UNITY TUTORIAL: GETTING STARTED

April 12, 2015 | misslivirose

INTRODUCTION

So you want to learn how to develop 3D experiences for games and virtual reality, but have no idea where to start? This is the place to do it!

This Unity tutorial is designed to help you take your previous experience writing code and learn how to turn that knowledge into building a 3D application from start to finish. My hope is to bring together some great resources to give you the tools to create richly immersive experiences, starting with a simple maze app.

Disclaimer: This tutorial is NOT meant to be an “end all, be all, best practices guide”, but is instead a collection of my own notes for getting the handle on basic Unity components.

Unity can seem imposing when you first start out, because if you don’t have prior experience working with 3-Dimensional objects or writing games, the IDE will seem kind of alien. That said, it’s easy to start making immediate progress right away after you figure out the basics of creating different game components.

There are a lot of ways to create 3D environments – this is a simple tutorial to get you started with developing in Unity without spending money on fancy rendering software or expensive assets: everything in this tutorial is done without a need to spend money on components, though you can upgrade your own experiences as desired as you go through. Unity provides a rich ecosystem of plugins and assets available as you build more in-depth games, and makes it relatively easy to get up and running without needing to go into the details of 3D modeling and animation.

You can find this tutorial broken into sections and the source code at: http://github.com/misslivirose/basic-unity-vr, though the VR code has been temporarily removed until the tools are officially released for Unity 5.

PREREQUISITES

- Basic knowledge of C# is helpful, but not required – this tutorial expects that you have a general understanding of writing simple scripts

ABOUT UNITY, MONO DEVELOP, AND 3D DEVELOPMENT

Up until recently, consumable content has been almost entirely relegated to a two-dimensional experience with the exception of one field: the gaming industry. The majority of information about Unity development is geared towards game developers, and while this tutorial will walk through the basics of gameplay, it will be interspersed with more general information about non-gaming applications for Unity's toolset.

This tutorial is for you if you have an interest in developing games or virtual reality applications but don't know where to start – we'll walk through the process of setting up the Unity developer environment, create a simple environment with basic texture applications, lighting, and particle effects, and add in a game play component that times our progress through the maze and lets us reset to the beginning. Once you have a basic environment, it becomes easy to add in a VR component (though due to the tools being in early beta, this will be added in later).

Unity is arguably one of the most common platforms that developers use to create 3D & virtual applications. We will be using the 5.0.0 version for this tutorial.

The actual coding that you'll do in your Unity applications will, by default, be done in Mono Develop, a lightweight code editor that comes as part of the package. For simplicity, I'll be doing the tutorial using Mono & Unity, but you can find plenty of resources for how to change the text editor.

Using NotePad++ as the default editor

Using Sublime Text as the default editor

Using Visual Studio as the default editor

There is a growing need for developers to become familiar with Unity and other 3D development tools as the virtual reality industry takes off. Developing 3D environments requires a focus on interactions, physics, and the ability to translate real world experiences into code, especially when focusing on building for VR. Scale and perception become a larger requirement for successful applications, and universes are completely immersive, and content exists in a sphere around the camera perspective rather than only on a single page or window in front of the viewer.

PROJECT SPECIFICATIONS

There are a few things that we'll cover in this tutorial in order to give you the basic skills to create 3D worlds, and to do so, we'll be walking you through how to make an interactive maze. . Specifically in this tutorial, we'll be walking you through:

- Building a 3D environment
- Creating a playable character
- Understanding scripting
- Creating an in-game UI
- Resetting the game
- Particle effects
- Creating a gameplay controller

INSTALLING UNITY

After completing this section, you will have Unity installed on your computer and be ready to start building your maze.

Action Item: Install Unity

The first thing that you'll want to do is download Unity from the Unity3d website. Unity is available for Windows and Mac, though you can also run it in Wine on a Linux distro of your choosing. While we won't get into the process of installing Unity on a Linux machine in detail here, there are a lot of great resources to help you get started with Unity on various Linux distributions. That said, some features are limited or unavailable when running through Wine (such as the asset store) and for the purposes of this tutorial, we'll be assuming that you're running Unity on Windows or Mac OS.

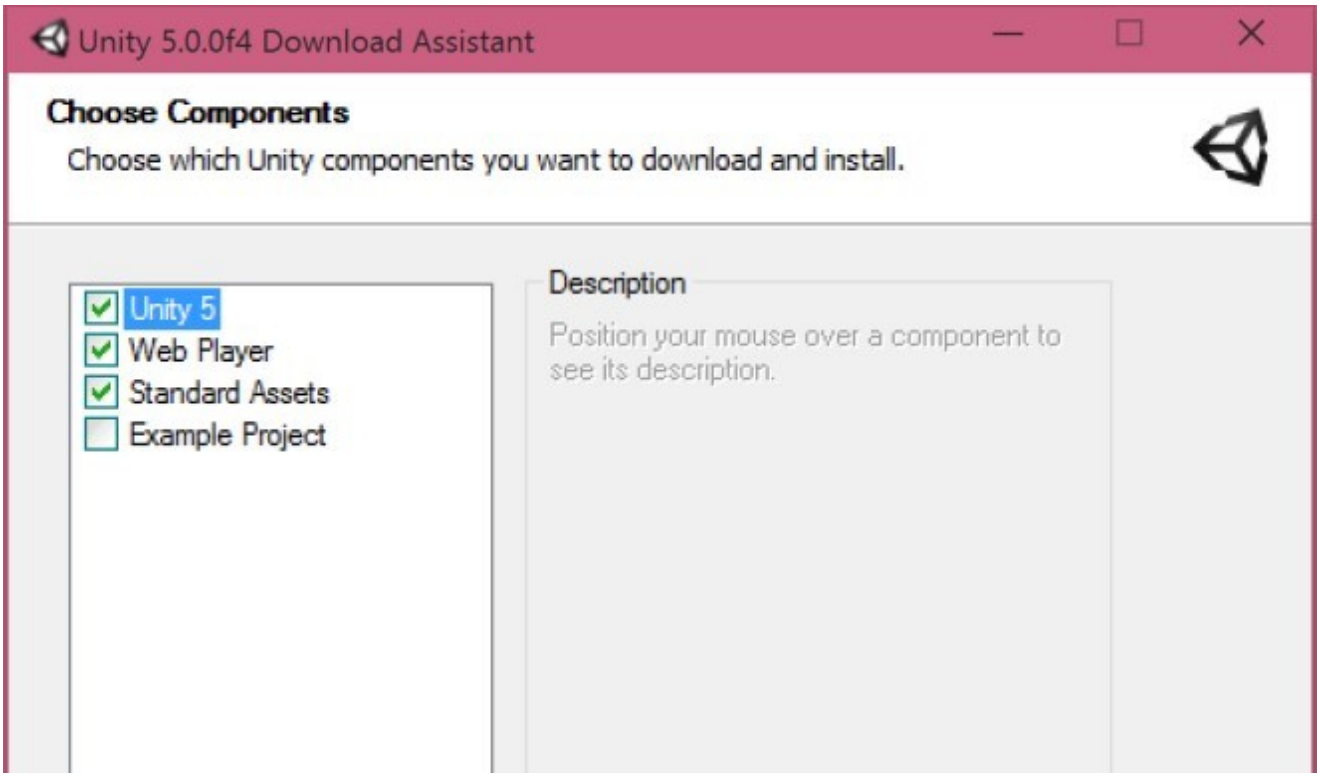
Note that the following links were tested on earlier versions of Unity, so you may see varying degrees of success with Unity 5.

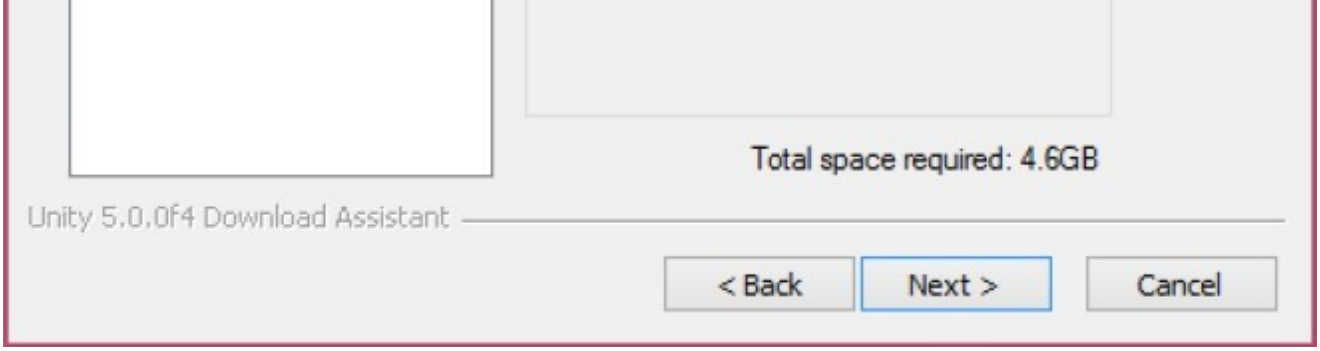
Install Unity on Linux using Wine

Scripting for running Unity on Linux (Tested on openSuse, Fedora, Ubuntu, and Debian)

Video walkthrough for installing Unity3D on Ubuntu

Once you begin installing, you will be prompted to seelect the packages you want to include. Make sure that you include Unity 5 and the Standard Assets. The Web Player is nice to have, and the Example Project can show off more complex features for building games in Unity, but they aren't required.





LICENSING INFORMATION

When you install & run Unity for the first time, you'll be presented with a licensing screen with two options: activate Unity 5 Professional Edition or Personal Edition. Everything that we show you in this tutorial is done using Unity Personal Edition, which is really enough to get you started in the beginning.

