

Realtime Content Creation in Unity3D

What We're Doing

In this tutorial, I will be demonstrating the process of interacting with meshes in Unity via scripting. Why would you want to do this? It's because... Well... I'm not sure, but I needed it for a project I'm working on, so I'm assuming that you need it too. There's just really nothing compared to the feeling you get when you manually place vertices via code.

But what are we doing?

When you complete this tutorial, you should have a game that looks something like the screenshot pictured above. The idea behind the game is to get the ball into the box. This is performed by drawing lines on the background for the ball to bounce off of.

Great! I'm on board!

Hold on there. Before you start this tutorial, there're some things you need to understand. Like when I put something in **bold**, it means that that word is a class name (like **Transform**, or **Collider**, or **LineRenderer**). I'll provide links to the docs for all the classes I put in here, so don't worry if you come across one you haven't heard before.

When I put something in *italics*, I am referencing a function, like *OnMouseDown()* and *Update()*, and *StealTheNoodles()*.

Regarding language, this tutorial contains no profanity- I mean, I'll be using JavaScript. I will try to get C# code put in here as well, but given that I'm on a deadline, that might have to come in the second edition.

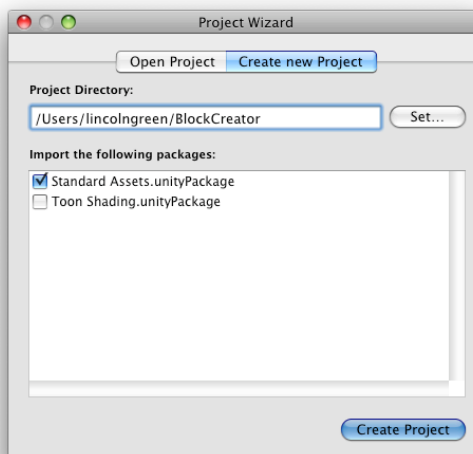
Completed Game

By Lincoln Green

Getting Started

Before I throw obfuscated code and screenshots at you, we need to determine how well you are acquainted with the Unity way of doing things. Click the link in that green box down there. If the stuff you read there is new to you, you should probably read that before you read this. But if you know it all by heart, or you're just really brave, you can dive in right here.

Create a new project and put in a name. I called mine BlockCreator. You should call yours something interesting, like Fuzzy Puppies on the Rampage.

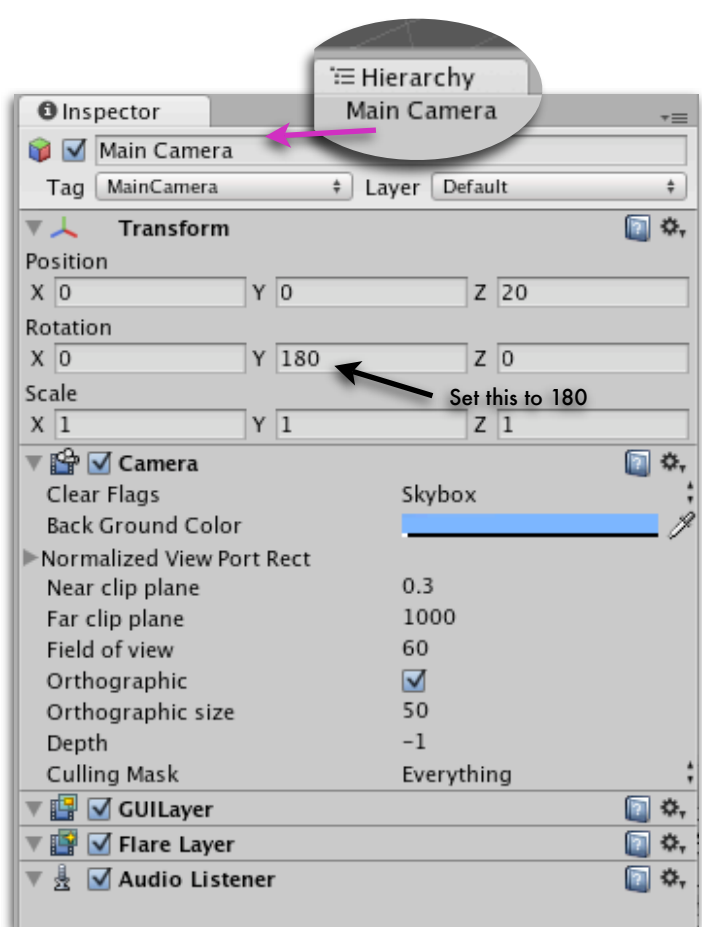


If you see stuff in a blue box, it means that you need to pay attention.

If you see stuff in a red box, it means that there is a common mistake lurking in the shadows.

If you see stuff in a green box, it means that there's an explanatory link or text, like [this one](#).

Everyone still alive? Good. Moving on, you will be presented with a clean window. For my project, I'm using the 'Wide' layout, so if the screenshots don't match up, it's possible that it's not your fault. Now the first thing to do is set up our camera. Since this is a 2D game, we need to set up our camera to be orthographic. Select the 'Main Camera' object in the hierarchy browser and take a look at the inspector. Turn on the 'Orthographic' checkbox and set the 'Orthographic Size' to 50. I also changed the background color of my camera to be a light blue(as opposed to the default dark blue). Now since we're working in two dimensions(x and y), and the camera is orthographic, the z variable is irrelevant for now. However, I set mine to '20' to get it out of the way for the next steps. You also do need to set the 'Y' rotation to 180. When you're done, your inspector should look similar to that screenshot.



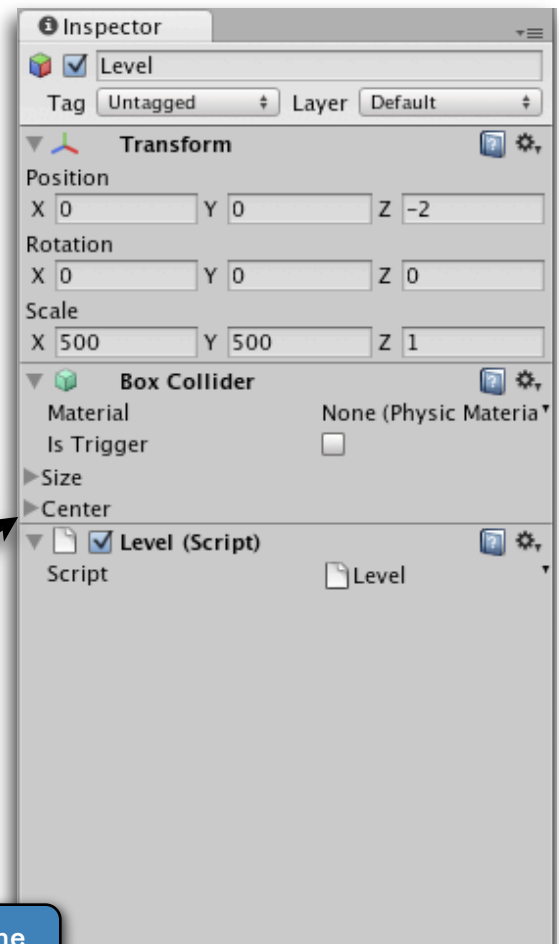
Setting up the level

In order to provide the dragging ability of our game(drag to create boxes), we need some way of intercepting mouse events. Fortunately for us, the **MonoBehaviour** class provides two useful functions, *OnMouseUp()* and *OnMouseDown()*. A glance at the docs for those methods informs us that we need a collider to receive these events. So...

