Lesson Four

Starting an Adventure!

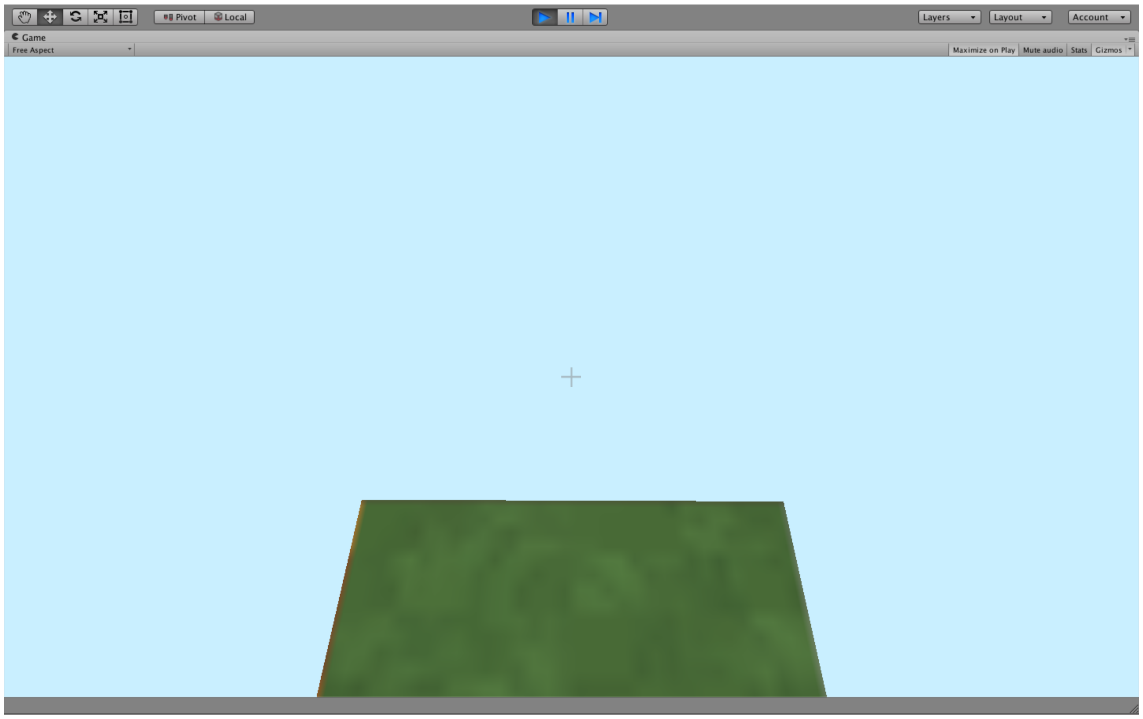


Figure 1: How LessonFour-Six begins.

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.

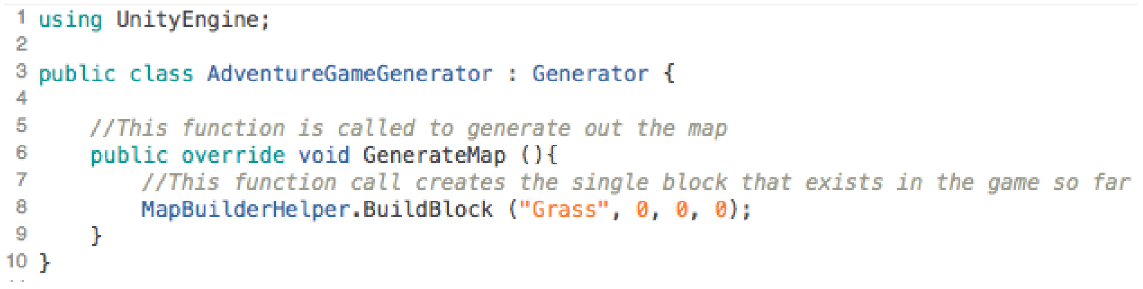
For the next three lessons we'll be ditching the "fixing broken games" format and instead making a mini Adventure game from scratch. In this lesson we'll focus on building up a meaningful virtual world and teach a powerful new programming technique: loops. The theme of this lesson will be doing more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

Definition!

