Lesson Two

Escape the Cave

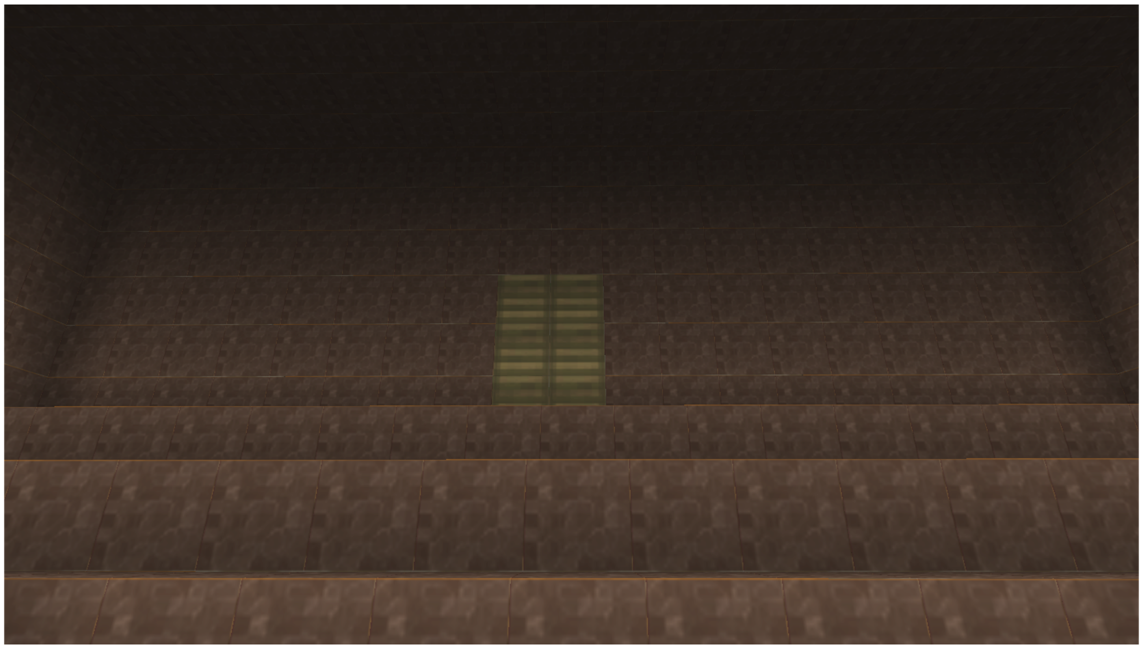


Figure 1: The door out of the cave (currently impossible to open).

