we'll need to make changes to some variables. For that we'll need to start looking at scripts. Scripts are files that store code. When the game engine runs it reads through the scripts in order to determine what to do. In this way they can be understood to be instructions for the game and everything that occurs within it. This means that in order to change something in the game (say the speed of the player) we'll need to change something in a script.

The script that we'll be looking at is called LessonOneGame.cs. You can open it from Unity in the Project View. Open up the "Codebase" folder, then the "Lesson1" folder. Double click on "LessonOneGame" to open it up. At this point, a new application called "MonoDevelop" will open that will actually let you modify the script.

Once MonoDevelop opens (this may take some time), you should see something that looks like Figure 4. (Note: you may get a message asking you to "convert line endings", go ahead and accept

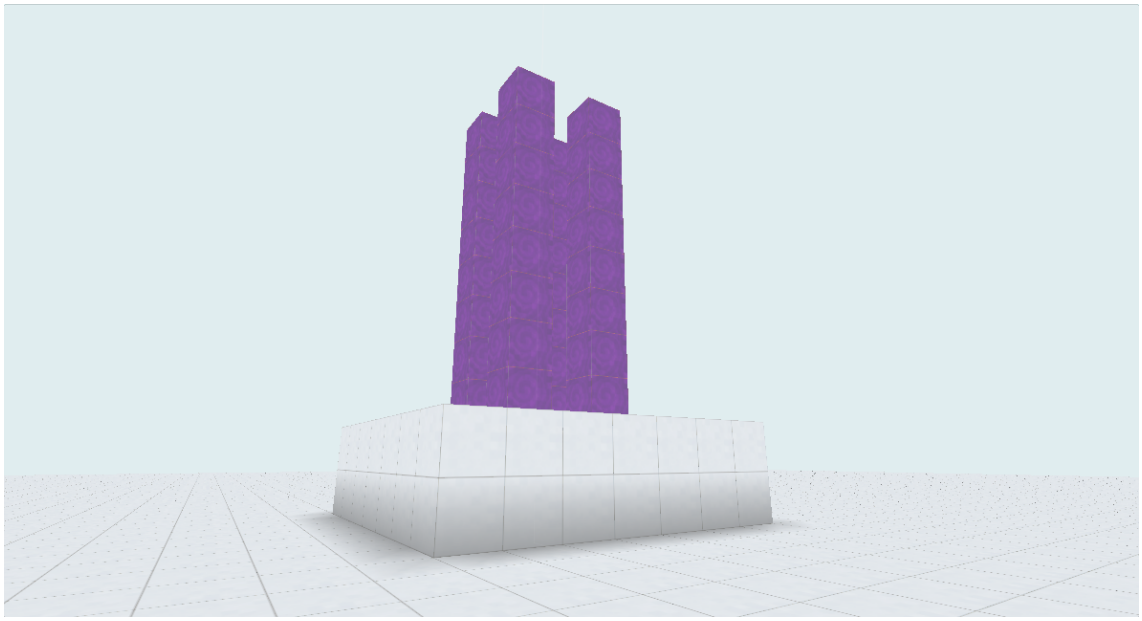

Figure 3: Protect the tower!

this. This is merely due to different system requirements.)

While even this simple game relies on many scripts, we'll be focusing on this script only for this section. Before we get much further let's go over a brief overview of the major sections of this script, and what exactly they're there for.

Definition!

COMMENT:
A part of code left by the code's author to explain what some code does in plain English. Starts with two forward slashes (//).

Definition!

KEYWORD:
An English word with a specialized purpose in code beyond its normal definition.

2.1 Overview of LessonOneGame

Throughout the script you'll see sections of plain English text that start with two forward-slashes (//). This is what's referred to as a **comment** Comments are sections of code written by a programmer in order to explain what's going on in a specific section of code. When the game runs, it completely ignores these comments, meaning they have no impact on the game.

At the top of the file you have a line that begins with the **keyword** "using". Keywords are generally English words that have a specific meaning in a programming language, in that they do or represent something beyond their general English meaning. For example, the keyword "using" allows us to import and reference code from other codebases. In this case, we need to import the "UnityEngine" codebase as that's the game engine that this script is used in.

Just after line 3 (the line starting with using) we get to the main section of this script: the definition of the class "LessonOneGame". The line itself may seem fairly confusing at first, but let's break it down. The first word (another keyword, you can tell since it's in teal as well) "public" means that this class can be referenced from other scripts in the game (as opposed to if we used the word "private" here). The second word **class** specifies that the thing we're doing on this line is defining a class. We'll get more into this in a later lesson, but for now it's enough to know that a class defines a chunk of code that share a similar purpose. The third word is "LessonOneGame", which is the name of this script and the class. The last section ": MonoBehaviour" means that

```
using UnityEngine;

public class LessonOneGame : MonoBehaviour {

    // This chunk of code (called a function) is run at the beginning of play time
    void Start () {
        //jumpValue determines the height of the player's jump
        float jumpValue = 1;
        //gravityValue determines how fast the player falls after a jump
        float gravityValue =20;
        //speedValue determines how fast the player moves
        float speedValue = 1;
        //npcSpeed determines the speed that the npcs move at
        float npcSpeed = 2;

        LessonOneGenerator.SetUpGame (jumpValue, gravityValue, speedValue, npcSpeed);
    }

    //This chunk of code (called a function) is run at every frame
    void Update () {

        //Check to see whether any of the enemies have reached the center
        if (LessonOneGenerator.PlayerLost()) {
            string lostString = "Player Lost";
            LessonOneGenerator.EndGame (lostString);
        }

        //Check to see whether all of the enemies cannot reach the center
        if (LessonOneGenerator.PlayerWon ()) {
            string wonString = "Player Won";
            LessonOneGenerator.EndGame (wonString);
        }
    }
}
```

Figure 4: The LessonOneGame.cs file opened in MonoDevelop.

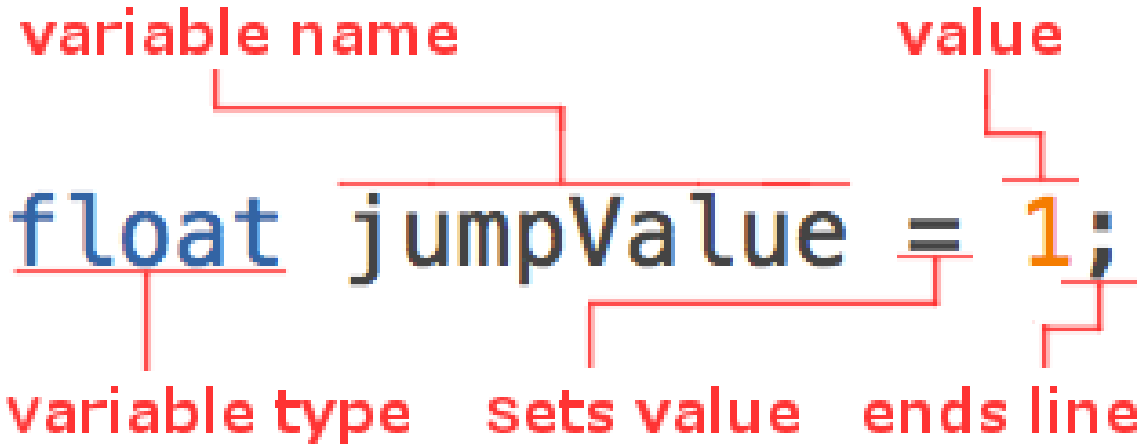

Figure 5: A breakdown of a variable definition line.

Definition!

CLASS:
A collection of code including both variables and functions with a similar purpose.

Definition!

FUNCTION:
A single section of code with a particular purpose, the equivalent of a paragraph in code.

this class named "LessonOneGame" takes some of its behaviors from the class "MonoBehavior". This process is called inheritance. You'll also notice that the line ends with a left curly bracket. If you click to the right of this curly bracket, you'll see a right curly bracket highlight on the last line of this script. Curly brackets opening and closing like this specify a chunk of related code, in this case a class. This is similar to how parenthesis surround a distinct thought in writing (Like this!).

Within the curly brackets of the class LessonOneGame we have two further chunks of code (held within curly brackets) referred to as **functions**. A function stores code that handles a particular activity. You can think of them as being like paragraphs for code with each line being a sentence. Like with classes, we'll talk more about these later, but for now all that's important to note is that the function "Start" runs at the beginning of the game just once, while "Update" runs every frame. Start handles setting up information about the game. Update handles checking whether or not the game should end and determines what text to display for the end screen. These two functions come from MonoBehavior, which specifies when the functions will be "called". We use the term "call" to refer to running each line of a function one by one from top to bottom. It's these two functions that you'll be editing through the entirety of this section.

2.2 Making the Game Winnable

In order to fix the broken game, we'll need to adjust variables. As we mentioned previously, variables are the building blocks of code. If we look at the function "Start" in "LessonOneGame" we can see four different *variable definitions*. A variable definition is a single line of code that creates a new variable, once created the variable can be used in later parts of the code. You can see a breakdown of one variable definition example in Figure 5. The word "float" indicates what kind of variable is being created and also specifies that this line is defining a new variable. We'll get into variable types in a few sections, but for now just note that we have variables named

"jumpValue", "gravityValue", "speedValue", and "npcSpeed".[1]

Since we know the major problem with the game seems to be how slow the player moves, it might make sense to try to raise the player's speed. All four of the definitions have an equals sign "=" and then a value after them. Unlike in math classes, this symbol doesn't mean two things have the same value, it actually stores the value on the right side into the variable on the left side. So the variable jumpValue currently stores the number 1, but we could change that to any other number we wanted to by changing the right side of the equals sign. We can use this fact to make the game winnable.

Based on the comments above each variable it looks like "speedValue" is our best bet. Try changing the number next to it from 1 to 5 and then playing the game again. WARNING: Don't forget to have a semi-colon (;) at the end of the line, so that it now reads:

```
float speedValue = 5;
```

Semi-colons are required to end individual lines of code as otherwise the game cannot determine where a line of code ends when reading a script. This will lead to an error and the game won't run till that error is resolved. Save the file after making this change and then open the Unity window.

Play the game by pressing the top play button on the Unity Window. As a quick note, while MonoDevelop does have a play button, it will not cause the game to play. Pressing that play button will lead to MonoDevelop trying to run the script as a standalone piece of code, which won't work well.

Now it's possible you *could* win the game. However, it's very unlikely that you can move fast enough to outrace the speedy snowmen. You might think that the issue is just that the player still isn't fast enough. That's possible! Why don't we try doubling the speed value from five to ten? The line should now read:

```
float speedValue = 10;
```

Try playing the game again before continuing.

Unfortunately, this increase of speed only makes the game much more difficult to control. Feel free to adjust speedValue more, trying an extreme value like 100 will make for an interesting but unplayable experience. We recommend setting the speedValue back to 5 for the remainder of this lesson, so that the line reads:

```
float speedValue = 5;
```

If adjusting the player's speed alone isn't enough to make the game winnable, then it makes sense that we should try adjusting another variable. We can determine which variable to adjust by thinking about what makes the game non-winnable: the snowmen can reach the tower before the player can build up a wall. To fix this, it makes sense that we should try to slow down the snowmen. Looking at the remaining variables it seems that "npcSpeed" is the variable we need to change. We'll start with an extreme change, let's make npcSpeed store 0.5.

However, if we just change the line to read:

```
float npcSpeed = 0.5;
```

[1]Note: the fact that all of these variable names have the first word uncapitalized, with the word afterward capitalized is referred to as "camel case". While its not required to name variables this way, it is a typical practice done by programmers to make it easier for readers to read the variable names.

Then we'll actually be unable to run the game! Make the change, save the script, and return to the Unity window. Look at the console (located on the bottom left of the Unity window; double click to open it) we'll see an error in red text that reads:

"Assets/Codebase/Lesson1/LessonOneGame.cs(20,23): error CS0664: Literal of type double cannot be implicitly converted to type 'float'. Add suffix 'f' to create a literal of this type". What does this mean?

Definition!

VARIABLE TYPE: An attribute of a variable, specified when it is defined, which determines what values can be stored into it.

Unfortunately we've just gotten our first type error (errors are problems with code that mean the game cannot run). All variables have a **type**, which dictates the kinds of values that can be stored in them. This can get confusing as there are actually multiple variable types for storing numbers.[2] For example all the four variables in this chunk of code are of the type "float", but including a decimal means that Unity thinks we're using another type called a "double". The major difference between a float and a double is that doubles are much more precise. For example, a float can hold values with up to 7 digits (0.3333333), while a double can hold values with up to 16 digits (0.3333333333333333). However, we generally don't need this level of precision.

In order to not confuse Unity into thinking we're using a double instead of a float we need to add a lower-case "f" to the end of the value we're storing in npcSpeed. This tells Unity that we want to create a float value instead of a double value, and then there's no problem storing this value into a float variable. The line should now read:

```
float npcSpeed = 0.5f;
```

Save the file and then return to the Unity window to play the game again.

This time it shouldn't be too difficult at this point to build up a wall to keep the snowmen away. Congratulations, you've made this game winnable! You might find that this speed of the snowmen is too slow for you, or maybe still a bit too fast. Feel free to adjust the value of the npcSpeed variable to adjust this. Just by changing this one variable you can directly impact the difficulty of the game.

Once you have a speed you're happy with feel free to move on to the next section of this lesson where we'll go beyond just making a game winnable and get into changing how the game feels to play.

2.3 Changing the "Game Feel"

The way it feels to control a virtual character in a video game is referred to as "game feel". In this section we'll cover making some additional changes that'll impact the way the player plays the game.

There are two float-type variables left in "Start" that we haven't yet considered. These being "jumpValue" and "gravityValue". As you might expect these impact the player's jumping and falling. One of the biggest complaints from video game players is that a a game's jump feels "floaty". In order to avoid that in the future let's see if we can replicate it here.

In order to turn a specific feeling ("floaty") into a gameplay mechanic it's useful to think about examples you might have seen from life. For example, you might have seen a gently floating feather, or bit of dust drifting through the air. We can then identify things that these "floaty" examples

[2] It may seem odd to have multiple types of variables to represent numbers, but its actually more efficient. Storing more information requires more memory, so it makes sense to use less "precise" variable types when we can. For example, if we know a variable will only hold a number between 1 and 100, there's no need to use a variable type that can store a wider range than that.

have in common and attempt to make the virtual character act more like them. The easiest of these shared attributes to implement is that both tend to fall very slowly. Let's see if we can make that happen in the game.

Of the two remaining variables, "gravityValue" makes the most sense in order to make the player fall less fast. Try altering the "gravityValue" to make the fall slower. Does it make sense to lower or raise the value? (Hint: You might have to make a large change before the effect is noticeable). Test out whether raising or lowering the value makes sense by making changes then testing how they impact the game.

Play the game and alter the gravityValue until it feels "floaty".

Depending on your ideas of floaty-ness, you might find that just changing the gravityValue doesn't seem to be enough. Especially given that the player character still moves at a relatively fast speed in the air when moving with wasd. To further floaty-ify the player, it might be necessary then to change speedValue as well. By adjusting "speedValue" as well you should be able to get a more satisfying floaty feeling. However, this will make the game impossible to win again, which is another reason to dislike floaty controls.

As mentioned above, most people who play video games don't like a floaty feeling with their controls. However, as with most rules, there is an exception. What if you wanted to set a video game on the moon, or some other place with low gravity? Making a player believe that a game takes place in a particular location takes more than just changing the way the game *looks*. The player character also needs to move as the player would expect for that location. You have to change how the game *feels*!

To demonstrate the importance of player controls to suggest a particular location, let's try transforming this Snowmen Blocking game into a Snowmen Blocking Game on the Moon. In order to make the player feel like they're on the moon we'll need to change both the gravity and the jump values. Make sure to reset the speedValue back to 5 (or even a bit higher!) as being on the moon wouldn't make one slower in terms of walking around speed. We won't give you exact values for gravityValue and jumpValue, play around with it! It is worth noting that jumpValue should be higher while gravityValue should be lower than its original value of 20.

Play the game to test your changes, and continue to make changes until you're happy with the feel.

Before too long you should find that you can bound around across the snowy landscape just like an astronaut on the moon. You'll also likely find that even though it doesn't impact the "gameplay" to any degree the game *feels* much more fun to play now. This altering of variables to get a specific feeling or behavior in a game is referred to as "tuning" and it's one of the most common tasks in game design.

2.4 Tuning and Relationships

In the process of tuning a game its sometimes useful to only change one variable, but keep a certain relationship between it and another variable. Let's say for example we wanted to change the speedValue variable from 5, but wanted to keep the current relationship between npcSpeed and speedValue (Where npcSpeed is 1/10 of speedValue at 0.5). Now you could make changes to speedValue and calculate what npcSpeed should be yourself, but that would get time consuming. Especially if you're making lots of little changes! (Say you wanted speedValue to be 4 then npcSpeed would be 0.4, or speedValue could be 4.5 then npcSpeed should be 0.45, and so on...). That doesn't sound like a lot of fun.

```
//speedValue determines how fast the player moves
float speedValue = 5;
//npcSpeed determines the speed that the npcs move at
float npcSpeed = speedValue;
```

Figure 6: Setting npcSpeed to speedValue.

Instead we can get the code to handle this for us! Remember that variables store specific values. However, they can also be used as if they *were* those values. For example, if we simply set npcSpeed to speedValue then npcSpeed will have the same value as speedValue, as seen in Figure 6. Trying to play the game now will be very difficult, as the snowmen will move at the same speed as the player.

Remember that our goal here is to have the same relationship between speedValue and npcSpeed; we want npcSpeed to be 1/10 of speedValue. We can do this very easily! Just change npcSpeed to be equal to "speedValue/10" so the line now should read:

```
float npcSpeed = speedValue/10;
```

You should now find that the game is now totally winnable! This is due to the fact that "/" in code is translated as a command to do division. So when Unity reads in the script it reads speedValue divided by 10 (or in this case 5 divided by 10 as 5 is stored in speedValue) which equals 0.5!

Try changing the value stored in speedValue, and you'll see that the relationship holds automatically. So setting speedValue to 4 will lead npcSpeed to equal 0.4 automatically and so forth. This is a powerful tool, as it not only allows tuning of two different variables to occur at once, but can even be used to change the relationship between two variables after the fact. So we can change "/10" to "/9" or whatever we like! Try changing this as well and see what kinds of different experiences you can make.

Division (using the "/" symbol) is not the only kind of mathematic operation that code can do for you automatically. We can use all of the "simple" mathematical operations with a single symbol. "+" tells the code to do addition, "-" tells the code to do subtraction, and "*" tells the code to do multiplication. See them all laid out with examples in the table below. We could have (for example) set npcSpeed with multiplication by changing the line to: "float npcSpeed = speedValue*0.1f;". Try using these different operations with the variables you have defined so far.

Name	Symbol	Example
Addition	+	float example = 4 + 5; (example equals 9)
Subtraction	-	float example = 4 - 5; (example equals -1)
Multiplication	*	float example = 4 * 5 ; (example equals 20)
Division	/	float example = 4 / 5 ; (example equals 0.8f)

2.5 Variable Types

So far we've dealt exclusively with variables that already existed, altering or "tuning" them towards different effects. In this section we'll go over creating and changing variable values with code instead of by hand.

Before we get too much further, we should take a moment to go over some of the most common variable types and what they store.

Type	English Name	Values Stored
uint	Unsigned Integer	0 to 4294967295
int	Signed Integer	-2,147,483,648 to 2,147,483,647
float	Float	-3.402823e38 to 3.402823e38
double	Double	-1.79769313486232e308 to 1.79769313486232e308
bool	Boolean	True or False
char	Character	Single text characters
string	String	Sequences of multiple text characters

The above table summarizes some of the most common primitive variable types. We classify these variable types as primitives via using a lower-case letter to define them in code. We've already mentioned "complex" variable types, though not by that name. Classes like LessonOneGame are complex variable types, and we'll explain those in detail in a later lesson.

Of these primitive variable types, you've already seen floats and interacted with doubles to some degree. There are two other types of variables that store number information here, signed and unsigned integers. You can think of an integer as being equivalent to a whole number in mathematics. So numbers that don't have decimal points that can be stored into integers. They're useful when it comes to counting things, where you can't just have "part" of something. Whether or not its signed merely indicates whether there can be positive or negative number values for them.

In the table we're also introducing three new types of variables entirely. First of all we have boolean values, which store true and false information. We'll come back to those in the next lesson. The last two types have to do with text information. We have characters, which can store a single text value ('a', 'b', 'c') and strings which can store sequences of text characters ("abc"). We have two examples of strings already in the script we've been working in, which can give you a sense of what these variable definitions look like. You'll note that just like how we had to add an f to the end of a float variable declaration in order to specify that it was a float, we specify that text is a string by surrounding it with quotation marks "like so". We have to do this as we technically use text characters throughout the code, including in variable names, so we need a way to differentiate when the text is a string.

2.6 Creating a Variable

Now that we've covered the most common types of primitive variables let's set about creating a new variable. To start with, it's possible individuals may want to know how long it took them to win or lose the game. To do that, we'll need to track how long the game is played for. Let's start by defining a new float variable to track the time. Instead of Start, try defining the new variable in Update. Let's use the name "timeSpent" and set it to 0 to begin with. You'll want to put the variable definition of the beginning of the Update function, something like Figure 7. As a reminder, variable definitions look like Figure 5. The first word ("float") determines the type of variable, and tells Unity that the line is creating a new variable. The second word ("timeSpent") is the name of the variable, and the way it can referred to later. The equals sign ("=") tells Unity to set the variable's value to it's right side ("0"). Then a semi-colon ends the line.

First ensure that the game still runs (you don't have to play it all the way through).

This will ensure that Unity can still read the script and that there have been no errors introduced. Its generally a good idea to ensure that the game still runs after every major change, as a

```
//This chunk of code (called a function) is run at every frame
void Update () {
    float timeSpent = 0;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost";
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won";
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 7: Update after adding timeSpent.

means of ensuring that the change didn't break anything. It's worth noting at this stage that you may see a warning pop up in the Unity console (bottom left corner of the Unity window) with the text:

"Assets/Codebase/Lesson1/LessonOneGame.cs(21,23): warning CS0219: The variable 'timeSpent' is assigned but its value is never used".

This is not an error, and won't stop you from playing the game. Warnings appear in the console for issues in the code that still allow it to run, such as defining a variable that is never used.

Now that you've confirmed that the game still runs we have two problems.

1. While we now have a variable to track the time spent playing the game, we don't have any means of displaying this information to the player.

2. The value of timeSpent will never change, meaning it won't actually hold the amount of time the player plays the game.

2.7 Displaying a Variable

We'll tackle the first problem first. We mentioned in a previous section that the "+" symbol tells the code to use addition, but it can actually be used for more than that. In certain situations the "+" symbol can be used to combine values of two different types. In this way we can turn a float (like our timeSpent variable) into a string (like our wonString variable). This can be accomplished in a number of ways, and we'll demonstrate one of them here. For example if we change the wonString line to read:

```
string wonString = "Player Won! Time Spent: "+timeSpent;
```

```
//This chunk of code (called a function) is run at every frame
void Update () {
    float timeSpent = 0;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 8: Update after changing the end of game messages.

Then timeSpent will be converted to a string and combined with "Player Won. Time Spent: " automatically. Now this is a really powerful and handy thing that programming languages can do, but it only works in specific cases and between certain types.

For example, you can convert numbers to strings but not strings to numbers in this way. Go ahead and change both the win and lose strings to include timeSpent. In the end your code should look like Figure 8. The text within the double quotation marks can be whatever you like, try writing your own win and lose messages with timeSpent included. Just try not to be too mean to the player, or they won't play again.

Play the game until you either win or lose and you'll see your new message appear! So that solves one problem, but we still have a fairly large one in that the value of timeSpent doesn't change from the initial value of 0 we gave it.

The solution to this problem is a more complex one, but we'll break it down into a number of smaller steps.

2.8 Updating Variables with Code

In order to have the timeSpent variable change from its initial value we'll need to talk about updating variables. So far, we've made use of the values in variables after defining them (in the Game Feel section), but haven't changed those values after defining them. As we mentioned at the very beginning of this unit, variables can vary (change). Let's start with a very simple example, look back at "Start" for a minute. On the line below npcSpeed's definition insert a *new* line that reads:

```
npcSpeed = 5;
```

This is the first time that we've updated a variable's value after defining it. We don't include the "float" variable type here, as that would create a *new* variable named npcSpeed, and we want

to use the same variable as defined above, but just change its value. We'll want to remove this line (and the following lines from this section) later as we're only using them as an example. We don't actually want the snowmen to be that fast!

If you run the game, you'll see that the snowmen all move at a much higher speed! This is how we update variables. We again use the "=" symbol to store a value on the right side into a variable on the left. The only thing missing here from the variable definition is the "float" type declaration. Since we've declared its type already the variable will stay at that type, meaning we don't need to include it when we update its value.

Updating a variable's value is important if a variable should store different values throughout the game. For example, if we're trying to track the time the player plays for! However, we've only scratched the surface when it comes to updating values. We can actually use the value stored in a variable and update that value *in the same line*. Remember the way the "=" symbol works. It takes the value on the right side of it and stores it in the variable on the left. That ordering is very important. As an example, change the line that updates the npcSpeed variable and change it to:

```
npcSpeed = npcSpeed+10;
```

When Unity reads through the script, what it does is take the value on the right hand side of the "=" symbol and runs any needed calculations first. So npcSpeed (which stores 0.5) is added to 10 (so it would be 10.5). Then that new value is stored back into the variable on the left hand side of the "=" symbol.

Go ahead and run the game, and you'll notice that the snowmen now move *much* faster. That's because their speed has a much higher value!

You can actually use addition (+), subtraction (-), division (/) and multiplication (*) in this same way. To prove it let's insert a new line just below our addition line that subtracts 10 from the npcSpeed then stores the new value back into npcSpeed. Before you play the game again, think about what this change should do. How should the value of npcSpeed that the game begins with relate to the last time you played? And the time before you added the addition line?

Once you've done that try using multiplication (*) and division (/) in the same way. Get rid of the two addition and subtraction lines and replace them with first just one multiplication line (so * 10 instead of + 10). Play through the game. Then add a division line by the same amount (/ 10). What would you expect to see with this change?

2.9 Using Mathematical Operations

Now that you're a coding math master we can get back to the original problem at hand. We want to make sure that "timeSpent" reflects the amount of time that the player has spent playing the game. For that we'll want to use a variable referred to as "Time.deltaTime". The variable "Time.deltaTime" stores the amount of time since the last time "Update" was called and is accessible due to the "using UnityEngine" line at the top of the script.

Let's go ahead and add "Time.deltaTime" to timeSpent. Add a new line below the definition of timeSpent where you set timeSpent equal to timeSpent plus "Time.deltaTime". Your Update function should now look like Figure 9.

Run the code now and either win or lose and you should see a display with a non-zero value.

However, this is only a fraction of a second, and definitely isn't representative of the player's entire play time. You may have already realized why that would be the case. Since "timeSpent" is defined as zero at the top of Update every frame, and "Time.deltaTime" added to it every frame, it

```
//This chunk of code (called a function) is run at every frame
void Update () {
    float timeSpent = 0;
    timeSpent = timeSpent + Time.deltaTime;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 9: Update with Time.deltaTime added.

will never be larger than one value of Time.deltaTime. Essentially we set timeSpent back to zero every Update before adding Time.deltaTime, therefore it only has the value of one Time.deltaTime stored in it. So we need some way to define timeSpent elsewhere so we don't set it back to zero every frame.

Definition!

VARIABLE SCOPE: An attribute of a variable, specified by *where* it is defined, which determines where it can be used and updated.

2.10 Variable Scope

At this stage we'll actually introduce something entirely new: **variable scope**. A variable's scope essentially determines where it can and cannot be used. At present you've only been defining variables within a function. We refer to this as a "local" definition. Variables with local scope cannot be used outside of the functions they are defined in. This means that when we define a definition in a specific function (as we have done so far) we cannot use it outside that function. For example, try adding a new line to Update instead of Start that reads:

```
npcSpeed=npcSpeed+5;
```

If you do that, the game will not run, as it will think that you're using a variable without defining it. You'll have to delete that line to get the game to play again. This might seem weird, because the variable "npcSpeed" clearly exists in Start, but this inability to access variables from other functions actually saves memory as it decreases how many variables the game has to keep track of at any one time.

The alternative to variables with local scope is variables with *global scope*. To set up a variable to have global scope we simply define it outside a function. Remove the variable definition of "timeSpent" from Update and move the entire line to be above Start. Your script should now look something like Figure 10.