Feature Overview for Unity Versions

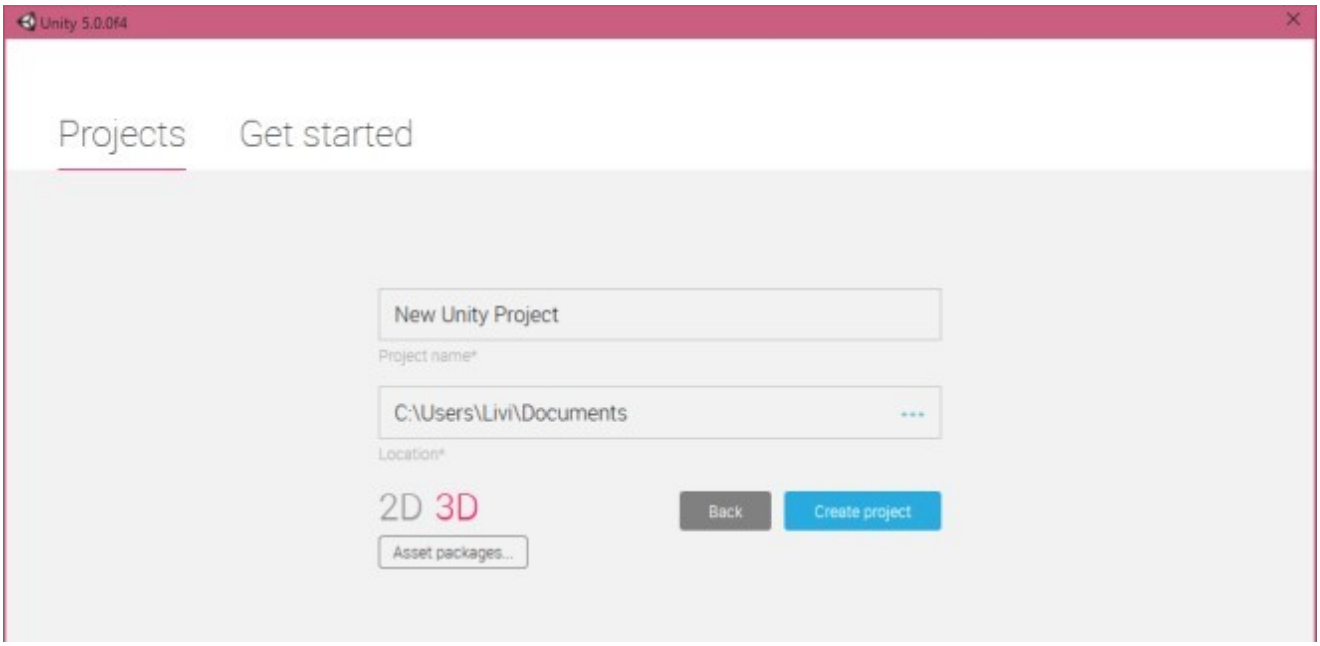
Recreating Unity Pro features in Unity Free – for Unity 4

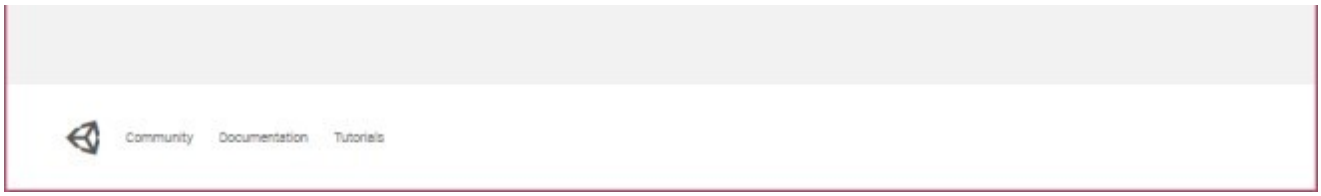
SIGN IN

Once you've activated Unity, you'll be prompted to sign in with your Unity account. If you haven't already created an account, you'll have the ability to create one within the application.

CREATE A NEW UNITY PROJECT

The first thing that we'll want to do once we've gotten Unity installed is create a new project. You will want to create an empty project set up with 3D defaults, but don't import any packages when you're prompted. We generally suggest not importing anything you don't need, since asset files will take up a lot of space, and importing packages in as needed while developing, which we'll go into later on.

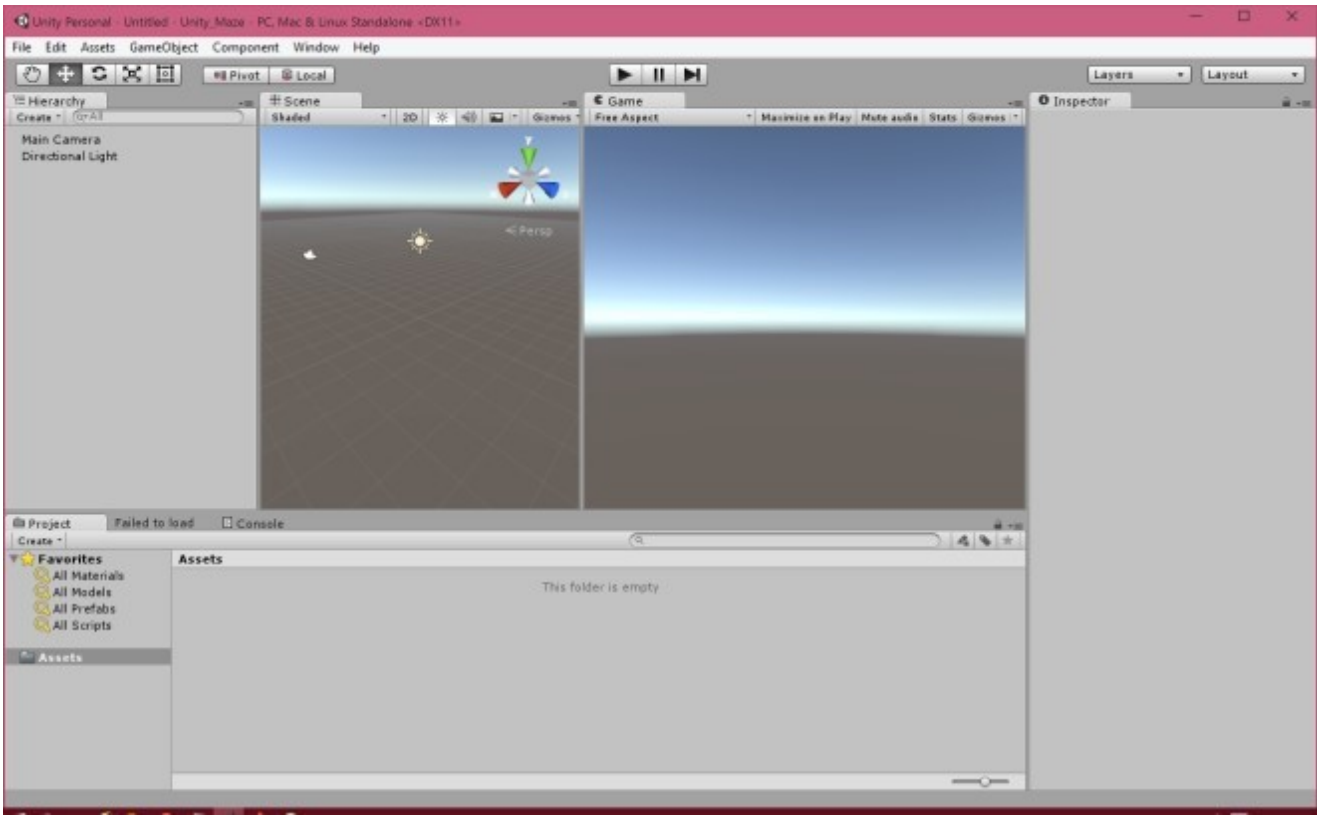




When you create your project, you will have an empty screen in front of you, called a ‘Scene’. This is the major component of a level in your game, and will be where you create all of your environment objects. Go ahead and save your scene – it’s helpful to do this often because Unity doesn’t currently have a great option for cleaning and refreshing projects as you make changes if it isn’t done automatically.

ABOUT THE UNITY EDITOR

On first glance, if you haven’t played around in Unity before, it can be a little overwhelming. Luckily, Unity Technologies has a series of informational videos that walk through the different aspects of the editor that can help familiarize you with the different components of the UI.



Action Item: Become comfortable with the Unity Editor
Interface Overview

The Scene View

Scene View Navigation – We’ll go into a little more detail about this when we create our first scene, but if you want a primer before getting started, or a reference for the behaviors and keyboard shortcuts, this is a helpful page to bookmark.

The Game View

UNITY AND SOURCE CONTROL

If this is your first time playing around with Unity, you may be realizing something: Unity projects can get big, and they can get big *fast*. At some point, if you continue building with Unity, you'll either want to implement some sort of source control or work on a project that involves collaboration, so it's important to get a feel for how this works with 3D Unity projects.

Action Item: Become familiar with setting up a Unity-friendly git repository with TheNappingKat's tutorial

BUILDING A 3D ENVIRONMENT

Okay, so now we've installed Unity and gotten our empty project made. By now, you should know a little about the editor, and we're ready to start building our maze.

First things first: when building 3D games (or any other type of interactive experience) in Unity, you need to know the fundamentals about the GameObject. Just about every aspect of your game, from the interface to the characters, player, environment, and affects, will be composed of various GameObjects with different sets of components. For the first part of the environment we're making, we'll be using a built-in Unity UI GameObject element called the Plane GameObject.