1 Introduction to Lesson Two

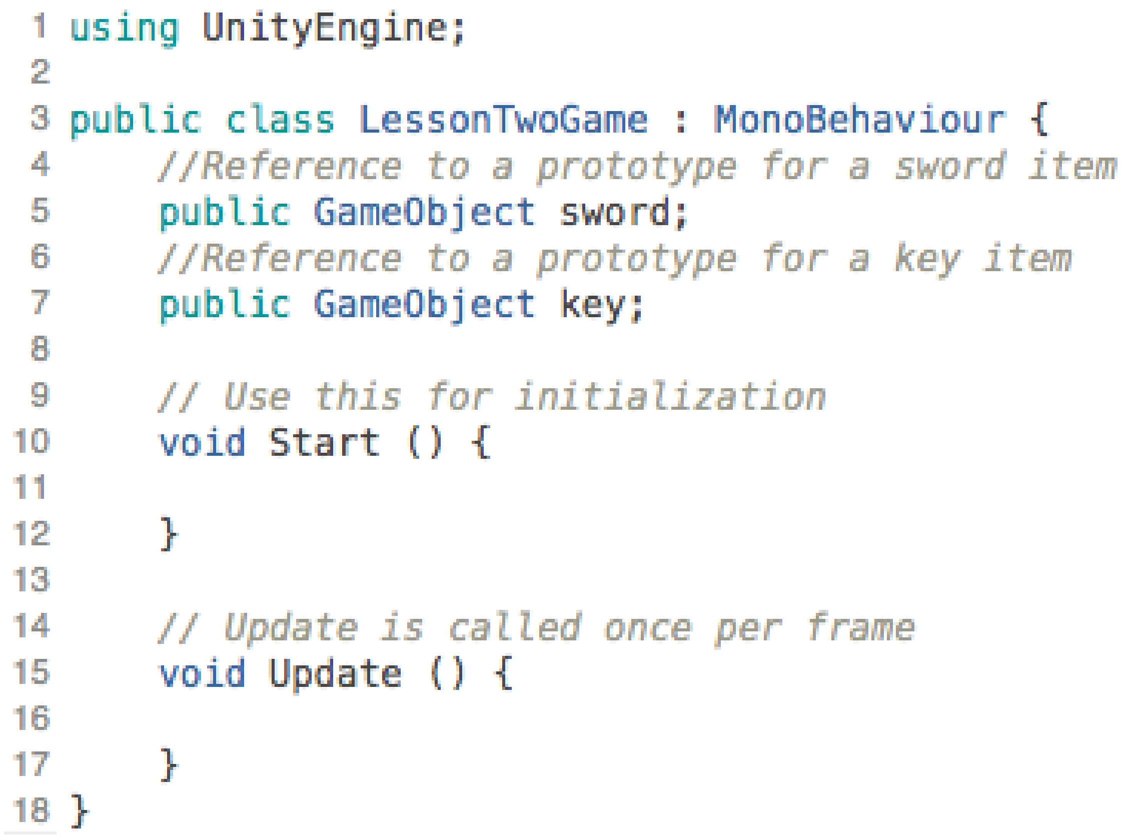
Games require player choice, the player making different decisions based on the situation to ensure victory. Code can actually make use of a similar (though simpler) form of decision making. In this section we'll look into how code can make decisions based on different information to fix a simple "Escape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



```
using UnityEngine;

public class LessonTwoGame : MonoBehaviour {
    //Reference to a prototype for a sword item
    public GameObject sword;
    //Reference to a prototype for a key item
    public GameObject key;

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

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.

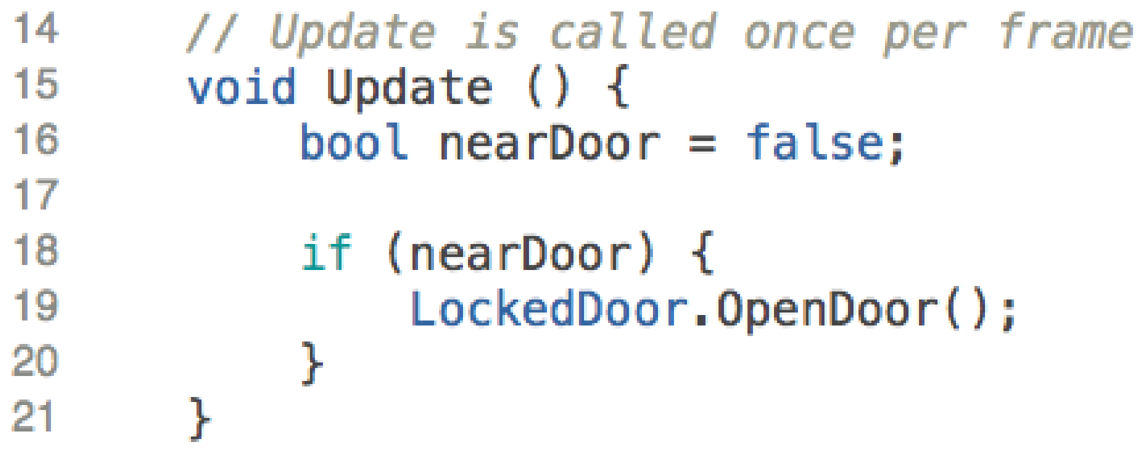
We mentioned how to represent conditions (the things that go into the parenthesis) in code in the previous lesson. There's actually an entire primitive variable type that stores true and false information: **booleans**. If the boolean within the parenthesis of the if statement is True, the code within 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



```
// Update is called once per frame
void Update () {
    bool nearDoor = false;

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

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.

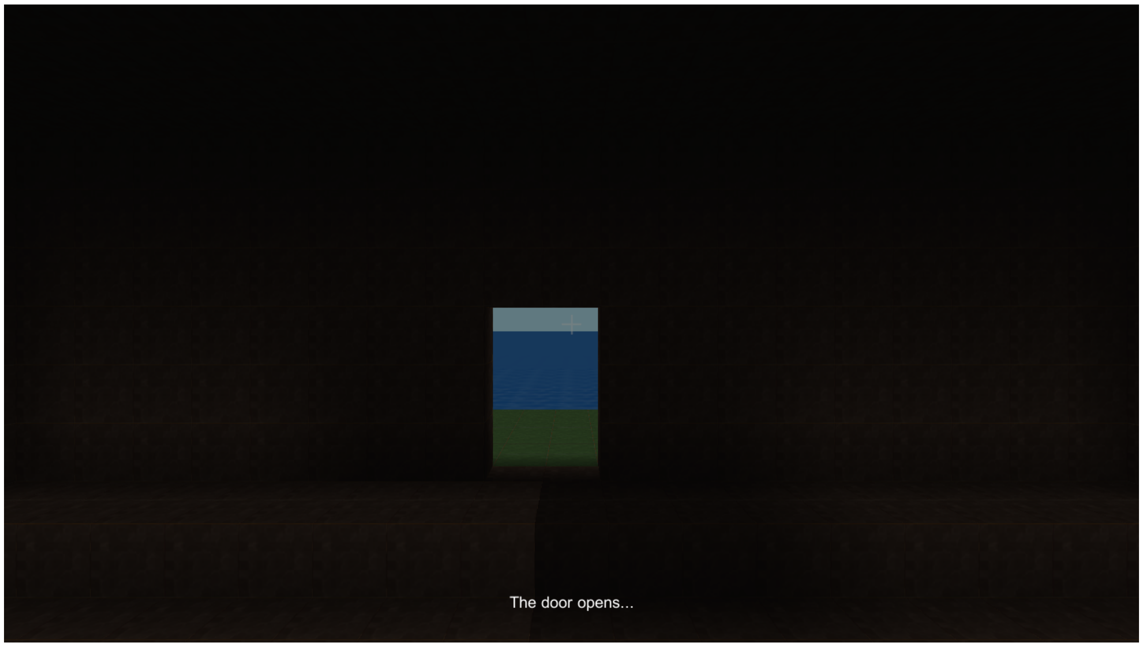


Figure 4: “Winning” the escape the room game a bit too early.