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

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:

```
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.

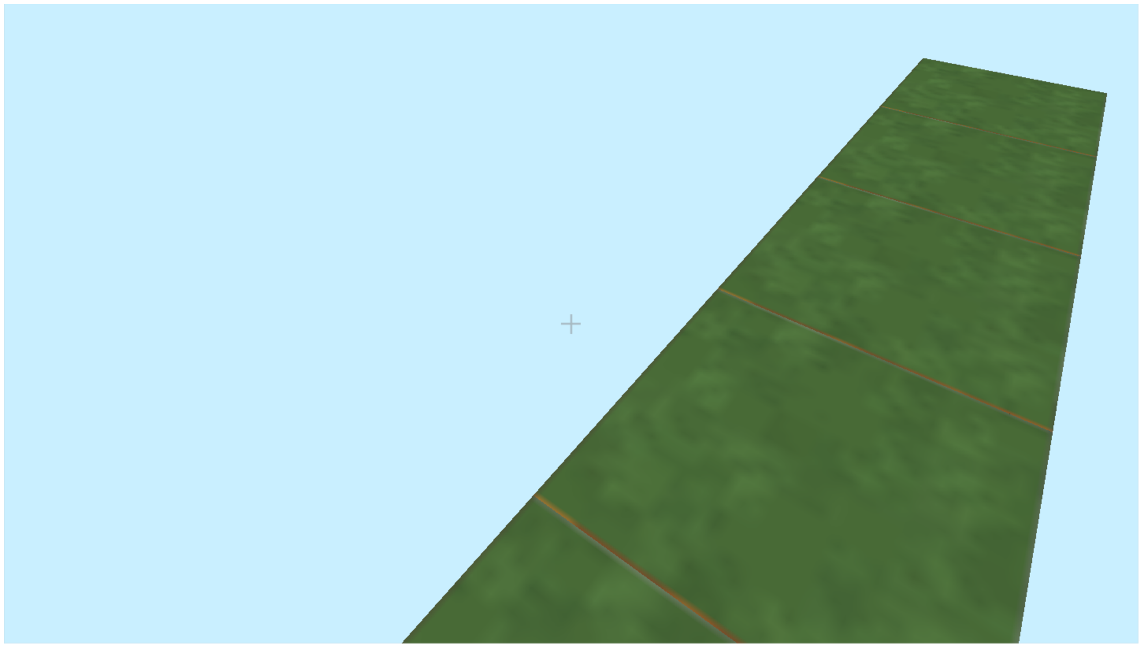
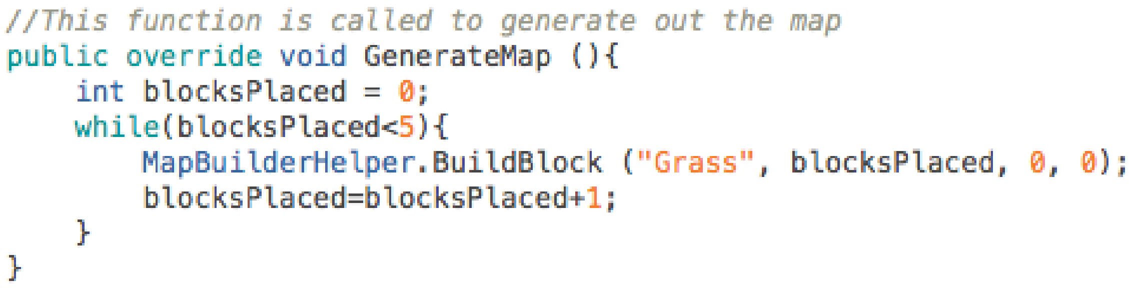


Figure 3: Five blocks floating in space, thanks to our while loop.



```
//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.