Positioning GameObjects in Unity

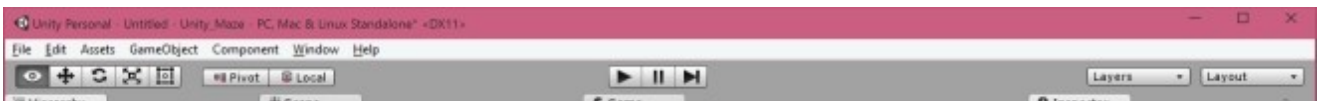
USING A PLANE TO CREATE A FOUNDATION

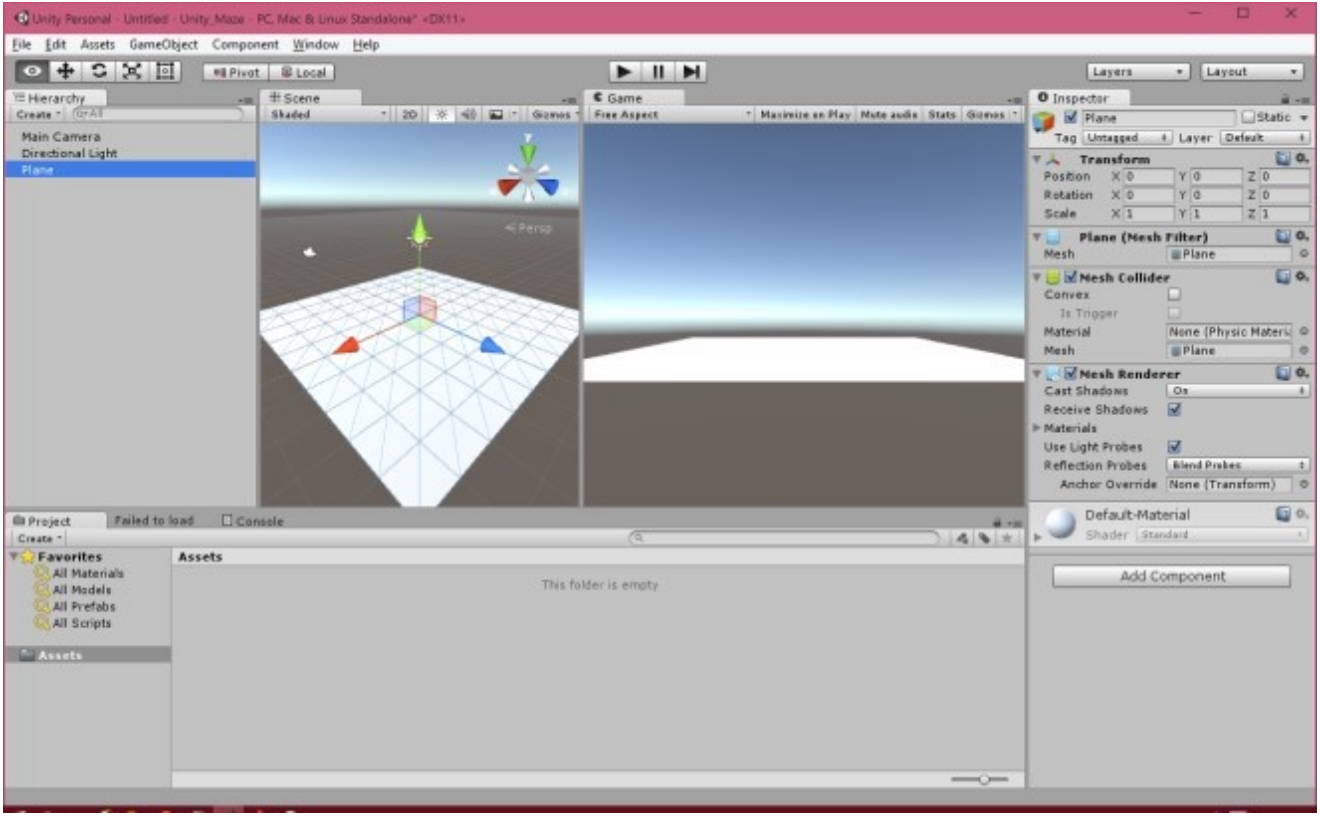
The first step in making an environment feel realistic to a player is to give it a foundation. If you try running your empty scene right now, all you'll see is a blank grey environment and a blue sky, which is to be expected – we haven't added any character controller or any objects to see. Later on, though, when we add a controller for the player, you'll notice that any time you step off of a plane, your character will fall until you kill the process and restart the level. By default, Unity 5 includes a Skybox enabled and has a directional light for your scene, so you won't have to worry about those off the bat – but we'll go into how to customize those later on.

So, let's get started!

The first thing we're going to do is create a GameObject using one of the prefabricated models (prefabs) that Unity has built in. As you build your game out with Unity, you'll become very familiar with the GameObject menu and the various components that make up objects in your environment.

Action Item: Create the foundation for your maze by creating a series of plane objects
We'll start by going to GameObject -> 3D Object -> Plane





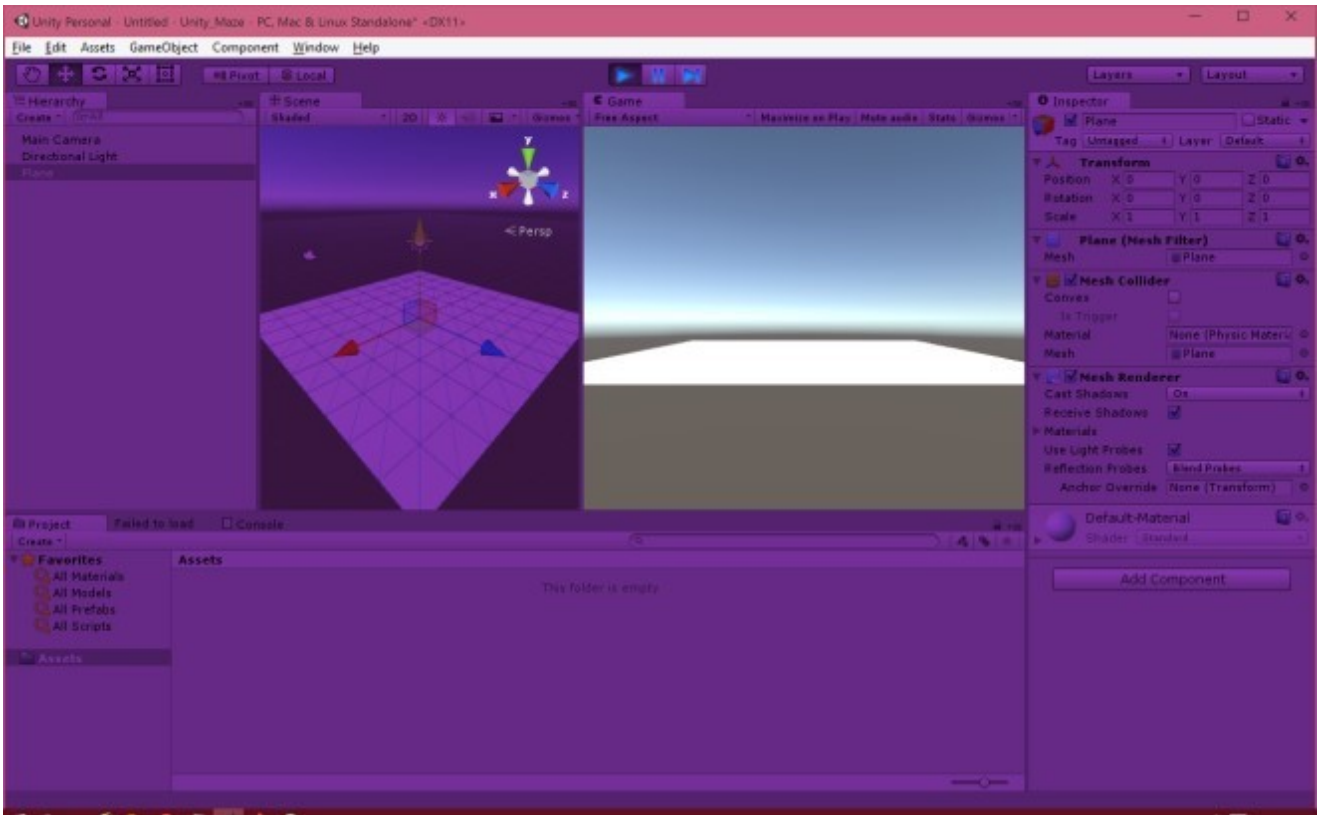
When you generate a plane GameObject, you should see a shape appear in the center of your view field. This will be the starting point for our maze, but right now, it's just a square.

The first hurdle with developing in 3D is getting used to specifying object sizes and locations relative to their parent container. In this case, the parent for the plane we've just created is our (theoretically) infinite grid, so the plane size will serve as our relative point of reference for sizing the rest of our maze elements.

Running the game now results in a static image where we can see the plane we've just created at the bottom of the field of vision. As we build out our maze, we'll want to change our view to place our new GameObjects, so let's cover a few basics with views:

- The fastest way to get precise camera control for the perspective you want is by holding down the alt/option key while you grab the view field and rotate (Mac). This switches the control mode to pan, but allows you to manipulate the view over all three axes – helpful when checking collision points.
- Dragging and dropping with the view mode set to 'Pan' allows you to view different areas of your map without changing the view angle. Pressing & holding the Control key will let you zoom in and out from a static point.
- To manually drag objects around on the grid, change the view mode to (!!arrow) keys. This can be helpful when placing objects in the general area you want them before using the coordinate system to polish up their exact location. You can specify which axis to move the object on by clicking the object and dragging along the arrow on the axis you want to change.

Important note: Do NOT make changes while you are in Game mode as they will not persist when you stop running your maze. It is strongly recommended that you change the color of your "In Game" UI by going to Edit -> Preferences -> Colors and changing the Playmode tint to something noticeably recognizable.