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;
```

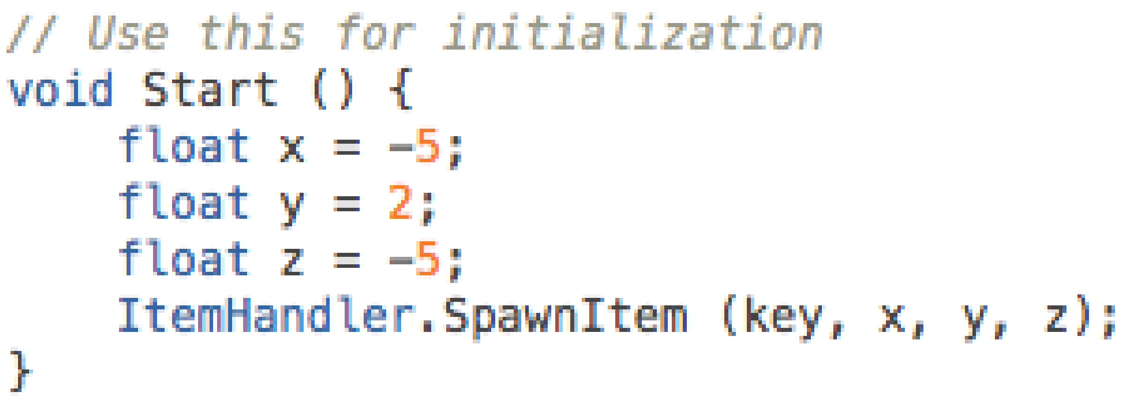
Play the game again (make sure to save first). You should now find that you can wander around the cave freely until you get right up to the door. At this point the game will pause and the door will open!

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

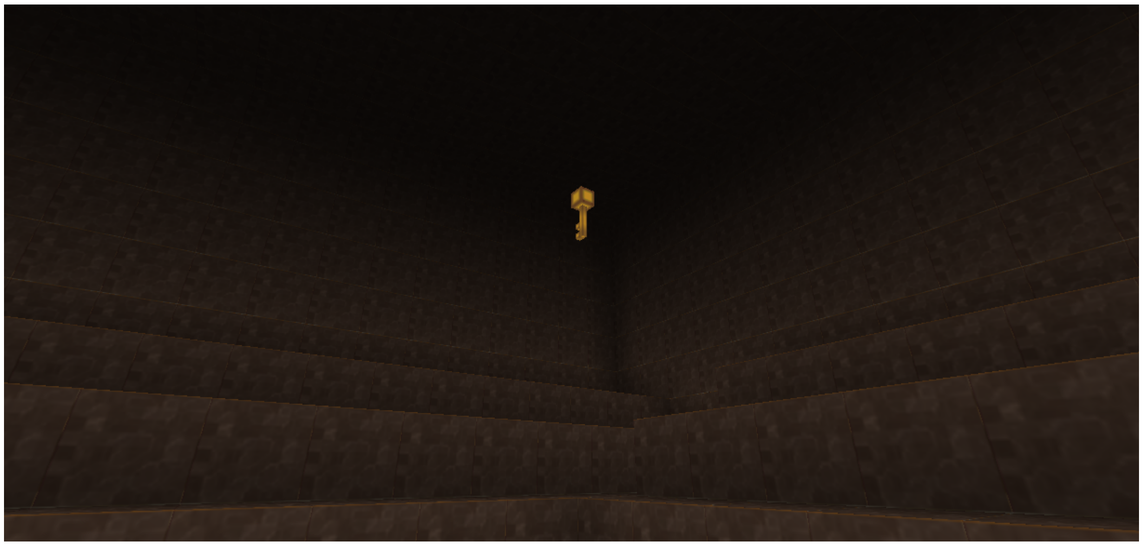


Figure 6: The key floating in the air thanks to a y value of 6.

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.

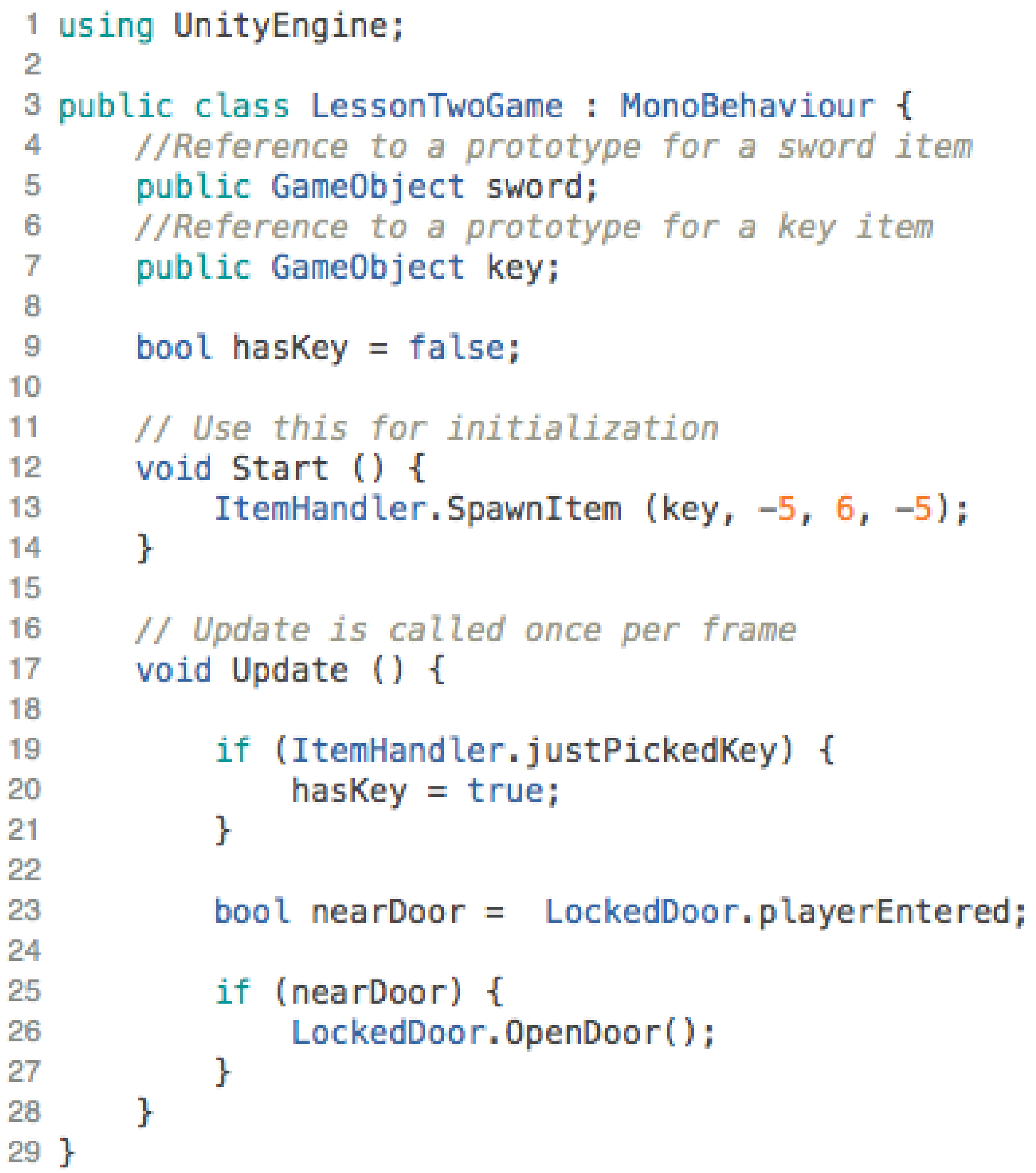