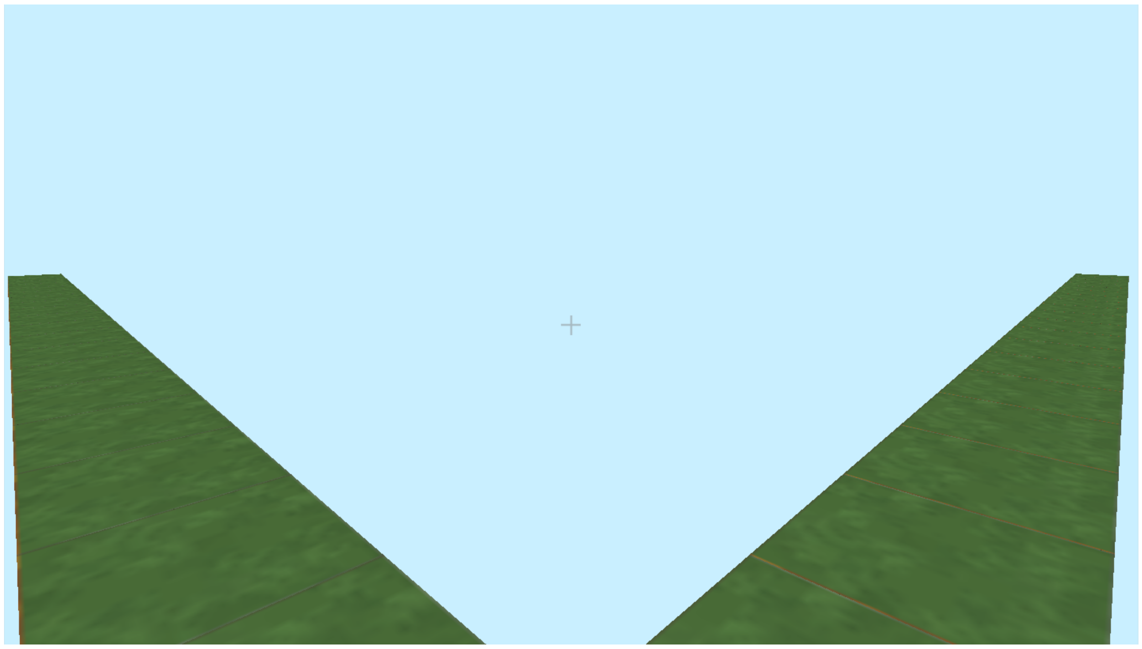


Figure 5: The effect of adding a second while loop with a line in the z-direction.

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

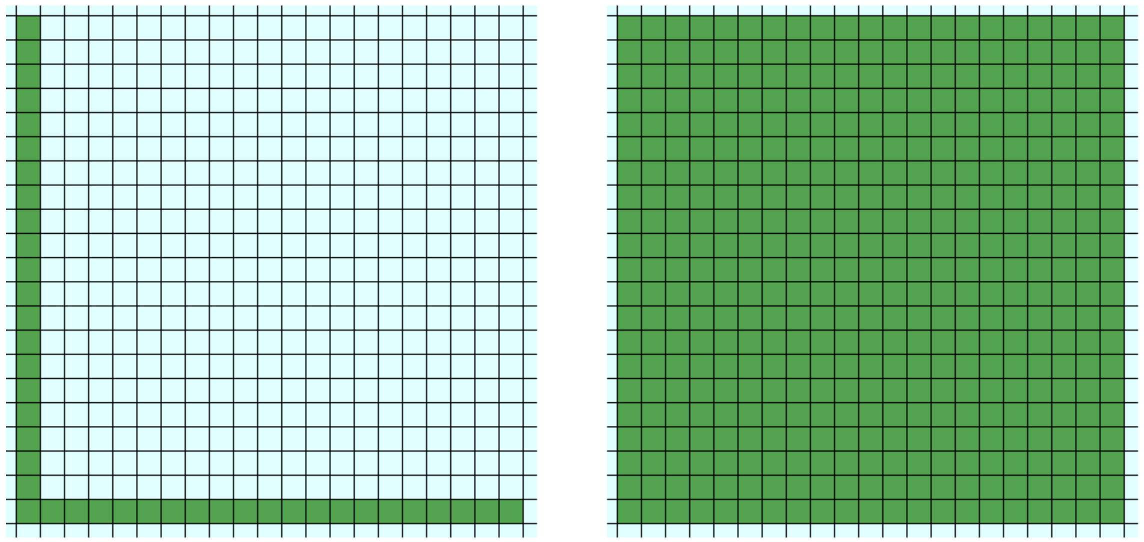


Figure 6: Left: An overhead visualization of what we currently have, two lines of length 20. Right: An overhead visualization of what we want. A square of 20x20 block area or 20 lines of length 20 placed side by side.

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.