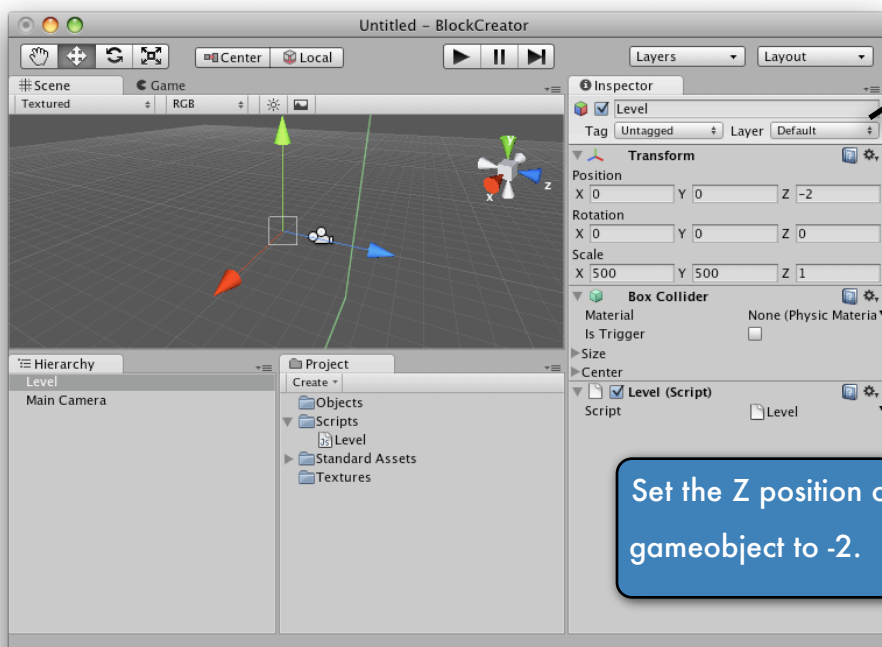
Create an empty **GameObject**. (GameObject > Create New)

Add a **BoxCollider** to said **GameObject**. (Component > Physics > Box Collider).

Set the scale of the GameObject you created to a very large value for the 'X' and 'Y' axes. I used 500. Leave the 'Z' scale alone. Also, make sure to set the z position of the box to -2. This step is very important, as it keeps the gameobject from colliding with objects that are actually part of the scene(i.e the ball and the box that don't exist yet). In the end, your setup should match my setup. Also feel free to name your gameobject. I named mine 'Level'.

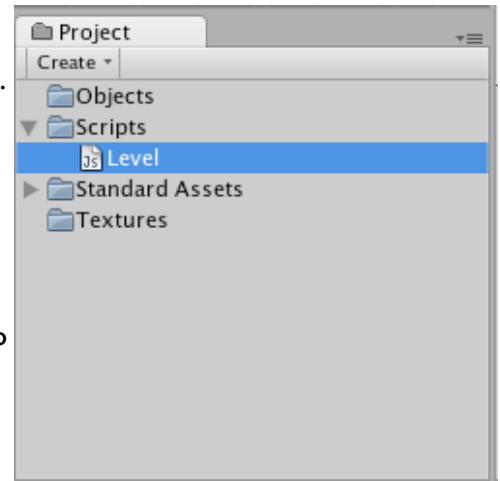


Set the Z position of the gameobject to -2.



Setup a Folder System

Create three folders and name them: "Objects", "Scripts", and "Textures". Now create a new script and call it 'Level'. This script will handle the creation of the block for the balls to bounce off of.



Let's Script!

Open up the 'Level' script in your favorite script editor. The first thing to do is define some variables.

Code:

```
var blockHeight : float = 1.0; //how tall are the blocks
var blockMaterial : Material;
var blockPhysicMaterial : PhysicMaterial; //What kind of
physical properties do the blocks have? (i.e rubber, wood,
metal, bouncy)

private var mouseHit : Vector3; //the position of the
mouse when a button was pressed
private var dragging : boolean = false; //is the mouse
button pressed?
```

When you define a var, it is by default visible in the Unity inspector. However, if you mark it as 'private', it will be hidden.

If you don't understand what these variables will be used for, don't worry - All will be explained in due time. Now let's write some actual code! Before we override the **MonoBehaviour** methods, we need to create a method that will grab the position of the mouse and turn it into a vector.

Code:

```
function MousePoint() : Vector3{
    var ray =
    GameObject.FindWithTag("MainCamera").camera.ScreenPointToRay
    (Input.mousePosition); //get the mouse position from the
    Input class, and use the main camera to cast a ray
    var hit : RaycastHit; //create a variable to store
    raycast data
    Physics.Raycast(ray, hit, 1000,
    kDefaultRaycastLayers); //cast a ray
    return hit.point; //And return the Vector3 of the
    point we hit
}
```

A Matter of Explanation

In the first line, we create a ray using the **Camera** function **ScreenPointToRay()**. We then create a **RaycastHit** variable (named 'hit'). When we do a **raycast(Physics.Raycast())** and pass the 'hit' variable into it, the 'hit' variable will be modified to represent the data that the **Physics.Raycast()** finds. In the last line, we simply return the vector of the 'hit' variable - the vector being the point that the raycast hit.

What's happening? We are taking the mouse position from the Input class. But the mouse position is returned in screen coordinates, so we use the camera to cast a ray. The point in 3D space that is returned in the 'hit' variable is where the mouse is.

Intercepting Mouse Events

So we have a function that will grab the mouse position and turn it into something we can use (a Vector3). Now we need to write the code that will actually intercept and interpret the mouse events.

```
function OnMouseDown(){
    mouseHit = MousePoint(); //Get the mouse position as a
    Vector3 and store it for future use
    mouseHit.z = 0; //We do this because we want all our
    objects to be at the same position on the z axis - it's a 2D
    game.
    dragging = true; //if the mouse is down, we're dragging
}

function OnMouseUp(){
    dragging = false; //if the mouse is up, we're not dragging
    anymore
}
```

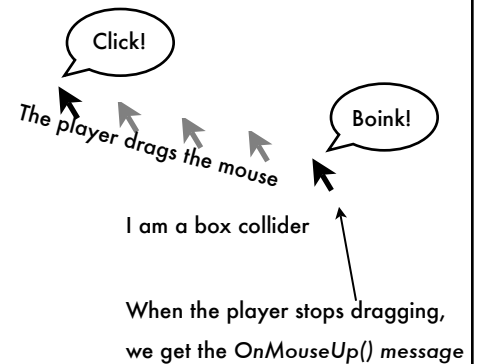
So what are we doing here? We'll start with `OnMouseDown()`. Remember, this method is triggered when the player clicks on the collider. Remember when we created a box collider and scaled it up? This is the collider that will receive the clicks. The first thing we do is store the current mouse position (the mouse position at the time of the click) as a Vector3. We will use this later. Next, we make sure that the z position of this vector is at 0. We do this because we don't want to create blocks at different z positions - we want them to all be at 0 on the z axis.

Next, we set the 'dragging' variable to be true. This is so the rest of the class will know that the player is clicking on the collider.

Now we move on to `OnMouseUp()`. This function is simple - all we do is notify the rest of the class that the player is not clicking on the collider any more.