```
1  using UnityEngine;
2
3  public class LessonOneGame : MonoBehaviour {
4      float timeSpent = 0;
5
6      // This chunk of code (called a function) is run at the beginning of play time
7      void Start () {
8          //jumpValue determines the height of the player's jump
9          float jumpValue = 1;
10         //gravityValue determines how fast the player falls after a jump
11         float gravityValue =20;
12         //speedValue determines how fast the player moves
13         float speedValue = 5;
14         //npcSpeed determines the speed that the npcs move at
15         float npcSpeed = 0.5f;
16
17         LessonOneGenerator.SetUpGame (jumpValue, gravityValue, speedValue, npcSpeed);
18     }
19
20     //This chunk of code (called a function) is run at every frame
21     void Update () {
22
23         timeSpent = timeSpent + Time.deltaTime;
24
25         //Check to see whether any of the enemies have reached the center
26         if (LessonOneGenerator.PlayerLost()) {
27             string lostString = "Player Lost! Time Spent: "+timeSpent;
28             LessonOneGenerator.EndGame (lostString);
29         }
30
31         //Check to see whether all of the enemies cannot reach the center
32         if (LessonOneGenerator.PlayerWon ()) {
33             string wonString = "Player Won! Time Spent: "+timeSpent;
34             LessonOneGenerator.EndGame (wonString);
35         }
36     }
37 }
```

Figure 10: Moving timeSpent to have global scope (line 4).

```
//This chunk of code (called a function) is run at every frame
void Update () {

    timespent = timespent + Time.deltaTime;
    int timespentint = (int)timespent;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost! Time Spent: "+timespentint+" seconds";
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won! Time Spent: "+timespentint+" seconds";
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 11: Using an explicit cast to get rid of decimals for timeSpent.

Play the game again, and you'll find that the value at the end of the game much better reflects your actual play time!

Definition!

CASTING:
A means of transferring a value from one variable type to another.

Definition!

IMPLICIT CAST:
A type of cast in which the type to cast to isn't specified, but assumed by the system (Unity).

Definition!

EXPLICIT CAST:
A type of cast in which the type to cast to is specified in parenthesis by the programmer.

2.11 Cleaning up the Display

There's quite a lot of decimals in this output though, don't you think? This is due to the fact that we were using floats, which track decimal values. We could instead use an integer value, which can only store whole numbers. However, we can't just make timeSpent an integer as Time.deltaTime is a float value! Instead we can convert timeSpent from a float to an integer right before we print it out via a process called **casting**.

We actually already used casting when we added a string value to timeSpent to make it into a string. That type of casting is referred to as an **implicit cast**. With implicit casts you as the programmer do not need to specify the variable type you want the variable converted to, as it happens automatically. However, implicit casts only work in specific cases (like casting from a number to a string, but not a string to a number).

To cast a float to an int we'll use something called an **explicit cast**. Instead of an implicit cast, an explicit cast requires you to spell out the type you want to convert to in parenthesis. So to cast to an integer you'd put (int) in front of the variable you wanted to cast from. We'll also need a new int variable to store the int "version" of timeSpent. So let's add the below line to Update:

```
int timeSpentInt = (int) timeSpent;
```

And then use timeSpentInt instead of timeSpent in the conversion to the string for end of game text. Your Update functions should now look like Figure 11.

Play the game, you should now see that your output is much easier to read!

However, while its much easier to read, the output isn't exactly clear. While we know that the number expressed is the seconds of game time, that's not something known to someone else

playing the game. We can make this more clear by adding the word "seconds" to the output. This is actually very simple! We've already used "+" earlier to combine a string value and both timeSpentInt (an int) and timeSpent (a float). We can actually use the "+" symbol to combine two strings as well! So we can add the word "seconds" to the end of the output simply by adding + " seconds" to the end of the line. The space before sounds is important as otherwise there won't be one between the number from timeSpentInt and the word seconds.

2.12 Adding Complexity

In the last section we created a single variable to add new information to the game. In this section we'll add new information that requires two different variables. Players of the game may want to know the rate at which they placed blocks through the game. If you are unfamiliar, the rate of block placement would be how many blocks were placed for a given amount of time. In this case we'll say blocks placed per seconds. We already have one of the variables required: the play time in seconds thanks to timeSpent. The other half of the information we need is the number of blocks placed throughout the game.

In order to track the number of blocks placed we'll use a very similar number of steps as with timeSpent, except that you'll need to use an int value instead of a float and we'll use Selector.blocksLastTick instead of Time.deltaTime. Selector.blocksLastTick holds the number of blocks placed since the last time "Update" was called, just like how Time.deltaTime held the time since "Update" was called. Try to walk through the entire process without glancing down at the image of the code further down, replacing timeSpentInt with your new variable. The name of your variables doesn't actually matter as it doesn't impact the game at all, its purely for yourself and other programmers looking at your code. If you'd like to remain consistent with us, we used the name "blocksPlaced".

Once you have the variable working and displayed to the player at the end of the game we can go ahead and calculate the rate of blocks placed per second. All it takes is dividing blocksPlaced by timeSpent (or formally Selector.blocksPlaced/Time.deltaTime) to get the average amount of blocks placed each second of the game. You previously used division with code with a variable and a number value. Division with two variables is exactly the same. Can you figure out how to calculate it without looking at the Figure? It might help to break this process into each of the steps that you'll need to go through to get this to work.

1. Define a new variable "blocksPlaced" that's an int with global scope.
2. Add a line in update to add Selector.blocksLastTick to blocksPlaced every frame.
3. Calculate the rate by defining a new variable with the value of blocksPlaced /timeSpent. (We used blocksPlacedRate)
4. Add blocksPlacedRate (or whatever you named the variable) to the output.

3 End of Lesson One

That's all for Lesson One! In this lesson you learned how to:

1. Alter scripts towards a specific purpose.

```
using UnityEngine;

public class LessonOneGame : MonoBehaviour {
    float timespent = 0;
    int blocksPlaced = 0;

    // This chunk of code (called a function) is run at the beginning of play time
    void Start () {

        //jumpValue determines the height of the player's jump
        float jumpValue = 1;
        //gravityValue determines how fast the player falls after a jump
        float gravityValue =20;
        //speedValue determines how fast the player moves
        float speedValue = 5;
        //npcSpeed determines the speed that the npcs move at
        float npcSpeed = 0.5f;

        LessonOneGenerator.SetUpGame (jumpValue, gravityValue, speedValue, npcSpeed);
    }

    //This chunk of code (called a function) is run at every frame
    void Update () {

        timespent = timespent + Time.deltaTime;
        int timespentint = (int)timespent;

        blocksPlaced = blocksPlaced + Selector.blocksLastTick;
        float blocksPlacedRate = blocksPlaced / timespent;

        //Check to see whether any of the enemies have reached the center
        if (LessonOneGenerator.PlayerLost()) {
            string lostString = "Player Lost! Time Spent: "+timespentint+" seconds. Block placement rate: "+blocksPlacedRate;
            LessonOneGenerator.EndGame (lostString);
        }

        //Check to see whether all of the enemies cannot reach the center
        if (LessonOneGenerator.PlayerWon ()) {
            string wonString = "Player Won! Time Spent: "+timespentint+" seconds. Block placement rate: "+blocksPlacedRate;
            LessonOneGenerator.EndGame (wonString);
        }
    }

}
```

Figure 12: The entire script including block rate

2. Use mathematic operations in code.

3. Differentiate different variable types

4. Convert between different variable types

5. Create new variables and integrate them into code

In the next lessons we'll cover using if statements in order to allow the game to make choices about when to run certain pieces of code, use loops to build up scenery, and using functions to create new possibilities with code.

4 Lesson One Glossary

Vocabulary Word	Definition	Examples
variable	The part of code that stores information for use.	jumpValue, gravityValue, speedValue, etc
comment	A part of code left by the code's author to explain what some code does in plain English. Starts with two forward slashes (//).	//jumpValue determines the height of the player's jump
keyword	An English word with a specialized purpose in code beyond its normal definition.	using, public, private
class	A collection of code including both variables and functions with a similar purpose.	LessonOneGame, Time, Selector, MonoBehavior, etc
function	A single section of code with a particular purpose, the equivalent of a paragraph in code.	Start and Update
variable type	An attribute of a variable, specified when it is defined, which determines what values can be stored into it.	float, double, int, bool, string, etc
variable scope	An attribute of a variable, specified by *where* it is defined, which determines where it can be used and updated.	[Not Appropriate]
casting	A means of transferring a value from one variable type to another.	[See Below]
implicit cast	A type of cast in which the type to cast to isn't specified, but assumed by the system (Unity). Only works between certain types.	string wonString = "Player Won! Time Spent: "+timespentint;
explicit cast	A type of cast in which the type to cast to is specified in parenthesis by the programmer.	int timespentint = (int)timespent;

Lesson Two

Escape the Cave

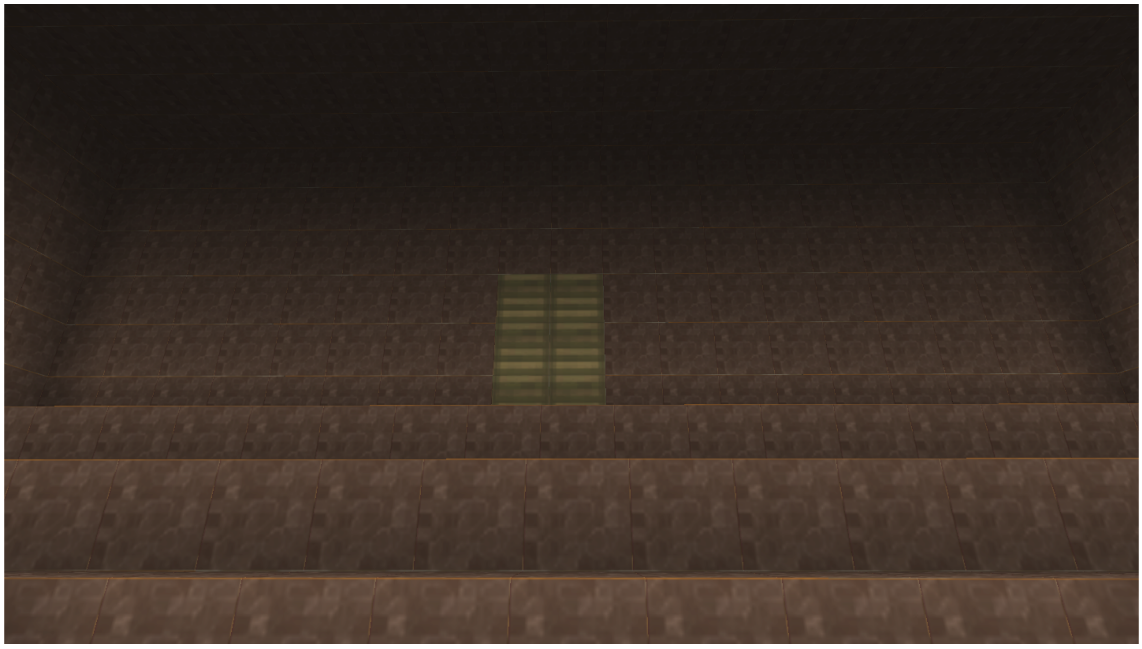

Figure 1: The door out of the cave (currently impossible to open).

1 Introduction to Lesson Two

Games require player choice, the player making different decisions based on the situation to ensure victory. Code can actually make use of a similar (though simpler) form of decision making. In this section we'll look into how code can make decisions based on different information to fix a simple "Escape the Room" game.

2 Lesson Two: Escape the Cave

To begin go ahead and open Unity back up if you closed it between lessons. Once you have the project open again (follow the same steps as from Lesson One), we'll want to open up a new scene. Go to the Project view, and open the Lessons folder then double click the "LessonTwo" scene file.

Click the "play" button at the top of Unity to check out the broken "escape the room" game. Remember you can hit the Escape or Backspace key to get your mouse back and stop the game at any point (this will be necessary to do as the game doesn't end). "Unclick" the play button to stop the game from playing when you're through.

When you play the game you should find yourself in a cave that looks like Figure 1. Unfortunately, there's presently no way out of the cave! You can't break apart the blocks as you could in the prior screen and the door doesn't do anything. It should be fairly clear what we need to do at this point, we need to make some way for the player to leave the cave! For that we'll need to open up the script we'll be using in this lesson. The script is called "LessonTwoGame.cs". You can find it by looking in the Project window within the Codebase folder, open it then open the

```
1 using UnityEngine;
2
3 public class LessonTwoGame : MonoBehaviour {
4     //Reference to a prototype for a sword item
5     public GameObject sword;
6     //Reference to a prototype for a key item
7     public GameObject key;
8
9     // Use this for initialization
10    void Start () {
11
12    }
13
14    // Update is called once per frame
15    void Update () {
16
17    }
18 }
```

Figure 2: What the original LessonTwoGame.cs should look like in MonoDevelop.

LessonTwo folder to open LessonTwoGame.cs or by searching LessonTwoGame in the search bar. Double click the script to open it in MonoDevelop. If MonoDevelop closed in between lessons this may take some time again.

2.1 LessonTwoGame.cs

Once you get LessonTwoGame.cs open in MonoDevelop you should see something that looks like Figure 2, Most of this should look the same as the LessonOneGame.cs file we edited in the prior lesson. Just as before the entire script is a single class that shares the same name as the script file (in this case LessonTwoGame). We also have both Start and Update functions again, which serve the exact same purpose as in LessonOneGame and act the same way.

Along with all the similarities from the previous lesson's script LessonTwoGame also has a couple new parts. We have two variables near the top "sword" and "key" of a new type "GameObject". Unlike the floats, integers, and strings of the previous section GameObject is capitalized, meaning it is a class (so this is a "complex" variable, as opposed to the "primitive" variables of the previous section). We'll explain why these two variables are here after we first make escape possible.

Definition!

CONDITIONAL STATEMENTS:
A line of code allowing for a "decision" to occur based on different conditions. A way of expressing if something should occur.

2.2 Conditionals Introduction

In order to fix the game, we'll use something called **conditional statements**. Conditional statements are used to perform different actions based on different conditions. That might seem a bit complex, so consider this example: say you wanted to know if Super Hero Movie (coming to theaters July 26th) was out already. You could answer this question by saying: "*If* it is July 26th or later, then Super Hero Movie is out". We refer to this statement as "conditional" as there is a condition (it is July 26th or later) before it is true.

In a video game, these types of conditional statements also come up. Say for example we had a fighting game with a visible "health bar" that flashes whenever the player fighter is near death. We could structure this statement as: "*If* the player's health is lower than ten percent, then the health bar should flash".

Definition!

IF STATEMENT:
The simplest way to express a conditional statement in code. Stores code that only runs if it's condition is true.

In code we have to represent these a little more formally. We can do that with something called an **if statement** An if statement looks a little like the following:

```
if(Condition){
    DoSomething();
}
```

How this works mechanically is that a condition is placed within the parenthesis of the if statement. If the condition is true (so in the health bar example if the player's health is less than ten percent) then the code between the left and right curly brackets runs (DoSomething() in this case). Otherwise, it does not. We typically indent the code within the if statement in order to better visualize that it is "within" the if statement (as we do with functions and classes as well). The left curly bracket ({) and the right curly bracket (}) specify the beginning and end of an if statement. If you place your cursor to the right of the right curly bracket, it's corresponding left curly bracket will highlight! Try this out, as it's an important thing to check, since not "closing" an if statement with a left curly bracket will lead to an error.

Definition!

BOOLEAN:
A primitive variable type that can store only true or false.

We mentioned how to represent conditions (the things that go into the parenthesis) in code in the previous lesson. There's actually an entire primitive variable type that stores true and false information: **booleans**. If the boolean within the parenthesis of the if statement is True, the code within the curly brackets of the if statement is called. If its false, it is not called. We can use this fact to make it possible to escape the room.

2.3 Make Escape Possible

To begin with, let's make it so that when the player gets near the door it opens (we can make it take more than that to escape later). For that we'll need a boolean value to store whether or not the player is close to the door. Defining a boolean variable is very much like defining a float variable, except that we use the shortened name "bool" (similar to how integers are defined with the shortened word "int"). In addition, instead of numbers a boolean can only ever store values of true or false. Let's define a new boolean variable named nearDoor in Update, the line should look like

```
bool nearDoor = false;
```

Don't forget to include the semi-colon! Remember that tells Unity when a line ends. Insert that line into the currently empty Update function. You could try to play the game again, but just creating the variable will have no impact on the gameplay as we haven't yet told the game to

```
14      // Update is called once per frame
15      void Update () {
16          bool nearDoor = false;
17
18          if (nearDoor) {
19              LockedDoor.OpenDoor();
20          }
21      }
```

Figure 3: Update after the addition of nearDoor and the if statement.

do anything with that variable. Remember that code doesn't have the knowledge a human has, it can't understand what "nearDoor" is for, we have to write the code to tell it how to use it.

To actually get the behavior we want, we'll need to next add an if statement. Use the same if statement structure from the previous section, using "nearDoor" in place of "Condition" and a new function call "LockedDoor.OpenDoor()" in place of "DoSomething()". In the end your Update function should look like Figure 3.

LockedDoor.OpenDoor() seems to come out of nowhere, so let's break it apart. "LockedDoor" refers to a class called LockedDoor, like the classes LessonOneGame and LessonTwoGame that we've edited before. "OpenDoor" is a function in LockedDoor. Previously we had functions "called" (run line by line) by the game engine, with Start called at the beginning and Update called every frame. But by simply giving Unity the name of the class, and the name of the function in that class we want to call, we can have a function like "OpenDoor" called whenever the line that has it is reached by Unity![1]

Make sure to save the file and then play the game again. Notice that nothing happens!

You'll note that nothing different happens even with the added if statement. If you think through it though, this makes sense. We already said that if the conditional in an if statement's parenthesis is false, then the code inside it doesn't run, and now we've seen proof of that. To make the code within the if statement run, we'll need to set nearDoor to true. Go ahead and make the change so that nearDoor has an initial value of "true".

Save and play the game again. You'll note very different, but still broken behavior.

While setting nearDoor to true *did* cause the door to open, it did so immediately, causing the screen to look a bit like Figure 4. Can you think of why that might be?

Since we set nearDoor to true, the if statement we added was called on the very first Update call. That meant the player had only the first frame of the game in the cave before LockedDoor.OpenDoor() is called, thus opening the door and ending the game. That's not the behavior we wanted!

[1]Note: Not all functions can be called in this way. For example, we couldn't call Start or Update like this. We'll get to why that is in a future lesson.

Figure 4: "Winning" the escape the room game a bit too early.

What we need is to have nearDoor not stay on true or false, but *vary* (as variables can) in value based on what's happening in the game. Happily there's another variable we can use to do just that! It's called LockedDoor.playerEntered. It's a boolean value that tracks whether or not the player has entered the "area" of the locked door. We can use its value to inform nearDoor of when to be true and when to be false. Replace your nearDoor definition line with:[2]

```
bool nearDoor = LockedDoor.playerEntered;
```

Play the game again (make sure to save first). You should now find that you can wander around the cave freely until you get right up to the door. At this point the game will pause and the door will open!

And with that you did it! You've made it possible to escape the cave! But it was a bit too easy wasn't it? In the next sections we'll add a little bit more challenge to escaping the cave and delve into more complex conditional statements.

2.4 Adding Complexity

It may have seemed odd that we had a class named "LockedDoor" that allowed the player to go through it without "unlocking" it. Let's see if we can add a requirement for a key so that makes more sense. In order to do this we'll need to make use of that "key" variable at the top of LessonTwoGame.cs.

[2]Note: Instead of using nearDoor at all, you could at this stage just simply put LockedDoor.playerEntered into the parenthesis of the if statement, as it's a boolean that will hold the values we want. However, we'll continue to refer to nearDoor throughout the rest of the lesson.

```
// Use this for initialization
void Start () {
    float x = -5;
    float y = 2;
    float z = -5;
    ItemHandler.SpawnItem (key, x, y, z);
}
```

Figure 5: Start function with all the information to spawn a key.

The variable "key" is of type GameObject, which is a special kind of class used by Unity to represent the "objects" in a game. It can also be used to represent a "blueprint" of a GameObject, which can then be built in the game. You can actually use this process to create several "versions" of the same blueprint in a single game, that's how we made the snowmen from the last lesson! This notion, that you can create many objects from one "blueprint" is an incredibly powerful use of code, and not just for making snowmen.

For now we just need the one key though. In order to do that we'll need to introduce a new function call.

2.5 3D Coordinates and SpawnItem

In order to place a key into the game we'll first need to know *where* it should be placed. Unity is a 3D game engine, therefore the location of every object in the game is defined by three numbers: an x coordinate, a y coordinate, and a z coordinate. You may have used x and y coordinates in a math class in a two-dimensional plot where points extended out from a central origin (at point 0,0). The way that Unity represents location is very similar to this 2D grid. But since we've got a 3D game engine, there's an added z coordinate to handle the extra dimension.

That may still seem confusing, so let's go ahead and show rather than tell. Add the following to the currently empty Start function:

```
float x = -5;
float y = 2;
float z = -5;
ItemHandler.SpawnItem (key, x, y, z);
```

Your Start function should now look like Figure 5. Adding the call to the function SpawnItem will create an instance of a "key" at the passed in location.

Play the game again, you should now notice a key! You can even go over and pick it up! (Though this won't have any different effect on the door yet.)

They key you see is currently at position (-5, 2, -5). That's not particularly meaningful though all by itself. To give you a better sense of how these coordinates work in 3D space let's change the

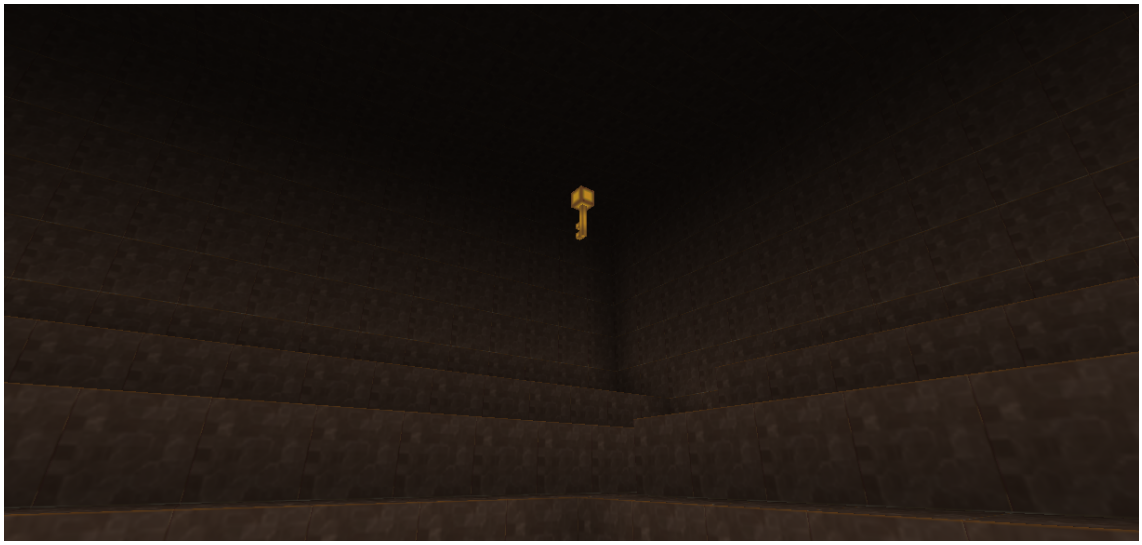

Figure 6: The key floating in the air thanks to a y value of 6.

y variable to equal 6. This will put the key 4 "blocks" higher and leave it floating in the air.

Play the game with the y variable now set equal to 6. Note the difference that makes!

It should be easier now to see how coordinates work. Feel free to change the x and z variable values, this will change the key's left to right and front to back position (if facing the door) respectively. Try to see if you can put the key somewhere interesting. Perhaps hidden in a corner? As reference the walls are located along the lines x = 10, x = -10, z = 10, and z = -10, so don't go beyond those values or the key will be impossible to get.

Play the game after making changes to see where you've placed the key. Make sure not to place the key beyond the cave walls! Keep making changes till you're happy with the key's placement. Once you are, continue on reading.

Now that you're happy with the lines that define variables x, y, and z there's a fourth line remaining to discuss. "ItemHandler.SpawnItem (key, x, y, z);" This is a function call, to the "CreateItem" function, but it's a bit different from those we've seen before. While it still has a class name in front of it (ItemHandler), followed by a period and the name of the function, we have variables placed inside the parenthesis! What's up with that? We've copied the function definition of SpawnItem from ItemHandler.cs below to make this a bit more clear. Worth noting this means we don't need to add the below code to the script, it's merely for reference.

```
void SpawnItem(GameObject obj, float posX, float posY, float posZ)
```

And here's the line we use to call SpawnItem with, for reference:

```
ItemHandler.SpawnItem (key, x, y, z);
```

If we look at what goes into the parenthesis of the function definition, you might see a couple connections. For one, there are a total of four different entities, each separated by commas. The first entity seems to have something to do with the GameObject type (in the definition we have

"GameObject obj" and in calling the function we have "key" which is of type "GameObject"). The following entities are all of the float type in both the definition, and calling of that function.

It should make sense then when we say that a function definition specifies the *types* and ordering of values needed to call this function (these are referred to as a function's "arguments"). Its a bit like a game of mad-libs or fill in the blank where we specify a type of word (adjective, verb) needed to go in a particular blank. Similarity a function's arguments specify the types of values needed to call a function. Functions like "LockedDoor.OpenDoor();" have nothing inside its parenthesis (no arguments) and therefore don't need any values at all to be called.

Worth noting that while "SpawnItem" therefore needs a GameObject and three float values (in that order) to be called, that's all it cares about. We've already shown that we can pass SpawnItem different float values (with the different values of x, y, and z). As a further example of how functions can be passed different values (as long as the types match up) let's try changing the GameObject value. Change the line where you call SpawnItem in Start to:

```
ItemHandler.SpawnItem (sword, x, y, z);
```

Play the game and notice the sword where you last saw a key.

So that proves it! SpawnItem doesn't care what values its passed, it'll work as long as we pass it a GameObject, and three float values. But why? There's actually a reason we've used "values" and not "variables" when talking about this. When Unity reads in the line "ItemHandler.SpawnItem(sword, x,y,z)" it actually translates each of the variables present into whatever value they store. For example if the current values are (sword = swordValue, x = -5, y = 6, z = -5) then unity translates it to"ItemHandler.SpawnItem(swordValue, -5, 6, -5)".

We can actually use this fact to clean up our code a little in that we can get rid of this translation step. Change SpawnItem back to using a key and replace the variables x, y, and z with the three values you used. That way we can cut out the middle-man. So if you used the values (x = -5, y = 6, z = -5) then SpawnItem should now read:

```
ItemHandler.SpawnItem (key, -5, 6, -5);
```

Play the game and see for yourself. We don't need variables to call SpawnItem, just values of the types it requires!

This is great as we compressed what was four lines of code into just one. We can just get rid of the x, y, and z variables as we don't need them anymore! Feel free to adjust the values you pass to SpawnItem till you're happy with where you place the key before moving on. You can make use of decimal values as well to have a key sticking partway out of the floor, or wall. Remember to put an "f" after any decimal values to ensure that Unity still reads it as a float value though!

Next we're going to make the key actually affect whether or not the door opens!

2.6 Making the Key Do Something

If we think about how keys and locked doors tend to work it's usually the case that locked doors won't open unless the opener has a key. We want to replicate this behavior. To do this we'll need to create a new boolean variable to track whether or not the player has the key. Let's call the variable hasKey. It'll need to be defined above Start so that it has global scope. Can you think why?

Go ahead and add hasKey with an initial value of false to above Start. You can use the line where we defined nearDoor as a reference. The only thing that needs to change is the name of the variable and where its located.

Unfortunately unlike with near door we cannot just grab the value from some variable in another class. The closest thing we have available to us are the boolean variables ItemHandler.justPickedKey and ItemHandler.justPickedSword. Similar to Time.deltaTime from the previous lesson ItemHandler.justPickedKey and ItemHandler.justPickedSword differ from frame to frame. They are only true if the player picked up the key (justPickedKey) or sword (justPickedSword) in the *previous* frame (they are false otherwise). We can't simply set hasKey to Itemhandler.justPickedSword as then hasKey will only be true for a single frame. We want it to be true forever after ItemHandler.justPickedSword is true for just one frame. Can you think of a way to do this?