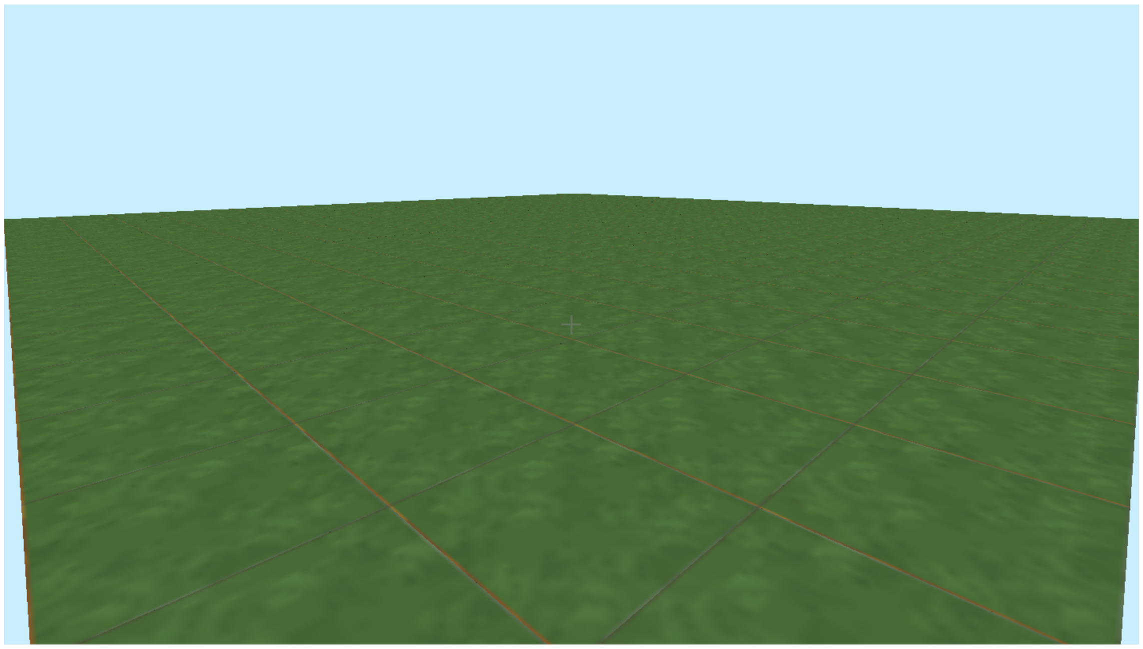


Figure 7: The full 20x20 block area!

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.

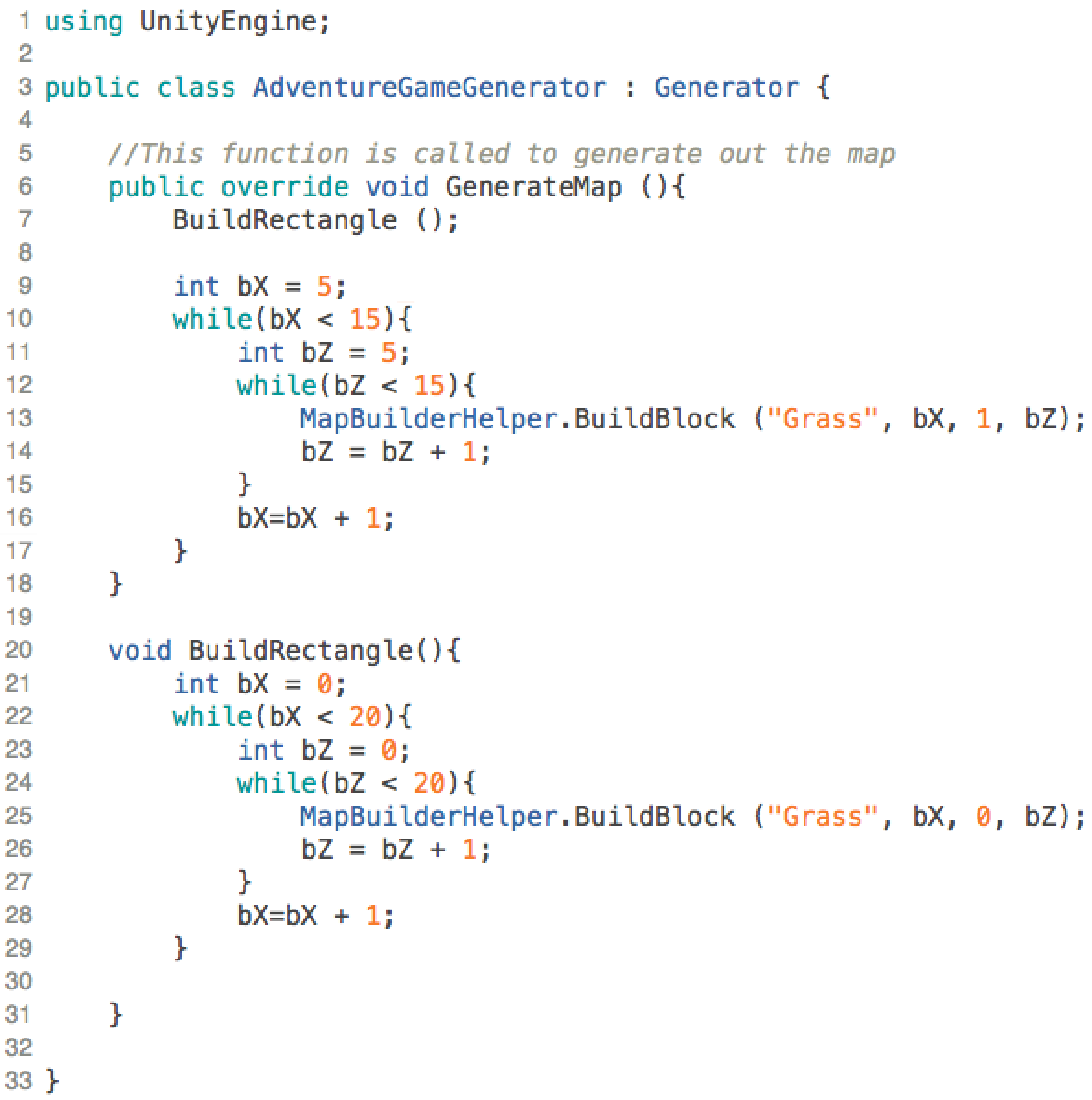
If you run the code again, you shouldn't see any changes! That might seem a bit odd, since we moved the code that did that from GenerateMap into BuildRectangle. But it should make sense if we remember that calling a function calls every line of code within that function. That means when the single line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



```csharp
using UnityEngine;

public class AdventureGameGenerator : Generator {

    //This function is called to generate out the map
    public override void GenerateMap (){
        BuildRectangle ();

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

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

    }

}
```