Mouse Event Methods

When the player clicks, we get the `OnMouseDown()` message

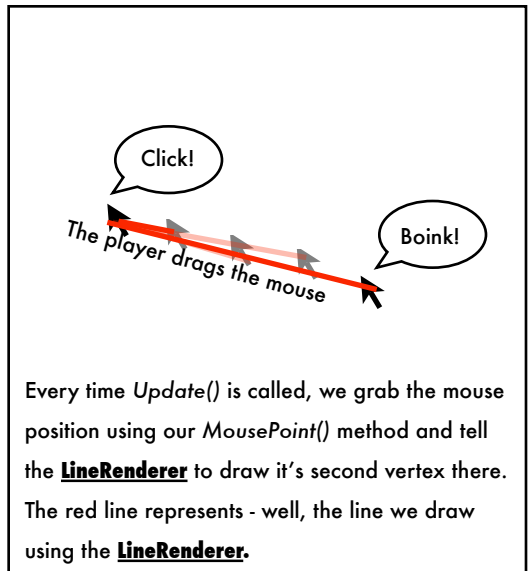


The `Update()` method

We will draw our line in the `Update()` method. In case I forgot to mention it, this line that we will be drawing is used to tell the player that something is happening when he drags the mouse.

```
function Update(){
    if(dragging){ //i.e if the mouse is down...
        GetComponent(LineRenderer).enabled = true; //if we're
dragging, we want to draw a line
        GetComponent(LineRenderer).SetPosition(0, mouseHit); //set
the first point of the line renderer to the position of the
mouse when the click occurred
        GetComponent(LineRenderer).SetPosition(1,
MousePoint()); //grab the current position of the mouse and
make the line draw to it
    } else { //if we've received a MouseUp event and we're not
dragging anymore, we
        GetComponent(LineRenderer).enabled = false; //stop drawing
the line
    }
}
```

Update()



Assuming you've read the blog post linked at the beginning of this tutorial, you should have a fairly good understanding of what's happening here, even if you've never heard of the **LineRenderer** component before. Basically, a **LineRenderer** is a component that renders lines based on values that the programmer(you) inputs. In this instance, we are putting two `Vector3s` into the **LineRenderer**. As you have probably figured out by this point, these two vectors are the `mouseHit` variable(which is of course assigned to the position of the mouse when the click occurred) and the current position of the mouse(returned by `MousePoint()`).

There's one last thing to do before we go back to Unity. Add this line to the end of your script:

```
@script RequireComponent(LineRenderer)
```

Note that there is no ';' at the end of this line.

This tells Unity that we want a **LineRenderer** component on whatever gameobject we're tied to. More on that in a minute.

Enjoy the fruits of your labor!

Congratulations! You are now at that glorious stage of the tutorial where you can press the big black button and see it change to the big glowy blue button. Almost. Yeah, there's still a little bit left to do. First, drop your 'Level' script onto the gameobject you created in the first part of the tutorial(the one with the big box collider). Note that Unity autmagically adds a **LineRenderer** component to the gameobject as well. That is the purpose of the '@script RequireComponent(LineRenderer)' line.

Next, check your script to make sure everything matches up:

What your script should look like:

```
var blockHeight : float = 1.0;
var blockMaterial : Material;
var blockPhysicMaterial : PhysicMaterial;

private var mouseHit : Vector3;
private var dragging : boolean = false;

function Update(){
    if(dragging){
        GetComponent(LineRenderer).enabled = true;
        GetComponent(LineRenderer).SetPosition(0, mouseHit);
        GetComponent(LineRenderer).SetPosition(1, MousePoint());
    } else {
        GetComponent(LineRenderer).enabled = false;
    }
}

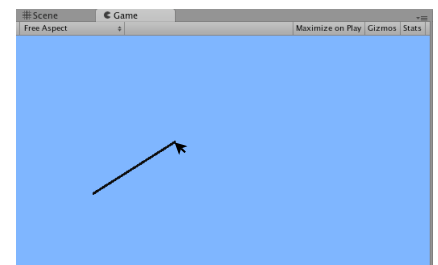
function OnMouseDown(){
    mouseHit = MousePoint();
    mouseHit.z = 0;
    dragging = true;
}

function OnMouseUp(){
    dragging = false;
}

function MousePoint() : Vector3 {
    var ray = GameObject.FindWithTag("MainCamera").camera.ScreenPointToRay(Input.mousePosition);
    var hit : RaycastHit;
    Physics.Raycast(ray, hit, 100, kDefaultRaycastLayers);
    return hit.point;
}

@script RequireComponent(LineRenderer)
```

At this point, you should be able to play your game without any errors. Drag, and a line will magically appear, stop dragging and it will disappear. Congratulate yourself, pat yourself on the back, go on vacation, and GET BACK TO WORK!!!



Part 2 - Let's add some blocks

I'm sure it was really fun, dragging the mouse around, watching the lines come out, but now it's time to get back to the scripting. We need to add one more function to our 'Level' class/component. This function is called `CreateBlock()` and it takes two `Vector3`s as arguments.

```
function CreateBox(p1 : Vector3, p2 : Vector3){  
}
```

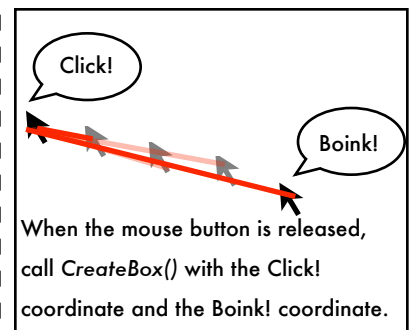
The purpose of this function is to construct the blocks for the ball to bounce off of. Rather than using a prefabricated mesh, we will construct the blocks manually, in code. The two arguments(`p1` and `p2`) are the beginning and end points of the blocks. To give you some context, look back at your `OnMouseUp()` function. Add the specified code:

```
function OnMouseUp(){  
    dragging = false; //if the mouse is up, we're not dragging anymore  
    CreateBox(mouseHit, MousePoint());  
}
```

The `CreateBox()` function will create a block based on the coordinates of the mouse at time of click and the mouse position at the end of the drag. Back to the `CreateBox()` function:

```
function CreateBox(p1 : Vector3, p2 : Vector3){  
    var newLedge : GameObject = new GameObject("Block"); //create a new  
    gameobject. This gameobject will hold the mesh we're creating.  
    var newMesh : Mesh = new Mesh(); //this is the mesh we're creating.  
    newLedge.AddComponent(MeshFilter); //this is what makes the mesh  
    available to the other mesh components  
    newLedge.AddComponent(MeshRenderer); //this is what makes the mesh  
    visible  
}
```

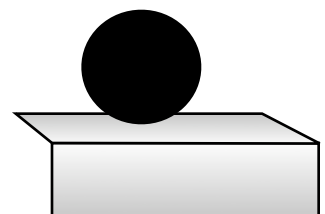
CreateBox()



Alright, so we've got a mesh, and we've attached it to a new gameobject. Next, we build the mesh. The process of building a mesh involves:

- 1.Creating vertices(`Vector3`)
- 2.Creating UVs(`Vector2`)
- 3.Creating triangles(lists of vertex indices)
- 4.Assigning materials, physic materials, etc.

The mesh we are creating will consist of 6 vertices. It will not be a *real* cube per sé, because we do not need some vertices due to working in 2D. To the right is a figure demonstrating the layout of the vertices, and how the sphere will interact with the blocks we're creating.