The solution is to make use of another if statement in the Update function (just above the other code we've written thus far) to check for the one frame when justPickedKey is true, and then to set the value of hasKey to true. Try to add this to Update without looking at Figure 7.

This is also why hasKey had to be global. We needed to still be able to tell whether we had the key (if hasKey is true) beyond a single frame of Update. If it was defined in Update, it would only be set to true for the single frame that ItemHandler.justPickedKey is true. Worth noting at this stage that there's no way for hasKey to go from true to false, since we didn't add one to the code. So once the player has the key, they cannot lose it.

Now that we have this information we can add an additional if statement check to ensure that the player has the key before opening the door. We could have just changed the if statement that uses nearDoor to use hasKey. But then we'd just transport through the door as soon as we grabbed the key!

Instead we need to check when both hasKey *and* nearDoor are true. The simplest way to do this is to actually put one if statement *inside* the other. If you think about the way if statements work this should make sense. If statements run the code within them if the conditional (the bool value in the parenthesis) is true. Therefore, the first if statement will run when it's condition is true, and the second one will run when it's condition and the first if statement's condition are true. When you're done Update should look something like Figure 8. We refer to if statements containing if statements as "nested" if statements.

Play the game again. First try to walk through the door without getting the key, then with the key. It works!

Congrats! You've managed to set up a simple puzzle to escape a room, where a player has to find or reach a key and then escape. However if someone besides you were playing and didn't notice the key at first, they may get confused as to what to do (especially if you've hidden the key). Let's see if we can fix that in the next section.

2.7 Giving the Player Hints

If you'll recall, in the last lesson you wrote text that displayed on either a win or a loss. We can actually display text like this even without the game ending. We'll use another function "GUIManager.SetDisplayText" to do this. Again, GUIManager is the name of the class, with SetDisplayText being the name of the function.[3] SetDisplayText takes two values to run, the first

[3]Note: GUI stands for Graphical User Interface, and is the part of a game that displays information to the player but isn't *in* the game (like a health bar, item being used, etc).

```
1  using UnityEngine;
2
3  public class LessonTwoGame : MonoBehaviour {
4      //Reference to a prototype for a sword item
5      public GameObject sword;
6      //Reference to a prototype for a key item
7      public GameObject key;
8
9      bool hasKey = false;
10
11     // Use this for initialization
12     void Start () {
13         ItemHandler.SpawnItem (key, -5, 6, -5);
14     }
15
16     // Update is called once per frame
17     void Update () {
18
19         if (ItemHandler.justPickedKey) {
20             hasKey = true;
21         }
22
23         bool nearDoor =  LockedDoor.playerEntered;
24
25         if (nearDoor) {
26             LockedDoor.OpenDoor();
27         }
28     }
29 }
```

Figure 7: LessonTwoGame.cs after updating everything for hasKey.

```
// Update is called once per frame
void Update () {

    if (ItemHandler.justPickedKey) {
        hasKey = true;
    }

    bool nearDoor =  LockedDoor.playerEntered;

    if (nearDoor) {
        if(hasKey){
            LockedDoor.OpenDoor();
        }
    }
}
```

Figure 8: Update with the nested if statements.

being a string type value containing the text to display, and a float type value determining the time to display the text.

To demonstrate the use of "SetDisplayText", let's add an introductory message to the game. In Start add the following lines:

```
string startText = "Attempt to Escape the Cave!";
float timeToShow = 3;
GUIManager.SetDisplayText(startText, timeToShow);
```

As an alternative you could instead do this with just a single line (remember that it's the value types that matter, we don't have to use variables for this):

```
GUIManager.SetDisplayText("Attempt to Escape the Cave!", 3);
```

Play the game and notice the helpful text that pops up at the beginning of the game. Feel free to play with the string value of the startText variable. Make it whatever you like! However, if the startText value gets much longer, you might also want to increase the timeToShow value (as it represents the time in seconds the text will display). Play the game again to make sure your new text appears.

Definition!

ELSE STATEMENT: A conditional statement like an if statement, which only runs the code inside it if the above conditional statement *doesn't* run.

Now that we've seen how easy it is to display text, let's add in a hint that the player needs the key. We could add it right to the startText, but then players might not look around as much, since they'd know to look for the key specifically. Instead, why don't we make it so that a hint to find the key pops up if the player tries to go through the door without one. For this we'll use a new way to express conditional statements, an **else statement**.

Else statements can only be located underneath another conditional statement (an if statement for example, though we'll introduce another kind of conditional statement soon). They work in a way similar, but opposite, to if statements. In that the code within an else statement will only run if the code in the above conditional statement *doesn't* run.

Let's show an example of else statements in action. Below the "hasKey" if statement in Update, we'll add an else statement. Inside the else statement's curly brackets we'll want a new call to GUIManager.SetDisplayText that displays text telling the player they need a key to go through the door. Make the changes as seen in Figure 9 to do just that. Remember we could instead just include the values instead of variables when we call "SetDisplayText". We only use the variables here to make it a bit clearer what's going on.

You will notice, unlike if statements, else statements do not have parenthesis within which a boolean value can go. This should make sense, as else statements don't have conditions of their own, only running if the statement above them does not.

Play the game. Try going up to the door without the key and notice our new statement pop up! Feel free to change the string value of keyText as well. You could even make it seem like the door is talking to the player by just changing the text to something like "Door: Now, now I won't let you through without a key!".

Can you think of anywhere else we could add an else statement? We could add an else statement and call to SetDisplayText below the "nearDoor" if statement, but that would display every frame of the game the player isn't near the door. That's almost the whole game! If you'd like, you could add an extra call to SetDisplayText in the if statement where the player picks up the key as that'll only occur once. Perhaps saying something like "Got key!". If you add that, test it out before continuing.

```
// Update is called once per frame
void Update () {

    if (ItemHandler.justPickedKey) {
        hasKey = true;
    }

    bool nearDoor =  LockedDoor.playerEntered;

    if (nearDoor) {
        if(hasKey){
            LockedDoor.OpenDoor();
        }
        else{
            string keyText = "The door is locked! Find a key!";
            float keyTime = 3;
            GUIManager.SetDisplayText(keyText, keyTime);
        }
    }
}
```

Figure 9: Update with the extra display text from not having a locked door.

```
// Use this for initialization
void Start () {
    ItemHandler.SpawnItem (key, -5, 2, -5);
    GUIManager.SetDisplayText ("Attempt to Escape the Cave!", 3);

    ItemHandler.SpawnItem (sword, 4, 2.5f, -5);
}
```

Figure 10: Start with the added call to spawn a sword.

2.8 Adding the Sword

Now just getting a single "key" doesn't make for a particularly interesting "Escape the Room" game, even if it is a simple one. What if the door instead required the player to find both a key *and* sword? It's dangerous to go alone, after all.

Actually adding the sword shouldn't be too difficult, as we added it instead of the key at one point. You'll need a call to SpawnItem with the sword value, and three new float values. Try to add it yourself before looking at Figure 10.

When you've load up the game, you should see something like Figure 11 if you use the float values shown in Figure 10.

Feel free to play around with different values for the sword and the key float values. You can put the key back into the air, or hidden in a corner. While you shouldn't put the key or sword beyond the walls of the cave, you can have them sticking partly out of the walls or floor of the cave. Play around with it and see what you can do!

2.9 Determine if Player has the Sword

Now that there's a sword in the game, it's possible that the player will pick up the sword before leaving. However, the game doesn't actually require that the player have it before "winning" the game! To do that we'll need to add an extra condition to require the player to have the sword before "LockedDoor.OpenDoor();" can be called. For that we'll need a new boolean to track if the player has the sword, just like how "hasKey" works. Remember that "ItemHandler.justPickedSword" holds if the player picked up the sword in the previous frame.

Using how we defined, and set the value of hasKey as an example, try to create a new boolean variable with global scope called "hasSword" and set its value to true if LockedDoor.justPickedSword is true. However you can also check Figure 12 if you get lost.

2.10 Make the Door Require the Sword: Logical Operators

Now that we have the hasSword variable, we need to actually make sure that the door doesn't open unless hasSword is true. At this point we could make this happen by adding yet another nested if statement, but that seems a bit extreme, don't you think? Take a look at Figure 13 to see what we mean. And if you think about it, some games might have even more conditions than this! Image four or five nested if statements, that would just get to be too much.

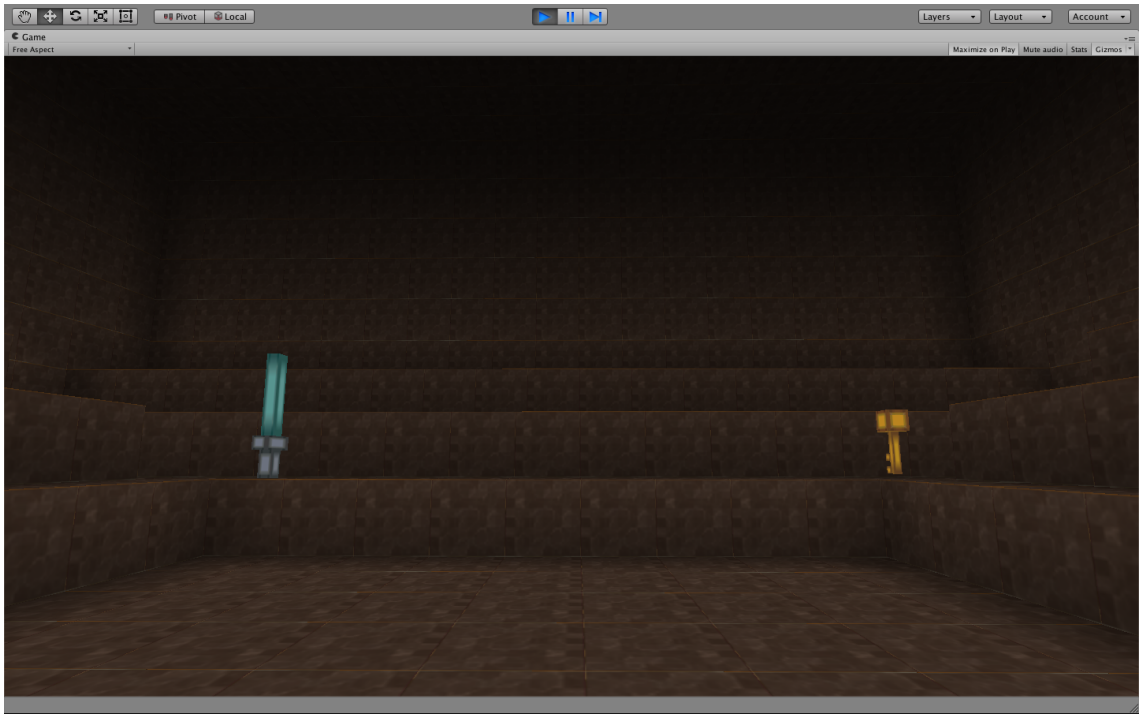

Figure 11: What you should see with the float values from Figure 10 for key and sword spawning. (though you don't have to use these values)

```
public GameObject key;

bool hasKey = false;
bool hasSword = false;

// Use this for initialization
void Start () {
    ItemHandler.SpawnItem (key, -5, 2, -5);
    GUIManager.SetDisplayText ("Attempt to Escape the Cave!", 3);
    ItemHandler.SpawnItem (sword, 4, 2.5f, -5);
}

// Update is called once per frame
void Update () {

    if (ItemHandler.justPickedKey) {
        hasKey = true;
    }

    if (ItemHandler.justPickedSword) {
        hasSword = true;
    }

    bool nearDoor =  LockedDoor.playerEntered;

    if (nearDoor) {
        if(hasKey){
            LockedDoor.OpenDoor();
        }
        else{
            GUIManager.SetDisplayText("The door is locked! Find a key!", 3);
        }
    }
}
```

Figure 12: The main section of LessonTwoGame after adding everything needed for “hasSword”.

```
if (nearDoor) {
    if(hasKey){
        if(hasSword){
            LockedDoor.OpenDoor();
        }
    }
    else{
        GUIManager.SetDisplayText("The door is locked! Find a key!", 3);
    }
}
```

Figure 13: Three nested if statements? That gets a bit hard to read.

Definition!

LOGICAL OPERATOR:
A symbol that works on conditions and can express either and (&&), or (||) or not (!) relationships.

Happily there's a solution to using so many nested if statements. They're called **logicial operators**, which makes them similar to the mathematical operators we've used in the last lesson (+, -, ., and *). There are three of them "and" (expressed in code using &&), "or" (expressed in code using ||) and "not" (expressed in code using !).[4]

Both && (and) and || (or) can be used to chain together multiple conditions to different effects, that works about as you'd expect. If we had a line of code that read "if(hasKey && hasSword)" in english we would say "if has key and has sword", in other words the condition is if the player has both the sword and the key. That's exactly what we want! If alternatively we used "if(hasKey || hasSword)" then the code in the if statement would be run if hasKey *or* hasSword was true.

Let's go ahead and replace the "if (hasKey) " line to read if(hasKey && hasSword)".

Run the game and try to get through the door with just the key. You can't do it! Now get the sword. You should get the end of the game screen!

The player now has to have the sword to escape the cave. The only problem there is that the display text from the else statement popped up and said that you needed a key, even though you got a key first! That's no good! What's even worse is that if you play the game again, and go up to the door with no items, it'll still display that same text referencing getting a key, even though you now need a sword *and* a key to get through.

Definition!

ELSE IF STATEMENT:
A conditional statement that works like a mix of an else and and if. It only runs if the above statements do not (like an else statement) and if it's own condition is met (like an if).

To fix this problem we're going to use the || (or) logical operator and the last of the conditional statements types, the **else if statement**. An else if statement is like a combination of an if statement and an else statement. It can only go after an if statement or another else if statement, like an else. However, it also has a condition like an if statement. What this means is the code within an else if only runs when the above statement is false, and its own condition is true!

Figure 14 shows an example of the else if statement in Update. The way that this block of code works now, if we follow it from top to bottom is that first "nearDoor" is checked whether it is true or false. If "nearDoor" is true, we check if the player has both the key and the sword ("if (hasKey && hasSword)"). If not, we next check if the player has either the key *or* the sword. If not, then we finally get to an else which runs since the above statement was false.

Let's walk through a potential situation to give an example of how this works. Let's assume the player had the key, but not the sword (hasKey is true, hasSword is false). As long as the player isn't close to the door, the top level if statement with nearDoor will never run as nearDoor

[4]We won't use the ! (not) symbol until the next lesson, but it translates much the same way as && (and) or || (or).

```
if (nearDoor) {
    if(hasKey && hasSword){
        LockedDoor.OpenDoor();
    }
    else if(hasKey || hasSword){
        GUIManager.SetDisplayText("You still lack an item, continue your search.", 3);
    }
    else{
        GUIManager.SetDisplayText("The door is locked! Find a key!", 3);
    }
}
```

Figure 14: The conditional area in Update after the inclusion of the else if.

will be false. However, if the player gets up to the door, then nearDoor will become true, and the code within the nearDoor if statement will run. Code runs from top to bottom so first the if statement is attempted. However, since hasSword is false, hasKey && (and) hasSword will not be true. Therefore we move on to else if statement, it's condition is hasKey || (or) hasSword. Since one of them is true, the code in the else if statement will run!

And that's it! You've now created a full escape the room style game without any inconsistencies and good prompts! If you'd like at this point you could act extra prompts. You could even add in two different else if statements, one to check for if the player has a key, the next to check if the player has the sword. We've put a couple examples of final Update functions below!

Finish up Update and change your displayed text to whatever you like to fit your own style. Can you make the game more spooky? More silly? Play the game and test out all the different combinations of actions and the effects! Once you're happy with it, have a friend play the game.

3 End of Lesson Two

That's all for Lesson Two! In this lesson you learned how to:

1. Use conditionals to express conditions in code (if, else if, and else)
2. How to call functions with different required value types
3. How to ensure that the player gets correct prompts
4. How to use logical operators (&& and ||)

In the next lesson we'll focus on a beat 'em up game. In it, we'll work on enemy behavior, game feel for combat, and more.

```
// Update is called once per frame
void Update () {

    if (ItemHandler.justPickedKey) {
        hasKey = true;
        GUIManager.SetDisplayText("Got Key!", 2);
    }

    if (ItemHandler.justPickedSword) {
        hasSword = true;
        GUIManager.SetDisplayText("Got Sword!", 2);
    }

    bool nearDoor =  LockedDoor.playerEntered;

    if (nearDoor) {
        if(hasKey && hasSword){
            LockedDoor.OpenDoor();
        }
        else if(hasKey || hasSword){
            GUIManager.SetDisplayText("You still lack an item, continue your search.", 3);
        }
        else{
            GUIManager.SetDisplayText("The door is locked! Find a key!", 3);
        }
    }
}
```

Figure 15: A final Update function with an added display for sword and key get.

```
// Update is called once per frame
void Update () {

    if (ItemHandler.justPickedKey) {
        hasKey = true;
        GUIManager.SetDisplayText("Got Key!", 2);
    }

    if (ItemHandler.justPickedSword) {
        hasSword = true;
        GUIManager.SetDisplayText("Got Sword!", 2);
    }

    bool nearDoor =  LockedDoor.playerEntered;

    if (nearDoor) {
        if(hasKey && hasSword){
            LockedDoor.OpenDoor();
        }
        else if(hasKey){
            GUIManager.SetDisplayText("You still need a sword!", 3);
        }
        else if(hasSword){
            GUIManager.SetDisplayText("You still need a key!", 3);
        }
        else{
            GUIManager.SetDisplayText("The door is locked! Find a key!", 3);
        }
    }
}
```

Figure 16: A final Update function with an additional else if statement to check for both items individually.

4 Lesson Two Glossary

Vocabulary Word	Definition	Examples
conditional statements	A line of code allowing for a "decision" to occur based on different conditions. A way of expressing if something should occur.	if, else if, and else
if statement	The simplest way to express a conditional statement in code. Stores code that only runs if it's condition is true.	if(hasKey){...}
boolean	A primitive variable type that can store only true or false.	nearDoor, hasKey, etc
else statement	A conditional statement like an if statement, which only runs the code inside it if the above conditional statement *doesn't* run.	else{...}
logicial operators	A symbol that works on conditions and can express either and (&&), or (||) or not (!) relationships.	and (&&), or (||), and not (!)
else if statement	A conditional statement that works like a mix of an else and and if. It only runs if the above statements do not (like an else statement) and if it's own condition is met (like an if).	else if(hasKey){...}

Lesson Three

I Ain't Afraid of No Ghosts

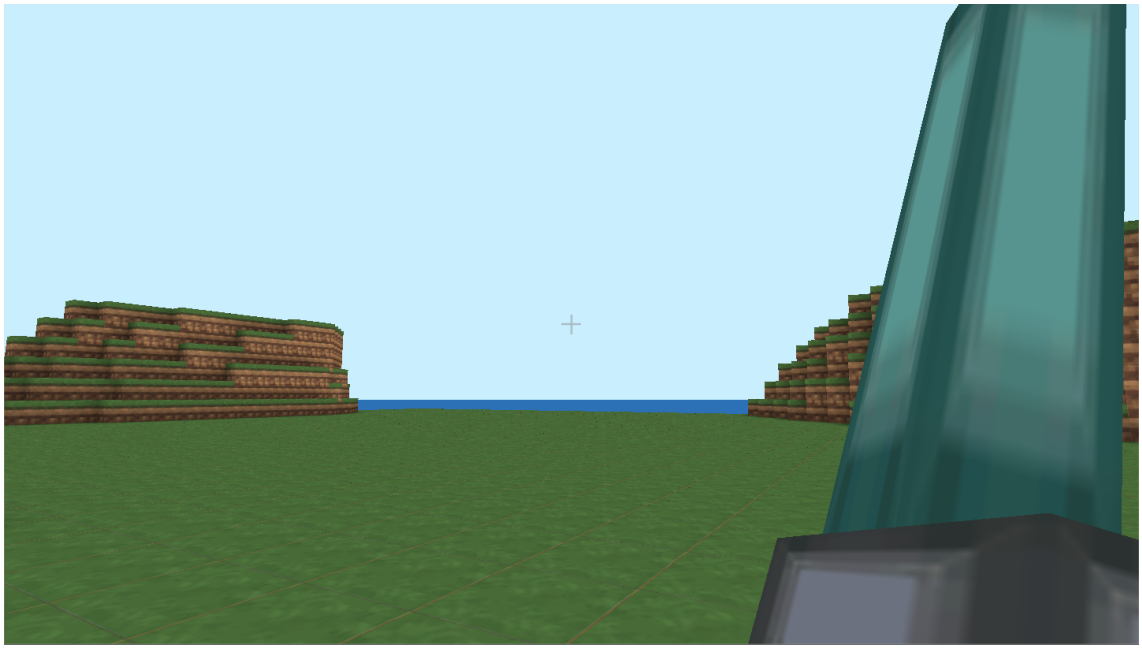

Figure 1: A section of the island that comprises Lesson 3.

1 Introduction to Lesson Three

In the past two lessons, you've learned about the building blocks of code (variables) and how to have your code make different decisions based on the situation (conditional statements). In this lesson we'll bring both of these pieces together and expand upon them in order to fix a *very* broken "beat 'em up" game. This lesson builds on lessons one and two, so we don't recommend attempting it until you've finished both of those.

2 Lesson Three: I Ain't Afraid of No Ghosts

If you closed Unity after Lesson Two, open it back up. Once Unity is opened, select the project from the drop down menu that pops up and wait for it to fully load. From there, go to the "Project" view, open the Lessons folder, and double click the "LessonThree" scene file.

Click the "play" button at the top of Unity to check out the currently broken "beat 'em up" game. Remember you can hit the Escape or Backspace key to get your mouse back and stop the game at any point (this will be necessary to do as the game doesn't end). Note: Windows users may not have the mouse disappear upon play without bringing up the pause menu with Escape then clicking "Resume".

When you boot up the game you should find yourself on a small island. Around the island you will find a number of randomly scattered green ghosts. Sadly, these ghosts don't seem to want to do anything.

However, the first thing you probably noticed was that the player moves much slower than in

```
1 using UnityEngine;
2
3 public class PlayerInfo : MonoBehaviour {
4     //This variable controls the player's speed
5     public float speed = 2;
6     //This variable controls the player's jump height
7     public float jumpValue = 1;
8     //This variable controls the impact of gravity on the player
9     public float gravityValue = 2;
10 }
```

Figure 2: PlayerInfo.cs before any changes have been made.

the last lesson. Additionally, falling seems to go much more slowly. Before we can fully explore the island, we need to deal with these issues. Let's open up the first script (of many) we'll be working with in this lesson. It's name is "PlayerInfo.cs". You'll find it in the Codebase folder, and then within the "PlayerScripts" folder. Double click on the "PlayerInfo.cs" file to open it up in MonoDevelop. As a reminder, MonoDevelop may take some time to open if it was closed after LessonTwo.

2.1 Script Usage and PlayerInfo

In both lesson one and two we only edited a single script. However, even the simple games of lessons one and two relied on a large number of scripts. Typically, each script contains a single class. Classes can, in turn, serve a variety of purposes in a game. For example, a class may be used to store information (like in lesson one where LessonOneGame held values for player speed) or a class may be used to store behavior information (like in lesson two where we added the behavior for the locked door to LessonTwoGame). PlayerInfo falls into the first of these two categories, storing variables that relate to the player.

Definition!

ACCESS MODIFIER: A keyword that limits what other code can "see" a variable, function, or class.

Figure 2 displays how PlayerInfo appears to begin with. By now you should be pretty familiar with the structure of classes. There's still the "using UnityEngine" line at the top of the script, followed by the class information "public class PlayerInfo : MonoBehavior". From there we have three variable definitions. However, these are a bit different than what we've seen before as all three have what's referred to as **access modifiers**. An access modifier is a keyword that determines whether some variable, function, or class can be seen by *other* scripts. The two most common access modifiers are "public" and "private". The access modifier public indicates that a variable, function, or class can be accessed by all other scripts in a project. We've seen it used in all classes thus far (including PlayerInfo) so that they can be used by other classes in the project. While we didn't see the word, it was also used in every single function and variable from other classes we've used so far (so like LockedDoor.OpenDoor() from LessonTwo and Time.deltaTime from LessonOne.

This might seem a bit complicated, so let's show an example. Go ahead change the values in the speed and gravity variables so you can move around a bit easier. We recommend "5" for speed and "20" for gravityValue, but feel free to play with other values for these variables and jumpValue.

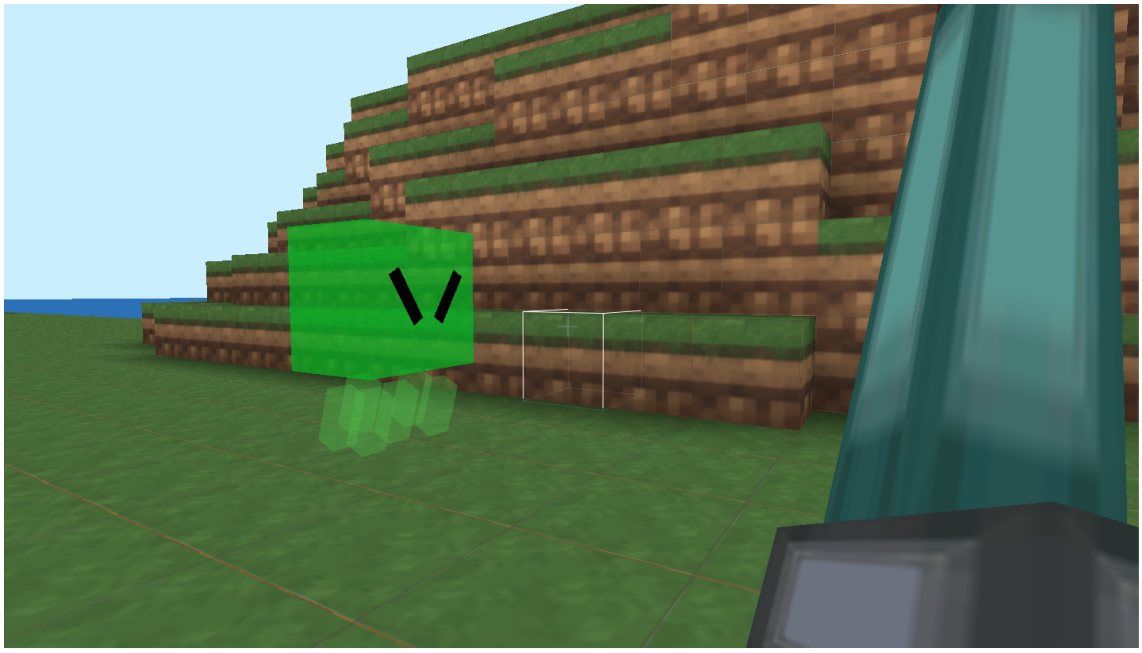

Figure 3: An example of an unmoving ghost.

Make your changes and play the game again, note the improved speed and falling! Feel free to continue to make changes to all three values, testing out their effects on the game world and player motion as you do.

But how did the variables in PlayerInfo.cs actually have this effect? The variables just exist in PlayerInfo.cs. Well in this case, because they're public *another* class (CharacterMotor.cs in this case) reads in the variable values to determine how quickly to move the player. That's the power of the public access modifier!

There's actually an opposite to the public access modifer called "private". We haven't seen it used yet, but it's actually the default access modifier for variables, functions, and classes. So unless otherwise specified, variables, functions, and classes can only be accessed inside the script where they are defined. This is the default as it helps programmers to control how information can be accessed, to be sure it isn't changed when it shouldn't be.

2.2 Fixing Enemy A.I.

Enemy Artificial Intelligence (AI) is a staple of "action" games, where various different enemies create challenge for a player based on their attributes and behaviors. However, our "beat 'em up" presently has no apparent enemy behaviors! All they do is stand around, as portrayed in Figure 3. You can go up to them and hit them with the sword (by left clicking). Why don't we try that?

If you haven't already, play the game so you can run up to an enemy and hit it with your sword.

That's not all that exciting, is it?

At the heart of AI is the idea that an entity will act differently depending on the situation.