Play the game again. First try to walk through the door without getting the key, then with the key. It works!

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

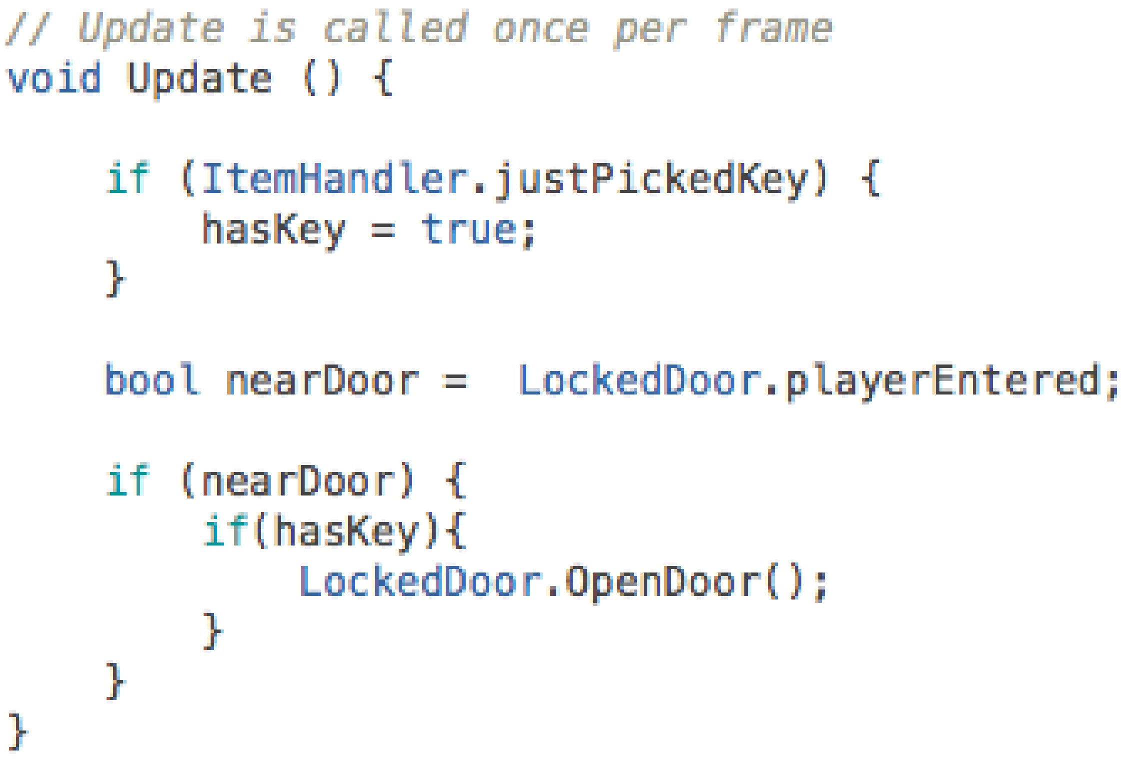
If you'll recall, in the last lesson you wrote text that displayed on either a win or a loss. We can actually display text like this even without the game ending. We'll use another function "GUIManager.SetDisplayText" to do this. Again, GUIManager is the name of the class, with SetDisplayText being the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