Creating the Vertices

The first thing to do is define some Vector3s to store the positions of the 6 vertices. Put this into your CreateBox() function:

```
var topLeftFront : Vector3 = p1;  
var topRightFront : Vector3 = p2;  
var topLeftBack : Vector3 = p1;  
var topRightBack : Vector3 = p2;  
var bottomLeftFront : Vector3;  
var bottomRightFront : Vector3;
```

The only difference between the upper front and back vertices is there z position. The reason we have to have front and back vertices is because the game is 2D, but the objects are 3D, and so the collisions are also 3D. Therefore, we must have some thickness on the top, or the ball will not collide with our ledge. Put this code in to setup the z position of the upper vectors.

```
topRightFront.z = 0.5;  
topLeftFront.z = 0.5;  
topLeftBack.z = -0.5;  
topRightBack.z = -0.5;
```

With the upper vertices setup, we move on to the lower vertices. Once again, the lower front vertices are the same as the upper front vertices, except that they are further down the y axis(to provide the height of the block that the player sees). Note that we do not need to set the z position of the vertices, because we are setting the lower vertices to be equal to the upper ones, which already have the z position set.

```
bottomLeftFront = topLeftFront;  
bottomRightFront = topRightFront;  
bottomLeftFront.y -= blockHeight; //remember the block height variable  
we defined?  
bottomRightFront.y -= blockHeight;
```

Now that we have the vertices defined, we just have to add them to our mesh(the newMesh variable we defined earlier).

```
newMesh.vertices = [topLeftFront, topRightFront, topLeftBack,  
topRightBack, bottomLeftFront, bottomRightFront];
```

Creating the UVs

I do nothing fancy as far as UVs - this code is pretty much taken straight from the documentation. All this block of code does is cycle through the vertices and create 2D coordinates for them based on their x and y positions in the 3D coordinate system.


```
var uvs = new Vector2[newMesh.vertices.length];
for (var i=0; i<uvs.Length; i++) {
    uvs[i] = Vector2(newMesh.vertices[i].x, newMesh.vertices[i].z);
}
newMesh.uv = uvs;
```

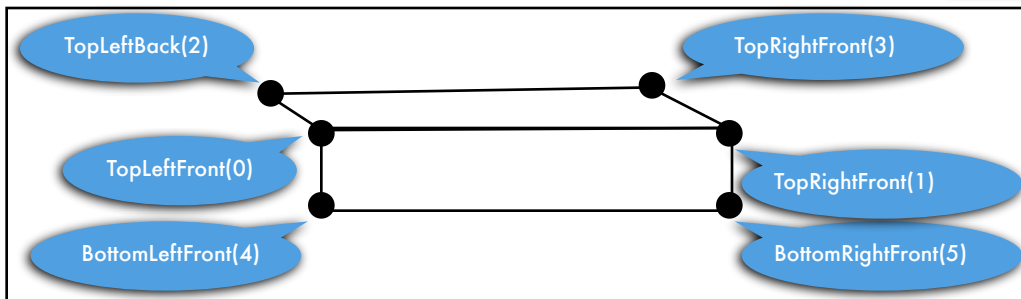
Creating the Triangles

Now it's time to create the triangles. Note that we're not really **creating** anything - we're simply telling the mesh how to connect the dots(vertices). If you look at the documentation for the **Mesh** component, you will see a 'triangles' variable. When you set this variable, you pass in sets of 3 numbers. Each set of 3 numbers tells the **Mesh** component which vertices to draw a triangle in. Note that you *must* pass in sets of three - if you don't, then the **Mesh** component might get confused, and there's nothing worse than a confused **Mesh** component.

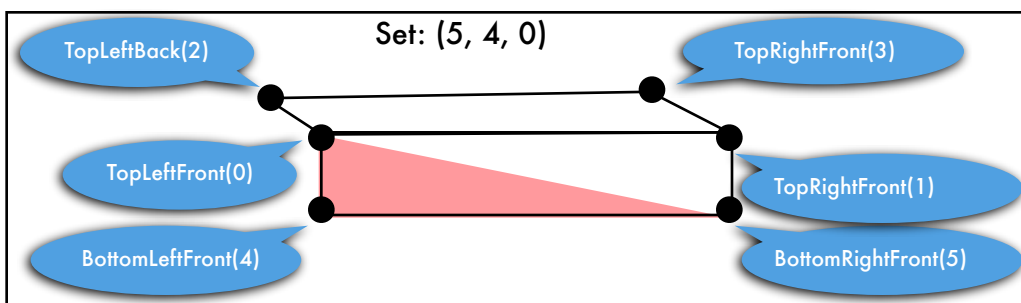
Here's the code:

```
newMesh.triangles = [5, 4, 0, 0, 1, 5, 0, 2, 3, 3, 1, 0];
```

A seemingly innocent little line of code. Now since this can be a little hard to envision, and I have an unfilled diagram quota, I'm going to draw an explanatory image.



And now, I will explain the explanatory image(weird, huh?). So as you've probably guessed, the black dots are vertices, the little blue balloons are the names and indices of the vertices. Easy, right? Now to show the effects of that line of code, I've got a series of diagrams. Except these diagrams show the triangles being filled in and which vertex set is doing it. The blue triangles are triangles that have previously been filled in, the red triangle is the one that is being filled in.



"I decided to be bold and adventurous, and filled in the triangles myself, without reading your tutorial. But now, they're invisible!"

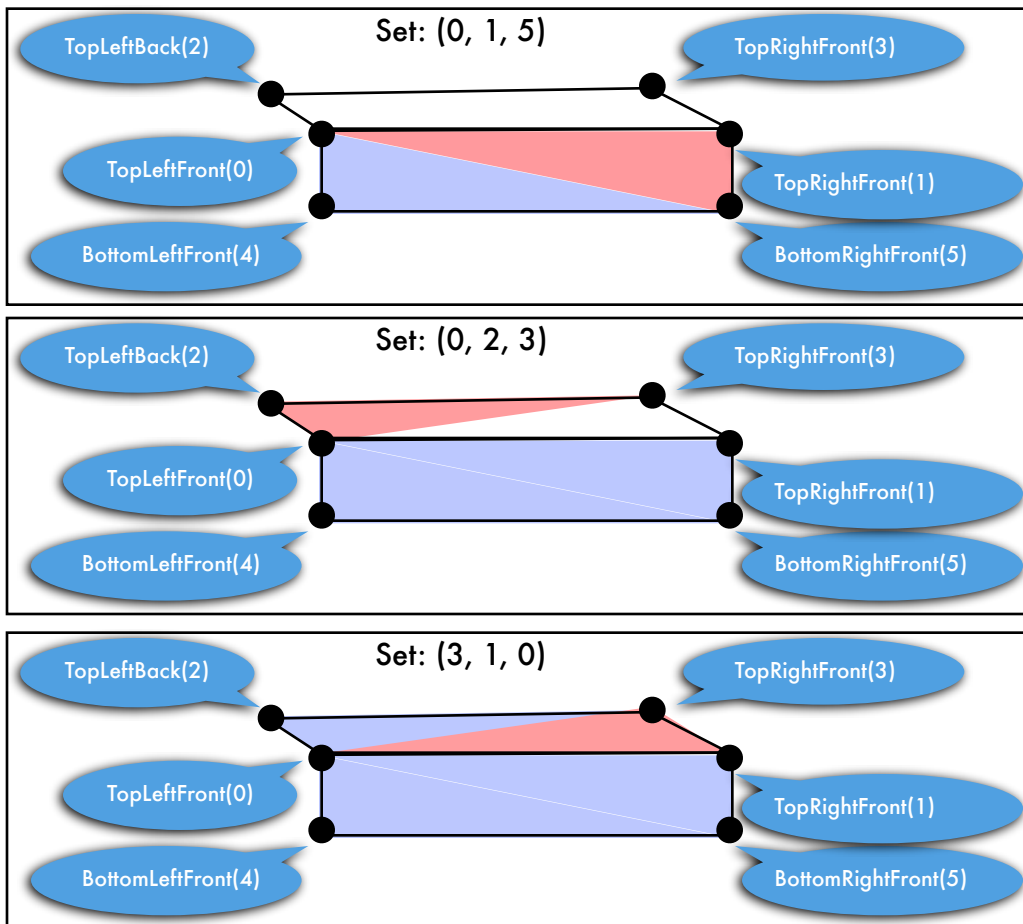
The problem here is that triangles are one sided, so the chances are, you put the vertex indices in the wrong order. Try flipping them around and see if that helps.