For example, if the player is close, then a ghost should attack. But if the ghost doesn't see the player, the ghost should search for the player. That structure kind of looks like the conditionals from Lesson Two, don't you think? In the AI we'll implement for these enemies we'll translate statements like that into code.

If you'll recall, we previously stated that classes served two main purposes, either holding information (like PlayerInfo) or behaviors. The next script we'll be working with is largely of the latter type, holding the behavior of the enemies. The script in question is Enemy.cs. It can be found in the Codebase folder, and then within the "NPC" folder. Double click on it to open it up in MonoDevelop.

When you load up Enemy.cs you should see something like Figure 4. The class Enemy looks very much like other classes we've seen before. It has "Update" and "Start" functions like we've seen before, we'll be using those to store the code to handle the enemies behavior. However, it also has a couple new features.

Definition!

RETURN:
A keyword which "returns" a value from a function to the line of code which called the function.

The first of the two new features can be seen in Enemy's other functions "GetSpeed" and "GetFiringRate". As you can see these functions are public, unlike Update and Start, meaning that they can be called by other code outside of this class. However, the new feature is the teal **return** keyword that both functions use.

Definition!

RETURN TYPE:
Part of a function definition, determines the type of value that function will return, or that it won't return anything if "void".

Though we haven't brought it up before, all functions have what's referred to as a **return type**. This return type specifies the type of value that a function will return, and it's determined by the word directly before a function name in a function definition. So for "Start" and "Update" this word is "void", which means that the function doesn't return anything (void is equivalent to "nothing"). However, both "GetSpeed" and "GetFiringRate" have "float" as their return type, meaning that they return float values when they are called. You may still be a bit confused about return types, but we'll give several examples further in the lesson.

2.3 Class Variables and Moving Enemies with movementController

The second new feature of Enemy is its use of two different new variables "movementController" and "attackController", both of which have types that are classes: NPCMovementController for movementController and RangeAttackController for attackController. It may seem odd to have a variable with the "type" of a class, rather than a primitive type (like float, boolean, or string) but it's actually very typical! We previously referred to classes as "complex variable types", but didn't explain why. If you think back to our other lessons, you might remember that classes had variables inside them (LessonTwoGame has seesPlayer, for example). The reason classes are referred to as "complex" variable types is that they can store *other variables inside them*. We therefore can have variables like movementController, that store *other* variables.

We'll demonstrate why this is important with an example. NPCMovementController handles (as you might guess from the name), an enemy's movement. It handles movement via setting a goal for the enemy and then calculating a path to get to the goal.[1] We'll use the function "SetTargetPositionGoal" as it takes in a GameObject and calculates a path to that GameObject's present position.

Go ahead and add the below line to Start:

```
movementController.SetTargetPositionGoal (player);
```

[1] You can check out a full list of publicly accessible functions in NPCMovement at the end of this lesson, though we'll go over most individually as they come up.

```
1  using UnityEngine;
2
3  public class Enemy : MonoBehaviour {
4      //A reference to this enemy's id (generated at start)
5      public string id;
6      //The speed this enemy will move at
7      private float speed = 3;
8      //The firing rate when the enemy is still
9      private float firingRate = 2;
10     //A reference to the GameObject of the player
11     public GameObject player;
12     //A reference to this enemy's NPCMovementController, which handles movement
13     public NPCMovementController movementController;
14     //A reference to this enemy's RangeAttackController, which handles attacking
15     public RangeAttackController attackController;
16
17     //Returns the speed of this enemy
18     public float GetSpeed(){
19         return speed;
20     }
21
22     //Returns the firing rate based on whether or not the enemy is moving
23     public float GetFiringRate(){
24         //Returns different values if the enemy is moving
25         if (movementController.moving) {
26             return firingRate * 0.5f;//Firing rate is half when moving
27         }
28         else {
29             return firingRate;
30         }
31     }
32
33     // Use this for initialization
34     void Start () {
35
36     }
37
38     // Update is called once per frame
39     void Update () {
40
41     }
42 }
```

Figure 4: Enemy.cs when it's first loaded into MonoDevelop.

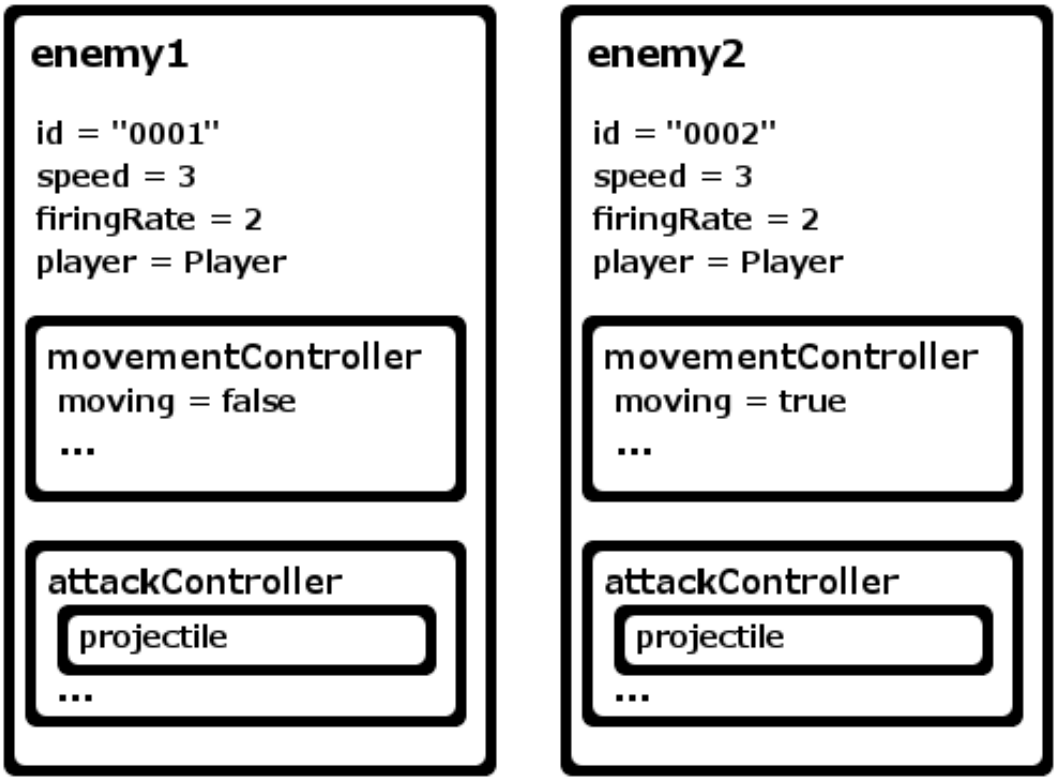

Figure 5: A diagram of two different Enemy objects during gameplay.

Play the game and wait a few moments without moving to find yourself surrounded by enemies.

Well that's something! But how is it working? We can actually do a small test to demonstrate.

Play the game again, but this time as soon as the game begins, move at least a few blocks away from the starting position.

The enemies didn't surround you that time, can you think why? The enemies instead surrounded the spot where you were to begin with. This is because the function call "SetTargetPositionGoal" was only ever called in Start, meaning that the function calculated a path for each enemy to the player's position *when Start was called*.

2.4 Class Variables and Functions

You may have wondered why we're called a function (SetTargetPositionGoal) by using a "class variable" (movementController.SetTargetPositionGoal) instead of just a class (NPCMovementController.SetTargetPositionGoal) as we have previously. This has a lot to do with the way that class variables work. We need to have different values stored in the different variables of movementCon-

troller. In the example that you just played through you probably saw the enemies taking very different paths to reach you and stopping at different times. That makes sense that they'd have to take different paths, given that they all start in different positions, right? They were able to store the value of this path, and follow it, as they could store *different* values!

Classes can have variables as we've seen before. A variable of a class type can actually store *different* values in the variables that it uses. That's how the enemies can have different paths. Take a look at Figure 5 which shows two different Enemy variables (the enemies you see in the game) during gameplay. Each enemy variable as variables of it's own, id, speed, and the GameObject "player" variable, which stores "Player" the actual object a player controls. These enemies are both "built" from the Enemy class that we've been modifying! It serves as their "blueprint" to use the same metaphor as we did with LessonTwo.

As you can see in Figure 5 not only do the two different enemies (enemy1 and enemy2 are their variable names) have different id values, but their movementController variables store different information as well. One is moving while the other is not!

This is the power of variables of a class-type (or "class variables). They will behave in the same way (have the same variables and functions), but can store very different values. That's why we call SetTargetPositionGoal with movementController and not with NPCMovementController. If we had to call that function with the class, instead of a class variable, then we could only store one of "path" value, and all the enemies would have to follow the same path.

Think of it of in terms of the "blueprint" metaphor we used in the previous lesson. A single "house" blueprint could be used to build multiple houses, as a single class can make multiple class variables. However, if we wanted to open a door on one of those houses, we wouldn't try to open the blueprint's door. Similarly, when we wanted to tell an enemy to compute a path to a goal, we don't tell NPCMovementController to do it, but an individual movementController variable. movementController is the house, NPCMovementController is the blueprint.

If you're still finding class variables to be confusing, continuing to fix the game should help in understanding what's going on.

2.5 OPTIONAL: Using Unity to Show the difference between objects

You can actually see these different values in Unity as well. If you play the game again, and then use "Escape" to get the mouse and then press the "pause" instead of the "play" button, you can see "0000" to "0009" in the "hierarchy" view. These are the enemy variables in Unity! If you click on them the "Inspector" view will show something that looks like 6. You can un-click the play button to get out of this mode. We'll use this way of interacting with Unity later, but for now it's just useful as a way to see the differences between the different enemies.

2.6 Making Enemies Move Intelligently

So far we've managed to make the enemies move all to the exact same point just once, after which they will again stay in place forever. That's not particularly useful. In the last section we were able to write what were essentially "rules" for how a door should act. Let's see if we can do the same here!

To begin with, let's see if we can just get the enemies to continually follow the player. To do that we'll need two things: the "ReachedGoal" function of movementController and an if statement. The "ReachedGoal" function returns a boolean value representing whether the enemy has reached its goal (true) or not (false). That means it has a "return type" of boolean. We can use this fact to

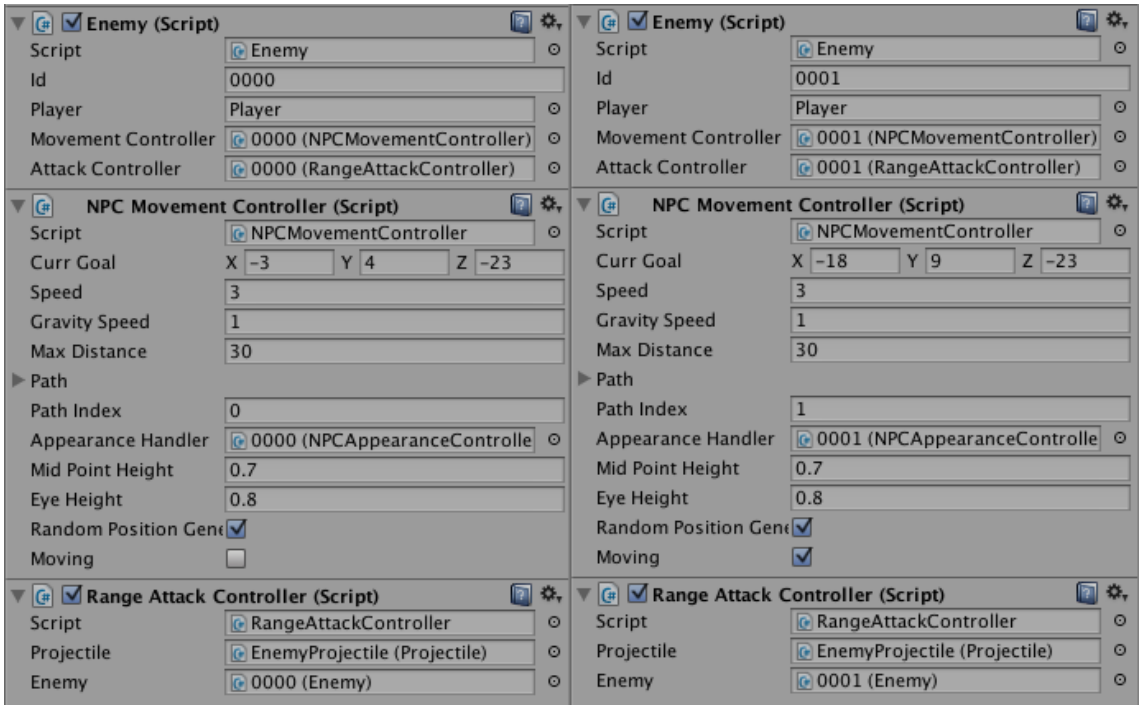

Figure 6: Unity's display of the public Enemy variables for "0000" and "0001".

determine when we need to use "SetPlayerPositionGoal" again. We'll want to put this new check and call in Update, so it occurs more times than just once at the start of the game. Place the following lines of code into Update:

```
bool reachedGoal = movementController.ReachedGoal();
if(reachedGoal){
        movementController.SetTargetPositionGoal (player);
}
```

We can't just call "movementController.SetTargetPositionGoal (player);" in Update as then the enemy would just continually recalculate a path, but never actually move along it. It also takes a lot of time to calculate a path and this would slow down the game.

Play the game again with the code added and saved in Enemy's Update. It may take some time (as they'll have to get to the starting position of the player before they'll plan out a new path), but eventually you'll have a small crowd of enemies following behind you.

You've done it! You've successfully managed to make the enemies follow you. Note that this Update code is being called in *each* Enemy individually as each of these enemies represents a different Enemy class variable. That's a lot to wrap your head around! But let's show you that it's working in game.

Play the game again. This time pay attention to the enemies at the "back" of the line following you as you move around. Try to see the moment they stop going to a different location and turn to where you are. That's the moment that "movementController.ReachedGoal()" is set to true as the enemy reached their last goal, and thus "movementController.SetPlayerPositionGoal (player);" is called and they head to your current location.

This shows you that this Update code is being called in each different enemy individually, as the moment they turn to face you, when "movementController.ReachedGoal()" returns true is different for each one! That's the trouble with writing good enemy behavior in code, it has to be able to work for different enemies in different situations!

However, this isn't particularly good enemy behavior. First of all, having the enemies immediately hone in on you seems pretty unfair. We probably want it so an enemy only chases the player if they can see the player. Can you think of a way to represent in code whether or not an enemy can see a player? Some kind of variable that could store whether or not something was true?

The answer is to add a new boolean variable to track whether or not the enemy can see the player. It'll need to have global scope (defined outside of a function in Enemy), can you think why? Let's name the variable "seesPlayer" with an initial value of false. Define it above the GetSpeed function.

Alright now we have a new variable, but that's not really particularly useful on its own. In trying to determine what to do next let's start with the effect we want to achieve and what we have so far, that's a typical approach for programmers in trying to determine what code to write. We know that we want the enemy to "chase" the player (as it has already) if it can see the player. Therefore it might make sense to go ahead and put the code in Update that we have so far (that we know makes the enemy chase the player) into an if statement with the "seesPlayer" variable as the condition. When you're done the script should look like Figure 7.

But we still don't quite have the behavior we want. For one, "seesPlayer" never gets set to true. Secondly, we haven't yet decided what an enemy should do *before* its seen a player (the code in Start still sends the enemy to the player's position). We could have it do nothing, so an enemy would just sit still until it saw the player. But then a ghost may never move! That's probably not

```
3  public class Enemy : MonoBehaviour {
4      //A reference to this enemy's id (generated at start)
5      public string id;
6      //The speed this enemy will move at
7      private float speed = 3;
8      //The firing rate when the enemy is still
9      private float firingRate = 2;
10     //A reference to the GameObject of the player
11     public GameObject player;
12     //A reference to this enemy's NPCMovementController, which handles movement
13     public NPCMovementController movementController;
14     //A reference to this enemy's RangeAttackController, which handles attacking
15     public RangeAttackController attackController;
16
17     bool seesPlayer = false;
18
19     //Returns the speed of this enemy
20     public float GetSpeed(){
21         return speed;
22     }
23
24     //Returns the firing rate based on whether or not the enemy is moving
25     public float GetFiringRate(){
26         //Returns different values if the enemy is moving
27         if (movementController.moving) {
28             return firingRate * 0.5f;//Firing rate is half when moving
29         }
30         else {
31             return firingRate;
32         }
33     }
34
35     // Use this for initialization
36     void Start () {
37         movementController.SetTargetPositionGoal (player);
38     }
39
40     // Update is called once per frame
41     void Update () {
42         if (seesPlayer) {
43             bool reachedGoal = movementController.ReachedGoal();
44             if(reachedGoal){
45                 movementController.SetTargetPositionGoal(player);
46             }
47         }
48     }
49 }
```

Figure 7: The class Enemy with the seesPlayer variable and additional if statement.

```
40      // Update is called once per frame
41      void Update () {
42          if (seesPlayer) {
43              bool reachedGoal = movementController.ReachedGoal ();
44              if (reachedGoal) {
45                  movementController.SetTargetPositionGoal (player);
46              }
47          }
48          else {
49              bool reachedGoal = movementController.ReachedGoal();
50              if(reachedGoal){
51                  movementController.SetRandomGoal();
52              }
53          }
54      }
```

Figure 8: Update of Enemy.cs after adding everything needed to keep the enemies constantly moving to random positions.

the best idea. How about instead we have the enemies move randomly across the island "searching" for the player?

Let's go ahead and add in the functionality to have the enemies move around the island randomly at first, even if they won't "switch" to seeing the player at this point. The function we'll want to use for this is "SetRandomGoal" of movementController. It works much in the same way as "SetTargetPositionGoal" except that it doesn't have any arguments. Replace the line in Start with:

```
movementController.SetRandomGoal();
```

This will cause each enemy to create an initial path to a random point (each point will be unique to each enemy), instead of to the player.

Play the game, and note that changed enemy behavior.

As you saw, enemies now move to a random position and then come to a stop. This is similar to how the enemies initially acted when we only had the "SetPlayerPositionGoal" line in Start. This should make sense if you think about it, as the only other code in Enemy.cs is in Update, and none of it will run as "seesPlayer" is not ever set to true (at the moment). However, what we'd like to have happen is for the enemy to keep moving to random positions until it sees the player. Can you think of how to keep the enemies moving to random positions if seesPlayer isn't true? Maybe if we say that we'll need to use an "else statement" in Update? And that you can use the code to have enemies chase the player as an example?

We'll need an else statement to handle when the enemy can't see the player. We'll then need to nearly replicate the code within the if statement to cause the enemy to find a new random path after it reaches its current goal. The difference is we'll need to call "SetRandomGoal" instead of "SetPlayerPositionGoal". When you're done, your Update function should look like Figure 8.

Play the game and note that changed enemy behavior.

Now you've got enemies moving everywhere! The last thing we're missing is figuring out when the enemy sees the player. To do that we'll make use of the "CanSeeTarget" function of movementController. CanSeeTarget will return a boolean value if the movementController in question can "see" the passed in GameObject. This is exactly what we need for updating the "seesPlayer" variable! We can use the below line in Update:

```
seesPlayer = movementController.CanSeeTarget(player);
```

Insert the "movementController.CanSeeTarget" line above the if statement in Update and play the game. Try and see if you can get an enemy to first start following you then "lose track" of you. As a warning, this will be very difficult.

It's possible to get the behavior where an enemy starts "chasing" you, but then "loses track" of the player. But it's actually really tough to tell! Can you think why that might be? Let's walk through a possible example to try to illustrate what's happening here.

Check out Figure 9 for an illustration of four different moments with just a single enemy to explain what's happening with the code at this stage. Before Start, the enemy has no path as it was just created (1).
After Start, the enemy has a path to a random point and begins to travel along it (2).
While traveling along the path, the ghost might see the player (seesPlayer will now be equal to true) (3).
But the ghost isn't done with its path (ReachedGoal would not return true) so it continues along it until the path is done (4).
However, the ghost cannot "see" the player in (4), so seesPlayer will be false. Therefore when it reaches the end of its path, it will pick a random goal instead of choosing the player as the goal (it will run the else statement since seesPlayer is false, instead of running the if statement).

That's no good! What we need is something where as soon as the enemy sees the player, it begins to chase the player. Essentially, we need to be able to set the enemy's goal to the player when it sees the player after previously *not seeing the player*. What we need to know is when seesPlayer switches from false to true.

There's a couple different ways to solve this. We could change where we call "CanSeeTarget" to be within the else statement. If we did that then we'd know that seesPlayer was false (so we got into the else statement), and then if seesPlayer became true we'd know it had just switched from false to true. However, that mean that we'd have run the other code in the else statement when that wasn't necessary. We'd essentially have had the enemy acting like it hadn't seen the player that frame even though it had!

To avoid running unnecessary code we'll use the "not" logical operator (!). As you might recall from the last section, logical operators work on conditions like how mathematical operators (+, -, etc) work on numbers. We'll start by storing the return value of "CanSeeTarget" into a temporary variable, let's call it "nowSees". Replace the "movementController.CanSeeTarget" line at the top of Update with the following two lines (they are separated on purpose, as we'll explain):

```
bool nowSees = movementController.CanSeeTarget(player);

seesPlayer = nowSees;
```

What we want is when nowSees is true *and* seesPlayer is false (before seesPlayer is set to nowSees). As that will mean that previously (as in the last time Update was called) the enemy couldn't see the player, but now it can. To say that a bit more formally we want a condition where

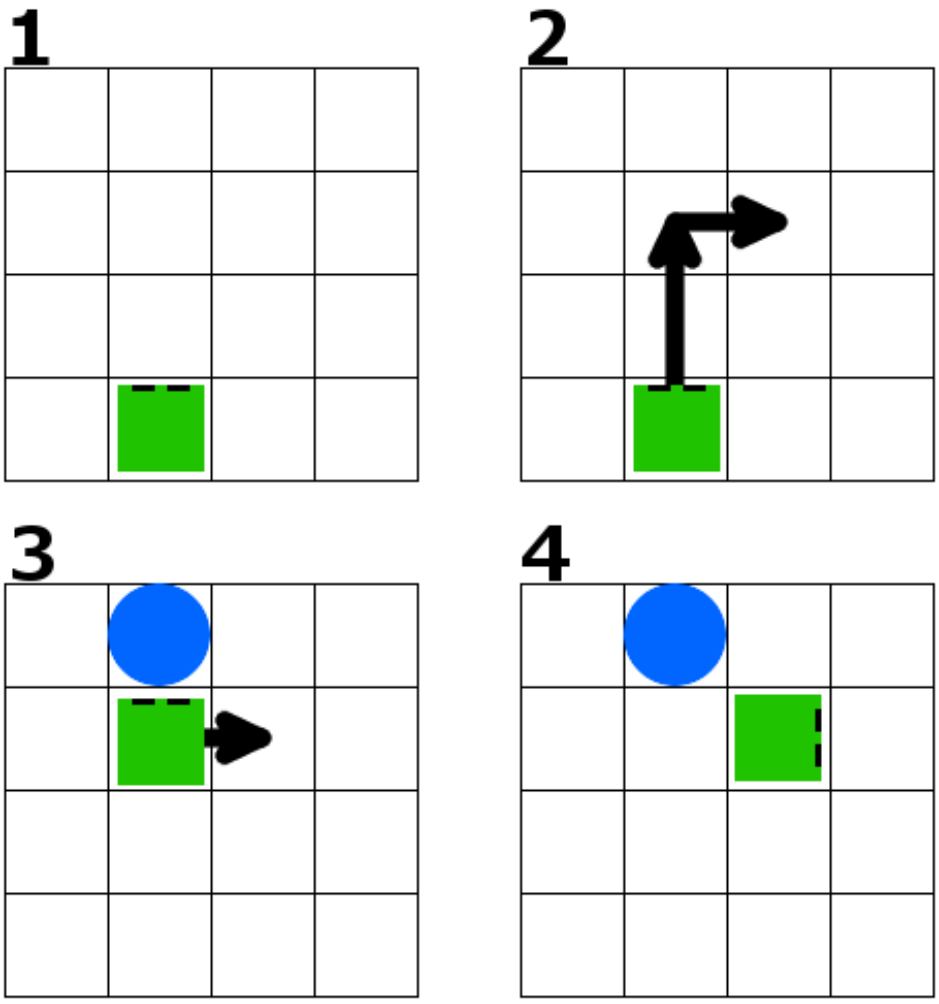

Figure 9: Example of how the current Enemy.cs script works. The green square is a ghost, the arrows are its path, and the blue circle is the player.

```
// Update is called once per frame
void Update () {
    bool nowSees = movementController.CanSeeTarget (player);
    if (nowSees && ! seesPlayer) {
        movementController.SetTargetPositionGoal(player);
    }
    seesPlayer = nowSees;

    if (seesPlayer) {
        bool reachedGoal = movementController.ReachedGoal ();
        if (reachedGoal) {
            movementController.SetTargetPositionGoal (player);
        }
    }
    else {
        bool reachedGoal = movementController.ReachedGoal();
        if(reachedGoal){
            movementController.SetRandomGoal();
        }
    }
}
```

Figure 10: Update of Enemy.cs after adding everything needed to have enemies search for then track the player.

nowSees is true and seesPlayer is not true. If we translate that into code we get the following condition:

```
nowSees && ! seesPlayer
```

This condition will be true when nowSees is true *and* seesPlayer is *not* true. If we put that condition into an if statement, then we can be sure the code in the if statement will run only when the enemy now can see the player, but could not see the player previously. Inside this if statement we could then put a call to "movementController.SetPlayerPositionGoal (player);" to make the movementController immediately calculate a path to the player and begin to follow that path. When you're done your Update function should look something like Figure 10.

Play the game and get some enemies chasing you. It should be much easier this time!

And with that you've done it! You've got the enemy moving intelligently. At this stage you could try to add additional behavior to make the enemy more intelligent. For example, you could add a special behavior in for when the enemy loses sight of the player (when seesPlayer is true and nowSees is false), perhaps having the enemy go ahead and move to the player's current location as that's close to where the enemy had last "seen" the player? That's up to you!

It might be surprising, but a lot of AAA games make use of rules-based enemy artificial in-

```
// Update is called once per frame
void Update () {
    bool nowSees = movementController.CanSeeTarget (player);
    if (nowSees && ! seesPlayer) {
        movementController.SetTargetPositionGoal(player);
    }
    seesPlayer = nowSees;

    if (seesPlayer) {
        if (movementController.ReachedGoal ()) {
            movementController.SetTargetPositionGoal (player);
        }
    }
    else {
        if(movementController.ReachedGoal ()){
            movementController.SetRandomGoal();
        }
    }
}
```

Figure 11: Update of Enemy.cs after removing the unnecessary "reachedGoal" variable.

telligence very similar to this! Of course, at the moment the ghosts running around aren't really "enemies" in that they make no attempt to attack the player. We'll change that though!

2.7 Cleaning up Code

Before making the enemies attack the player, let's take a moment to clean up the code we have so far.

If you'll recall, we don't actually need to use a boolean variable as a condition in an if statement. In the previous section we put a direct boolean value (true and false) inside an if statement.

Using this information it should be pretty easy to see that we don't need to use either "reachedGoal". We can simply use the fact that "ReachedGoal" returns a boolean value to place the call to that *inside* the if statement! Go ahead and remove both the "reachedGoal" variables from Update and just used the "ReachedGoal" function instead. When you're done Update should look like Figure 11. We can do the same thing with two less lines of code, not bad!

We'll continue to use returned values directly from functions (instead of storing them in unnecessary variables) in the next section.

2.8 Making the Enemies Attack!

We used movementController in order to control Enemy movement. As you might guess, we'll be using attackController to handle attacking. Go ahead and insert the following line near the top of

Update:

```
attackController.Fire(player);
```

This will cause the enemy to shoot off a projectile at the passed in GameObject (player). Let's see what the effect of this is!

Go ahead and play the game. Notice the behavior of the enemy's and the projectiles they shoot almost constantly.

This behavior isn't particularly intelligent, is it? Since the enemies just fire constantly it tends to leave stray shots firing in every direction. One simple change we could make is to have the enemies only shoot if they can currently see the player. Move the call to "Fire" to inside the "seesPlayer" if statement.

Move the "Fire" line to only be called when the enemy can "see" the player. Play the game, and see what effect this has!

This behavior is definitely improved, but its still not perfect. The biggest problem seems to be that the projectiles "explode" before hitting the player. That's because these projectiles are being shot when the player is out of range!