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

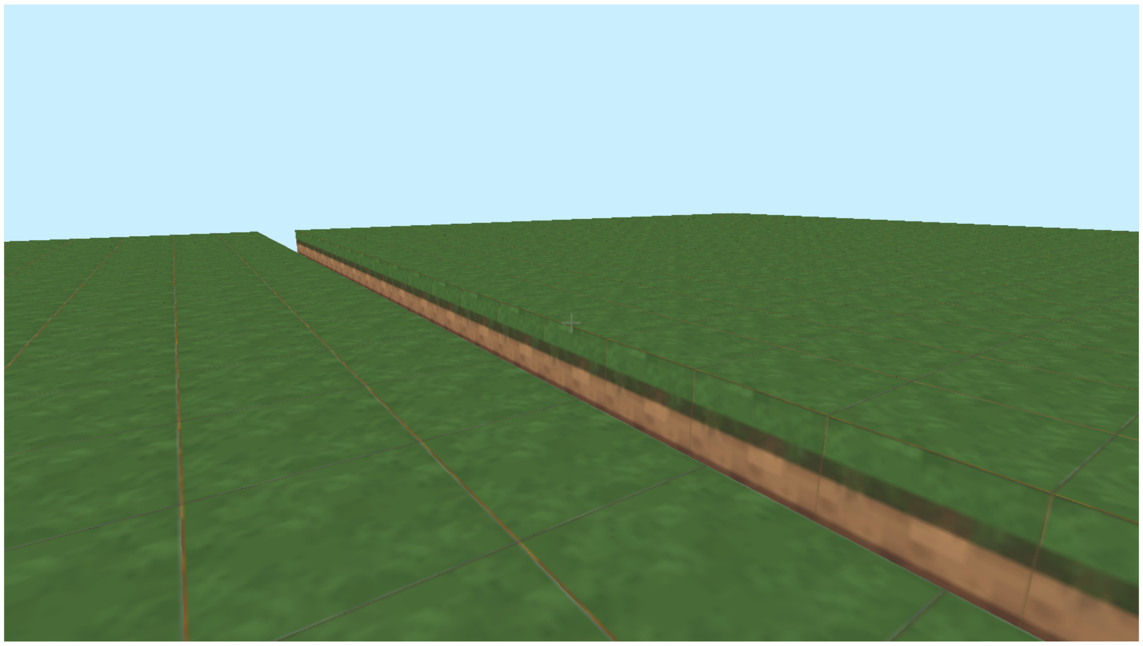
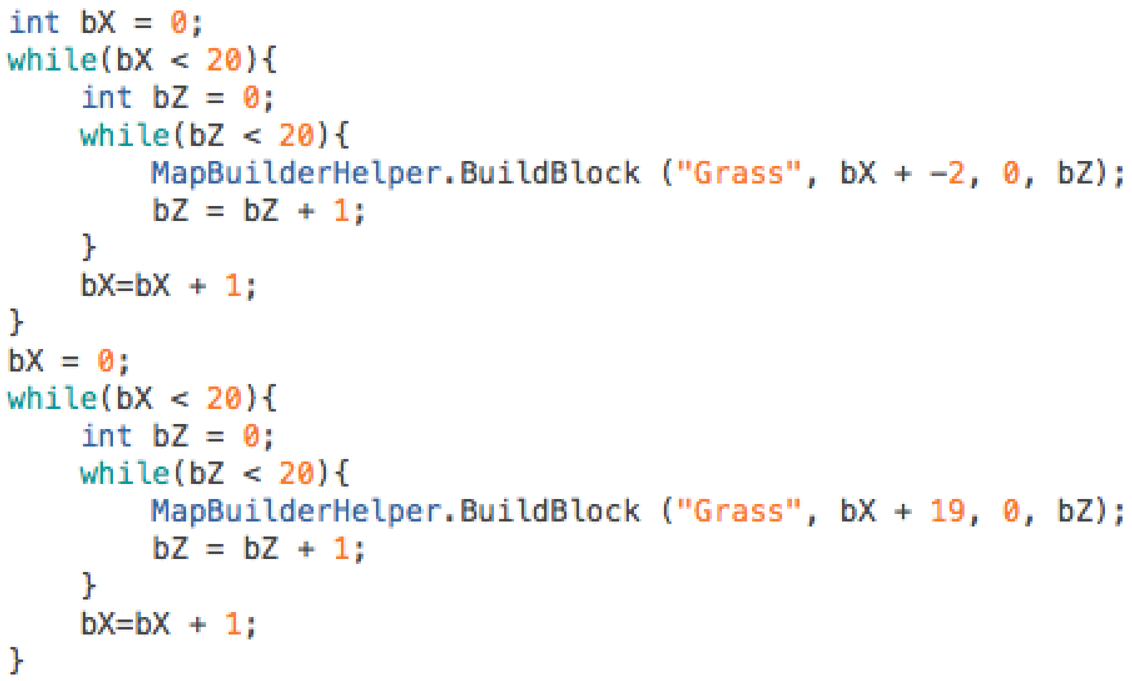
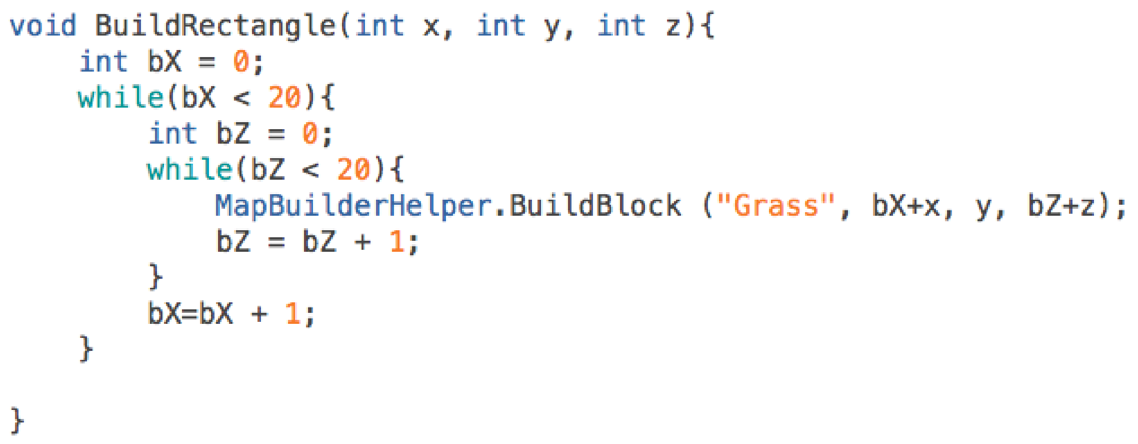