Example: if you had:

`newMesh.vertices = [5, 4, 1, 2, 5, 6];`

Make it:

`newMesh.vertices = [1, 4, 5, 6, 5, 2];`



Pretty cool, huh? Now there's one last little itty bitty piece of code you have to type in. Basically, we have to tell our shader how to light our object.

```
newMesh.RecalculateNormals();
```

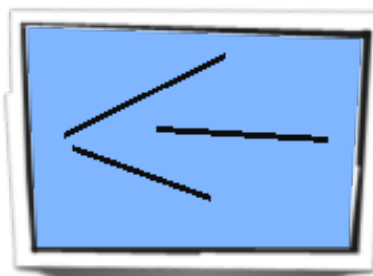
All we've got left to do now as far as creating boxes is to tie everything together. We need to tell our 'newLedge' gameobject what its mesh is, give it a material, a physic material, and a collider.

Remember the blockMaterial and blockPhysicMaterial we defined a while ago? Here's where those two variables come in. Note that we check to see if they exist before using them.

```
newLedge.GetComponent(MeshFilter).mesh = newMesh;
if(blockMaterial) newLedge.renderer.material = blockMaterial;
newLedge.AddComponent(MeshCollider);
if(blockPhysicMaterial) newLedge.GetComponent(MeshCollider).material = blockPhysicMaterial;
```

Now if you've correctly typed in all your code and saved your file, you should be able to jump back to Unity and hit the play button. See the block world unfold before your very eyes... Power! Glory! HONOR!!!

Power, glory, and honor aside, the blocks are kind of ugly at the moment. Don't worry about that - we'll fix that in the next section of this tutorial, "Spitting on Your App". No wait, that should be "Adding Spit and Polish to Your Game."

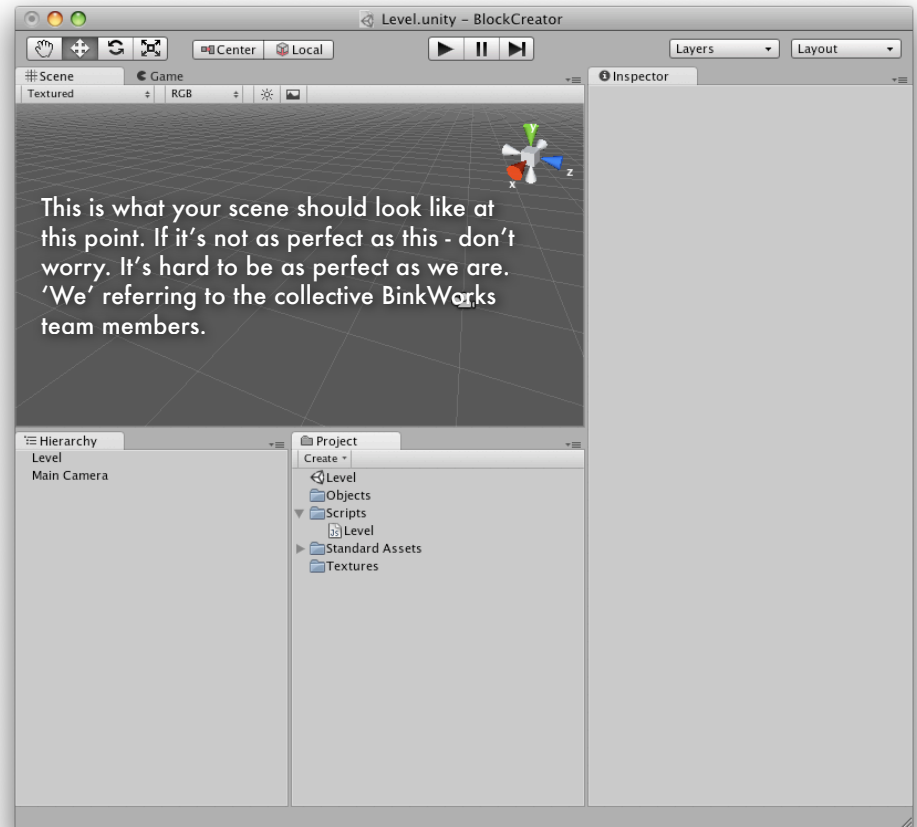


Finished product

Make sure to save your Unity scene file! When saving, name it 'Level'. If you don't, there may be complications further along in the tutorial.

Part 2

HAI LOOK AT
THAT FUNNY
ROTATED TEXT!



Creating Gameplay and Adding (some) Polish

You have come far...

Welcome to Part 2 of the BinkWorks realtime content generation in Unity3D tutorial. In the last section, you learned how to generate blocks for a ball to bounce off of. In this section, we're going to... add the ball! Now if you remember, the objective is to bounce a ball into a box. So we will add a ball, a box, and a playing area. We will also add a bit of polish. This section should not take as long as the first, as there's less scripting and more setup in Unity.

What happens in this section

In this section, we will:

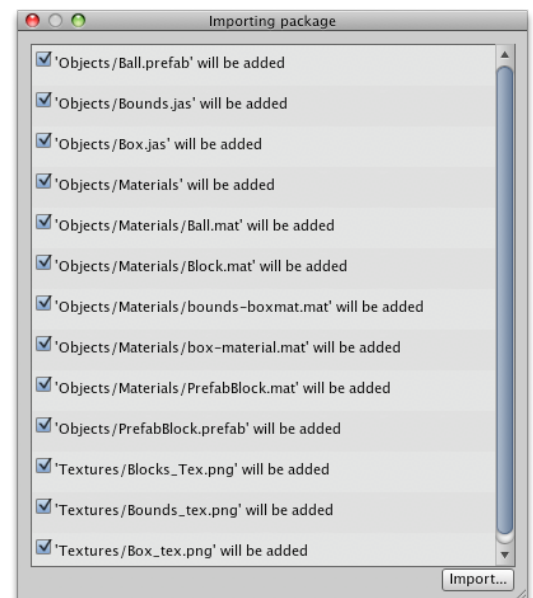
1. Give the player the ability to win
2. Add a visible playing area
3. Add some (admittedly kind of awful) art.