We can get the projectiles range by using attackController's "GetProjectileRange", which will return the float value for the max distance the projectile can travel. We can then use movementController's "GetDistanceToTarget", which takes in a reference to a GameObject, and calculates the distance between it and the enemy. These two functions together will help us change the enemy's behavior to only shoot when the player is in range.

Definition!

RELATIONAL OPERATOR:
A symbol or symbols that represents a boolean relationship between two values.

What we want is to only shoot (have Fire be called) if the distance between the player and enemy (GetDistanceToTarget) is less than the distance the projectile can fly (GetProjectileRange). That phrasing should make it sound like an if statement might be useful. However, we'll have to use something new called a **relational operator**. A relational operator is a symbol or symbols that represents a boolean relationship between two values. For examples there's ">" which is the greater than symbol. In code if we had a line:

```
bool test =  4 > 2;
```

Then test would store a value of true. Since it's "true" that 4 is greater than 2. You may have run into this symbol and it's opposite less than (<) in a math class. They work in code in a very similar way, as you can guess! For example if we used less than instead of greater than:

```
bool test =  4 < 2;
```

Then test would store a value of false.

But how is this useful to you? Well remember the phrasing we used earlier: "if the distance between the player and enemy (GetDistanceToTarget) is less than (GetProjectileRange)". Now that you know about relational operators, you should see that we can make use of them to check for this exact condition and to only "Fire" when it is true. Replace the line that calls "Fire" (inside the seesPlayer if statement) with the following lines (as a note, both GetDistanceToTarget and attackController should be on the same line in update, we just can't fit those on the same line here):

```
bool inRange = movementController.GetDistanceToTarget(player)
         < attackController.GetProjectileRange();
if(inRange){
        attackController.Fire(player);
}
```

```
void Update () {
    bool nowSees = movementController.CanSeeTarget (player);
    if (nowSees && ! seesPlayer) {
        movementController.SetTargetPositionGoal(player);
    }
    seesPlayer = nowSees;

    if (seesPlayer) {
        if(movementController.GetDistanceToTarget(player)<attackController.GetProjectileRange()){
            attackController.Fire(player);
        }
        if (movementController.ReachedGoal ()) {
            movementController.SetTargetPositionGoal (player);
        }
    }
    else {
        if(movementController.ReachedGoal ()){
            movementController.SetRandomGoal();
        }
    }
}
```

Figure 12: Update with Fire now only firing when it makes sense to do so.

Play the game, and notice the effect of these changes!

Does what's happening make sense? Now Fire will only be called when the player is closer to the enemy than that enemy's max range. This doesn't mean the enemy will always hit, but it does mean that the enemy won't just shoot wildly. And that looks more intelligent!

Remember that you don't need to put an variable into an if statement, just a boolean value. That means we can actually don't need inRange, we can put "movementController. GetDistanceToTarget(player) < attackController.GetProjectileRange()" directly into the if statement! Check out Figure 12 for how Update should end up looking.

You've done it! Now you have an enemy that moves and shoots at least somewhat intelligently! At this point you could add more if statements to make it behave more intelligently on your own. Think about the condition that you want to create (for example: if the player is within range, the enemy should always move towards the player as if it heard it), then translate that into code (for example: if(movementController.GetDistanceToTarget(player) < attackController.GetProjectileRange())), and determine where in Update it makes sense to put this if statement. Feel free to add as many behaviors as you like, see how smart of an enemy you can make!

2.9 Creating End Conditions

You may have already gone and "destroyed" all the enemies in the game and been left to nothing to do but stop playing. You may have also noticed that no matter how many of the projectiles hit you, you can never lose! If not its worth noting at this point that there is no way to win or lose the game! For both we'll make use of brand new scripts and we'll need to make use of the relation operators as well.

2.10 Losing

We'll start with making it possible to lose. To do that we'll want to open up a new script "Projectile.cs". As you can probably guess, this script handles the behavior for projectiles. It can be found in the Codebase folder, and then within the NPC folder. Double click "Projectile" to open it up.

There's a lot going on in the "Projectile" class! Feel free to take a deeper look at it, but the function we need is called "HitPlayer" and it's right at the top of the class. In this function we can specify what should happen when a projectile hits the player. Right now all that happens is that the projectile is destroyed via the line "Destroy (gameObject);".

For now, let's go ahead and make the player lose if they get hit by just one shot (we can make this less difficult later). We can end the game by calling the line:

```
GameManager.EndGame ("You Lose!");
```

EndGame will immediately stop play and display whatever string value is passed to it. Therefore, we can use it for both winning and losing! Go ahead and insert that line somewhere *before* the line that destroys the projectile in HitPlayer.

Play the game! You'll almost immediately see that this is an unreasonable level of difficulty.

Because the enemies move randomly, it can be only seconds before the first time the player is hit by a projectile. Just one shot isn't even enough time to act!

Many games use a "hitpoints" system. Hitpoints typically represent a player or enemy's health, and the player or enemy doesn't die till they lose all of their hitpoints. Let's see if we can use something like that here! To do that, we'll actually need to introduce a new variable to track the player's hitpoints. Can you think of where might be a good place to store this value?

Open PlayerInfo.cs back up and add a new integer variable named "hitpoints" (there's no need for decimal points here). We'll store this information here since that's where the rest of the player information is, and we have access to it inside HitPlayer (since PlayerInfo playerInfo gets passed into the function). Since we'll want to be able to both get and alter the value of this variable, it will need to be public. Set its initial value to something high at first, perhaps a hundred? In total the line should look like:

```
public int hitpoints = 100;
```

Add that to PlayerInfo then return to Projectile.cs.

What we want now is to have the value of hitpoints get lower every time a Projectile hits the player. Remember how we can alter the value of a variable as we did in LessonOne, we'll want to use a line like:

```
playerInfo.hitpoints = playerInfo.hitpoints -1;
```

This will change the value of hitpoints stored in playerInfo to be one less every time the player is hit by a projectile. We used a similar technique in LessonOne. It works because the right side of the equals sign is evaluated first, meaning that it gets the value of playerInfo.hitpoints and subtracts one from it. Then the equals sign stores that new value into the same variable. This has the effect of making playerInfo.hitpoints go down by one every time this line is called!

The last thing we'll want to do is add a call to EndGame with "You Lose!" (feel free to change the text used) only if hitpoints is less than 0. Translating that statement into code we get:

```
if (playerInfo.hitpoints < 0) {
```

```
//This function is called when the player is hit
private void HitPlayer(PlayerInfo playerInfo){
    //TODO; What should happen when you hit the player?
    playerInfo.hitpoints = playerInfo.hitpoints -1;
    if (playerInfo.hitpoints < 0) {
        GameManager.EndGame ("You Lose!");
    }
    //Destroy the projectile
    Destroy (gameObject);
}
```

Figure 13: HitPlayer when it is complete.

```
        GameManager.EndGame ("You Lose!");
}
```

Insert both the line changing hitpoints value and the new condition into HitPlayer and play the game (your HitPlayer should now look like Figure 13. If you're willing to wait a very long time, you may still be able to die, but a hundred projectiles is a lot of projectiles.

You've done it! You've created a lose condition! Feel free to alter the starting value of hitpoints to make it a little bit more challenging, and avoiding ghosts more important. Try values below 10 or so for a pretty challenging experience! While you're in Projectile.cs feel free to change the "projectileSpeed" or "maxLifeTime" variables as well, to make the projectiles easier or harder to dodge. Play around with it!

2.11 Winning

Last but not least, we have to create a win condition. For this we'll need to open the last of the scripts we'll be accessing in this lesson Sword.cs. Sword.cs can be found in the "Codebase" folder and then within the "Items" folder.

While Sword mostly looks like the code we've seen before, for the first time we have a class that doesn't "inherit from MonoBehavior" (: MonoBehavior) instead it inherits from "Item". This allows the "Use" function be called when the item is held and used. But what we really care about is the HitEnemy function. Currently all it does (as you can see) is destroy the Enemy on a hit "Destroy (enemy.gameObject);".

What sort of win condition should we have? Maybe something like if the player has destroyed a certain number of enemies? There are ten enemies in the game, but to begin with let's say the player need only defeat 1 enemy to win. So let's go ahead and put in a new call to "GameManager.EndGame" but this time with text related to winning:

```
GameManager.EndGame ("You Win!");
```

Insert the EndGame line into HitEnemy and play the game. Destroying just a single enemy seems a bit easy, doesn't it?

```
int enemyCount = 0;

//This method is called when the sword hits an enemy
private void HitEnemy(Enemy enemy){
    Destroy (enemy.gameObject);

    //TODO; Add code here for handling the win condition
    enemyCount = enemyCount + 1;

    if (enemyCount > 9) {
        GameManager.EndGame ("You Win!");
    }
}
```

Figure 14: HitEnemy when it is complete.

What if the player instead had to kill many of the ghosts, say all ten to win? That sounds like it's a better win condition, but how could we represent that in code? Well we'll need some way to store the value of how many enemies have been destroyed so far. To do that, we'll need to add a new variable to Sword with global scope. Since we're counting something, we have no need of decimals so we can use an integer value. Let's call it "enemyCount" and it'll need to have an initial value of 0 so we count up. We don't need to access this variable outside of Sword.cs so feel free to make it private, or simply not specifying an access modifier.

```
int enemyCount = 0;
```

Add that line to above HitEnemy. We'll also need to add one to it every time HitEnemy is called. Within HitEnemy we'll need the line:

```
enemyCount = enemyCount +1;
```

Once we've done that, let's figure out what the condition can be before the Win message should be called. It should be something like: enemyCount is greater than nine (since there are ten ghosts we want to be destroyed). To translate into code we'd use something like:

```
if (enemyCount > 9) {
        GameManager.EndGame ("You Win!");
}
```

Add everything to HitEnemy such that it ends up looking like Figure 14. Play through the game and try to get all the enemies!

And that's it! You've created a win condition! There's other things you could add here. For example, you could of course add hitpoints to enemies as well, decrement the enemy hitpoints in HitEnemy until they fall too low then call "Destroy (enemy.gameObject);". But that's up to you! You have a full, working "beat 'em up" game.

2.12 End of Lesson Three

That's all we have for Lesson Three! Feel free to play with your game more, altering the speeds and behaviors of the various pieces till you're happy with them. In this lesson you learned how to:

1. Use access modifiers (private/public) to restrict access
2. Make use of functions that "return" values
3. Create complex enemy behavior with if statements
4. Call on class variables to run similar behavior on individual objects
5. How to use relational operators to get boolean values from numbers

In the next lesson we'll focus on the actually building up and creating your own unique world to play a simple version of an adventure game.

2.13 movementController Functions

Function Name	Arguments	Behavior	Return Type
SetTargetPositionGoal	GameObject value of the target	Sets the enemy's goal to be the target's current position.	[Not Any]
GetDistanceToTarget	GameObject value of the target	Calculates distance between the target and this enemy.	Returns a float value of the distance between the target and this enemy.
CanSeeTarget	GameObject value of the target	Calculates whether or not the enemy can see the target GameObject.	Returns a boolean value, true if can be seen, false otherwise
SetRandomGoal	[Not Any]	Calculates a random position and sets the enemy's current goal to it.	[Not Any]
ReachedGoal	[Not Any]	Determines whether or not the enemy has reached its current goal.	Returns a boolean value, true if the enemy has reached the goal, false otherwise.
Stop	[Not Any]	Immediately stops the enemy and sets its goal to its current position.	[Not Any]

2.14 attackController Functions

Function Name	Arguments	Behavior	Return Value
GetProjectileRange	[Not Any]	Calculates the largest possible distance a projectile could travel	Returns a float value of max projectile distance.
Fire	GameObject value for target	Creates and fires a projectile at the target, if it can.	[Not Any]

2.15 Lesson Three Glossary

Vocabulary Word	Definition	Examples
access modifier	A keyword that limits what other code can "see" a variable, function, or class.	public and private
public	The access modifier that allows access of a class, function, or variable outside of the class where its defined.	public float GetSpeed()
private	The access modifier that doesn't allow access of a class, function, or variable outside of the class where its defined..	private float speed = 2.0f;
return	A keyword which "returns" a value from a function to the line of code which called the function.	return speed;
return type	A keyword which "returns" a value from a function to the line of code which called the function.	void
getter	A slang term for a type of function that is publicly accessible and returns the value of some private variable	GetSpeed
relational operator	A symbol or symbols that represents a boolean relationship between two values.	> and <

Lesson Four

Starting an Adventure!

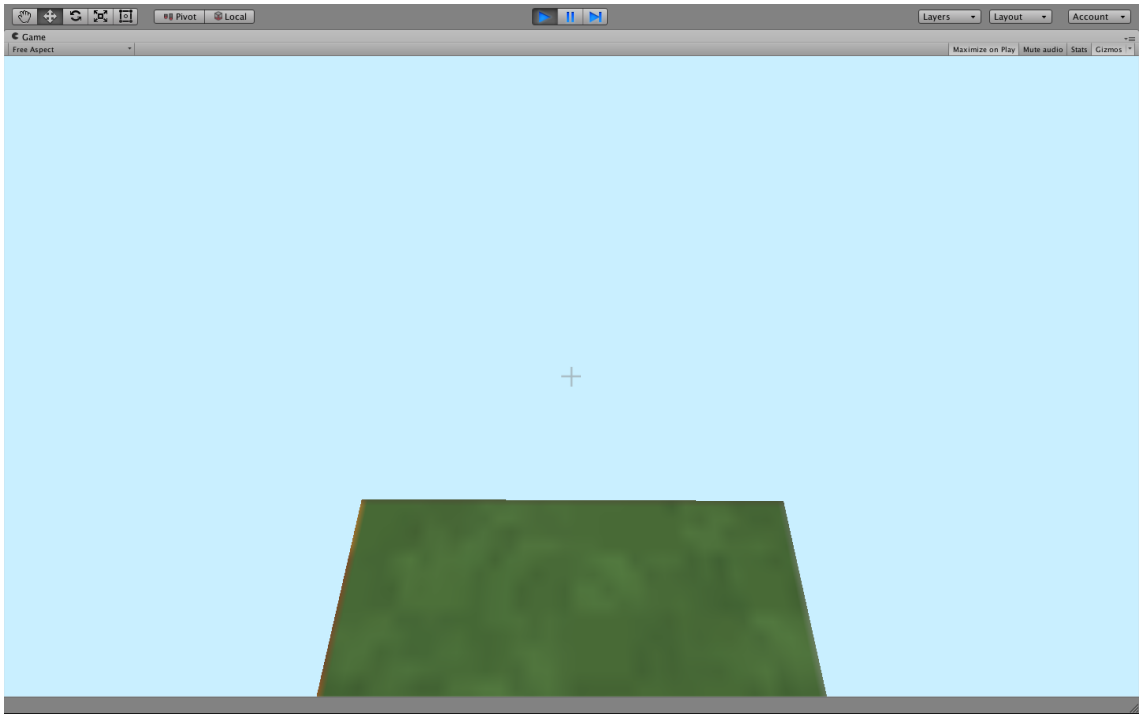

Figure 1: How LessonFour-Six begins.

1 Introduction to Lesson Four: Starting an Adventure!

Adventure games like the Zelda series feature a protagonist going on a journey through a virtual world. These adventures tend to involve fighting monsters, finding treasure, and ultimately saving the day. They are also notoriously difficult to make, due to all the effort it takes to create something a player will see as a "real" world.

For the next three lessons we'll be ditching the "fixing broken games" format and instead making a mini Adventure game from scratch. In this lesson we'll focus on building up a meaningful virtual world and teach a powerful new programming technique: loops. The theme of this lesson will be doing more and more with less and less code, and we'll be using loops to do that.

2 Lesson Four: Literal Worldbuilding

First we should go ahead and open up the scene we'll be using throughout Lessons four, five, and six. Open Unity back up, and you'll find the scene file titled "LessonFour-Six" in the Lessons folder. Double click to open it.

Click the "play" button at the top of Unity to check out what we have so far (not much). Remember you can press the Escape or Backspace Key to get your mouse back so you can stop the game by "unclicking" the play button.

When you boot up the "game" as it exists you should find yourself standing on a single block

```
using UnityEngine;

public class AdventureGameGenerator : Generator {

    //This function is called to generate out the map
    public override void GenerateMap (){
        //This function call creates the single block that exists in the game so far
        MapBuilderHelper.BuildBlock ("Grass", 0, 0, 0);
    }
}
```

Figure 2: How LessonFour-Six begins.

suspended in the sky (like in Figure 1). This isn't really a sufficient "world" for an adventure, now is it? Our first task then should be pretty obvious. We need to build up a world for this adventure to happen inside. That's the task that will take much of this lesson, we'll be building up "floating islands" as the setting for this adventure game.

If you'll remember from lesson one, you can place blocks by right clicking. We could (in theory) build up the whole world by just hand. You might already recognize how time consuming this would be, but let's try to build up a small platform just to get a sense of how long it would take.

Play the game again. Try to get to the edge of the platform and "place" a block by right clicking onto the "side" of the starting block. Try to place at least a few blocks. It's easier to jump and place a block below you, but we want to make a wide platform that can be walked around on.

This approach is not only dangerous (with the risk of falling), but it's also time consuming! Placing a single block probably took you at least a few seconds, and at that rate building up even a "small" world like the cave in lesson two could take half an hour!

You might be thinking there's a better way, and there is! Once again, we need to do some programming.

2.1 AdventureGameGenerator

Let's go ahead and open the the script we'll be working with in this lesson "AdventureGameGenerator". You can find AdventureGameGenerator by opening the following folders "Codebase", "Environment", "Map", and then "Generators". Double click "AdventureGameGenerator" to open it. In the Generator folder you can see "Generator" files from other lessons.

For this codebase, the "Generator" scripts create the world of the game. Previously, we've written these scripts for you. But in this lesson you'll be doing it yourself.

Once MonoDevelop opens AdventureGameGenerator, it should look something like Figure 2. This script looks similar to many we've seen before, but its worth noting that instead of "MonoBehavior" the class inherits from a class named Generator ("AdventureGameGenerator *: Generator*"), as the Generator-type script is what makes a game world. There's a single function "GenerateMap" in the class that is called to actually (as you might guess) generate out the map. It is publicly accessible (it uses the "public" accessibility modifier) so that it can be called outside of "AdventureGameGenerator". It also has a new keyword "override" we'll get to that more in a later lesson. For now just know that this keyword means AdventureGameGenerator's GenerateMap is called by the codebase to create the world of the game.

As you saw when you played the game, the only thing presently getting created is a single

"Grass" block at location 0, 0, 0 (just before where the player is). This corresponds to the only line in GenerateMap: "MapBuilderHelper.BuildBlock ("Grass", 0, 0, 0);". "MapBuilderHelper" is a class that helps to build the map. "BuildBlock" is a special function set up to be called by the class instead of by a class variable (since we won't need more than one "MapBuilderHelper"). The first argument is a string value that determines the type of block with the next three variables corresponding to the x, y, and z position of that block in space. To demonstrate, let's add a second line to GenerateMap.[1]

```
MapBuilderHelper.BuildBlock ("Grass", 1, 0, 0);
```

So this code should create a "Grass" block at position (1, 0, 0). Can you think of where that will be placed in terms of the first block? Remember that x-values handle the left (negative x values), and right (positive x values) position of an object in 3D space, y-values similarly handle down and up position, and z-values handle forward and backward position. With this information we should see that this new block will be placed to the right (as 1 is a positive value) of the other block.

After adding the above line to GenerateMap play the game again, and note the difference!

It worked! We successfully got a second block to show up.

You might be wondering at this point if the plan is to call "MapBuilderHelper.BuildBlock" for every block we want placed in the game. That'd be a lot of function calls, and a lot of lines of code. To avoid that, we'll be using a new structure in code called a **loop**.

Definition!

LOOP:
A programming structure that repeatedly calls the same chunk of code.

2.2 Basic Loops: While Loop

A loop is a type of structure that is able to run the same lines of code over and over again. There are a variety of different types of loops, but we'll be focusing on one of the more simple varieties referred to as a **while loop**. A while loop's structure looks like the following:

Definition!

WHILE LOOP:
A simple kind of loop that repeats a chunk of code *while* some condition is true.

```
while(Condition){
  DoSomething1();
  DoSomething2();
}
```

As with an if statement a while loop has a "condition" that determines if the code within its curly brackets will run. However, unlike an if statement whose code will run just once *if* its condition is true, a while loop will run the code over and over again *while* the condition is true. Therefore in the example above calls to the functions DoSomething1 and DoSomething2 would repeat as long as "Condition" was true. So this code would run DoSomething1(), then DoSomething2(), then DoSomething1(), then DoSomething2(), etc. Why this is useful may not be immediately obvious, so let's put together an example.

Let's insert a working example into the code. Replace the current code inside "GenerateMap" with the following where we use a while loop to repeatedly call some code, including the "Build-Block" function. We'll walk through this code in a bit more detail once you play the game.

```
int blocksPlaced = 0;
while(blocksPlaced < 5){
  MapBuilderHelper.BuildBlock ("Grass", 0, 0, 0);
  blocksPlaced=blocksPlaced+1;
```

[1]Note: You can find a list of all the usable types of blocks (beyond "Grass") at the end of the lesson.

```
}
```

Play the game!

But wait, this had the opposite effect of what we wanted! The only block is the Dirt block located at (0, 0, 0). That's a step backward, since we went from two blocks to one. Let's look over the lines inside GenerateMap to try to see what's going on here.

First, we define a new integer variable "int blocksPlaced = 0;". Second, we have the start of the while loop, with its condition "blocksPlaced < 5". This condition translates to: "blocksPlaced less than five". So the while loop will run while that is true. Third, we have the call to place a Grass block at position (0,0,0): "MapBuilderHelper.BuildBlock ("Grass", 0, 0, 0);". Fourth we increase the value of blocksPlaced by one, this is very important as otherwise we will never hit the **terminal condition**.

Definition!

TERMINAL CONDITIONAL: The condition which will lead a while loop to stop repeating.

The terminal condition is the condition that causes the while loop to stop repeating. Essentially its when the condition that keeps the while loop going is false. For example for "blocksPlaced < 5" the terminal condition is when blocksPlaced is equal to 5. If there exists a while loop where the terminal condition cannot be reached, then it will continue forever. That will have the effect of causing the game to be stuck repeating a while loop, making it look like the program (in this case Unity) has frozen. You may have seen this behavior before in other programs! Feel free to test this by deleting the line "blocksPlaced=blocksPlaced+1;"! But be prepared to force quit Unity, and possibly even restart your computer. As you can see, making sure the terminal condition is reached is very important.

Given that we only ever call "MapBuilderHelper.BuildBlock ("Grass", 0, 0, 0);" repeatedly, it makes sense that we only see a grass block at position (0,0,0). The code just places the same block to the same position repeatedly. We can fix this by changing the BuildBlock call to instead "change" where it places the blocks. Replace the call to BuildBlock in the while loop with:

```
MapBuilderHelper.BuildBlock ("Grass", blocksPlaced, 0, 0);
```

Play the game! Take a moment to count how many blocks are now floating in the air.

2.3 Walking through the While Loop

As you counted for yourself, there were five blocks floating in the air instead of just one. They spread out to the right as blocksPlaced was passed in to the "x" position and was increasing. That still might be confusing so let's take a moment to explain what's going on. For reference, your GenerateMap function should look like Figure 4 at this point. To show what's going on let's walk through what occurs when GenerateMap is called:

1. GenerateMap starts with "int blocksPlaced = 0;", which creates an integer variable called "blocksPlaced" and stores the value "0" in it.

2. The next line of GenerateMap is the beginning of the while loop. It begins by checking if the condition "blocksPlaced<5" is true. Since blocksPlaced currently stores 0 and 0 is less than 5, the lines of the while loop begin to be called.

   i. The while loop starts by building a "Grass" block at location (blocksPlaced, 0, 0). Since blocksPlaced's current value is 0, it places a block at (0, 0, 0)
   ii. The second line of the while loop increases the value of blocksPlaced by one. blocksPlaced now stores a value of 1.

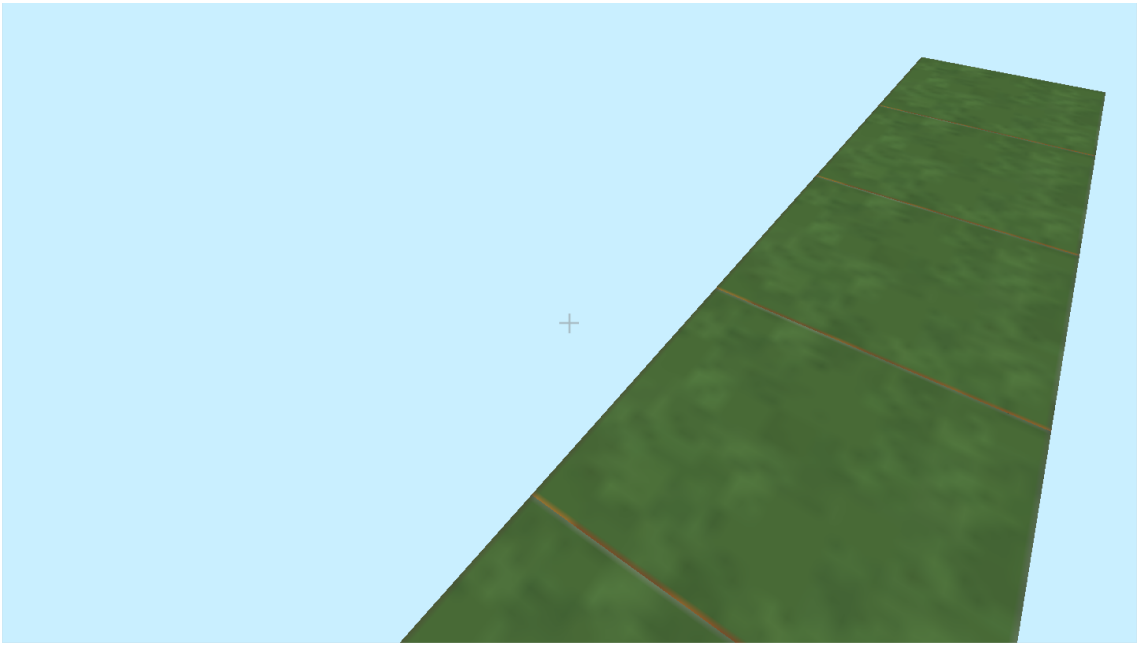

Figure 3: Five blocks floating in space, thanks to our while loop.

```
//This function is called to generate out the map
public override void GenerateMap (){
    int blocksPlaced = 0;
    while(blocksPlaced<5){
        MapBuilderHelper.BuildBlock ("Grass", blocksPlaced, 0, 0);
        blocksPlaced=blocksPlaced+1;
    }
}
```

Figure 4: GenerateMap set up to create five grass blocks.

3. Because this is a while loop, we next return to the start of the while loop and it's condition "blocksPlaced<5". Since blocksPlaced now stores a value of 1 and 1 is less than 5, the while loop begins to be called again.

   i. The while loops build a "Grass" block at location (blocksPlaced, 0, 0). Now at (1, 0, 0).
   ii. blocksPlaced's value is increased by 1, it now stores 2

4. The while loop is called with blocksPlaced at a value of 2. It goes through the same process as when blocksPlaced was at 0, and 1. First it checks the condition, then places a block (2,0,0), and then increases blocksPlaced.

5. The while loop is called with blocksPlaced at a value of 3

6. The while loop is called with blocksPlaced at a value of 4

7. At the end of calling the while loop with blocksPlaced at 4 (the last step), blocksPlaced is increased to 5. Therefore, when the condition "blocksPlaced<5" is checked, it is false as 5 is not less than 5. We have reached the terminal condition and the while loop stops running.

As you can see, the while loop is called with blocksPlaced having values of 0, 1, 2, 3, and 4. That explains the five blocks and their positions!

2.4 Multiple and Nested Loops

So far we've managed to make a pretty short line of blocks with a while loop. Let's try changing the number of times the while loop will run. How about 20? Change the condition of the while loop to:

```
blocksPlaced < 20
```

This way blocksPlaced will run with values between 0 and 19 for a total of 20 times (we start counting at 0 instead of 1 since blocksPlaced starts with a value of 0).

Make that change to the while loop and play the game again. You'll notice the difference.

So we've got a much longer line of blocks (four times as long), but that's still not all that useful (except maybe if we wanted a tight-rope walking game). What we want is a nice big area to move around on. A square of blocks, instead of a line.