```csharp
using UnityEngine;

public class LessonTwoGame : MonoBehaviour {
    //Reference to a prototype for a sword item
    public GameObject sword;
    //Reference to a prototype for a key item
    public GameObject key;

    bool hasKey = false;

    // Use this for initialization
    void Start () {
        ItemHandler.SpawnItem (key, -5, 6, -5);
    }

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

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

        bool nearDoor =  LockedDoor.playerEntered;

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

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**.

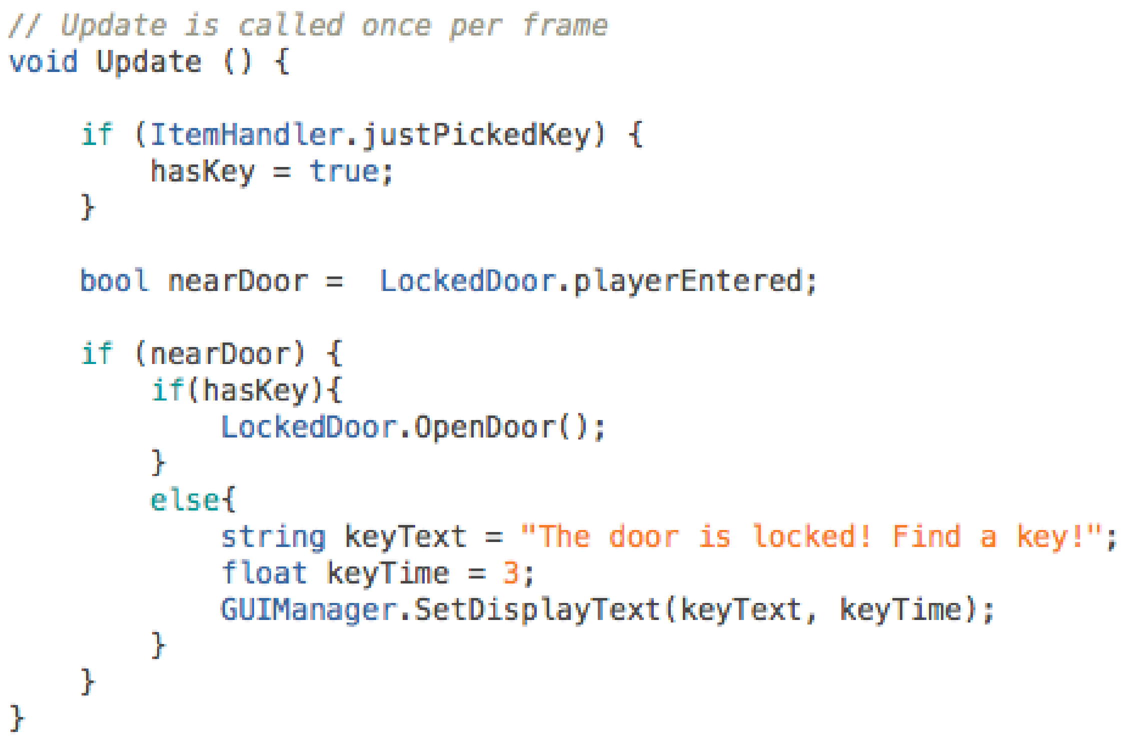
Else statements can only be located underneath another conditional statement (an if statement for example, though we'll introduce another kind of conditional statement soon). They work in a way similar, but opposite, to if statements. In that the code within an else statement will only run if the code 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.

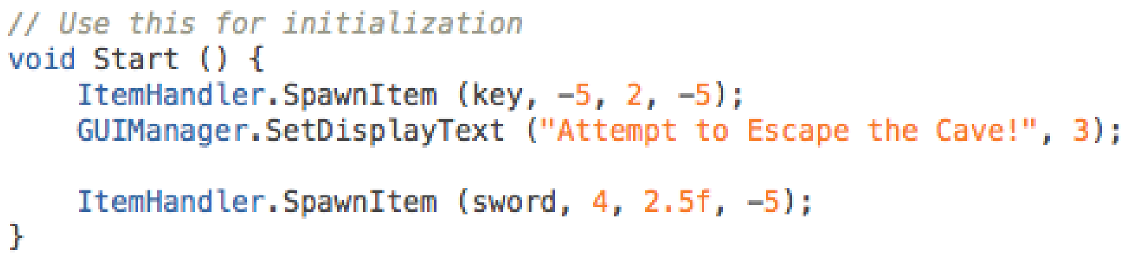
Play the game. Try going up to the door without the key and notice our new statement pop up! Feel free to change the string value of keyText as well. You could even make it seem like the door is talking to the player by just changing the text to something like "Door: Now, now I won't let you through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