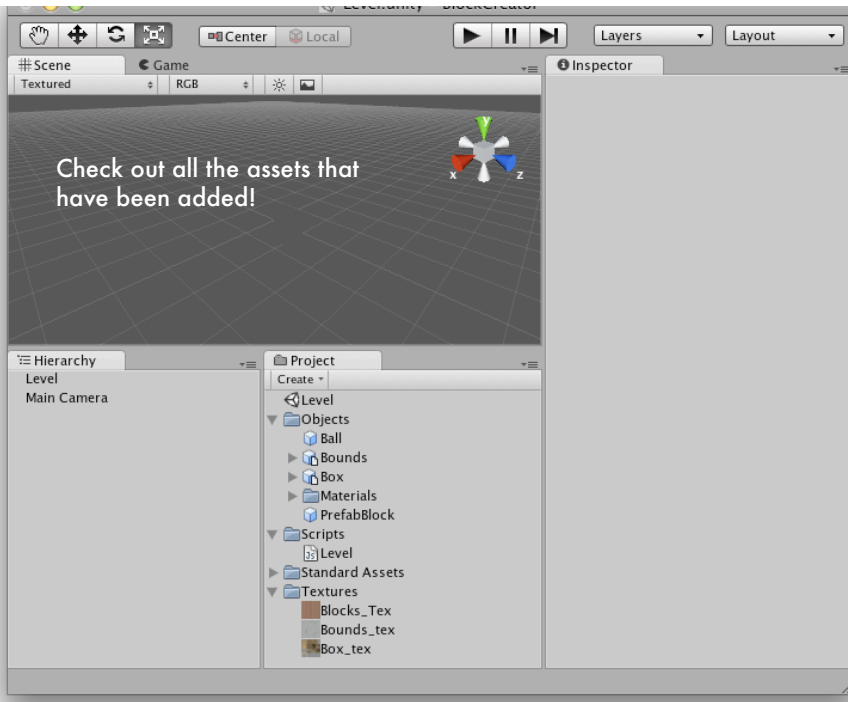
First step: Download Files

In this section we will have to make use of some custom meshes. Rather than making you do your own work, I've done it for you. Ain't I great? Anyway, download the stuff from [this link](#)....You're done? Good.

Now unzip it and open up the Unity asset package(double click it in the Finder). Make sure that your 'Block Creator' or 'Fuzzy Puppies' or 'Whatever you named it' project is open. If it is not, then the assets will not be added.

You will be presented with a dialog similar to the one on the right. Make sure all the items are checked and click the 'Import...' button. The assets should now be added to your project, resulting in...


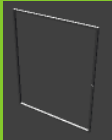





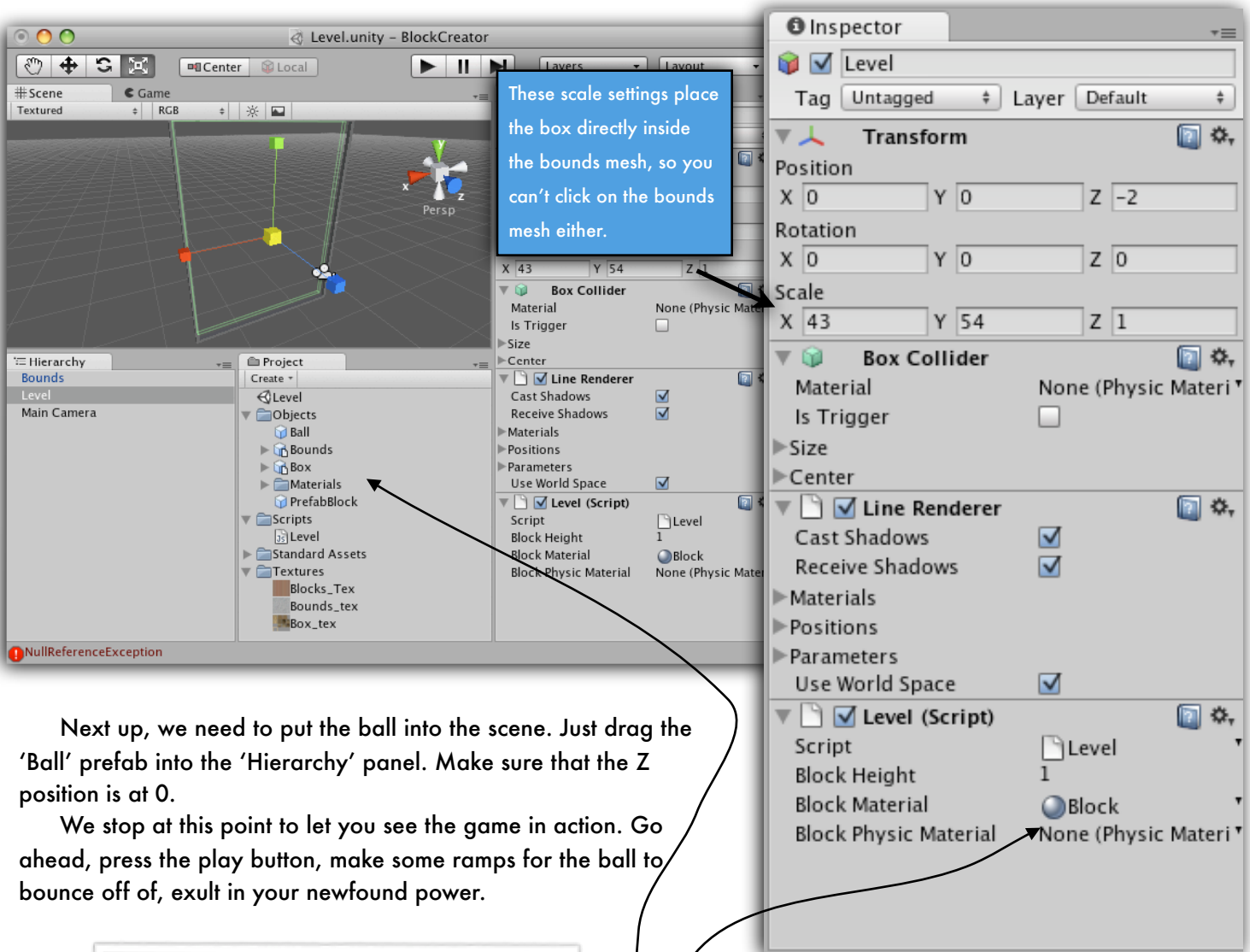
Import Settings...

When you select a mesh in the 'Project' tab, the Unity inspector changes to reveal a section labeled 'Import Settings'. This panel has many useful features, including a 'Generate Colliders' option. Turn this on, and when you drag your mesh into the level, Unity will automatically add a mesh collider to the object.

...This window. As an explanation, I've put together a table of the included assets and their purpose(s).