Figure 9: Two squares of grass!



```
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

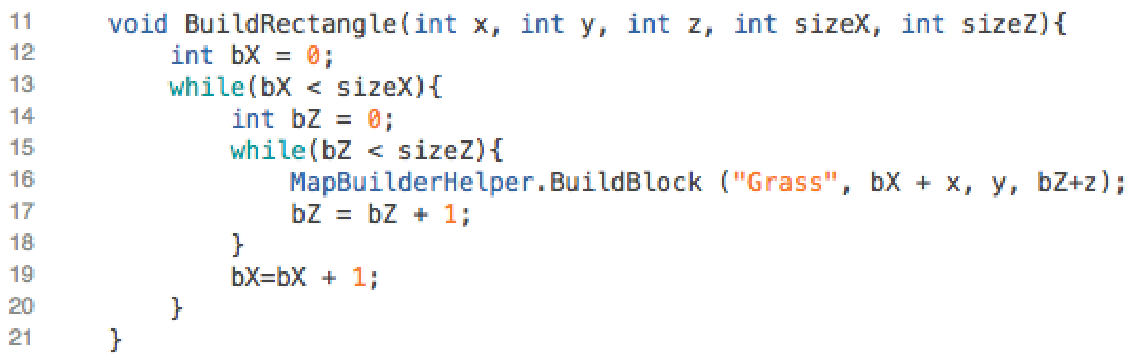
We already added an argument to control where the rectangle is built on the x axis, so it makes sense to add arguments for both the y and z axises. We called these arguments “y” and “z”. You should be able to use how we used the x argument to figure out how to add y and z arguments to control those axises. 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



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

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.

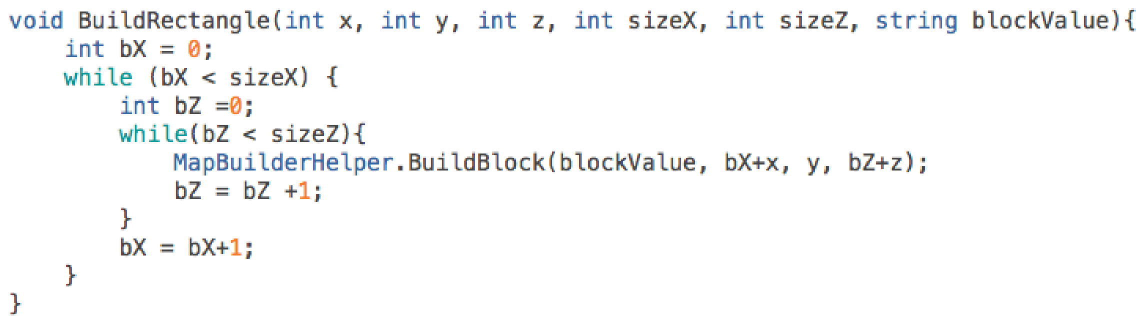
You've successfully made a a reusable and useful function. Feel free to play around with calls to BuildRectangle in GenerateMap. Add more calls to BuildRectangle to spruce up your island, add a second "floating" island, or even a set of small "stepping stone" islands. Test out your changes by playing 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".

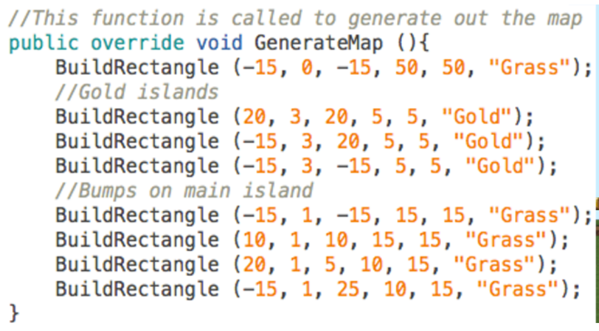
Play the game, and note the difference! Don't worry, you won't be hurt by walking around on this lava island.

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");
}
```