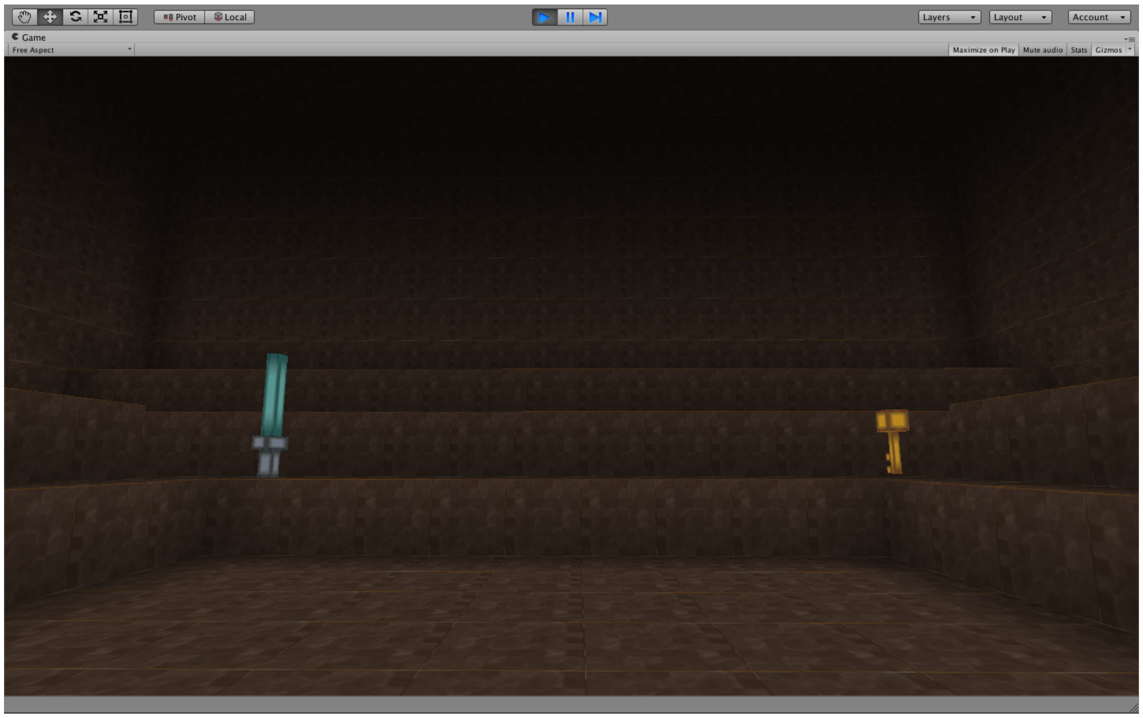
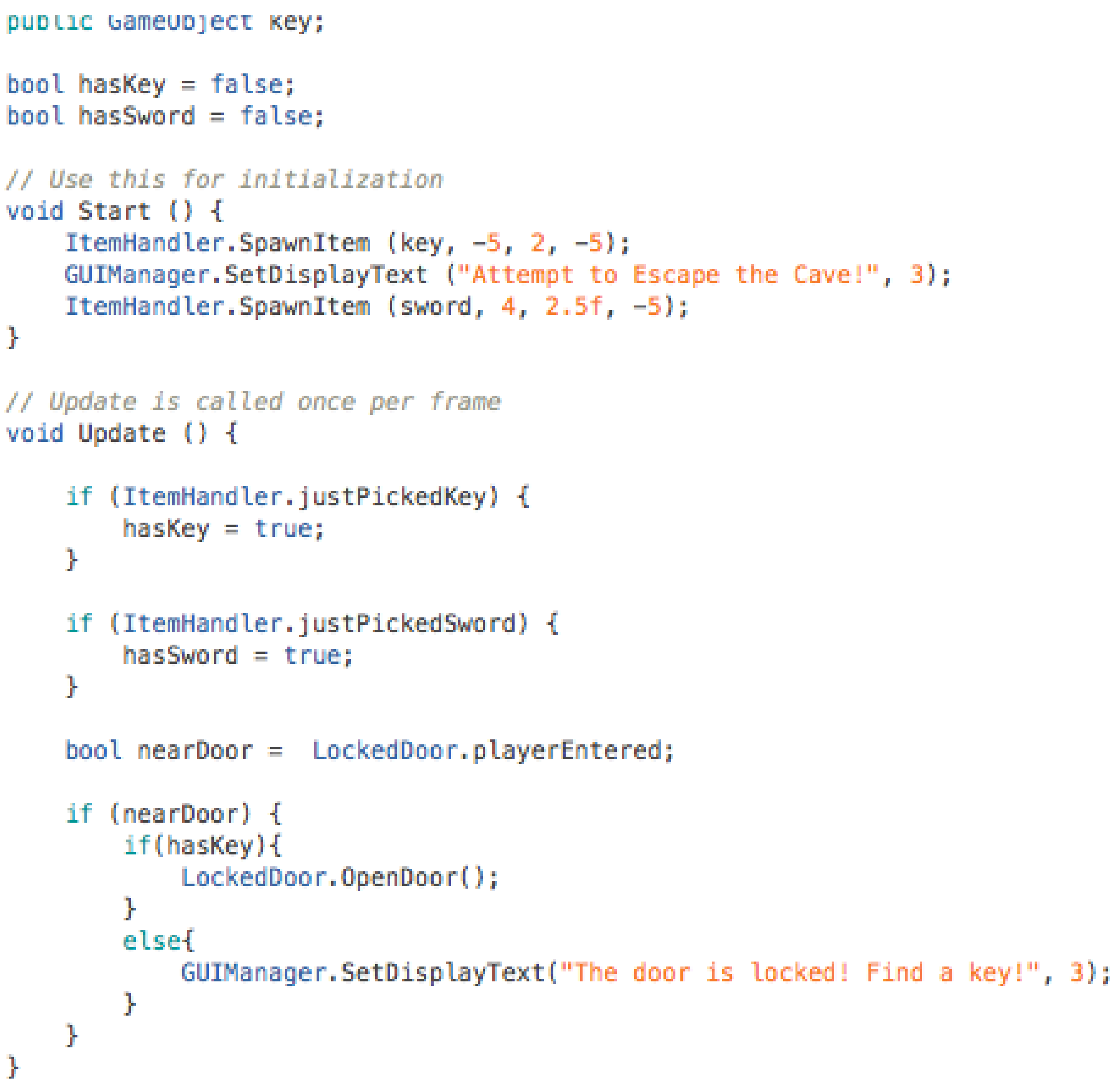
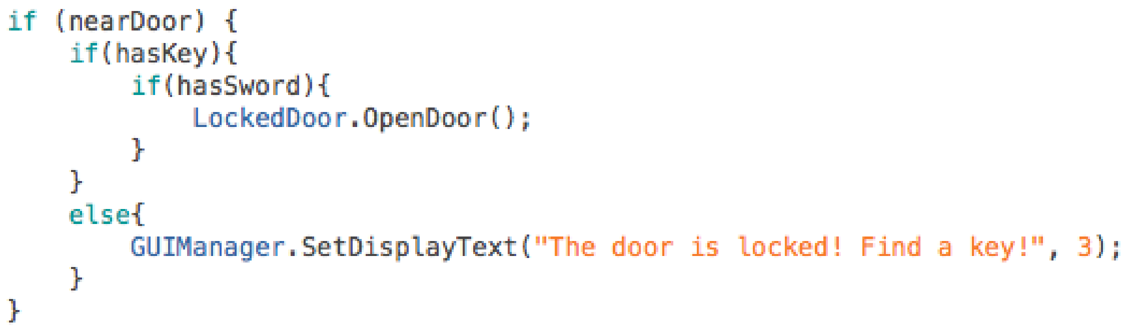