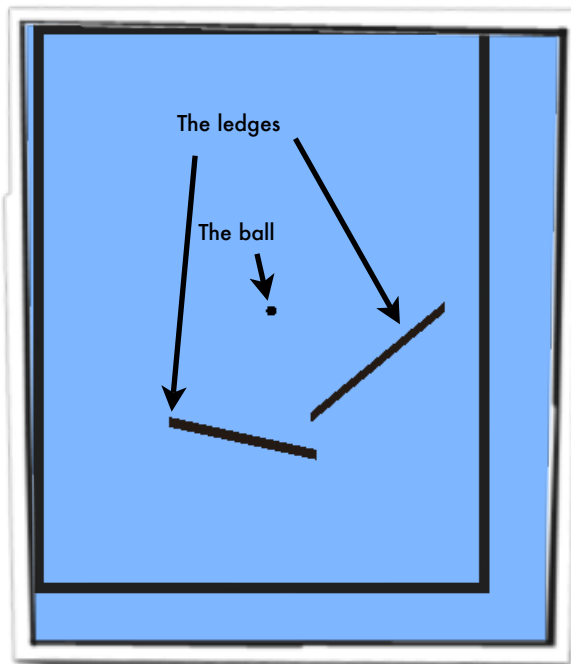
Asset	Purpose	Image
Ball	This is a Unity prefab. When dragged into your project, it is the ball that bounces around.	
Bounds	This is a mesh that determines the playing field of the game. You can make your own, if you're adventurous.	
Box	This is the box that the ball must get into on order for the player to win.	
PrefabBlock	This is a prefab. When dropped into your project, it will produce a block, much like the blocks we create when we drag the mouse. One usage for this object would be to create a ledge for the Box to sit on.	

Alright, so we've got our objects. Now it's time to put together a level. First, drag the 'Bounds' object into your level, and set its position to (0, 0, 0). Also, make sure it has a **MeshCollider** component attached to it. Now remember that big box collider we set up a while ago? You'll need to change the scale of that object to match the rectangle defined by the bounds mesh. Otherwise, the player will be able to create ledges outside of the playing area, which really makes no sense at all. Take a look at the screenshots below for an example of what you should be seeing.



Next up, we need to put the ball into the scene. Just drag the 'Ball' prefab into the 'Hierarchy' panel. Make sure that the Z position is at 0.

We stop at this point to let you see the game in action. Go ahead, press the play button, make some ramps for the ball to bounce off of, exult in your newfound power.

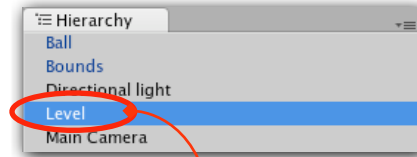
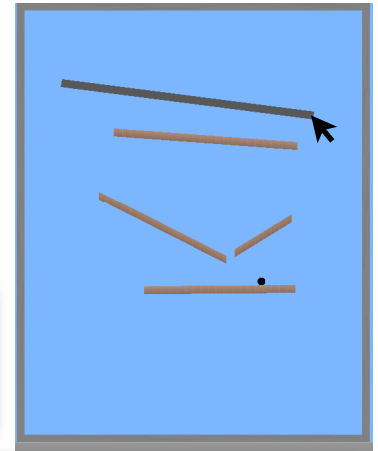


Regarding Size...

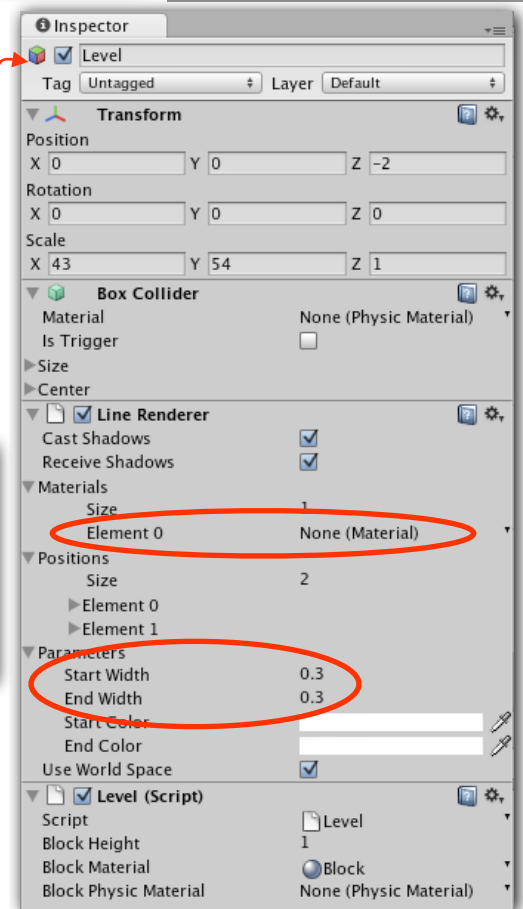
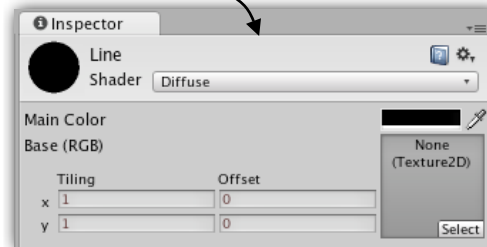
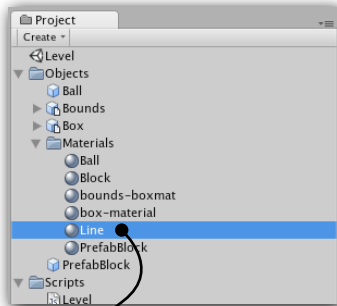
If the game is a little small, you can go to the inspector for the Main Camera object and set the 'Orthographic Size' to a smaller value. The odd thing about the 'Orthographic Size' variable is that the smaller you make it, the larger the objects look.

Now before we continue, let's give our stuff some texture. The texture for the 'Bounds' object is already assigned, but we need to assign the textures for the ledges. To do this, select the 'Level' object (the object with the box collider). In the inspector, there is a variable called 'Block Material'. Drag the 'Block' material (located in the 'Materials' folder) from the 'Project' pane into the 'Block material' variable.

Now even though we have assigned the textures and materials, they still don't show up because the scene doesn't have a light source. Create a new directional light (GameObject > Create Other > Directional Light). Change the rotation to whatever you see fit. I've set my rotation to: (X: 45, Y: 180, Z: 0). Now if you run your game, your textures should show up on the objects. However, there's still a problem. If you look at the screenshot to the right, you'll see that when we drag, we get a large, ugly gray line. To make this look better, select the 'Level' object and take a look at the inspector for the **LineRenderer** component that's attached to it. We're going to set the 'Start Width' and 'End Width' variables to be a little bit thinner. I set mine to 0.3. Note that I have circled the variables we are editing in the next few steps.

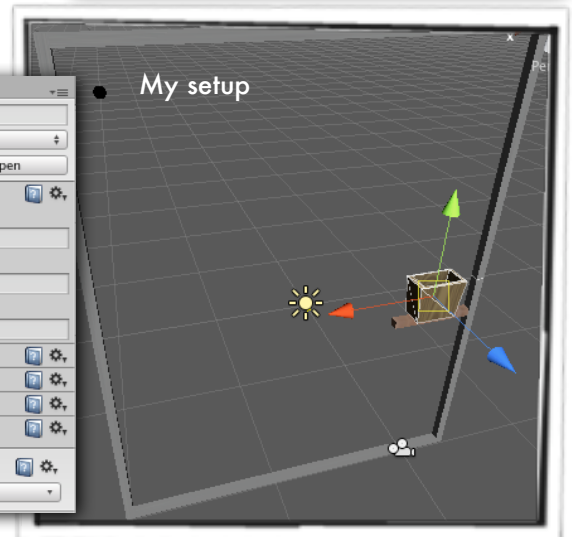
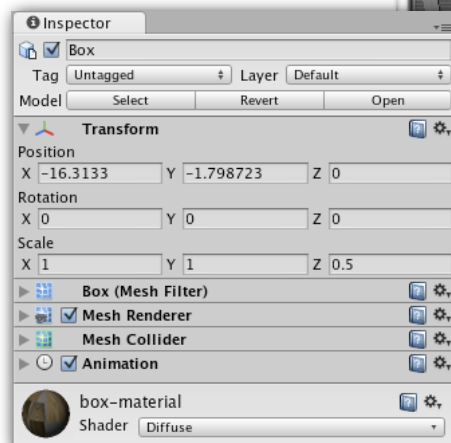


Our line is still pretty ugly. Let's change its color. Create a new material (Assets > Create > Material). Name the new material 'Line'. You can pretty much do anything you want with this material - I've just made mine a solid black thing. When you're done playing around with your material, go back to the 'Level' object inspector and put it into the 'Element 0' slot of the 'Materials' array. Alright, I admit it, it's still pretty ugly. But this tutorial isn't about the art, it's about the technical side of things, right? Right?