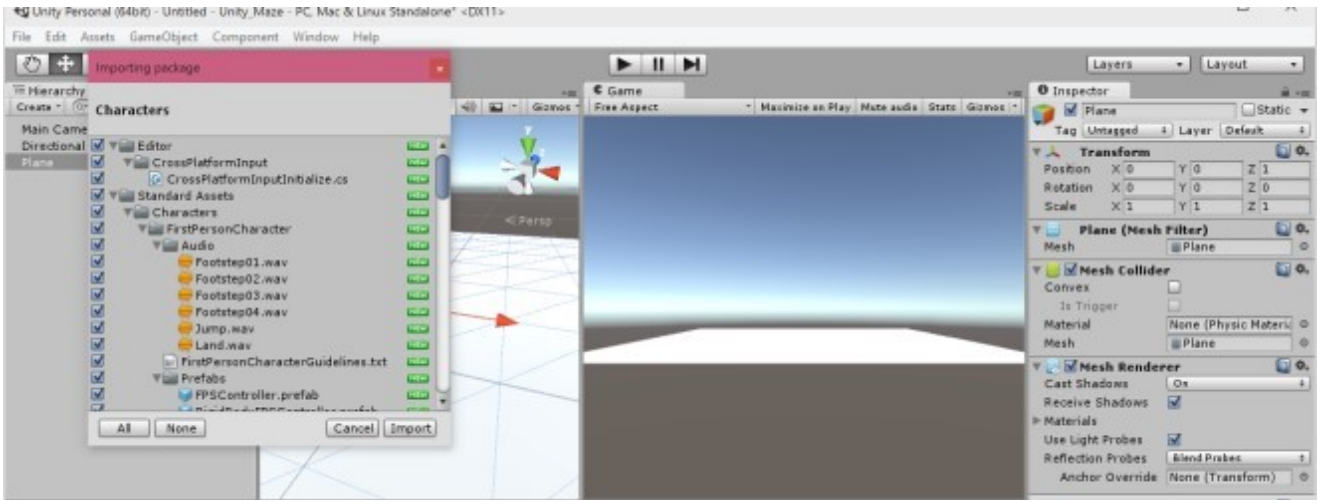


Action Item: Understand the relationship between GameObjects and Components
Generally speaking, the relationship between a GameObject and a Component is fairly straightforward: a **GameObject**is the object's representation in the world and it's "physical" aspects, such as location and size, and it contains a set of**components** that define its characteristics and behaviors, such as it's motion controls or how it lights the area around it.

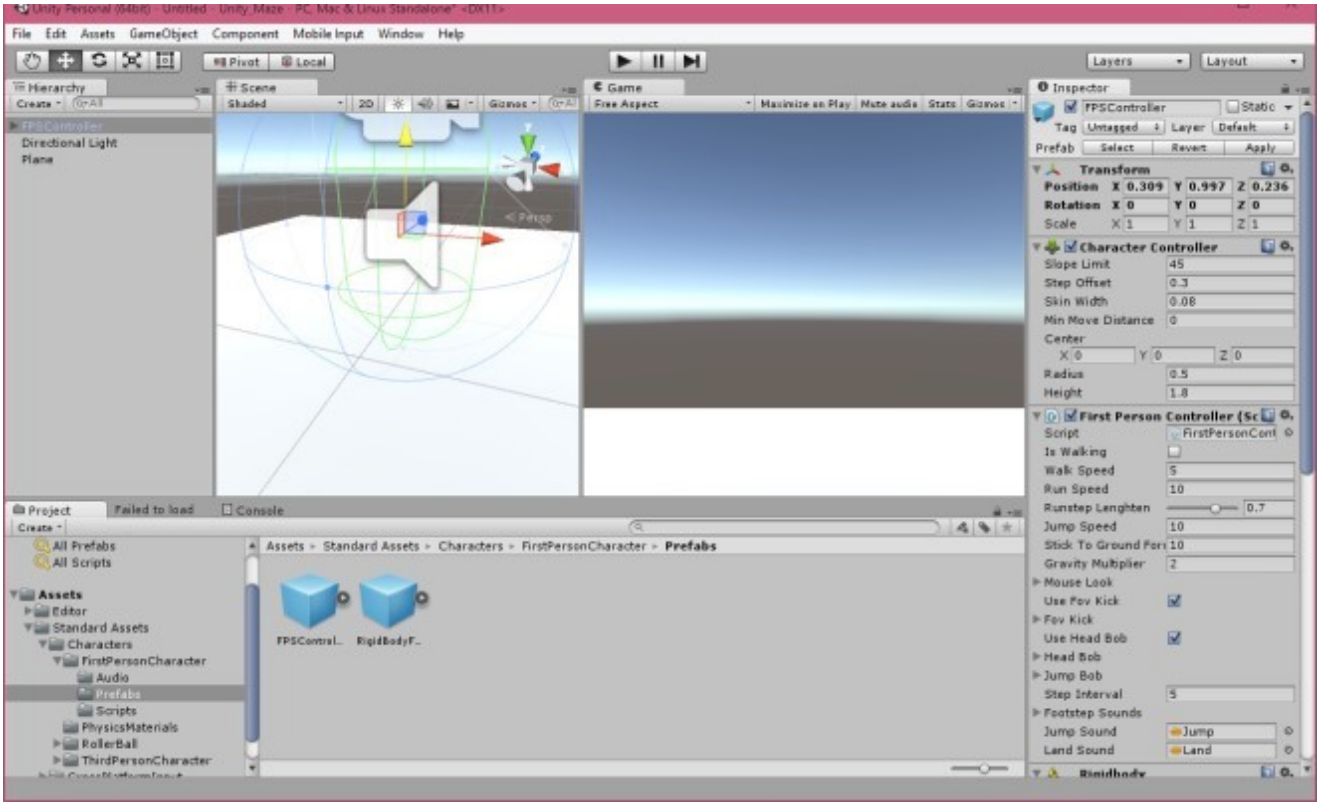
CREATING A FIRST PERSON CONTROL

Action Item: Create your first character controller using the built-in Unity FPS character component
We won't get very far with playing our game without a controller, so let's pause now to make a controller so that we can manipulate the camera and move around the board.

Because we created our project without any assets, the first thing that we'll want to do is import the standard Character Controller asset package.



1. Go to Assets -> Import Package -> Characters to import the character package into your project. You can uncheck the Third Person Prefab to save space, but it's not strictly necessary, especially if you feel like playing around with the various character controllers.
2. In the Asset directory at the bottom of the screen, expand the Standard Assets folder, choose the Characters folder, open the First Person Character directory, and double-click 'Prefabs'. You should see two prefabricated controllers, an FPSController and a RigidBodyFPSController
3. Grab the FPSController and drag it into your scene over your plane. Make sure that the character is fully above the plane, or the collider won't take and the character will fall through the floor.
4. Delete the Main Camera. The FPSController comes with a camera component, so we no longer need the one that Unity created by default.



When you run the game now, you will have a first person view of the plane we've created. Notice that many of the typical controls for a first-person controller have been added for us automatically. You can move with the arrow or WASD keys, jump with the space bar, and the view adjusts automatically for us when we move around the plane.

If you jump off the plane at this point, gravity kicks in and you'll fall through space until you restart the scene.

BUILDING OUT THE MAZE FLOOR

This next part of the tutorial allows you to get creative. There are different ways that you can choose to build a maze in Unity, but we'll be going through one that lets you add on to the maze easily by having the floor follow our maze pattern rather than relying on the walls alone for the plan. Whether you choose to use one large base or break it out into the sections as shown is up to you, but there are several benefits to creating the floor layout in pieces rather than using one large foundation:

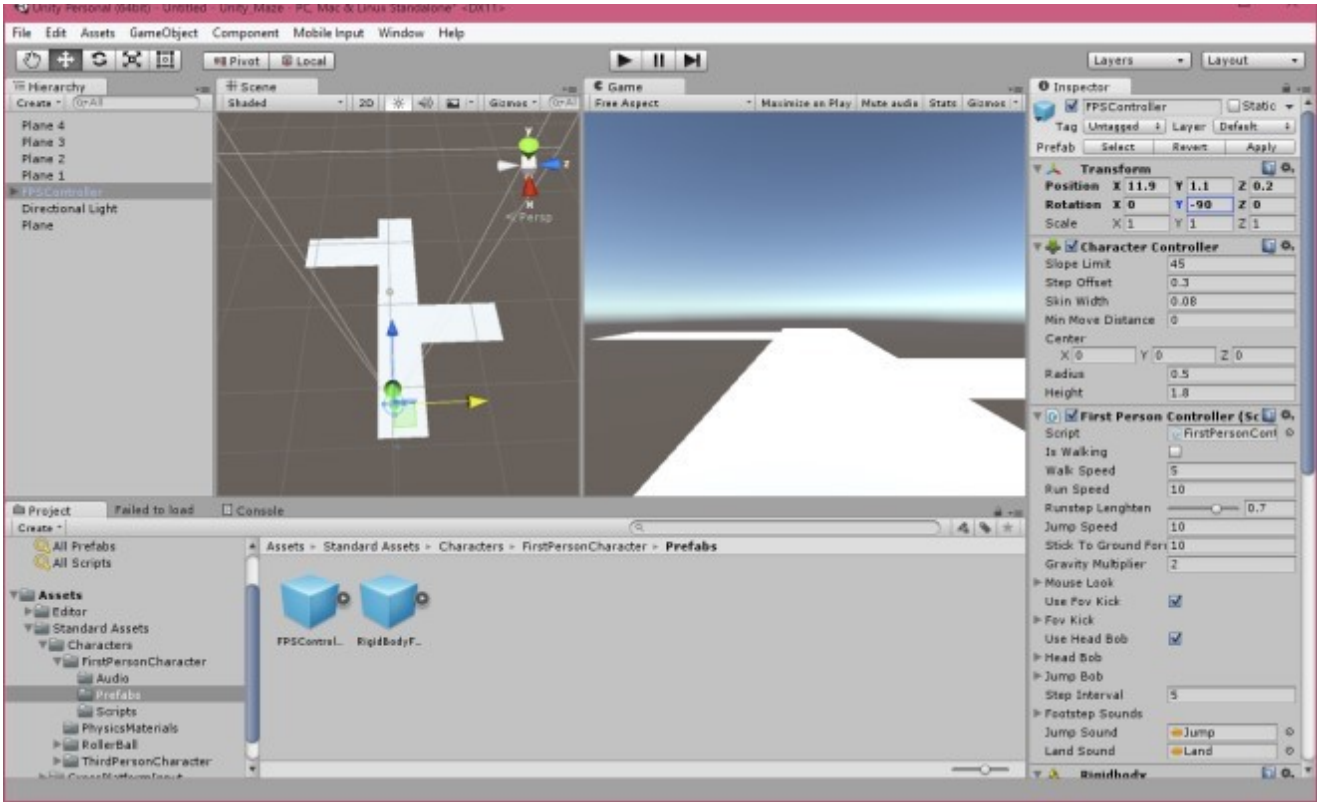
- Placing walls is easier since you have the coordinates of the floor already
- You can view the entire maze from any angle while you're building
- Allows for a better holistic view of the layout than you can with a single base
- Gameplay is more controlled since the user will not be able to run around the maze outside of the paths you've created

Action Item: Build a single pathway for the maze

1. Select the plane that we've created and modify the size to scale for your maze. Notice the perspective tool at the top right of our viewer, which shows which direction each of the axes will change. In our example, we'll be changing our plane to have an X scaling factor of 5 and a Z scaling factor of .5
2. Create a second plane to make a crossroad. You can do this by creating another plane from the GameObject menu, but I prefer to just copy and paste the existing one since it keeps the Y and Z sizing consistent across the pathways.
3. Rotate the second plane by selecting it and changing the Y rotation to 90. This will make a path that is perpendicular to the original one. Move the new plane where you want it on the environment. For the example, we'll be changing the X location from 0 to 25.