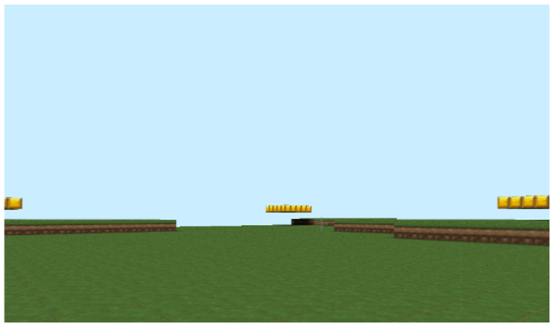


Figure 14: Our "final" island on the right and the code to make it on the left.

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!

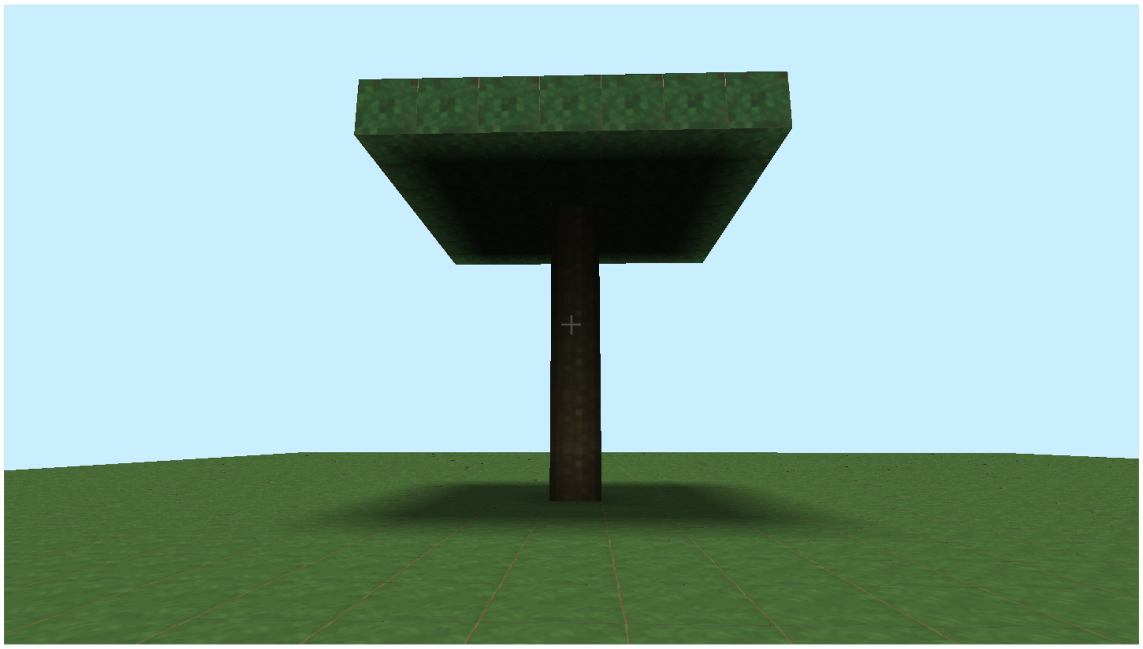


Figure 15: Our “flat” tree.