What if we used two different while loops? We have one where we had the changing variable (blocksPlaced) in place of the x-coordinate. We could try adding a second one to handle the z-coordinates. Let's try it. Add (do not replace) the following code below the first while loop in GenerateMap. It's an exact duplicate of the first while loop, except with blocksPlaced in place for the "z" coordinate instead of x.

```
while(blocksPlaced < 20){
        MapBuilderHelper.BuildBlock ("Grass", 0, 0, blocksPlaced);
        blocksPlaced=blocksPlaced+1;
}
```

This looks like the first while loop, except that we put blocksPlaced in the z-coordinate position of the BuildBlock call.

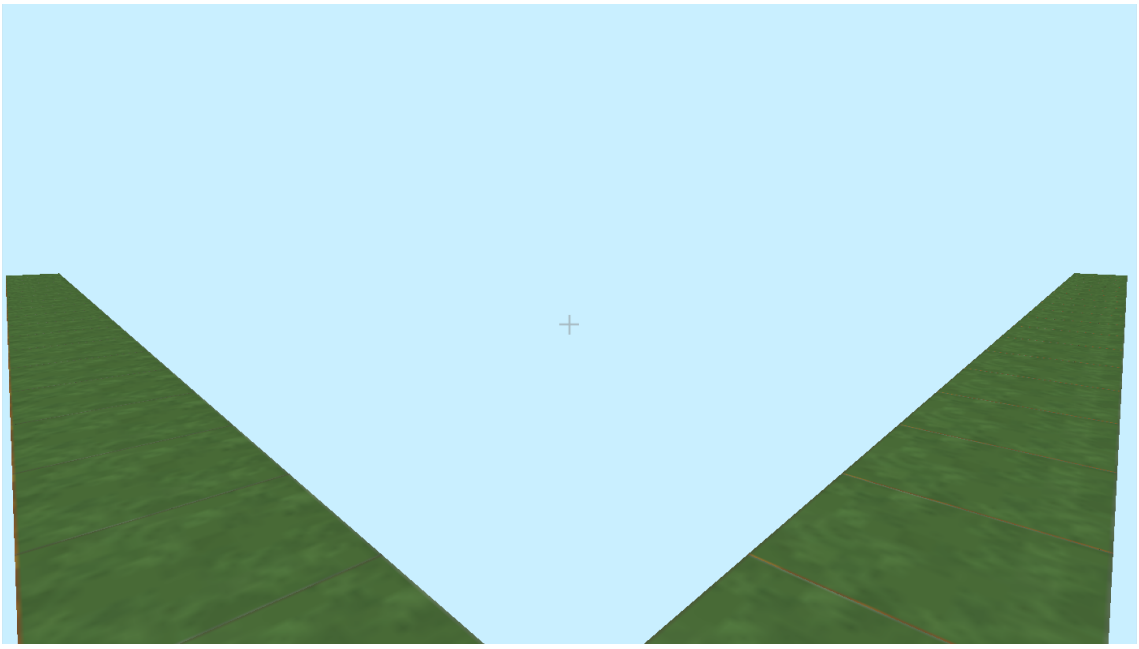

Figure 5: The effect of adding a second while loop with a line in the z-direction.

Add the above code to GenerateMap then run the game.

Hold on, that's not quite right! Despite having a whole new while loop inside GenerateMap, there appears to be no change. This is due to the fact that none of the code in the second while loop is called, as its condition is never true, can you see why?

The issue is that after the first while loop blocksPlaced stores a value of "20". This means that the second while loops condition "blocksPlaced $<$ 20" is never true. What we need to do is to "reset" blocksPlaced back to 0 before the second while loop is called! Add the following line of code *in between* the two while loops.

```
blocksPlaced = 0;
```

Add the above code to GenerateMap then run the game. You should see something that looks like Figure 5. An improvement, but not entirely what we were looking for!

When you run the game you'll see two lines of blocks. One line extending out into the positive x direction (to the right), and one into the positive z direction (going forward) both of them coming from the origin located at point (0,0,0).

Given that our goal is a square area, what we want is an area of 20 blocks by 20 blocks. You could think of this then as 20 lines that are 20 blocks in length placed next to each other. if you think about it this way the "L" shape that we've been able to make so far is 1/20th of what we need.

That may be a bit confusing, so check out Figure 6 to see a visualization of what we're talking about. On the left you can see what we've got now, two lines of length 20. On the right you can see what we want. A square with a 20 by 20 block area. If we could add 19 more lines to what

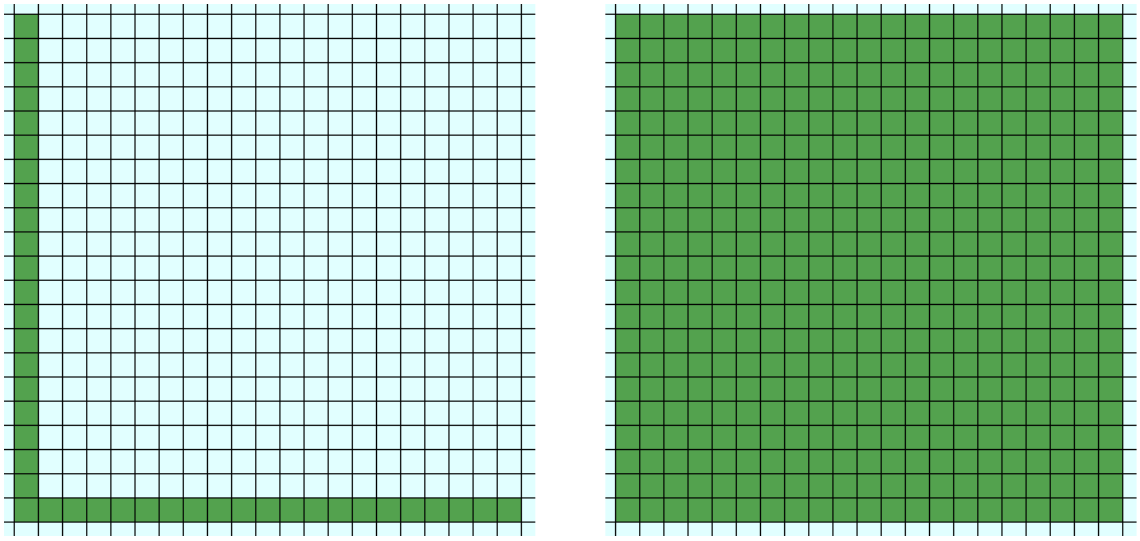

Figure 6: Left: An overhead visualization of what we currently have, two lines of length 20. Right: An overhead visualization of what we want. A square of 20x20 block area or 20 lines of length 20 placed side by side.

we have now, we could get what we want! But how to do that? The answer is to use a technique called nested loops.

You might recall when we used "nested" if statements in lessons two and three. Nested loops are exactly the same notion. It's when we put a loop *inside* another loop. We'll go ahead and show you the code to add and then walk you through it. Replace *all* the code in GenerateMap with:

```
int bX = 0;
while(bX < 20){
  int bZ = 0;
  while(bZ < 20){
    MapBuilderHelper.BuildBlock ("Grass", bX, 0, bZ);
    bZ = bZ + 1;
  }
  bX=bX + 1;
}
```

Play the game with the new code inside GenerateMap. You should see something that looks like Figure 7.

It worked! We managed to make a pretty huge area with just 9 lines of code! Imagine the time it would have taken to place that all by hand (probably around 20 minutes for these 400 blocks), doesn't this seem better? Try messing around with the numbers in the conditions. What happens if we replace both 20's with 50's? What happens if we only change one of the 20's to another number?[2]

[2]Note: Using values larger than 100 or so will lead to a bit of a lag before the game will start up. That should make sense though, as the game is placing over a 1000 blocks.

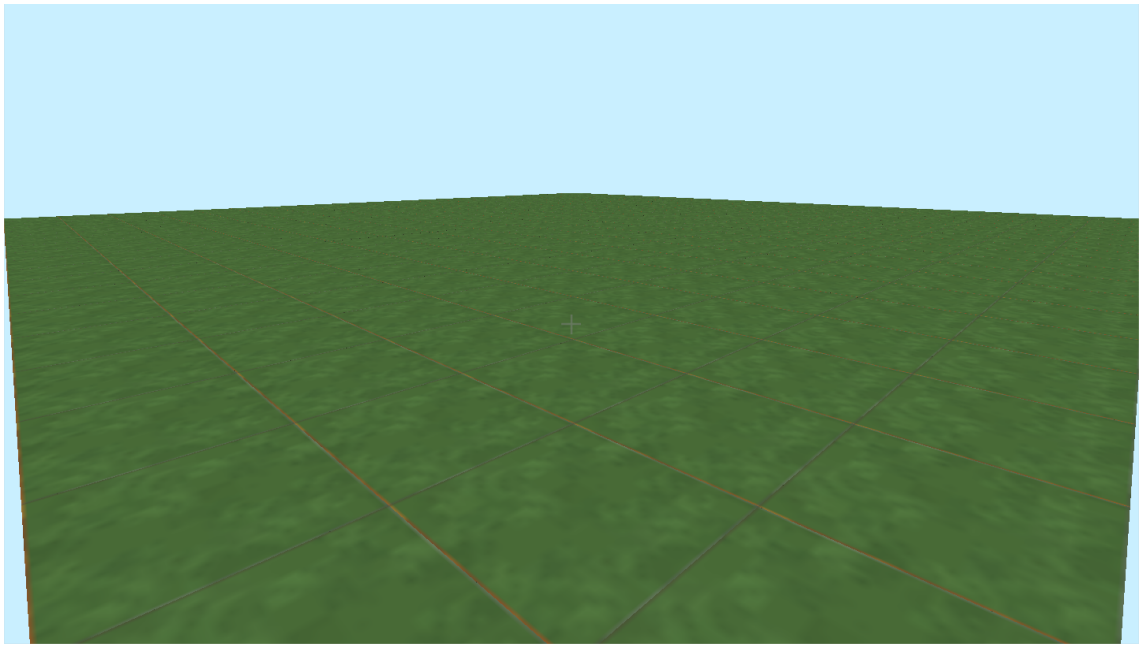

Figure 7: The full 20x20 block area!

2.5 Walking the Nested Loops

You might still be wondering how exactly the two while loops actually make the big square area we saw? We can say that they draw 20 lines of length 20 blocks, but that doesn't really. Let's walk through the code (not all the way through as that would take some time), and see what's going on:

1. The first line of GenerateMap "int bX = 0;" creates an integer variable called "bX" and gives it an initial value of 0. (We replaced blocksPlaced as the variable name with bX to save space and make its role more clear).

2. The second line is the beginning of the first while loop. The condition is checked (bX $<$ 20). Since bX is 0, the condition is true and the while loop begins.

   i. The first while loop starts by creating an integer value "bZ" and setting its value to 0
   ii. The next line of the first while loop starts the second/nested while loop. It's condition is (bZ $<$ 20). Since bZ is 0, the condition is true and the second while loop begins
   iii. The second while loop places blocks and updates the value of bZ until bZ equals 20. This has the effect of placing a block at (0, 0, 0), (0, 0, 1), (0, 0, 2)... all the way to (0, 0, 19). Drawing a line of 20 blocks.
   iv. The last line of the first while loop updates the value of bX by one. It now has a value of 1.

3. The first while loop returns to the top, bX at 1 is less than 20, so it begins again.

   i. The first while loop starts by creating an integer value "bZ" and setting its value to 0
   ii. The condition (bZ < 20) is true and the second while loop begins
   iii. The second while loop places blocks and updates the value of bZ until bZ equals 20. This has the effect of placing a block at (1, 0, 0), (1, 0, 1), (1, 0, 2)... all the way to (1, 0, 19).
   iv. The last line of the first while loop updates the value of bX by one. It now has a value of 2.
   ...

This process continues creating a line from (2, 0, 0) to (2, 0, 19) when bX is equal to 2, a line from (3, 0, 0) to (3, 0, 19) when bX is equal to 3 and continuing until bX equals 20 and the first while loop stops. It might seem odd that bZ is "redefined" several times. But given that it has local scope (it doesn't exist outside a single run of the first while loop), its actually similar to creating *different* variables that happen to have the same name.

That explains how the two while loops work! It should make more sense now when we say that we're drawing individual lines (drawn by the inner/second while loop) with different starting positions determined by the first while loop. This practice of walkthrough through code can be very helpful!

2.6 Making Multiple Squares

After all that we have a large square platform floating in the air, but that doesn't look particularly "realistic" (as far as a floating island can be realistic). Most land structures are a bit "lumpier" than just a floating square. What if we added a second square? Let's try adding a second square *starting* a square at a different location. That'll require a second set of two while loops. Let's add (do not replace) the below code at the end of GenerateMap:

```
bX = 5;
while(bX < 15){
  int bZ = 5;
  while(bZ < 15){
    MapBuilderHelper.BuildBlock ("Grass", bX, 1, bZ);
    bZ = bZ + 1;
  }
  bX=bX + 1;
}
```

Play the game after adding the above code to GenerateMap. You should see a second raised section now coming out of the ground!

This more "bumpy" ground looks a bit more natural. Can you see how it worked? We changed where we started building the square from (0,0,0) to (5,1,5) by changing bX and bZ to start at 5, and called "BuildBlock" with a y-value of 1 (putting this square one block *above* the other). We also determined the size of the square of blocks by changing the conditions for both while loops (just as you did if you played with the condition values of the first while loops!). Play around with the starting values and terminal conditions to change the "bump" position. Can you use what you've learned here to make a third "level" of bumpiness?

Even by reusing values of bX, it still took a lot more lines of code (nine) to create this second square. It also involved the duplication of quite a bit of code, which is something programmers try to avoid. In general with programming its best to try to *reuse* code as much as possible. And given that we want to add many more squares to our floating islands we need some way to cut down on the code.

There's a solution to this problem! And its for you to make your very first function.

2.7 Making your First Function

You have used functions in every previous lesson and even in this one. By now you should be pretty comfortable with the form that they take. While they can have additional information (like accessibility modifiers), a function definition only requires three things: a return type (typically void), a function name, and parentheses with the functions arguments (if any). In code that looks something like:

```
void FunctionName(){
        //things in function
}
```

For now let's make a function named "BuildRectangle" with a return type of "void", and no arguments. It'll need to be defined outside of GenerateMap but inside the class AdventureGameGenerator. Remember that every function has a "return type" determining the value type it returns and that "void" means that it will return no value. Inside it just place the code that builds the first square, copy and paste the code that creates the first square (the two while loops) into BuildRectangle. Inside GenerateMap remove the code that replaced the two while loops and replace it with a a single call to BuildRectangle, you'll also need to redefine "bX" since now it's no longer defined in GenerateMap (just add an int before its first usage). Unlike with "BuildBlock" you don't need to specify the class when calling "BuildRectangle" since it is defined inside this class. When you're done your code should look like Figure 8, but try to make the changes before you check it out.

Play the game, and notice what effect these changes had on the game world.

If you run the code again, you shouldn't see any changes! That might seem a bit odd, since we moved the code that did that from GenerateMap into BuildRectangle. But it should make sense if we remember that calling a function calls every line of code within that function. That means when the single line in GenerateMap calls BuildRectangle we can call all nine lines of code inside BuildRectangle!

While this cut down the size of GenerateMap, all it really did was "move" those lines into BuildRectangle. However we'll make changes to BuildRectangle, which will allow us to draw both the "bump" and many other platforms with just a single function call for each one!

2.8 Adding Arguments

Our end goal here is to alter BuildRectangle such that it can build different sized rectangles at different locations. To do that we'll need to add arguments to BuildRectangle.

In prior lessons we talked about arguments as being value types that are required to call a function. These values seemed to be able to impact the behavior of the function (such as placing the key and sword in different locations in Lesson Two). This is due to the fact that a function can *use* the values from the arguments in the code it runs.

```
1 using UnityEngine;
2
3 public class AdventureGameGenerator : Generator {
4
5     //This function is called to generate out the map
6     public override void GenerateMap (){
7         BuildRectangle ();
8
9         int bX = 5;
10        while(bX < 15){
11            int bZ = 5;
12            while(bZ < 15){
13                MapBuilderHelper.BuildBlock ("Grass", bX, 1, bZ);
14                bZ = bZ + 1;
15            }
16            bX=bX + 1;
17        }
18    }
19
20    void BuildRectangle(){
21        int bX = 0;
22        while(bX < 20){
23            int bZ = 0;
24            while(bZ < 20){
25                MapBuilderHelper.BuildBlock ("Grass", bX, 0, bZ);
26                bZ = bZ + 1;
27            }
28            bX=bX + 1;
29        }
30
31    }
32
33 }
```

Figure 8: AdventureGameGenerator.cs after the addition of the BuildRectangle function.

This may be unclear, so let's add in an example. Inside the parenthesis of BuildRectangle add "int x". Then change the call to "BuildBlock" inside it from BuildBlock ("Grass", bX, 0, bZ) to BuildBlock ("Grass", bX + x, 0, bZ) such that x is now added to bX.

```
void BuildRectangle(int x){
  int bX = 0;
  while(bX < 20){
    int bZ = 0;
    while(bZ < 20){
      MapBuilderHelper.BuildBlock ("Grass", bX + x, 0, bZ);
      bZ = bZ + 1;
    }
    bX=bX + 1;
  }
}
```

Trying to run the game now will be met with the following error:

"Assets/Codebase/Environment/Map/Generators/AdventureGameGenerator.cs(7,17): error CS1501: No overload for method 'BuildRectangle' takes '0' arguments"

This error pops up since we added an argument (int x) to BuildRectangle, but the call to the function in "GenerateMap" doesn't have an integer value passed in to it. Remember arguments specify values that *must* be included when calling the function. So let's give it one! Change the call to BuildRectangle in GenerateMap so it now reads:

```
BuildRectangle (-2);
```

Play the game, can you notice the difference?

The difference is that the 20 by 20 square is now two units to the left! If we look at the changes we made though, this should make sense. We add "x" to every block's placement in BuildRectangle. Since x was "-2" that had the effect of making every x value 2 lower. So 0 became -2, 1 became -1, 2 became 0 and so forth. This meant that instead of stretching from 0 to 19 in x values the blocks stretch from -2 to 17!

But we haven't yet shown the true power of functions. Add the following line below the first BuildRectangle call in GenerateMap:

```
BuildRectangle (19);
```

Play the game. The difference should be obvious. There are now two 20 by 20 blocks separated by a single block opening like Figure 9.

With a single function call we managed to get another while 20 by 20 square of blocks! But how did that work? Well, just like when we used BuildRectangle with an argument of -2 and it created a square of blocks shifted two units to the left, calling BuildRectangle with 19 created a square of blocks shifted 19 units to the right! Try changing the value and playing the game again on your own. Can you make the gap between the two squares smaller so they look like one shape? Or make the gap so wide you can't jump it? We'll be getting rid of this second island in a bit, but messing with its location can be useful for fun and to help understand arguments.

If you're still confused one way to think of it is that a call to BuildRectangle (as it is) with any integer value can be swapped out with the code inside BuildRectangle with "x" replaced with the

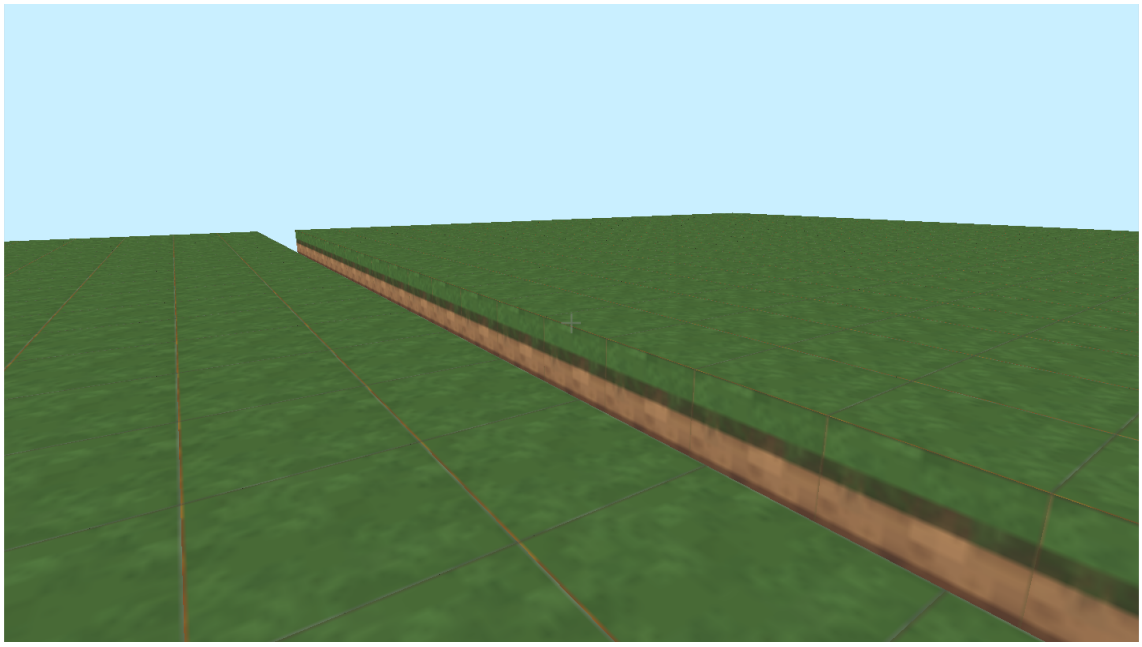

Figure 9: Two squares of grass!

```
int bX = 0;
while(bX < 20){
     int bZ = 0;
     while(bZ < 20){
          MapBuilderHelper.BuildBlock ("Grass", bX + -2, 0, bZ);
          bZ = bZ + 1;
     }
     bX=bX + 1;
}
bX = 0;
while(bX < 20){
     int bZ = 0;
     while(bZ < 20){
          MapBuilderHelper.BuildBlock ("Grass", bX + 19, 0, bZ);
          bZ = bZ + 1;
     }
     bX=bX + 1;
}
```

Figure 10: The equivalent of calling BuildRectangle with -2 and 19.

```
void BuildRectangle(int x, int y, int z){
    int bX = 0;
    while(bX < 20){
        int bZ = 0;
        while(bZ < 20){
            MapBuilderHelper.BuildBlock ("Grass", bX+x, y, bZ+z);
            bZ = bZ + 1;
        }
        bX=bX + 1;
    }

}
```

Figure 11: BuildRectangle after adding the "y" and "z" arguments.

value. In that way BuildRectangle(-2) and BuildRectangle(19) are equivalent to Figure 10. If you like, feel free to replace the calls to BuildRectangle inside of GenerateMap with the code in Figure 10 to confirm this for yourself, but make sure to switch it back afterward!

But that's not all we can do with arguments! The end goal here is to be able to make a "bumpy" island even easier than before, and we'll do that (and more) by adding more arguments.

2.9 More Arguments, More Control

We already added an argument to control where the rectangle is built on the x axis, so it makes sense to add arguments for both the y and z axises. We called these arguments "y" and "z". You should be able to use how we used the x argument to figure out how to add y and z arguments to control those axises. But you can check Figure 11 to see how we did it.

Once you've added these two new arguments to the function definition, we'll need to add two arguments to the calls to BuildRectangle in GenerateMap. Calls to BuildRectangle now need to have three values passed in to them, which will determine where the rectangle will be positioned in space! Let's see if we can use this fact to remake our "bumpy" island with only two calls to BuildRectangle! With our bumpy island we had one rectangle of size 20 starting at (0, 0, 0) and one rectangle of size 10 starting at (5,1,5). So to place two rectangles at those positions we just need the calls:

```
BuildRectangle(0, 0, 0)
BuildRectangle (5, 1, 5);
```

Replace the code in GenerateMap with the above two BuildRectangle calls, and play the game.

Now hold on, that's not quite right! The second rectangle was located at the right position, but it was size 20 instead of size 15! That's because BuildRectangle is still set up to only create rectangles of size 20, but we can change that. We just need two new arguments to control the size in the x axis and the z axis! We called these arguments sizeX and sizeZ. Where to use them in BuildRectangle shouldn't be too tough, you just need to have reach of them replace one of the two

```
11      void BuildRectangle(int x, int y, int z, int sizeX, int sizeZ){
12          int bX = 0;
13          while(bX < sizeX){
14              int bZ = 0;
15              while(bZ < sizeZ){
16                  MapBuilderHelper.BuildBlock ("Grass", bX + x, y, bZ+z);
17                  bZ = bZ + 1;
18              }
19              bX=bX + 1;
20          }
21      }
```

Figure 12: BuildRectangle with all of the location and size arguments.

20's currently in it. Essentially we're using them to determine how many times to call both while loops.

Try to add both arguments on your own, but you can always check out Figure 12. Now that we've made those changes we'll need to make changes to our two BuildRectangle calls in GenerateMap. We can finally make the bump look right! The two calls should look like:

```
BuildRectangle (0, 0, 0, 20, 20);
BuildRectangle (5, 1, 5, 10, 10);
```

Play the game, this should look exactly like the "bumpy" island we made before! Except unlike before we managed to accomplish this bumpy island with only two lines of code.

You've successfully made a a reusable and useful function. Feel free to play around with calls to BuildRectangle in GenerateMap. Add more calls to BuildRectangle to spruce up your island, add a second "floating" island, or even a set of small "stepping stone" islands. Test out your changes by playing the game after each change to ensure that its doing what you think it should. However, before you get too far ahead with designing your game world, you might want to read through the next section as we'll be adding one more argument that'll give you further control of your island aesthetics.

2.10 Changing your Island "Type"

So far in this lesson we've changed the size and position of grassy islands floating in the sky. However, we don't have to use grassy islands. Remember that the call that's actually placing blocks is "MapBuilderHelper.BuildBlock", which takes as one of *it's* arguments a string value for the block to use. We've used "Grass" so far, but that doesn't need to be the case. Change the string value in the call to BuildBlock in BuildRectangle from "Grass" to "Lava".

Play the game, and note the difference! Don't worry, you won't be hurt by walking around on this lava island.

Well that's definitely different than the grass islands! However, if we just manually change the string value inside BuildRectangle, then we can still only have Islands constructed of a single block "type". What if instead we passed in a string value as an argument to BuildRectangle? We could then use that string value when we called BuildBlock, thus meaning that each island could be of a unique "type". We used the argument name "blockValue" for this string value, placing it

```
void BuildRectangle(int x, int y, int z, int sizeX, int sizeZ, string blockValue){
    int bX = 0;
    while (bX < sizeX) {
        int bZ =0;
        while(bZ < sizeZ){
            MapBuilderHelper.BuildBlock(blockValue, bX+x, y, bZ+z);
            bZ = bZ +1;
        }
        bX = bX+1;
    }
}
```

Figure 13: The final BuildRectangle function with all arguments.

```
//This function is called to generate out the map
public override void GenerateMap (){
    BuildRectangle (-15, 0, -15, 50, 50, "Grass");
    //Gold islands
    BuildRectangle (20, 3, 20, 5, 5, "Gold");
    BuildRectangle (-15, 3, 20, 5, 5, "Gold");
    BuildRectangle (-15, 3, -15, 5, 5, "Gold");
    //Bumps on main island
    BuildRectangle (-15, 1, -15, 15, 15, "Grass");
    BuildRectangle (10, 1, 10, 15, 15, "Grass");
    BuildRectangle (20, 1, 5, 10, 15, "Grass");
    BuildRectangle (-15, 1, 25, 10, 15, "Grass");
}
```

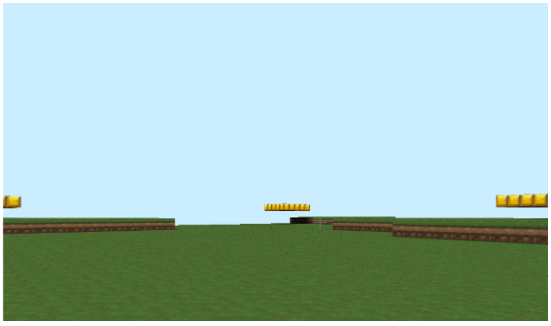

Figure 14: Our “final” island on the right and the code to make it on the left.

after all other arguments in the “BuildRectangle” function definition. Try to add this argument yourself without checking out our final BuildRectangle function in Figure 13. Feel free to mess around with your island locations and types till you’re happy. While you can find a full list of the types accessible to you at the end of this lesson. We recommend testing out using ”Snow”, ”Dirt”, ”Stone”, ”Sand”, ”Portal”, or ”Gold”. Our final code and islands looks a bit like Figure 14. But make something unique to you! You might want to adjust these islands later when we add in more elements to our Adventure Game (for example, by the end of this lesson we will add enemies). But you can at least make a cool starting point now, and change it as necessary!