How do we figure out the location? *The location is determined by figuring out the size of the original plane and multiplying that by the desired location. Our X scaling factor for the first plane was 5, with an original size of 10 units, which means that the length of the entire pathway is 50 units, centered on the X axis at the origin. So we need to put the new pathway at the end by setting X to +/- (5 x Scaling Factor).*

We'll pause here to introduce a new GameObject to the maze we're building by adding in walls. You can layout the entire maze floor first if you'd like, but we recommend building the floor and walls in sections to make it easier to layout the objects as you go.



Action Item: Build walls around your existing paths
A maze is no fun when you can see to the exit, so our next step is to add walls. Do do this, we'll need to add a new type of object. Unity provides a **cube** GameObject, which we'll be using to make our walls.

Create a cube using the following properties (if you're following the same pattern we used above) by going to GameObject -> 3D Object -> Cube:

- Position (x:0, y = 2, z = 2.75)
- Rotation: (x: 0, y:0, z:0)
- Scale: (x: 50, y: 4, z = 0.5)

This will give you one wall that runs alongside the original floor we laid out, stopping a little short of the crossing path at the end. If you've built your own design and used different sizes, you'll need to modify the position and scale for the walls.

How do we figure out wall placement? *There are a few things to consider when placing your walls in a 3D environment. You want your wall to be consistently placed – in this example, we are placing all of our walls entirely outside of the floor, right at the edge, but you could also do your wall placed just inside of the floor or splitting the middle – it's up to you and it makes the most sense to decide this based on the scaling you're using so you don't end up with awkward decimals.*

POSITIONING

- x: this number will determine the location along the X axis. This should be the center of where your floor plane is.
- y: this number should be the positive value of 1/2 the height of your wall – the positioning is from the

origin, in our case, 0, and this will prevent our walls from sticking through the bottom of the maze.

- z: this number should be 1/2 of the width of your floor plane + 1/2 of the width of the wall. In our case, the width of the plane is 1 (5 units) and the wall is half a unit (note that this is relative to the parent, so in our case, this is 0.5 units when compared to the plane) for a z position of 2.5 + .25 = 2.75.

ROTATION

For this tutorial, you will only need to rotate along the Y axis to create different angles for your walls and planes.

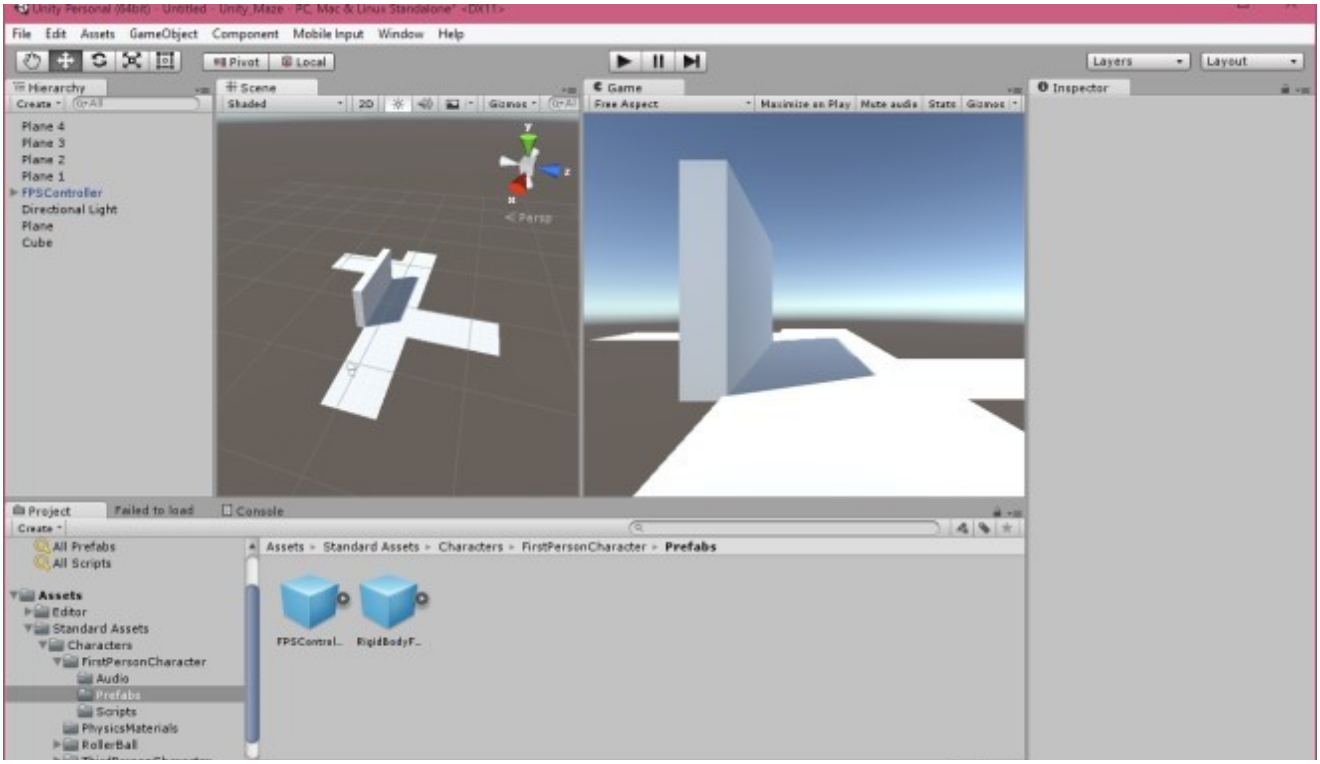
SCALE

Scaling with Unity can be tricky before you're used to it – scaling is done relevant to the parent of an object, which can vary for different objects. In this example, the cube has a different scaling than the plane, specifically that the cube is 1/10 the scale of the plane by default. The best example of this is the comparison of the wall to the ground:

Let's say we are measuring in meters. We specified a scaling factor on the Z axis (width) of the plane at .5. Here, one "unit" for the plane is actually 10 "meters", so the floor is 5m across.

We made a cube with a default of 1 unit^3 – the scaling here is 1:1 with "meters", so the cube is 1m^3. When we change the z-axis scaling to .5, our cube shrinks to .5m instead. You can see how this gets confusing with relating differently scaled objects, but thinking about it in terms of absolute measurements can help if you get tripped up.

For our wall, we have a scale of x:50 (since the base is 5*10 units), y:4 (you can choose the height here of your walls to vary the effect) and z:0.5 since there isn't a need for wall thickness to increase and keeping it small allows more flexibility with building paths.



Now that you know the basics about building the skeleton for your maze, you can finish building your own maze design or copy the one that we've provided in the source code.

Action Item: Finish the general maze layout using the steps above to create your maze

ADDITIONAL RESOURCES FOR ENVIRONMENT BUILDING:

Generally speaking, if you are planning on building a more complex environment and game experience, you'll need to incorporate a more advanced element into your scene – the Terrain GameObject. We won't go into too much detail here, but you can find additional resources on building complex environments with the Terrain Editor in Unity below.

Building a more complex environment with the Terrain Editor

Next up, we'll explore the Asset Store and apply textures to our maze.

INTRODUCTION TO THE UNITY ASSET STORE

Right now, we have the bare basics of our maze and a general first person controller, so it's time to make our environment feel a little more welcoming. Unless you already have a passion for graphic design, 3D modeling, or photography, it's pretty likely you'll need to use the **Unity Asset Store** to get textures and models for your game objects.

Action Item: Explore the Unity Asset Store
You can access the Unity Asset store online and through the program with the shortcut CMD/CTRL + 9. The Asset store requires Unity 3.3 or later, so if you're using an older version or running Unity in Wine on Linux, you won't be able to access the store directly.

Workaround for accessing the Asset Store on Linux

The Unity Asset store is a centralized location to find resources for building games and environments, and allows content creators to purchase assets, ranging from full scenes to character sprites and ground textures, so that you don't have to make everything yourself.

Want to skip the asset store for the terrain? Unity has a few built-in textures in the Terrain Asset package that can be included in your project by going to Assets -> Import Package -> Environment.

FINDING AND APPLYING TEXTURES

Right now, we have a completed maze that is just about impossible to navigate due to all of the monotone colors, so we're going to make the environment more welcoming with textures. This will also add an element of realism based on the textures you choose.s

Launch the Unity Asset store. If this is your first time using the store, you'll need to create an account.

Action Item: Create an account on the Unity Asset store

We're going to be downloading a texture pack from the Unity Asset store and importing it into our project. You can pick any texture you'd like – for this tutorial, we'll be using a couple of free store downloads, but you can also use the built-ins.

MODIFYING THE FLOOR TEXTURE

Action Item: Download and import a texture package from the Unity Asset store to apply to the ground of the maze.