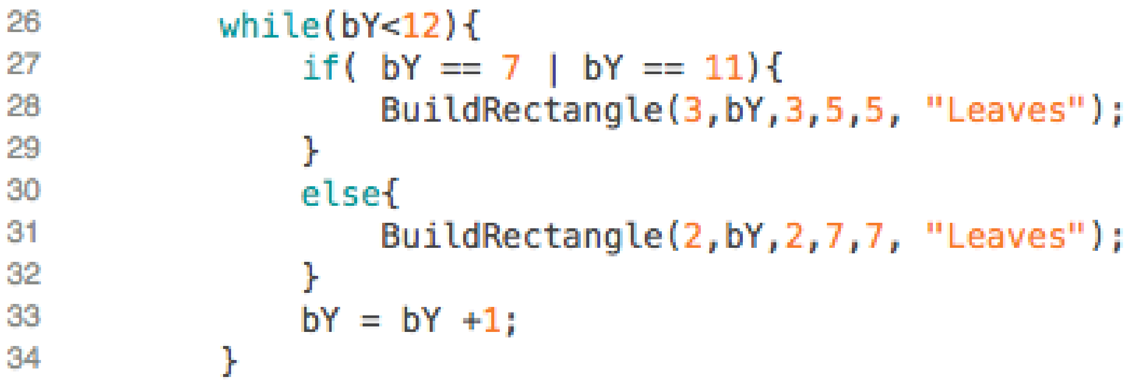
You should now see a much more tree-looking tree. That’s great! But it could still be better. What if we instead changed the leaves so that the first and last of the rectangles was a couple blocks smaller? It’d give the tree a “rounded” appearance. To do that we can make use of conditional statements (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.

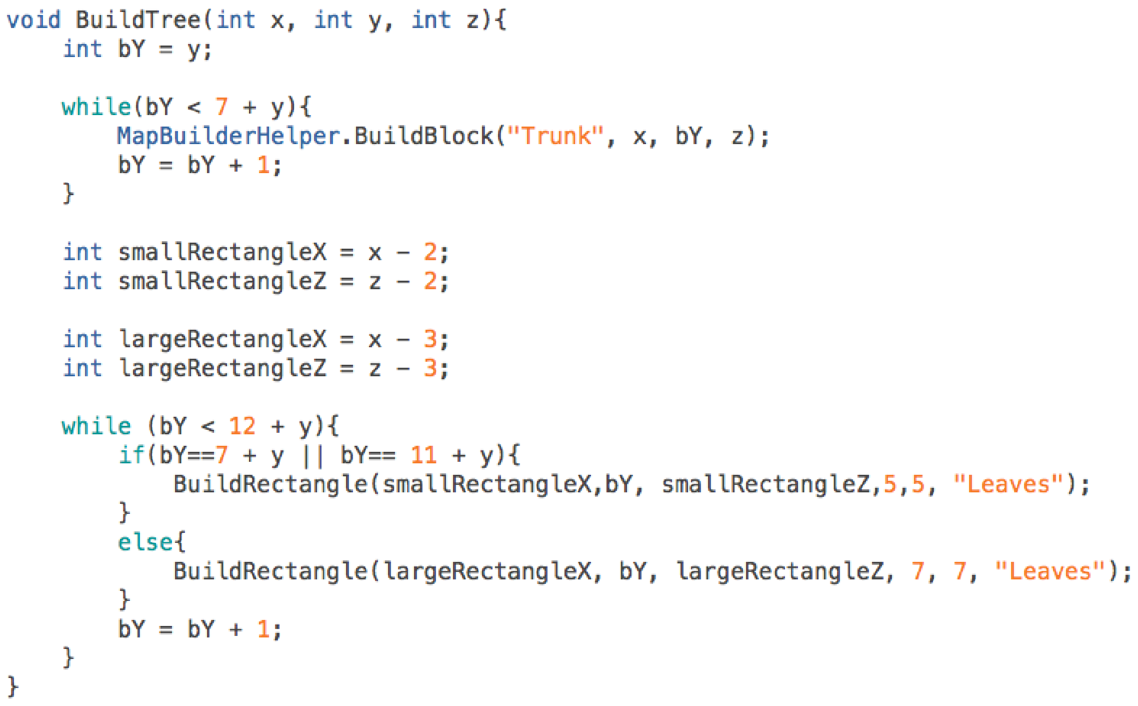
Play the game and note the changes. If you prefer the original appearance, feel free to change the code back. You could also experiment with other conditional statements with other conditions!

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.



```
while(bY<12){
    if( bY == 7 | bY == 11){
        BuildRectangle(3,bY,3,5,5, "Leaves");
    }
    else{
        BuildRectangle(2,bY,2,7,7, "Leaves");
    }
    bY = bY +1;
}
```

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.