Figure 11: What you should see with the float values from Figure 10 for key and sword spawning. (though you don't have to use these values)



```
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 **logical 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!

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).

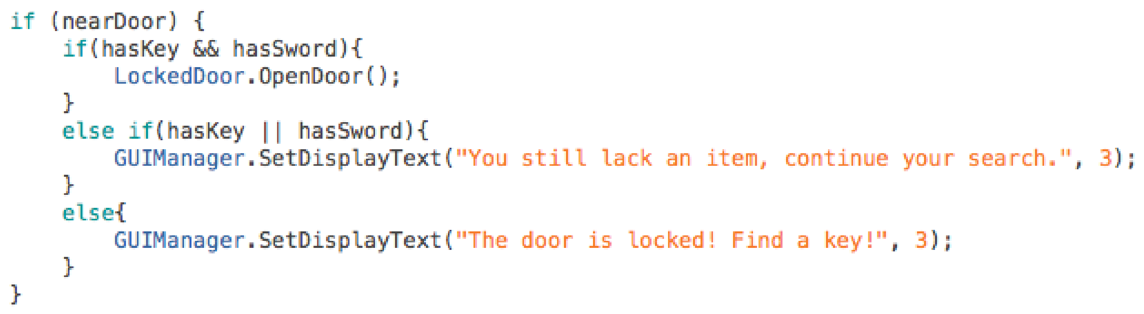
The player now has to have the sword to escape the cave. The only problem there is that the display text from the else statement popped up and said that you needed a key, even though you got a key first! That's no good! What's even worse is that if you play the game again, and go up to the door with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

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!