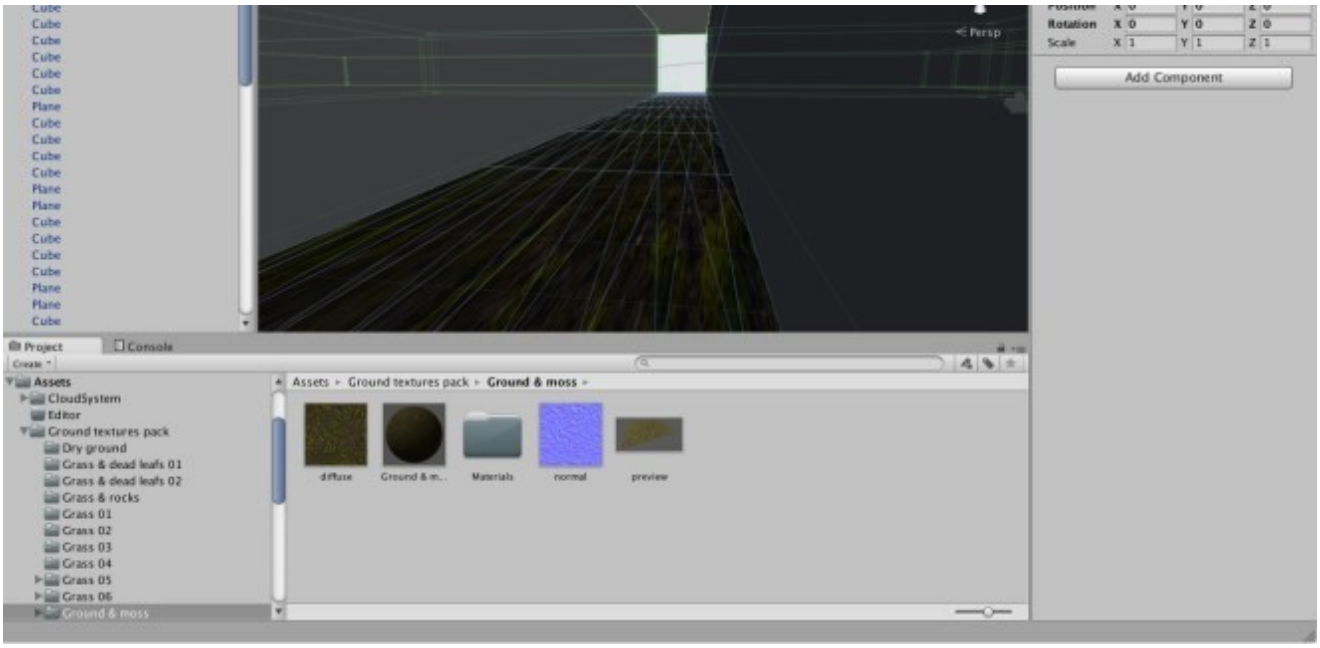
1. From the asset store home page, select "Textures & Materials".
2. Under the list of available categories, select the type of ground you want to use.*Note: The "Ground", "Nature", and "Organic" categories contain natural textures, such as grass, rock, or dirt. Man-made material textures can be found under "Bricks", "Concrete", "Metal", "Pavement", "Roads", "Tiles", and "Wood".*
3. Select the asset package to download for your maze floor. This tutorial uses the Ground Textures Pack'Ground Textures Pack', which is currently available for free from Nobiax/Yughues.
4. Import the desired textures into your application when prompted after the download completes. If there is a specific texture you want, you can select just that, but for now, you can just import them all to try out the different textures in your maze.

Once the import completes, you should see a new folder in the Asset window in Unity for the new texture pack. If you used the Ground Textures Pack, the folder name will just be the name of the imported package. Expand the folder to show the different textures.

There are a few different ways that you can add textures to the various flooring components, depending on your personal preference, but the easiest way is to just grab the texture from the asset folder and drag it onto the map where you want to set the texture. In this case, we're using the 'Ground & Moss' texture.

1. Expand the asset folder that you want to use and find the folder for your chosen texture.
2. Expand the texture folder for your chosen texture.
3. Drag the texture – in this case, the file named 'diffuse' – onto your map where you want to change the ground visuals. A green circle with a plus sign will appear when you can drop the texture on your GameObjects.
4. Repeat this for any components you want to give textures to.

Once you've added your desired textures to the ground of your maze, try running the game again – you'll notice that it's much easier to distinguish the pathways from the sky with the contrast from the new textured ground. We want to do the same thing with our walls to polish off the maze foundation.



MODIFYING THE WALL TEXTURE

Action Item: Download, import, and apply a texture package from the Unity Asset store to apply to the walls of the maze.

1. From the asset store home page, we'll be going to "Textures & Materials" again.
2. Under the list of available categories, select the type of wall you want to use.
3. Select the asset package to download for your maze walls. This tutorial uses the Nature Textures Pack, also from Nobiax/Yughues.
4. Import your desired textures into your application the same way you did with the floor textures.

When you run your game now, you'll see that the walls and ground now have the new look applied to them. Depending on the scale of your maze, you may need to adjust the tiling of the texture.

For more information about textures in Unity, we recommend checking out the following resources:

How do I add Textures?

Textures – Official Unity Tutorials

Normal Maps

Beginner Texture Tutorial

NORMAL MAPS AND TILING

You may need to manually add in normal maps to your textures in order for them to get the 3D feel to them in your environment. Normal maps are specific to materials and serve as a layer that specifies how lighting renders on your object to give the appearance of shadows and depth. To specify normal maps, you will need

to change the type of your texture from “Diffuse” to one that supports an additional normal map. Then, follow the steps above for adding textures to include a normal mapping image to your material.

(OPTIONAL) LIGHTING AND SHADOWS

Unity 5 includes a directional light when you create a new scene, but you can modify the lighting style if you’d like to try something different. The types of lighting that Unity supports are:

- Directional Light
- Point Light
- Spotlight
- Area Light
- Reflection Probe
- Light Probe Group

You can play around with creating lighting effects the same way you’d create other GameObjects:

1. Right click in the hierarchy, select “Create” to open the drop down menu, or go to GameObject -> Light to choose a lighting type.
2. Position your light to your liking to cast shadows in your maze

You can change the various properties of your light(s) in the inspector. Adjusting the color & intensity will change the way the light appears on your maze and increase the contrast between light and shadows. There are several options for additional lighting effects that you can add to your lights for different appearances.

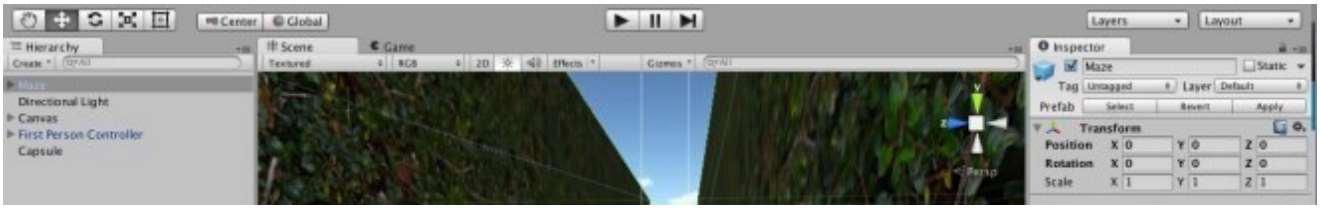
Full list of properties for lights and descriptions.

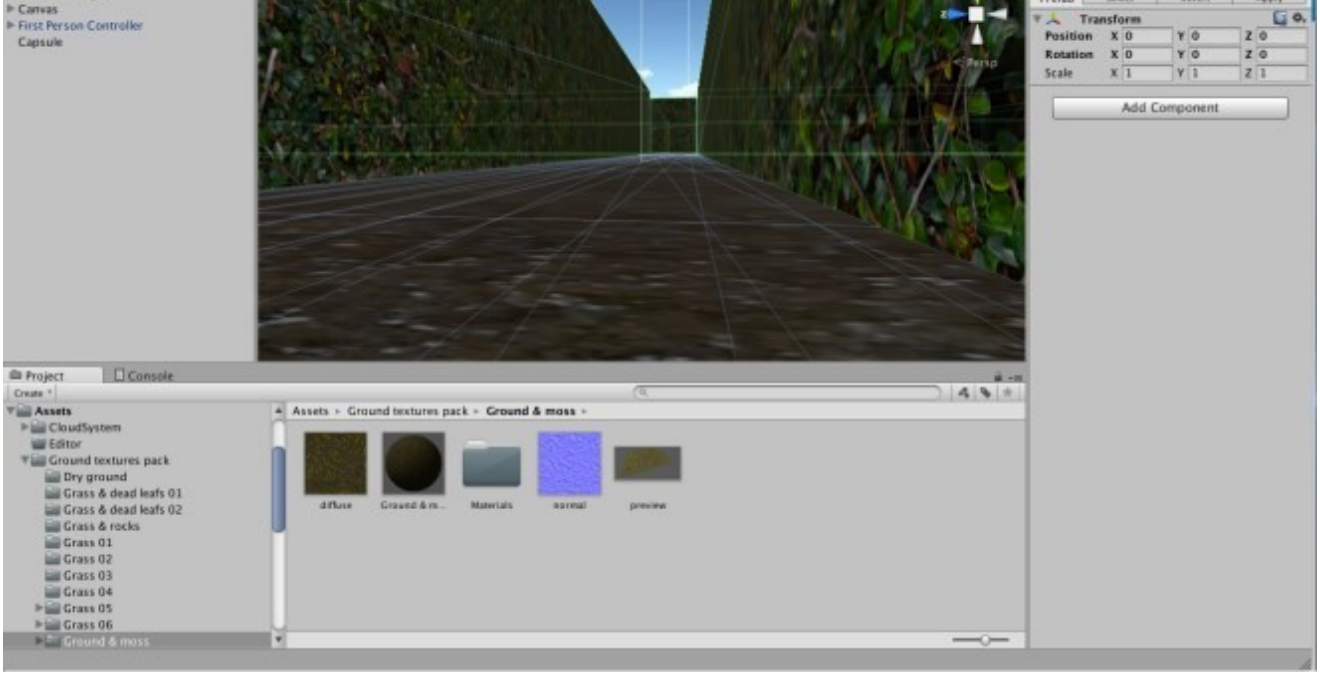
(OPTIONAL) ADDING A CUSTOM SKYBOX

With Unity 5, the scene comes with a skybox by default, but you can change this depending on how you want your environment to feel. To change the skybox:

1. Import your desired skybox from the Asset Store
2. Select your FirstPersonCharacter controller and click “Add Component”
3. Under “Rendering” choose “Skybox”
4. Drag your new skybox into the Custom Skybox field

Run your maze – you can now run through it and you should see the sky surrounding the entire environment. You can download and try out additional skyboxes from the Unity Asset store.