2.11 OPTIONAL: Adding Foliage

Despite our additions, your islands still probably look a bit bare. What if we made it a forested island? To do that we’ll need to go through a similar process as with creating islands. We’ll experiment in order to create a single tree in GenerateMap, then place the code that makes a tree into a function, and then add arguments to that function in order to place many different trees. While this section is not required, it should serve as a useful model for yourself in terms of building other structures with code on your own, such as bushes, statues, or even houses!

We’ll build up the tree bit by bit. Let’s start with the trunk. A trunk can be thought of as a single “line” moving upward (in the y direction). We know how to draw lines! We’ll need to add a new while loop into GenerateMap to handle drawing the trunk (we’ll add the other parts of the

tree after we get this first part down).

To draw a line we'll need to use a while loop, similar to how we drew a "line" of Grass earlier. Except that we'll need to call MapBuilderHelper.BuildBlock with "Trunk" instead of "Grass" and use a variable's value to change the "y" position, instead of the x or z position. All together, that should look something like:

```
int bY = 0;
while(bY<7){
  MapBuilderHelper.BuildBlock("Trunk", 5, bY, 5);
  bY = bY +1;
}
```

This will create a straight line of "Trunk" blocks starting at position (5, 0, 5) and going to position (5, 6, 5). If we instead wanted the blocks start at a different y-value we could change the initial value of bY. Changing it to be equal to 1 would lead to a line of blocks stretching from (5, 1, 5) to (5, 6, 5). To change the x or z values of the line we'd have to change the call to "BuildBlock". You may have to use different values depending on the world you made!

Add the code to make the trunk, save, and play the game. Ensure that the trunk is in fact present in the scene.

We've got a trunk! The next thing we need is leaves at the top of our tree. We can make use of the fact that the variable "bY" ends with a value of y *just above* the trunk to place a square of leaves at that height with our "BuildRectangle" function. Remember that the rectangle of blocks made by BuildRectangle isn't centered on the x, y, and z coordinates that we call it with. Instead it starts in what can be thought of as the lower-left "corner" of the square (if seen from above). This means we can't call BuildRectangle with values of x=5 and z=5 as then the square of leaves will have its lower left corner on the top of the trunk! Instead we'll need to call it using code like:

```
BuildRectangle(2,bY,2,7,7, "Leaves");
```

We start the rectangle with an x value of 2 and a z value of 2, and use values of 7 for sizeX and sizeZ. That way the rectangle's center will be just above the trunks. Since there will be three values before we hit the trunk (2,3,4), and three after it (6,7,8). That might be a bit confusing, so let's go ahead and play the game to see it in action.

Add the call to BuildRectangle to make the "Leaves" rectangle atop the trunk and then play the game. You might notice that the tree looks a little odd.

The tree we made looks a bit flat on top, doesn't it? It looked something like Figure 15. While the rectangle of leaves is centered on the trunk as we suggested, having a rectangle of leaves isn't actually enough. We need the top of the tree to have more *volume* than that.

What if instead we created several rectangles by using a while loop to further increase the value of bY? These "stacked" rectangles would therefore look more like a tree! Let's try it, replace the single line that builds a leaf rectangle with the following while loop:

```
while(bY<12){
  BuildRectangle(2,bY,2,7,7, "Leaves");
  bY = bY +1;
}
```

Play the game with the new while loop. Check out what the tree looks like!

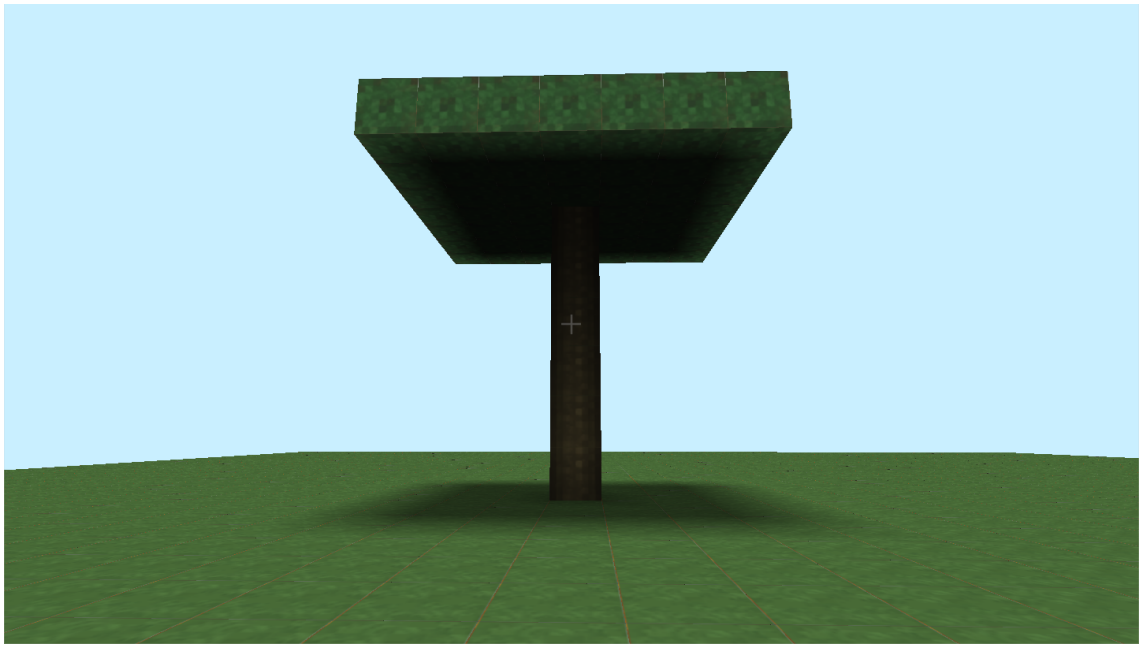

Figure 15: Our "flat" tree.

You should now see a much more tree-looking tree. That's great! But it could still be better. What if we instead changed the leaves so that the first and last of the rectangles was a couple blocks smaller? It'd give the tree a "rounded" appearance. To do that we can make use of conditional statements (if and else) and a relational operator that we have introduced yet equal to (==). As you might recall, relational operators (like < meaning less than) allow us to get boolean values from numbers. Equal to (==) determines if two values are the same, and is true if they are, it therefore acts similarly to how we normally think of the equals sign working. We could translate the desired effect into the following statement: if bY is equal to 7 or bY is equal to 11 then draw a smaller rectangle. In code that would look like the following:

```
if(bY == 7 || bY ==11){
  BuildRectangle(3,bY,3,5,5, "Leaves");
}
```

Then we could place the original call to BuildRectangle into an else statement. In the end your code should look something like Figure 16.

Play the game and note the changes. If you prefer the original appearance, feel free to change the code back. You could also experiment with other conditional statements with other conditions!

Once you're happy with your tree the time has come to place it into it's own function. You could just copy and pasting the code to create another tree, but that would be a lot of duplicate code and just a lot of code in general. Instead let's create a new function called "BuildTree". For now, we'll give it three integer arguments named x, y, and z for handling the initial position of the bottom of the trunk. To make these argument values actually serve that function we'll need to make several changes in the chunk of code that makes a tree.

```
26          while(bY<12){
27              if( bY == 7 | bY == 11){
28                  BuildRectangle(3,bY,3,5,5, "Leaves");
29              }
30              else{
31                  BuildRectangle(2,bY,2,7,7, "Leaves");
32              }
33              bY = bY +1;
34          }
```

Figure 16: The while loop to create the leaves after the inclusion with the if and else statements.

```
void BuildTree(int x, int y, int z){
    int bY = y;

    while(bY < 7 + y){
        MapBuilderHelper.BuildBlock("Trunk", x, bY, z);
        bY = bY + 1;
    }

    int smallRectangleX = x - 2;
    int smallRectangleZ = z - 2;

    int largeRectangleX = x - 3;
    int largeRectangleZ = z - 3;

    while (bY < 12 + y){
        if(bY==7 + y || bY== 11 + y){
            BuildRectangle(smallRectangleX,bY, smallRectangleZ,5,5, "Leaves");
        }
        else{
            BuildRectangle(largeRectangleX, bY, largeRectangleZ, 7, 7, "Leaves");
        }
        bY = bY + 1;
    }
}
```

Figure 17: The entirety of the BuildTree function.

Let's start by adding the "y" argument to be used in the function. We'll need to update every place we previously had hard-coded y values. There are five of these in the BuildTree function. They are as follows: the initial value of bY, the first while loop's condition, the second while loop's condition, and the two "==" checks if you used them. Add "y" to the value used for each of these to fully incorporate the argument y's value. That way, regardless of where we put the tree on the "y" axis, the tree will look the same. If we didn't include a "+ y" in the while loops, for example, then a starting value of bY might cause the while loops to never be called if it was already greater than 7 and 12!

Adding in the arguments for "x" and "z" is a bit tricker. For the call to "BuildBlock" for the trunk, we can just replace the two 5 values. But the leaf rectangles positions must be slightly different. The larger of the leaf rectangles must start at two less than the trunk's x and z values, with the smaller leaf rectangle starting at three less than that. We made some variables to store these values to make it a bit easier to see what's going on. You can check out Figure **??** for our version of BuildTree.

And with that you can add trees to the game world! This should help to make your world less barren looking. If you want to make further changes to the BuildTree function you could add additional arguments to handle the trunk size or the size of the leaf area. You could even change the type of blocks being used. How about a lava tree? Or a stone tree?

At this point we're done with the instructions for the game world. If you can think of other structures you'd like to make (a boulder? a house?) feel free to make them at this time. Start the same way we did at with the tree, making changes in GenerateMap to test things out. Then, put them into a function so you can use them more than once, and add arguments to the functions to place the structures in different locations! Build up a game world that you're satisfied with, as we'll be using it in Lessons 5 and 6.

2.12 A little bit of Adventure

While this lesson was mostly about building up the world for the adventure game, let's add some adventure elements here at the end. To begin with, we'll need to open up another script named "AdventureGame.cs". You can find it under the "Codebase" folder then under the "LessonsFour-Six" folder. Open it up and we'll make a couple quick changes.

This script should mostly look familiar to you, we have a sword variable of type GameObject like in lesson two along with Start and Update functions. We also have a new variable of a new type: "npcManager" of type "NPCManager".

Our goal with this class is to add a sword and enemies to the game, such that a player needs to find or reach the sword (preferably from a challenging location) and then defend themselves from the enemies. We can spawn the sword very easily using the game ItemHandler.SpawnItem function we used in lesson two. As a reminder SpawnItem takes as arguments a GameObject, and three float values for the location. If need be, feel free to check back in lesson two for some examples.

Add the code to spawn the item into Start and then play the game to check where it's place. Play around with it's location until you're happy that it's suitably challenging to get!

Once your happy with the location of the sword let's add in a call to a function to create some enemies. The function we'll need to use is npcManager.SpawnEnemies. It takes five arguments: 1.) An integer value for the number of enemies to spawn. 2-4.) Float values for the x, y, and z coordinates of the center of the area to spawn enemies in. 5.) A float value for the range of the area to spawn enemies in.

That might seem a but confusing so let's show an example call to the function and explain the effect:

```
npcManager.SpawnEnemies(10, 0, 0, 0, 10);
```

This call will create ten enemies in an area centered on point (0, 0, 0) with the area having a max size of 10. That means that an enemy could be located at (10, 0, 0) or (-10, 0, 0), or (0, 10, 0) or (0, -10, 0) or (0, 0, 10) or (0, 0, -10) or anywhere between those points. The 10 defines an area centered around the x, y, and z coordinates passed in. So for example (-5, 5, -2) is a potential place the enemies could be placed via this function, (-20, 2, 3) would not be a possible location as -20 is less than -10.

We explain the use of npcManager to you like this as enemies have to be placed on a floating island as otherwise they will fall forever. You can make a call to SpawnEnemies several times, so it might be could to put at least a few enemies on every island.

Add the call or calls to SpawnEnemies in Start, and play the game. Notice that you now have enemies behaving how you made them behave from Lesson Three. You can even win and lose.

The enemies behaved like you made them from lesson three! How is this possible? Well, each of the enemy ghosts still has the same script on it (Enemy.cs) that we edited in lesson three. Scripts like this can be used in multiple games, or levels! We can use this fact to build on the work done with this lesson into the next.

3 End of Lesson Four

That's all we have for Lesson Four! Make any changes to the islands, foliage, or enemies so that you're happy with the tiny game world you have made. We'll be making adjustments in the next section as we create a simple quest and add NPCs to your world. In this section you learned how to:

1. Use loops to create lines of blocks
2. Make use of nested loops to create more complicated structures with blocks
3. Use conditional statements with loops to better control the effect of the loop
4. Spawn enemies within a given range.

3.1 Lesson Four Glossary

Vocabulary Word	Definition	Examples
loop	A programming structure that repeatedly calls the same chunk of code.	while, for, etc

while loop	A simple kind of loop that repeats a certain chunk of code *while* some condition is true.	while(Condition){
		DoSomething();
		}

terminal condition	The condition which will lead a while loop to stop repeating.	[Not Appropriate]

3.2 List of Blocks

String Value	Description
“Grass”	The original block used in AdventureGameGenerator. Appears to have grass on top with a dirt bottom.
“Dirt”	A block that appears to be made totally of dirt, similar to the bottom part of the grass block.
“Stone”	A grayish block made of stone.
“Leaves”	A green block that appears to be entirely made of leaves.
“Trunk”	A light brown block 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” block.
“Cactus”	A green block that appears to be a cactus. The bottom and top have a pink flower while the sides have needles
“Pumpkin”	A block that looks like an uncarved pumpkin. The top has a stem.
“PumpkinLight”	A block that looks like an carved, and lit pumpkin. The top has a stem and the front has a face.
“Lava”	A block that appears to be made of unmoving lava.
“Portal”	A block with a swirly purple design on it.
“Brick”	A block that appears to be constructed from brown bricks.
“Sand”	A block that appears to be constructed from sand.
“TNT”	A block with the letters TNT on it and a fuse on top. (It does not explode).
“Gravel”	A block that looks like the stone block, but with darker edges.
“Bookshelf”	A block that appears to be a section of a bookshelf. Has books along its sides and a design on the top and bottom.
“Obsidian”	A block that looks similar to the Stone block but much, much darker.
“Gold”	A block appears to be made of pure gold.
“Diamond”	A block that appears to be made of a blue gem.
“Iron”	A block that appears to be made of iron.
“SnowGrass”	A block like the Grass block but that appears to be covered in snow.
“Snow”	A block used in Lesson 1 that appears to be made of snow.

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.

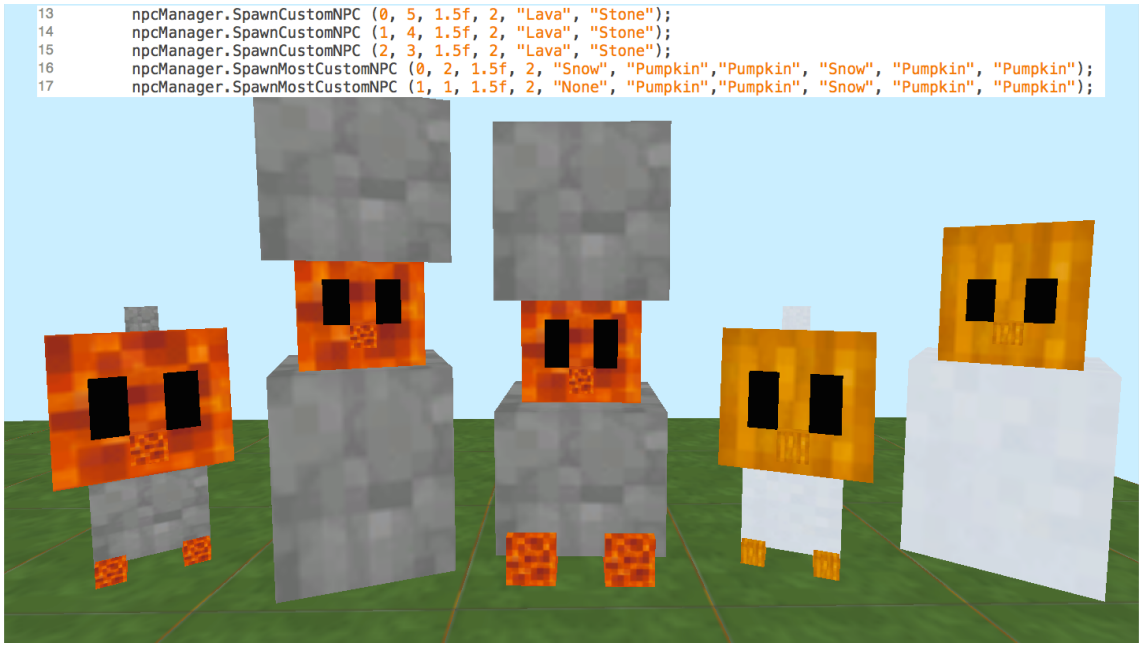

Figure 3: The NPCs and the commands that make them.

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.

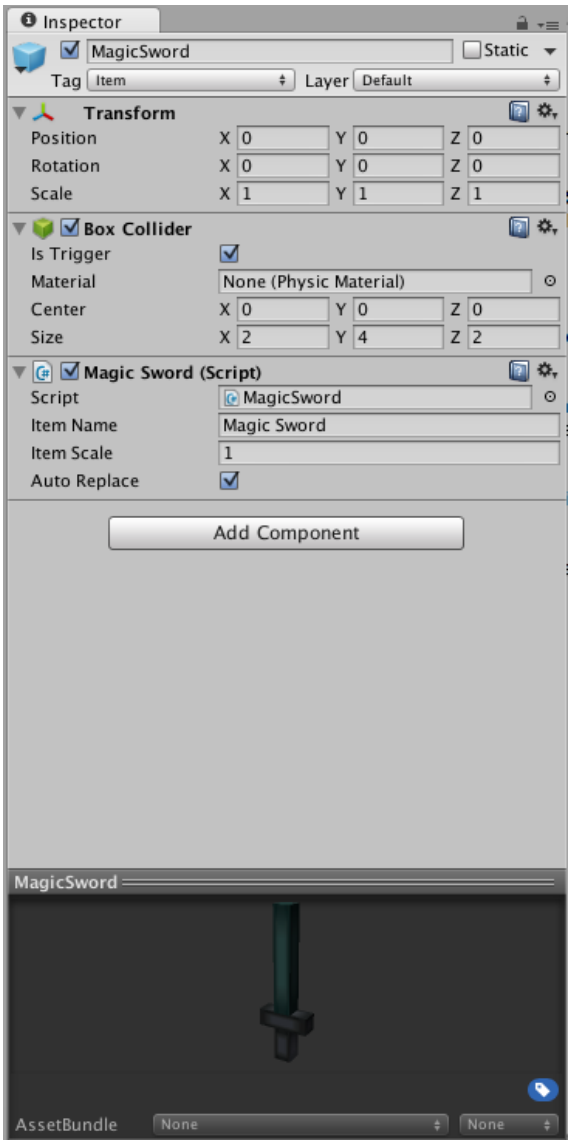

Figure 11: The model MagicSword with the magic sword script (MagicSword.cs) attached.

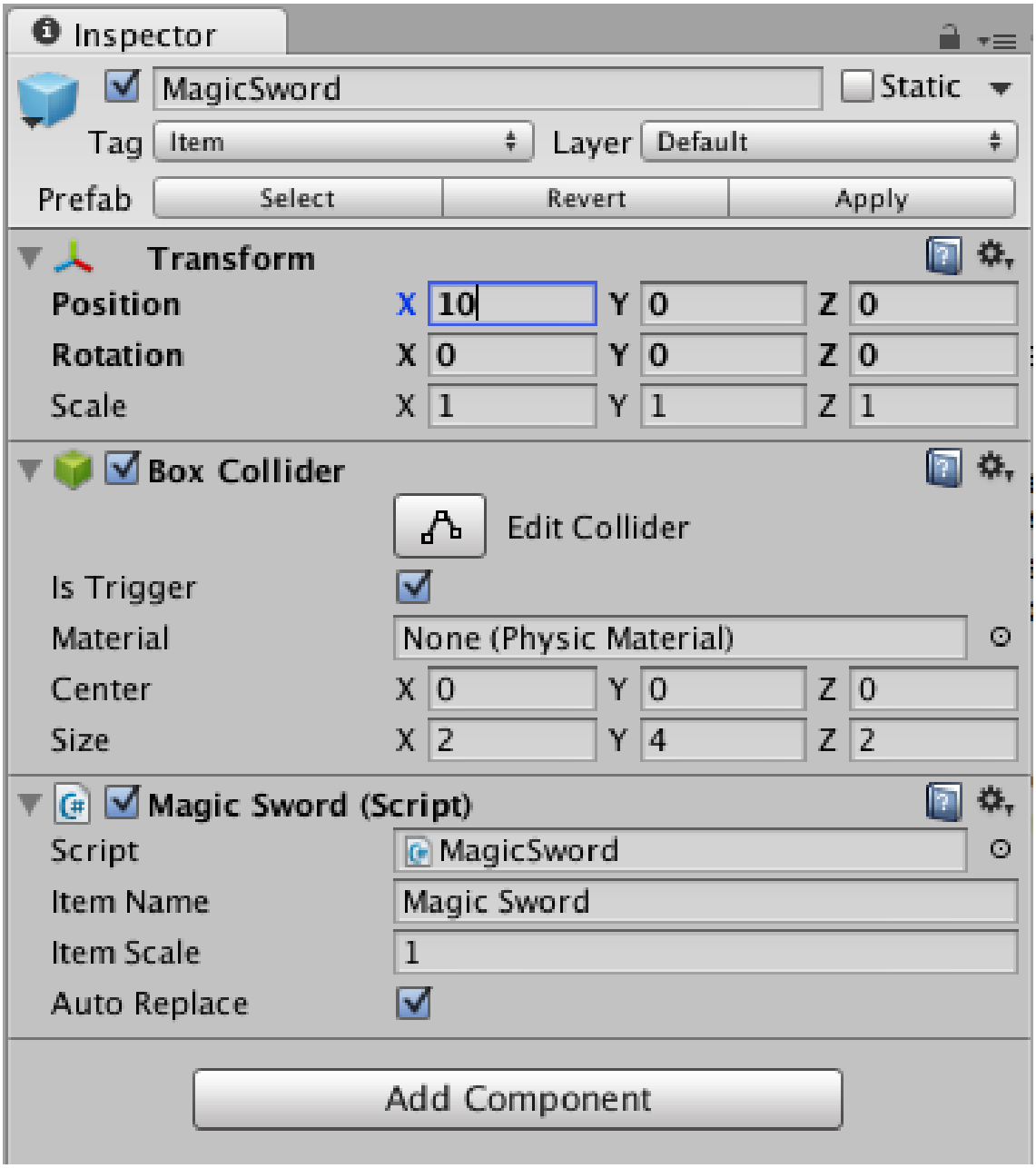

Figure 12: The inspector view of the MagicSword object

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:

