Lesson Seven

Where No One Has Gone Before

1 Introduction to Lesson Seven

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

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

2 Lesson One Extension

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

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

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

3 Lesson Two Extension

LessonTwo.unity is a simple "escape the room" game. At the end of lesson two you ended up with a pretty simple puzzle to find a key or sword that you had hidden in a small cave. There are a number of ways to change this game. For example you could create a new kind of item needed to escape (like in lesson five), make a bigger or more maze-like cave (using what you learned in lesson four), or add a monster or boss in the cave (like in lesson six).

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

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

4 Lesson Three Extension

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

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

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

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

5 Adventure Game Extension

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

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

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

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

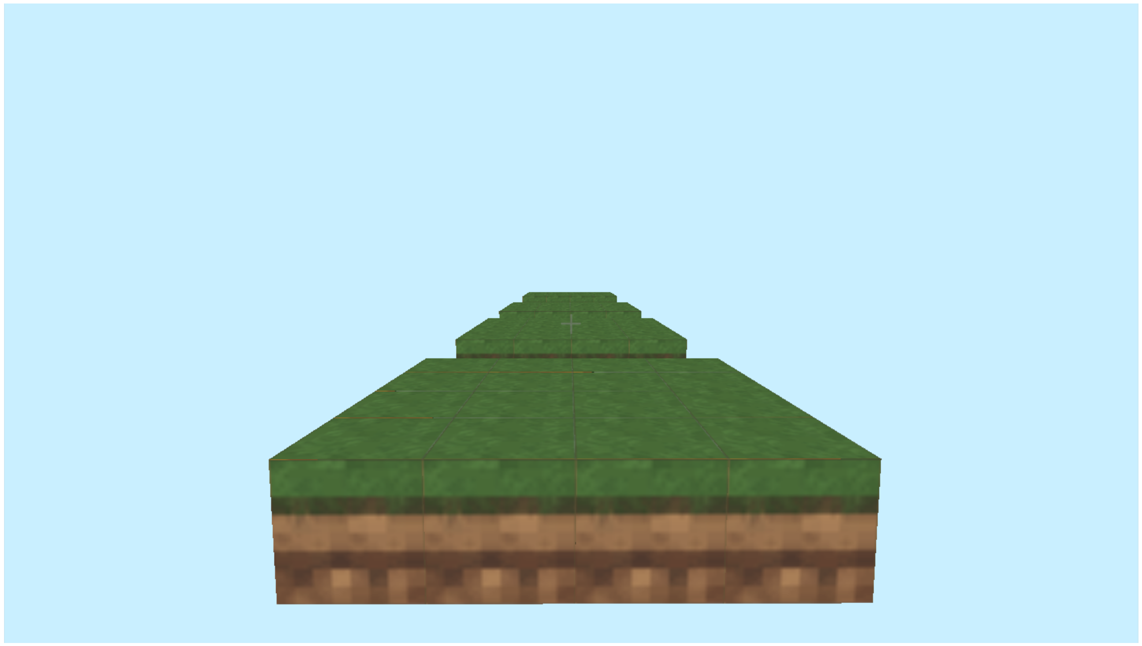

Figure 1: The simple Platformer game

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

6 Platformer

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

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

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

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

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

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

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

7 Murder Mystery

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

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

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

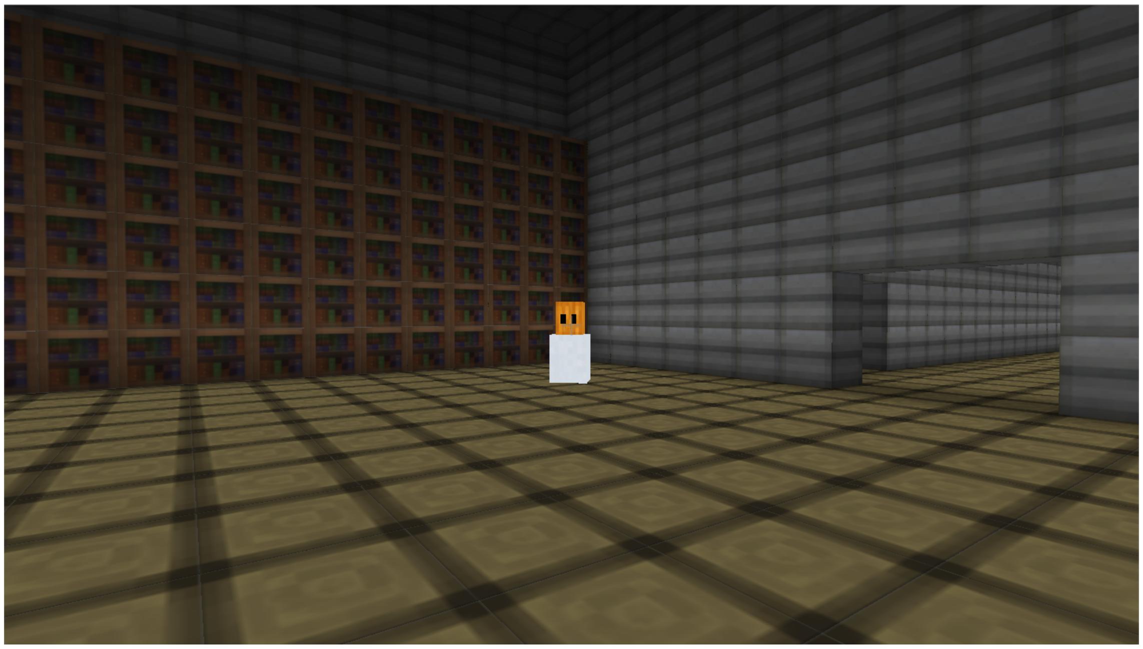

Figure 2: The simple Murder Mystery game

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

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

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

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

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

8 End of Lesson Seven

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

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