Now that we've got a nice environment set up, it's time to add a few components that make our game more enjoyable. We're going to include a basic GUI control so that you can see your time throughout the maze, a gameplay mechanism for winning, and a few other components to make the experience more challenging.

ADDING A GUI

The first thing that we're going to do is include a basic user interface that displays on our screen.

Action Item: Create the container for a timer

1. In your project hierarchy, click Create -> UI -> Canvas
2. Under your Canvas, right-click and add a UI -> Panel item under the Canvas item
3. In the Inspector for your Panel, change the scale to x:.25, y:.25, z:.25 – we don't want this to cover the entire screen.
4. Change the color of the panel to be more visible, also in the Inspector tab
5. If needed, zoom out to see the full UI display. Adjust the positioning of your Panel to your desired location – in our example, we're putting this in the top right corner.
6. Making sure that the Panel is selected in the scene Hierarchy, right click one more time and create a UI -> Text item in the Panel

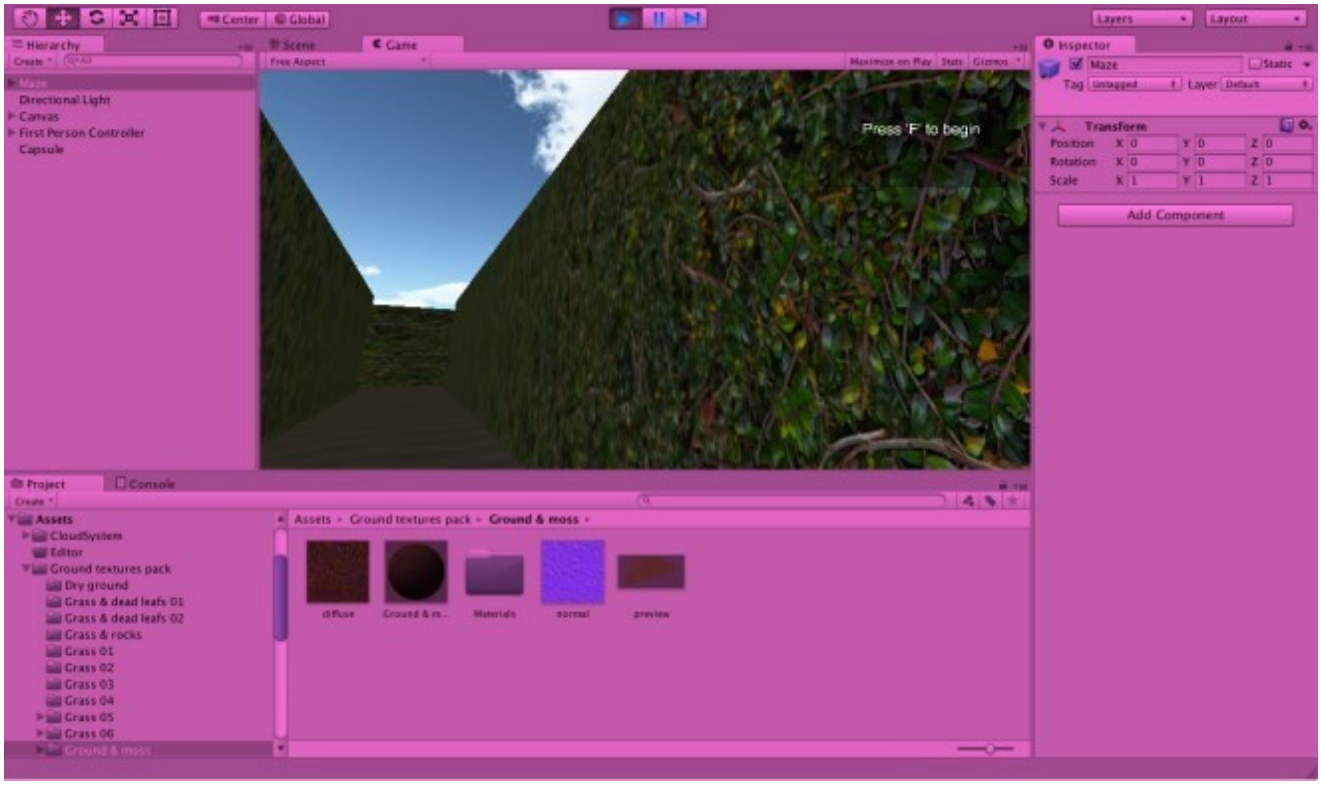
If you run your game now, you should see your panel floating in your camera view screen. We'll create a script to show and hide this later, but for now, let's finish up with the timer. We'll want to adjust our text block first so it's visible when we add in the timing component.

Action Item: Add placeholder string for a timer

1. Select your text item and find the properties for the text block. For now, we can put a placeholder text in, so under the Text (Script) box, type in a temporary string.
2. Under the 'Paragraph' header, check the 'Best Fit' box and set the max size to your desired size – we chose 60. You might need to play around here to get the look you want.
3. Change the color of your text to contrast with your panel.
4. Set the Vertical and Horizontal Overflow to "Overflow".

5. Change the Alignment to center horizontally and vertically.

Now, when you run the game, you should see your static panel in your view and the text in front of you.



GETTING STARTED WITH SCRIPTING

In Unity, all of the game play is done through scripting. You have three choices for scripting, JavaScript, Boo, and C# – we'll be doing this tutorial in C#. By default, scripting is done in MonoDevelop, a tool included with Unity, but you may have changed this going through earlier parts of the tutorial – you can use any editor you'd like to edit scripts, but MonoDevelop is pretty good for the small ones we'll be writing in this game.

Many of the preconfigured assets that are available through Unity come with their own scripts, such as the character controller we added. For some good scripting resources and a primer, check out the following links.

Unity3D Wiki – Scripting

Official Unity Scripting Reference

Unity Scripting Lessons

IMPLEMENTING THE TIMER

Since we've got the placeholder text in order, it's time to actually give it some functionality. Since we're writing our first custom script for the timer, we'll launch MonoDevelop (or your IDE of choice) when we create the new file.

Action Item: Create a timer script to time your maze

1. Under your Assets folder in the Project hierarchy, right-click and select Create -> C# Script. Name the new script ‘TimerController’
2. Drag your TimerController file onto your Text GameObject and double-click the script to open it in your editor.

By default, Unity will generate a code template when you create new scripts from within the editor. The boilerplate code gives you everything you need for the basic functionality of your script, including two methods that will manage the initialization of the script and update it on each tick of the gameplay.

```
using UnityEngine;
using System.Collections;

public class TimerController : MonoBehaviour {

  // Use this for initialization
  void Start () {


  }

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

  }
}
```

The two methods are pretty straightforward: `Start()` is called when the scene is first rendered, and `Update()` is called on subsequent frames of the gameplay. To create the timer text, the first thing we’ll want to do is add the following directive:

```
using UnityEngine.UI;
```

This will allow us to interact with our GUI objects on the screen so that we can update our text box text. Next, we’ll declare our variables for two objects – the text box we’re editing, and a float to store the timer. Above the `Start()`function, add the following lines:

```
static float timer = 0.0f;
public Text text_box;
```

Unity uses their editor to interact with the scripts we write, so we'll actually be assigning the `text_box` item back in Unity, rather than in code, so it's important to make sure that the `Text` item is actually visible outside of the script file.

Once we have those lines set, the next thing we'll do is include an incremental addition to the `Update()` method and update our UI to show the running time:

```
// Update is called once per frame
   void Update () {
    timer += Time.deltaTime;
    text_box.text = timer.ToString("0.00");}
```

With that, we're almost done with our first script – all that's left is assigning our UI Text box to be the one we update with the timer.The final code should look like this:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

static float timer = 0.0f;
public Text text_box;

public class TimerController : MonoBehaviour {

  // Use this for initialization
  void Start () {


  }

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

```
        timer += Time.deltaTime;
        text_box.text = timer.ToString("0.00");
      }
    }
```

Save your script and return back to Unity.

To set the target for our `text_box`:

1. Select the Text box in the Hierarchy and scroll down to the bottom of the Inspector.
2. Under the TimerController (Script) header, select the dot on the far right of the box next to ‘Text_box’
3. Choose the Text object in the window that pops up by double-clicking

When you run your game now, your GUI should display a running timer in your text box, trimmed to two decimals.

WINNING THE GAME

Now that we have the timer to track our pace, it’s time to add a “finish line” of sorts – you can extend the gameplay in a variety of ways that we won’t necessarily cover here, but for simplicity we’ll just be adding a hidden goal that the player needs to find. To do this, we’ll need to do a couple of things:

- Create a GameObject where we want the end of our maze to be
- Add a ‘Game Over’ UI element
- Write a script to handle finding the ending trigger, displaying the Game Over UI, and resetting the character position & timer

CREATING A FINAL GAMEOBJECT

Action Item: Add a capsule 3D object to represent the end point of the maze
The first thing we want to do is create an element in our maze that looks different enough for the player to recognize that they’ve gotten to the finishing point. You should add this far enough away from your spawning point (the initial position of your character controller) that the maze doesn’t immediately give away the final element – in this case, we are going to place our “game over” object in the far corner from the starting point.

To create our ending point:

1. In the hierarchy, click Create -> 3D Object -> Capsule
2. Drag the capsule to your desired location

PARTICLE SYSTEMS

Right now, we have a capsule, but it looks pretty boring so we're going to add a particle system to make it more interesting.