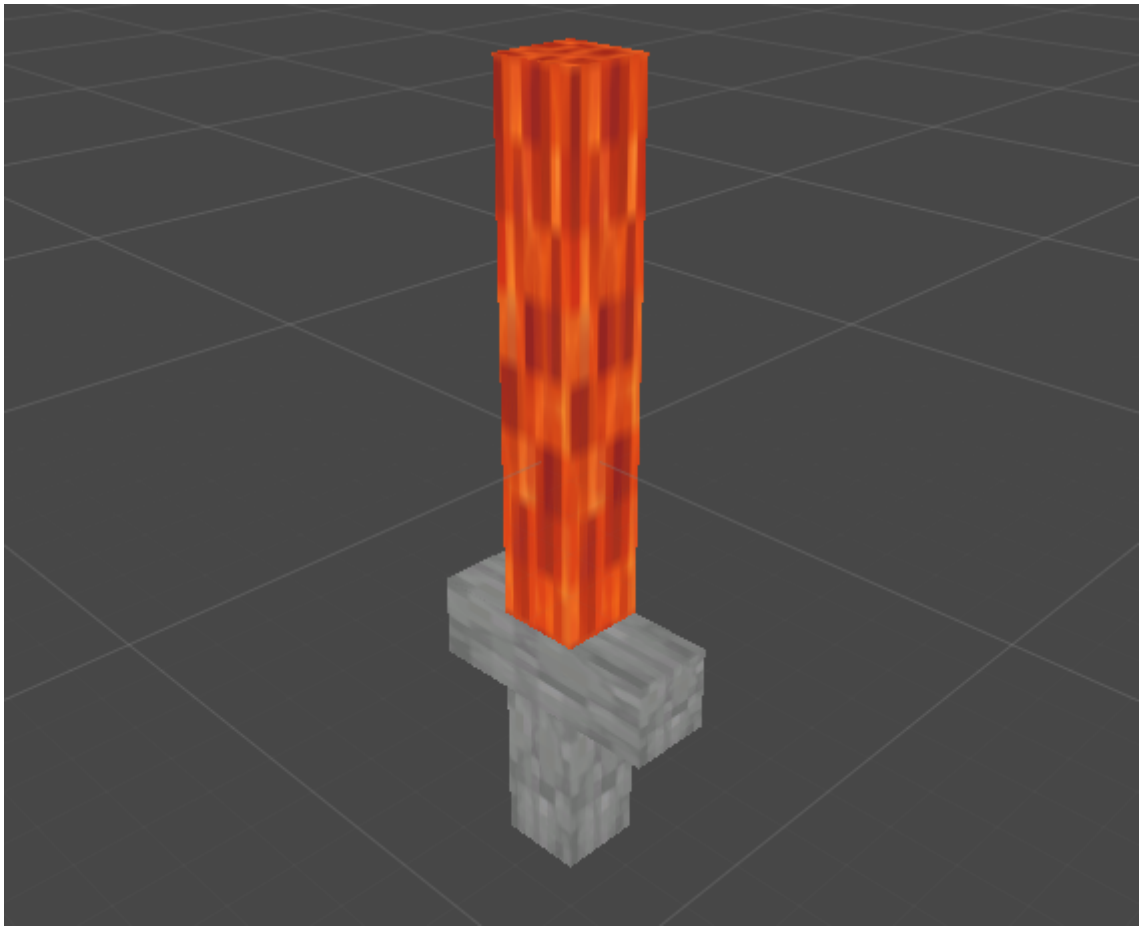

Figure 13: Our final MagicSword.

```
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.

Lesson Six

Who's the Boss?

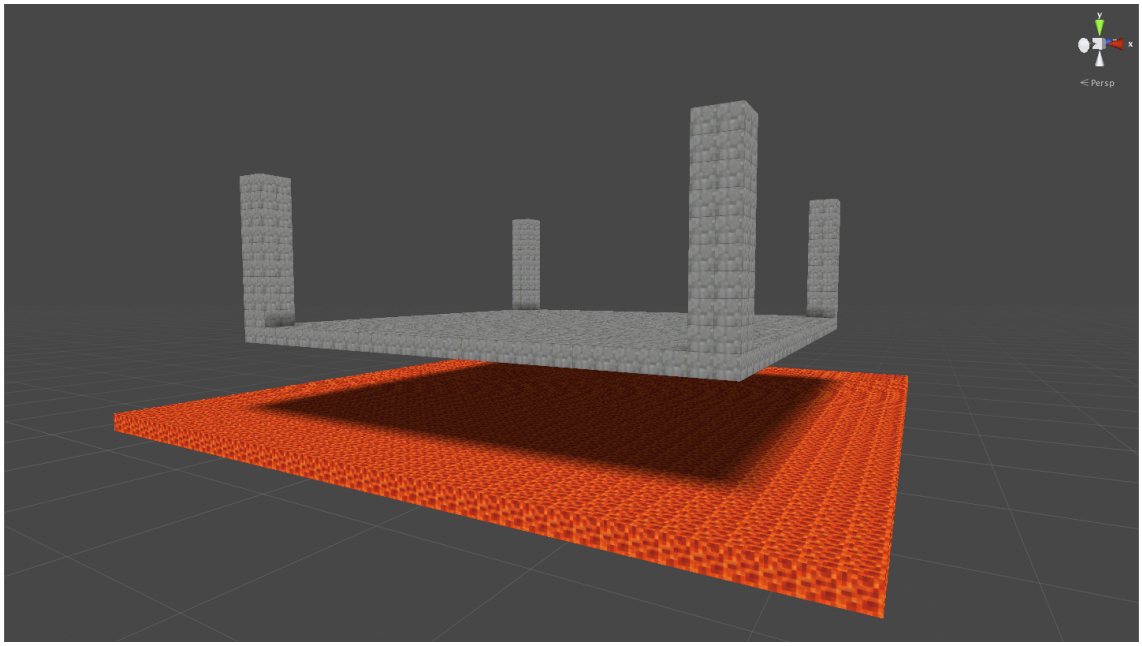

Figure 1: Our boss area as an example

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

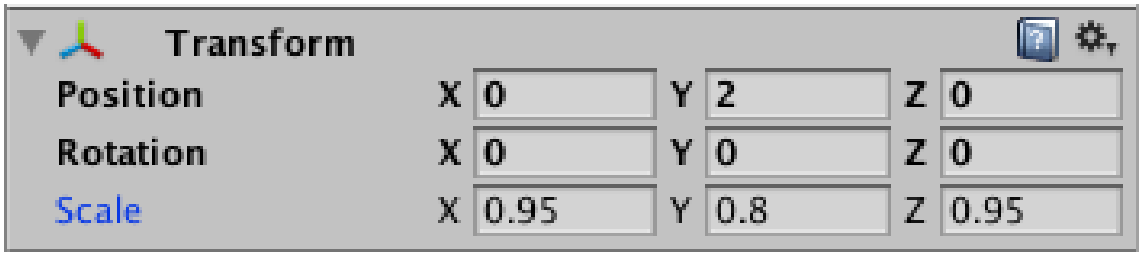

Figure 3: The player's initial transform in Unity's window.

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

Definition!

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

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!

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.

Definition!

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

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.

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

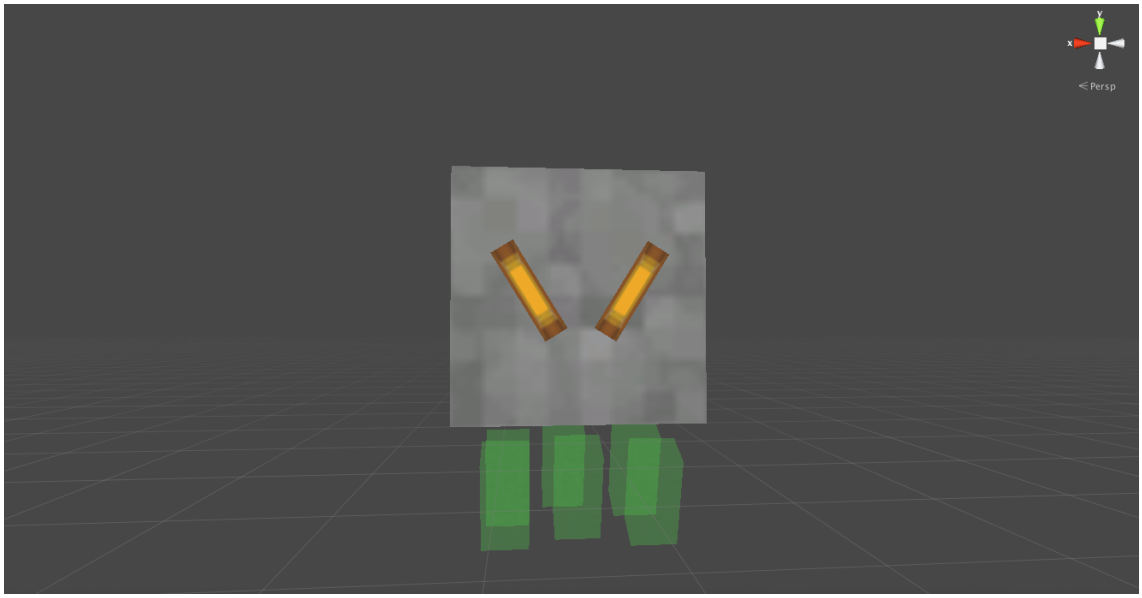

Figure 4: An example of our own version of the boss ghost.

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

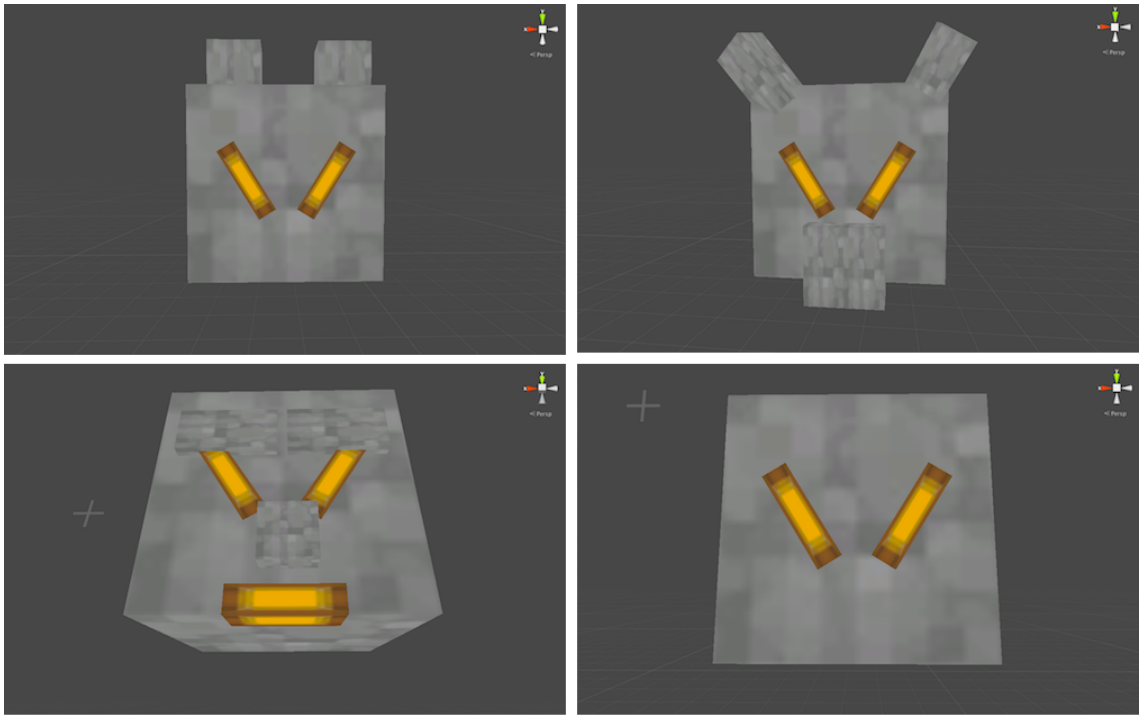

Figure 5: Four possible boss ghosts.

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)

```
4 public class BossAttackController : RangeAttackController {
5
6     public override float GetProjectileRange ()
7     {
8         return base.GetProjectileRange ();
9     }
10
11    public override void Fire (GameObject target)
12    {
13        base.Fire (target);
14    }
15 }
```

Figure 6: BossAttackController.cs with the initial setup.

```
34    //Called to shoot a projectile
35    public virtual void Fire(GameObject target){
36        if (shotTimer == 0) {
37            GameObject projectileObject = Instantiate<GameObject> (projectile.gameObject);
38            projectileObject.transform.position = transform.position;
39
40            Projectile firedProjectile = projectileObject.GetComponent<Projectile> ();
41
42            firedProjectile.Fire (target.transform.position);
43            shotTimer = 1.0f/enemy.GetFiringRate();
44        }
45    }
```

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

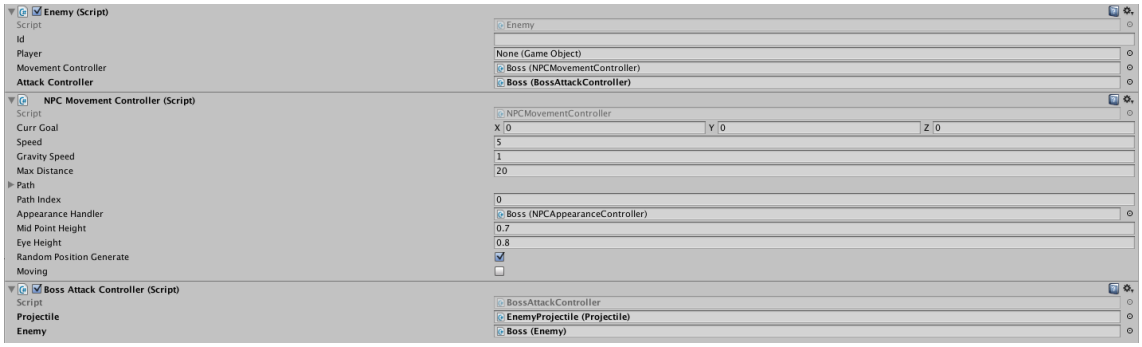

Figure 10: Inspector View of Boss components after set up

```
4 public class Projectile : MonoBehaviour {
5     //The speed this projectile will move at
6     public float projectileSpeed = 10;
7     //The current life (counts up till maxLifeTime)
8     private float lifeTime = 0;
9     //The max possible amount of time (in seconds) the projectile can be alive
10    private const float maxLifeTime = 1f;
11    //Destination of this projectile
12    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

```
11      public override void Fire (GameObject target){
12          Vector3 differenceVector = target.transform.position-transform.position;
13          float distToPlayer = differenceVector.magnitude;
14      }
```

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

```
11      public override void Fire (GameObject target){
12          Vector3 differenceVector = target.transform.position-transform.position;
13          float distToPlayer = differenceVector.magnitude;
14
15          if (distToPlayer < GetProjectileRange () / 2) {
16              FireSpread (target);
17          }
18          else {
19              FireFast (target);
20          }
21      }
```

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.

Definition!

NULL:
The keyword used to represent missing or nothing.

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

```
4 public class AdventureGame : MonoBehaviour {
5     public GameObject sword;
6     public NPCManager npcManager;
7
8     public GameObject boss;
9
10
11    // Use this for initialization
12    void Start () {
13        ItemHandler.SpawnItem (sword, 0, 3, 10);
14        FriendlyNPC friendlyNPC = npcManager.SpawnMostCustomNPC (1, 1, 1.5f, 2, "None", "Pumpkin","Pumpkin", "Snow", "Pumpkin", "Pumpkin")
15        friendlyNPC.SetQuest ("FirstQuest");
16    }
17
18    // Update is called once per frame
19    void Update () {
20        if (boss == null) {
21            GameManager.EndGame ("You win! Thanks for playing!");
22        }
23    }
24 }
```

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

Lesson Seven

Where No One Has Gone Before

1 Introduction to Lesson Seven

Lesson seven serves to test the skills you've learned throughout these lessons, along with giving you more freedom to create your own game. In this lesson you're allowed free reign to alter any of the four games you've created so far or two new games (a platformer and a murder mystery).

We present each of the prior games in the sections below, discussing potential ways you could extend them into more interesting experiences. Below that we introduce the two new games, discuss how they work, and suggest ways you could extend them. You'll have to pick one (or more given time) of these games to start from to make something uniquely your own.

2 Lesson One Extension

LessonOne.unity is a simple "tower defense" game. At the end of lesson one you should have ended with a game where the player has to protect a tower from some snowmen. It's a fun, short game but not particularly complex. You could consider changing the games in a number of ways, and we've included a few suggestions below to get you started. In general there are three main scripts to look at. LessonOneGenerator.cs generates the map, LessonOneGame.cs handles the game itself and is the script you edited in lesson one, and SnowmanController.cs sets up the snowmen's positions and goal.

- Create an initial race to defend the tower. If you open up LessonOneGenerator.cs you can see the selection of for loops and calls to "MapBuilderHelper.BuildBlock" to build up the world. If you changed this to include (for example) a series of platformers the player had to jump through before building up a defense that'd definitely change the game. You'd also have to change the player's starting position, but you can do that by changing the Player object's transform's position values in the Inspector view.
- You could add enemies to defeat. You could add the boss or ghosts to the world (along with the sword). Perhaps even a quest with an NPC to get a sword in order to fight them off?
- You could change the shape of the tower area. If you open up LessonOneGenerator.cs you could make a cliff on one side of the tower that would require new kinds of defenses to protect it.
- You could make multiple towers. Via changing LessonOneGenerator.cs and SnowmanController.cs (the "SetGoal" function) you could make multiple towers for the player to defend for an increase in difficulty.

There are of course many possibilities and many ways you could adapt this game to make it more to your liking. If you need a reminder on how to generate environments we recommend checking out lesson four, or checking the Platformer section below for some examples.

3 Lesson Two Extension

LessonTwo.unity is a simple "escape the room" game. At the end of lesson two you ended up with a pretty simple puzzle to find a key or sword that you had hidden in a small cave. There are a number of ways to change this game. For example you could create a new kind of item needed to

escape (like in lesson five), make a bigger or more maze-like cave (using what you learned in lesson four), or add a monster or boss in the cave (like in lesson six).

Regardless of what you want to do, there're a few scripts that can help. LessonTwoGame.cs is the script you altered in lesson two, it handles the placing of the sword and key, and the logic for when to open the locked door. LessonTwoGenerator.cs handles the generation of the cave and the door. LockedDoor.cs handles the logic for "opening" the door, along with being the invisible force that keeps you from pressing right up against it (check the LockedDoor object in the Hierarchy view of LessonTwo.unity to see what we mean). Below we've included suggestions on ways to change the game to make it more interesting.

- Make it a climb to get to the locked door. Alter LessonTwoGenerator to make the cave steeper and require a climb to get to it and the items needed to escape.
- You could add enemies to defeat. What if you extended the cave in LessonTwoGenerator and then used a call to NPCManager.SpawnEnemies to spawn some enemies in one of the cave "rooms". Then getting the sword might be necessary to get the key!
- Add a new item, and maybe another lock. You could use the same techniques from lesson five to create a magic key needed to get by a first door, before moving on to the original door and its lock and key system.
- Add additional code to LessonTwoGenerator to "bury" the key and sword, requiring digging to get it. As you might recall from lesson two, you can't break through the stone that makes up the cave but you can use calls to MapBuilderHelper.BuildBlock to build dirt or stone passages that the player has to clear out to advance. Maybe with enemies hiding behind certain walls?
- Add an NPC in the cave to add a little story. It could be fun to add a little narrative to the game by including an NPC to talk to about the cave or "magic" locked door.

4 Lesson Three Extension

LessonThree.unity is a simple "beat 'em up" game, where the player must fight off a large wave of ghosts. If you've gone through lessons four through six, you're already "broken" this game as destroying ten ghosts doesn't lead to a win state (you changed Sword.cs in lesson five)! So if you wanted to extend this beat 'em up game, you'd have to do something about that. Additionally you may want to change the number of enemies or introduce waves of enemies to make it more like a true "beat 'em up" game.

You should be able to make all the changes you want in this game via changing a few different scripts and components. TerrainGenerator.cs controls the generation of the initial map. There are in addition four "Invisible Wall" objects that you can see in the Hierarchy. These are the objects that stop the player from "leaving" the island that is initially generated. BeatEmUpGame.cs is a generalized script that could be used to instantiate different NPCs (perhaps to give different quests in order to control waves of enemies?), but currently contains nothing. Sword.cs controls the behavior of the sword in the game, and the placement of the "Sword" object in the hierarchy determines where the Sword object initially spawns (it currently spawns "on" the player). Enemy.cs handles the behavior of enemies and the number that are initially spawned are determined by the "Num To Spawn" variable on the Admin object's NPC Manager component.

Below you can find a few suggestions on how to make the game more interesting, but feel free to mix and match or come up with your own!

- Make multiple waves of enemies. The easiest way to make multiple waves of enemies would be to spawn a series of NPCs, each of whom would start a unique "challenge" via spawning a given number of enemies with "NPCManager.SpawnEnemies". Maybe make a boss area that the final NPC teleports the player to? Up to you!
- Change the shape of the island to make a series of challenges. You could get rid of the current code in TerrainGenerator.cs and instead replace it with a series of block placements to create larger and larger islands, each with more and more enemies on it.
- Make an additional challenge to collect cats/coins along with fighting off the enemies so that the player can "trade in" these items for a more powerful weapon (the MagicSword) using the ItemHandler as you did in lesson five.

5 Adventure Game Extension

LessonFour-Six.unity represents a simple adventure game that you built entirely from scratch. But there's a number of things missing from it that could make it more of a true adventure game. For example, most adventure games have small towns with NPCs that serve to add "flavor" to a game, along with offering the possibility for trade or healing. In addition most of the time enemies appear in "dungeons", instead of just out on the map. Lastly most adventure games involve more than just combat, with most having elements of "platformer" or "puzzle" games as well. You could extend LessonFour-Six.unity to add these things, or something entirely different!

You should be fairly familiar with the scripts that make LessonFour-Six.unity work at this point, but we'll go over them briefly. AdventureGameGenerator.cs generates the map. AdventureGame.cs handles the code to run the game, such as the initial placement of NPCs. MagicSword.cs handles the use/behavior of MagicSword. Sword.cs handles the behavior of sword objects, as Enemy.cs handles the behavior of the enemies. Various different scripts extending from Quest.cs can be used to create NPC dialogue and quests. BossAttackController.cs handles the attack behaviors of the boss. PlayerInfo.cs holds the variables that are specific to the player. Lastly, ItemHandler.cs controls the spawning and collection of items in the game.

With these in mind there are a large number of changes you could make to the game, including all of the suggestions above. We explain how to tackle a couple of them at a very high level below.

- Adding a town with random NPC "conversations". To create a town we'd recommend creating a single house using the same approach we used to create trees in the optional part of lesson four. Then modify it to create a function that places houses across an area. Then create NPCs, each with their own quest that cannot start, but change each Quest's CannotStart function to be "flavor text" for that NPC. Things like "I love my town!" or "Pleasant day out, isn't it?".
- Adding a dungeon with puzzles. To create a dungeon with "puzzles" we'd recommend changing AdventureGameGenerator.cs. You could start by building up a single dungeon room, turn that code into a function, and then place lots of dungeon rooms side by side to create the whole dungeon. To create a puzzle you can use the LockedDoor.cs example from lesson two

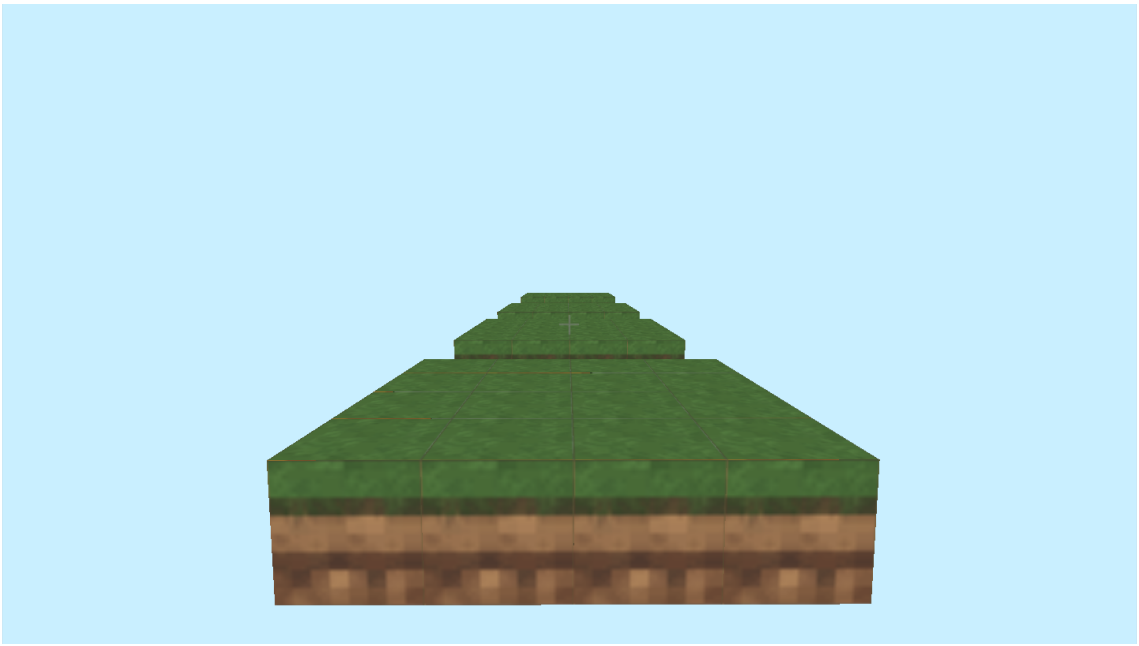

Figure 1: The simple Platformer game

to create a door (perhaps to the boss) that requires finding a Key to pass it. You could also add a "platforming" challenge by having a room that the player can only cross by jumping on small floating platforms. At the end put the boss you could add an NPC for the player to "save" to win the game.

6 Platformer

Platformer.unity contains everything needed for a very simple platformer game. A "platformer" is a genre of game where a player attempts to make a series of jumps from "platformers", while potentially collecting items and dodging enemies. Before reading any further we recommend opening up the file, and then pressing the "play" button to play the platformer game as it exists so far. As a warning, if you don't make a jump you'll have to exit out of the game yourself as there's no "lose state". When you play the game you should see something like Figure 1. Making each jump from platform to platform should lead you to a final long platform with an NPC who will congratulate you for making it.

The game is very simple at present. There are no items to collect or enemies to avoid, and there's not even that many jumps to make. In addition, there's nothing to stop you from falling forever if you miss a jump, which is really annoying. But you could change that!

There are only three real scripts that create this entire tiny game. They are:

- PlatformerGame.cs (located in Codebase > Platformer > PlatformerGame.cs), handles spawning the one NPC and assigning it the quest EndQuestPlatformer.

- EndQuestPlatformer.cs (located in Codebase > Platformer > EndQuestPlatformer.cs), handles the end text and actually ending the game.
- PlatformerGenerator.cs (located in Codebase > Environment > Map > Generators > PlatformerGenerator.cs), handles the creation of all of the different platforms starting with the one the player begins on.

There are a lot of ways to extend this game, and we give a brief description of how to make a few extensions below.

- Change the platforms. The simplest thing to do would be to alter the arrangement of the platforms, to make a longer game or to alter the challenge. To do that you can look at the examples in PlatformGenerator.cs and just go from there. Make sure to reposition the npc in PlatformerGame.cs if you change the end point!
- Add items to collect. Using the same approach as lesson five, you could add coins/cats to be collected, meaning that you could have branching paths of platforms. Then it'd be pretty simple to require a given number of coins/cats before the game will end. You could even create your own brand new item to collect!
- Add a "death". You could change PlatformerGame.cs to have a reference to the player and then check to see if player.transform.y (how high the player is) goes below a certain value. If you do that you could create a "lose state", or just "teleport" the player back to the starting point if the player gets too low.
- Add enemies. It'd be easy enough to add enemies at certain points to make dodging enemy bullets part of the game.
- Add an item or NPC who changes player's jump. It'd be pretty easy to alter PlayerInfo's jumpValue's value. Then you could give the player "mega jumps" that would be necessary to make it between the final platforms.

7 Murder Mystery

MurderMystery.unity contains a very, very simple "mystery" game. In mystery games, sometimes called "secret box" or "walking simulators", a player must solve a mystery by talking to NPCs and completing simple puzzles. Before continuing, go ahead and open MurderMystery.unity and play it. You'll need to talk to the NPC, find the hidden "sword" and then return it to "solve the mystery". You should see something like Figure 2 when you start. As a hint if you can't find the sword: while most of the blocks are unbreakable, not all of them are.

The present game lacks many of the common attributes of mystery games. For example, there's only one NPC to talk to. In most mystery games a player has to talk to many different NPCs (some who will even lie to the player) and must use what they say to solve the mystery. One other major difference is the lack of a story. Who has been killed? Who would have the motivation to murder them? You can make changes to add these attributes and more.

There are only three real scripts that create this entire tiny game. They are:

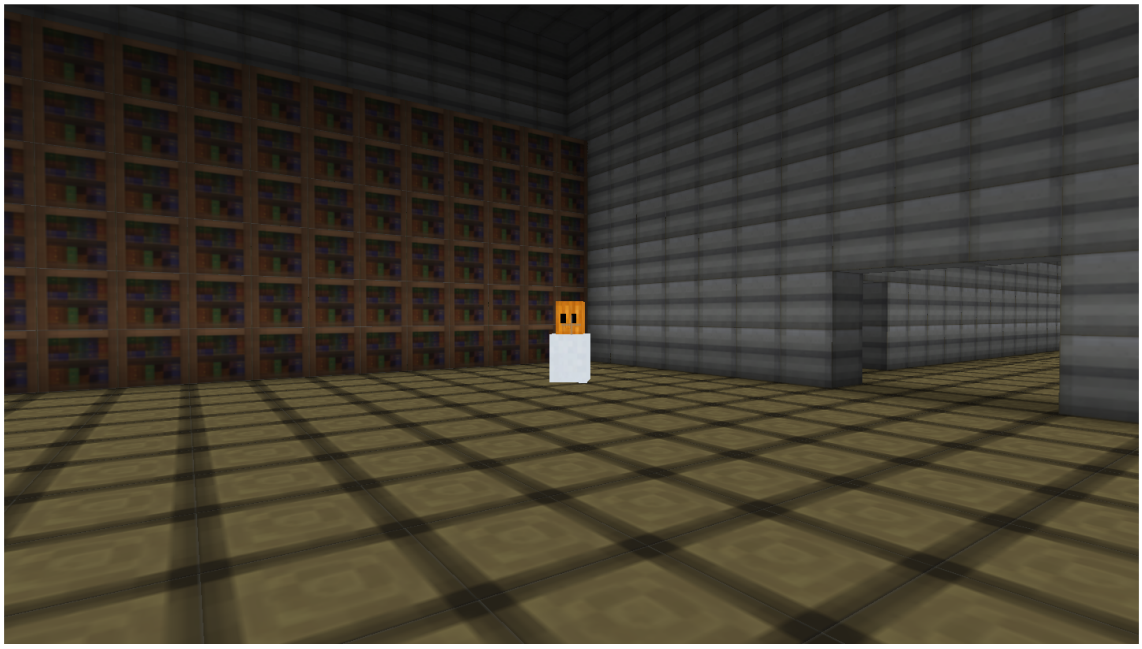

Figure 2: The simple Murder Mystery game

- MysteryGame.cs (located in Codebase > MurderMystery > MysteryGame.cs), handles spawning the one NPC and assigning it the quest MysteryQuestStart.
- MysteryQuestStart.cs (located in Codebase > MurderMystery > MysteryQuestStart.cs), handles the initial dialogue and the end condition.
- MysteryGenerator.cs (located in Codebase > Environment > Map > Generators > MysteryGenerator.cs), handles the creation of the "manor", that the game takes place in.

There are many possible ways to extend this game, we go through a few and how we'd suggest tackling them below.

- Add more characters. This is a simple change, simply adding an NPC to each room and giving them some manner of flavor text. You could make it so that after talking to one NPC, another's dialogue will change based on creating and changing variables in PlayerInfo marked as "public static".
- Add new items as red herrings or as pieces of a puzzle. You could place a Magic Sword in one room, and put clues in NPC dialogue so the player knows that it can't be the murder weapon. Alternatively, you could require the player find a key to open an area before they can access the sword (like the locked door in lesson two).
- Change the environment. You could add further decoration to each room, perhaps to explain the "personality" of each occupant. For example, you could create a bed function like the tree function in lesson five, and put beds of different sizes in each room. Alternatively you

could make the manor bigger or smaller, perhaps with smaller hallways to make it creepier. You could even make more fantastic environments like areas to represent the "dreams" of the different NPC suspects that the player warps to if they get too close (using the quest system).

- You could make an exciting "Third Act". You could change MysteryQuestStart.cs so the game doesn't end when the player brings the item to the original NPC. Instead it prompts the true murderer to transform into a "boss" or "ghost" and attack!

8 End of Lesson Seven

The purpose of lesson seven is to give you a chance to flex your creativity with the skills you've learned, along with challenging you to learn more. While working on it, we recommend that you have friends play through your game. It can be hard to tell while you're making something how hard it'll be for someone else and having others play your game is the best way to test this. You might even get cool ideas for new things to try!

If you liked these lessons and want to try to build your own games from scratch, we recommend starting with the official Unity3D tutorials. They'll help you to build games that aren't in this engine, and can give you even more freedom to create something unique to you.