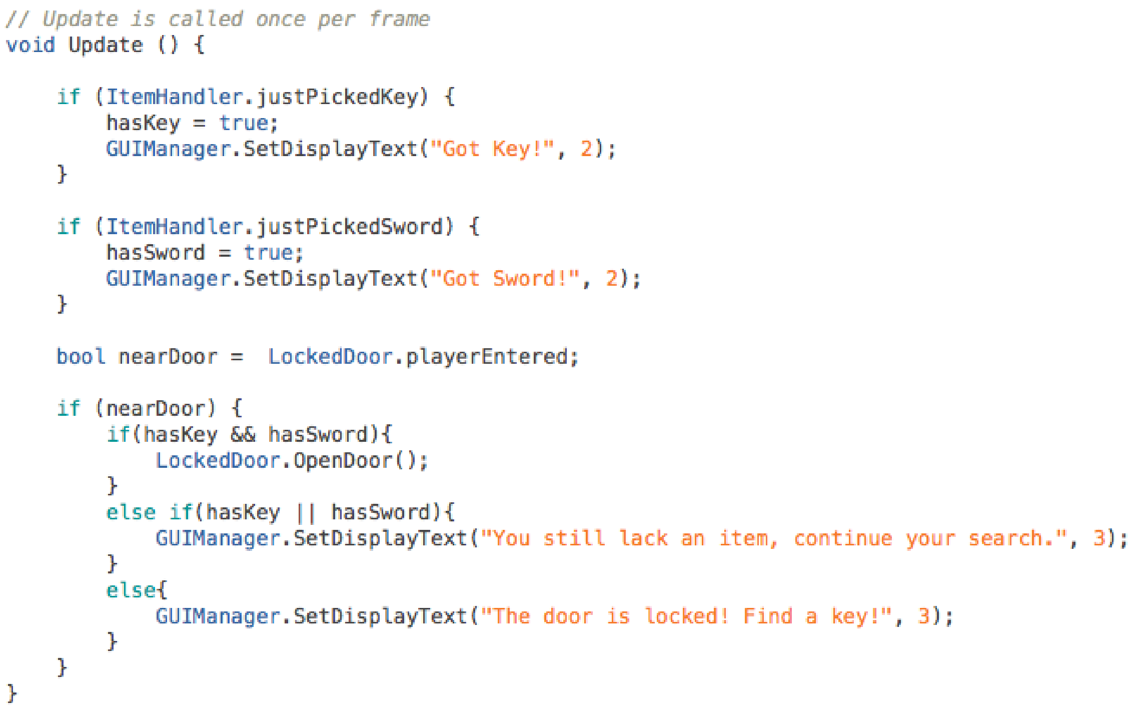
Finish up Update and change your displayed text to whatever you like to fit your own style. Can you make the game more spooky? More silly? Play the game and test out all the different combinations of actions and the effects! Once you're happy with it, have a friend play the game.

3 End of Lesson Two

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

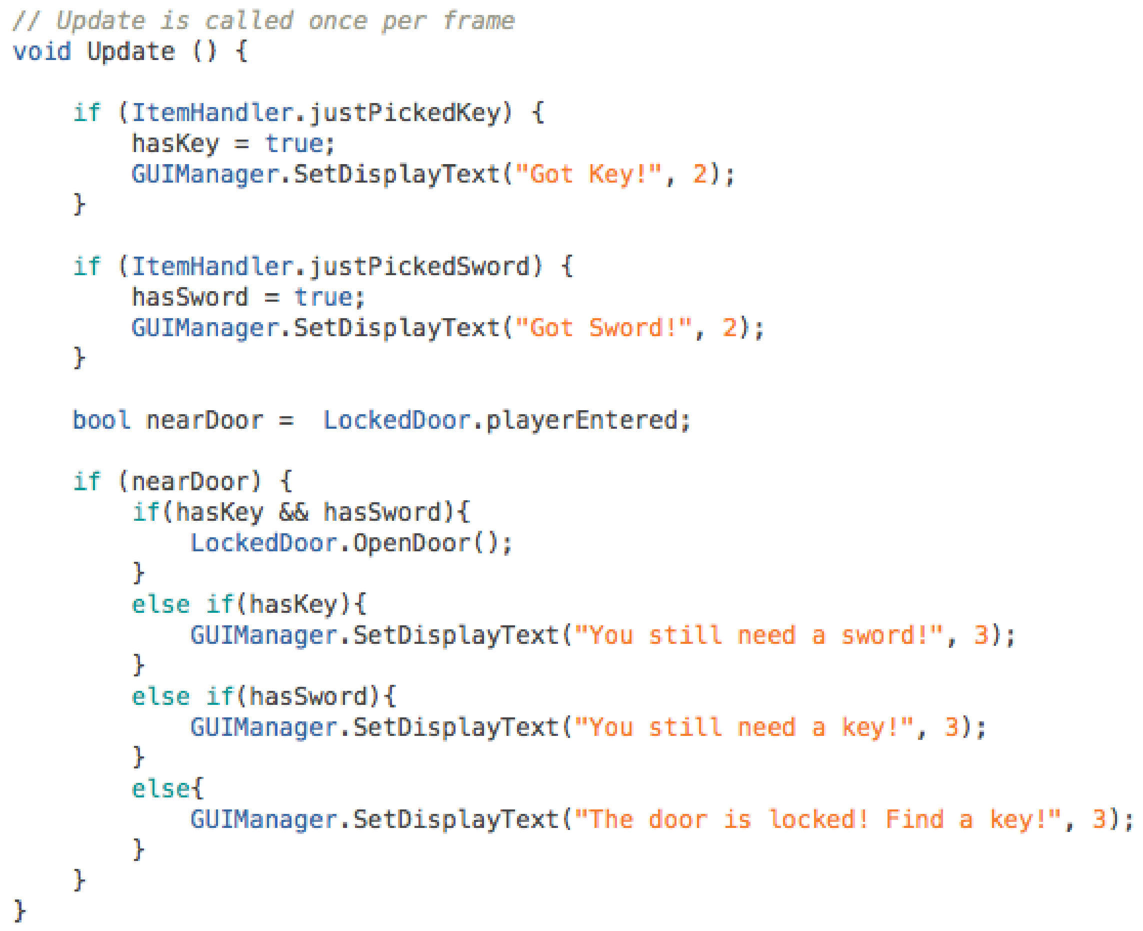
1. Use conditionals to express conditions in code (if, else if, and else)
2. How to call functions with different required value types
3. How to ensure that the player gets correct prompts
4. How to use logical operators (&& and ||)

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){...}