Action Item: Create a particle system effect for our end point
Particle systems are effects that you can use in Unity to give game objects their magic. You can use them during animations, to create interesting lighting effects, give your environment wow factor, represent in-game interactions – the opportunities are pretty much endless. In this section, we're going to be giving our capsule a particle system using the built-in Unity particle asset package so it gives the user the memo that it's not a default component of the maze.

- The Particle System – a beginner overview tutorial from Unity3D on what particle systems do and how to implement them
- Particle system documentation

The first thing we'll need to do to create our particle system is import the Unity asset package, as this comes with a variety of effects that we can use in our maze. You can find a lot of additional particle effects on the asset store, but for now, we'll just be using the default since Unity gives a lot of flexibility with how you can manipulate particles in different ways.

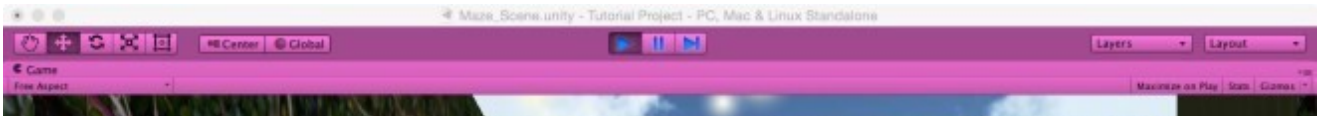
1. Go to Assets -> Import Package -> Particles
2. Import the asset package into your project
3. Select your capsule and click "Add Component"
4. Choose Effects -> Particle System

What you should be seeing now is a series of glowing objects flying out of your capsule. We want to make the effect a little less intrusive so it's not too obvious from the rest of our maze, so we're going to go ahead and change some of the characteristics of our particle system and change how it renders. You can be creative, or copy what we ended up with by changing the follow particle system attributes in the Inspector on your capsule by ticking the header for particular characteristics and making the following changes:

1. Start Speed: 1
2. Start Lifetime: 4
3. Inherit Velocity: 4
4. Start Size: .4
5. Shape -> Angle: 0
6. Color Over Lifetime -> Gradient (we made ours gold)

After changing the particle effect, we want to hide the mesh for the capsule by unchecking "Mesh Renderer" on the capsule object in the Inspector.

Lastly, we'll rotate our capsule -90 degree on the X axis so it appears our particles are floating up into the sky and gives the player a sense of stepping into the ring they form.



PUTTING IT ALL TOGETHER

Action Item: Write the GamePlay Controller
Now that we have all the pieces together, there are just a few more steps to finishing up our game play. We're going to implement a timer feature that does the following:

- Captures how long it takes to go through the maze
- Recognizes when we collide with our particle system (the game ending trigger)
- Displays our "Game Over" UI
- Resets the timer and the character position for another chance to go through the maze

We're going to do this by improving our earlier timer script. To do this, we'll have to make a few changes to where we have the script in the game and add a few new lines of code to our current script.

Action Item: Update Timer Script
The first thing that we'll need to do is add a few more variables to our timer script to track 1) our character's starting position 2) whether or not the timer is running and 3) which character controller in the scene is being used. Since we only have one, this is straightforward, but we'll still need to reference it in our script, so in the variable goes.

Action Item: Add in global variables
To implement these, add the following lines of code to your TimerController script under the timer and text_box objects:

```
public bool isRunning = true;
```

```
Vector3 startPosition;
public CharacterController characterController;
```

Action Item: Store the starting position
In order to reset the position of our camera once the player has finished navigating through the maze, we need to store our initial coordinates for the character controller in the `startPosition` object. The 'Vector3' object stores these coordinates when the game is launched, so we'll be including this in our `Start()` method.

```
// Use this for initialization
void Start () {
  startPosition = characterController.gameObject.transform.position;
}
```

Now that we have the player's initial starting point, we'll need to create a new function that triggers the behavior of our end game. We've already initialized the boolean that tells us whether or not the timer is running, so we can move right into our `OnTriggerEnter` function. This will be called when our player controller collides with our particle system.

Note: If you were creating a more complex gameplay system, you would need to check which object was colliding with our capsule in the `OnTriggerEnter` *event, but since we only have one character and our particle system capsule, we'll skip this part now for simplicity.*

Action Item: Create the `OnTriggerEnter()` and `Reset()` functions
The `OnTriggerEnter()` function is what Unity will call when your object recognizes that there is an overlap between it and another GameObject – in this case, we are writing the function for our capsule, and whatever object collides with it (in this case, our character) will be passed in as an argument.

```
void OnTriggerEnter(Collider other)
    {
        isRunning = false;
        Reset ();
    }
```

At this point, we should get an error if we switch back into Unity since we haven't added our reset function. In this case, we've pulled it out as a separate function so we can modify the reset behavior without getting strange behaviors in`OnTriggerEnter()` but for the basic behavior, you could add in the `Reset()` code to the collision trigger if you so chose.

Add in the `Reset()` function by copy and pasting the following function. `Reset()` will move our character controller back to the start of the maze, begin the timer at zero again, and restart the timer.

```
void Reset()
    {
        characterController.gameObject.transform.position = startPosition;
        timer = 0.0f;
        isRunning = true;
    }
```

Action Item: Change the `Update()` function to check if the timer should be running
With those two functions in place, we need to make one minor change to our `Update()` call so that the timer checks to make sure it's running before incrementing the number of seconds on the clock. This is an important step because we will later want to implement a keypress to start the timer, and we need it to stay paused until it's reset when the maze is completed. To do this, we'll just be putting an if-statement around the current `Update()` code to check the boolean we declared above:

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

        if (isRunning) {
                        timer += Time.deltaTime;
                        text_box.text = timer.ToString ("0.00");
                }
    }
```

After that change, we're just about ready to go – but we've got a few more tweaks to do in Unity itself so that the script runs. The full code should look like this:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class TimerController : MonoBehaviour {
```

```
    static float timer = 0.0f;
    public Text text_box;
    public bool isRunning = true;
    Vector3 startPosition;
    public CharacterController characterController;

    // Use this for initialization
    void Start () {

        startPosition = characterController.gameObject.transform.position;
    }

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

        if (isRunning) {
                        timer += Time.deltaTime;
                        text_box.text = timer.ToString ("0.00");
                }
    }

    void OnTriggerEnter(Collider other)
    {
        isRunning = false;
        Reset ();
    }

    void Reset()
    {
        characterController.gameObject.transform.position = startPosition;
        timer = 0.0f;
        isRunning = true;
    }
}
```

Action Item: Update the script references in Unity

When we were first testing out our script, we had originally attached it to our GUI, but we're going to change this since we now have a collider to work with. Back in Unity, select the text GUI and delete the script out of the Inspector by clicking the drop down Settings menu on the script and selecting 'Remove Component'.

Attach the script to the Capsule by dragging and dropping the script from the Project directory onto the Capsule GameObject.

Try running your game. When you run into the capsule effects now, your character will automatically begin back at the start and the timer will reset.

Action Item: Wait on a key press to start the game
The last thing that we're going to do is add a function in our TimerController that allows us to trigger when we'd like to start our game. This will allow us to be in control of when the game play actually starts, whereas the current behavior simply starts the gameplay over immediately. We will need to:

1. Prevent the timer from running automatically
2. Display an instructional text in the UI while our timer isn't running
3. Modify our `OnTriggerEnter` function to reset and wait for the player input

First, we want to change our timer declaration. Over in our `TimerController.cs` script, change

```
public bool isRunning = true;
```

to

```
public bool isRunning = false;
```

This will prevent our game from starting automatically, but that's okay – we'll take care of that in a bit. For now, switch back over to Unity and locate your GUI Text object. Change the default text (ours is 'Hi!') to "Press F to begin." Then, switch back over to the capsule, and under the Inspector for the script, uncheck the 'isRunning' box.

Finally, we want to make a few changes to our `OnTriggerEnter()`, `Reset()`, and `Update()` methods that will allow us to control when we want the game play to begin.

Action Item: Finalize script functions
In the `Reset()` function, we want to change the behavior from immediately resetting the game and starting the timer to having the timer reset and allowing for another player to try. We're also going to remove the `isRunning = false`line from our `OnTriggerEnter()` function and move that into `Reset()` after we put the player back into the original position. Now, `OnTriggerEnter` will just call `Reset()`, and the two functions should look like:

```
    void OnTriggerEnter(Collider other)
```

```
    {
        Reset ();
    }

    /* Call Reset once we start the game over. */
    void Reset()
    {
        characterController.gameObject.transform.position = startPosition;
        isRunning = false;
        timer = 0.0f;
        text_box.text = "Press 'F' to begin";
    }
```

With these two functions in place, all that's left is to add in a check to see if the player has pressed the 'F' key to start the timer. We will add the following block of code under the existing lines in the `Update()` method. This will check first that the timer isn't running (we don't want to reset if the game is in progress) and if there was a registered key down input on the 'F' keyboard key. If these are both true, then we will begin the timer to start the game.

```
if (!isRunning & Input.GetKeyDown (KeyCode.F)) {
          isRunning = true;
      }
```

The final script we have will look like this:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;

public class TimerController : MonoBehaviour {

    static float timer = 0.0f;
    public Text text_box;
    public bool isRunning = false;
    Vector3 startPosition;
    public CharacterController characterController;
```

```
    // Use this for initialization
    void Start () {

        startPosition = characterController.gameObject.transform.position;
    }

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

                if (isRunning) {
                        timer += Time.deltaTime;
                        text_box.text = timer.ToString ("0.00");
                }

                if (!isRunning & Input.GetKeyDown (KeyCode.F)) {
                        isRunning = true;
                }

    }
    /* We want to check when the character collides with a trigger object,
       in this case, the particle system cylinder that ends the run. */
    void OnTriggerEnter(Collider other)
    {
        Reset ();
    }

    /* Call Reset once we start the game over. */
    void Reset()
    {
        characterController.gameObject.transform.position = startPosition;
        isRunning = false;
        timer = 0.0f;
        text_box.text = "Press 'F' to begin";
    }
}
```

With that, you have a basic maze game running! You can deploy to your platform of choice with Unity's build settings – have fun!