Alright, so we've added the textures...

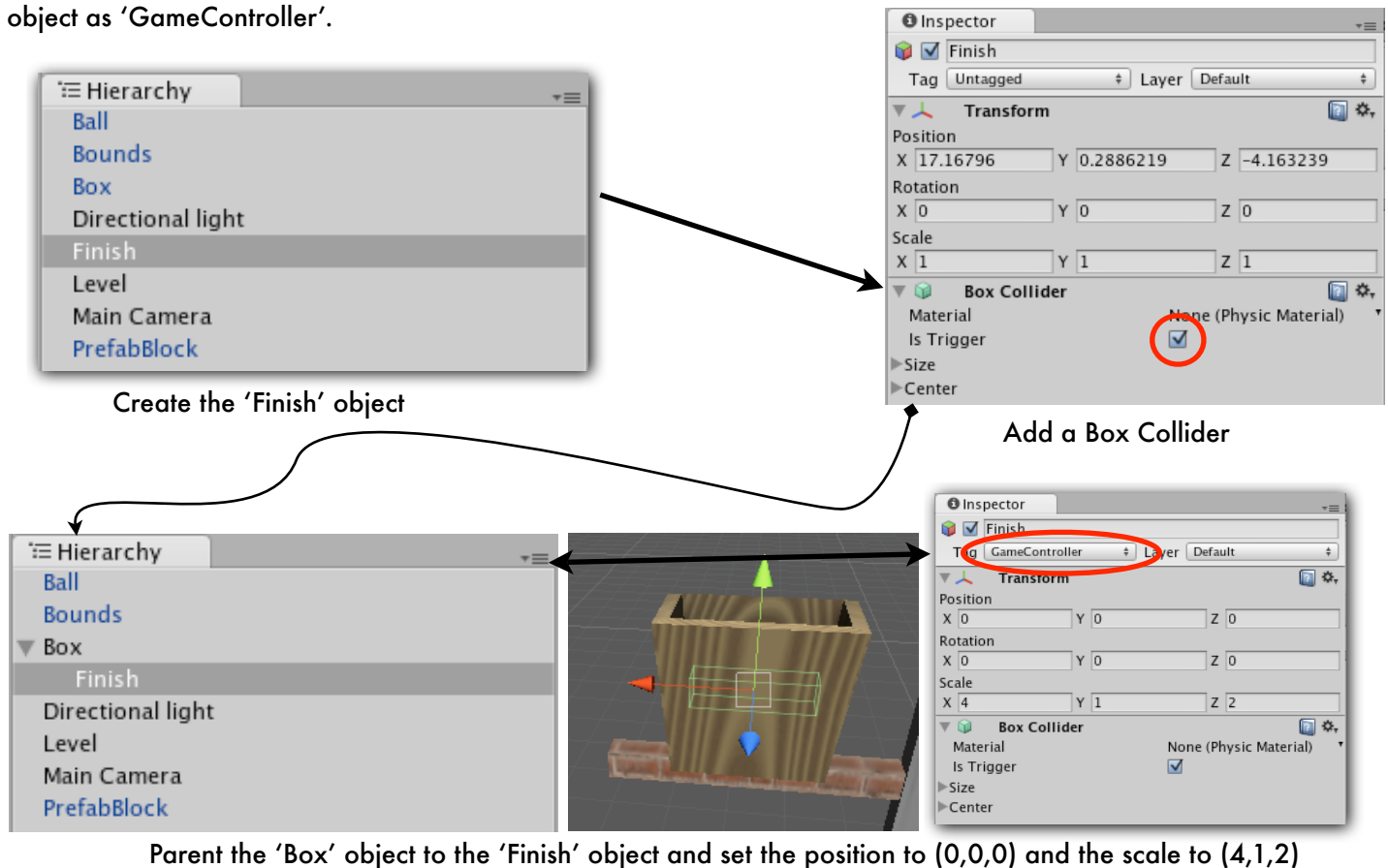
Let's put our box into the game. Drag out the 'Box' object and place it somewhere in the play area. Remember to make sure the Z position is at 0. Also, set the Z scale to 0.5. The 'Box' is as long as it is wide, so if you don't set the Z scale to 0.5, it will be colliding with the box collider that you put on the 'Level' object. If you have not placed the box at the bottom of the play area, you will probably want to drag out a 'PrefabBlock' and place it beneath your box, since this game is supposed to be realistic (Har har har) and we don't want things floating around. You also may want to place the Ball somewhere other than the middle of the screen. I've included a screenshot of my setup and inspector settings.



Back to the scripts

Alright, so we've got the ball, we've got the box, we've got the ball in the box. Now what? There's a series of steps here, so I've got some numbered screenshots for you to take a look at.

First thing to do is create an empty gameobject (GameObject > Create Empty). Call your new gameobject 'Finish'. Got that? Great. Now add a **BoxCollider**, and set the 'isTrigger' variable to be true. Now what you're going to do is drag the 'Finish' object onto the 'Box' object. Now, when you move the 'Box' object, the 'Finish' object moves with it. Set the position of the 'Finish' object to (0,0,0). By doing this, you will make sure that the 'Finish' object is always inside the 'Box' object. Also, set the scale to (4,1,2) as shown in the screenshots. Note that I've also tagged the 'Finish' object as 'GameController'.



Now what was the point of that? We have to have a method of determining if the ball is in the box. We can't use the collider of the 'Box' because if we did that, then the player would win if the ball bounced off the outside of the box. So instead, we added a new collider, made it a trigger (so the other colliders would ignore it) and placed inside the box. Now we need to create a script that will respond to the trigger events sent by our new collider. Add a script to your 'Scripts' folder and call it GameController. Add it to the 'Finish' gameobject you created (The one with the trigger). Now before we move on to scripting, let's create a prefab for our 'Box' object. After all, we don't want to go through this setup mess for every level we make.

1. Create a new prefab and name it 'Finish'. (Assets > Create > Prefab)
2. Place it in your 'Objects' folder.
3. Drag the 'Box' object from your 'Hierarchy' pane onto the prefab you just created.

Triggers

Triggers are basically colliders that send messages, but don't collide. When another collider enters a trigger, the gameobject that the trigger is attached to receives a `OnTriggerEnter()` message. For a detailed explanation of triggers, visit the [Blurst technology blog](#).

We're done in Unity for now, so open up your 'GameController' script and define some variables.

```
var nextLevel : String;
var loadDelay : float;
var winObject : GameObject;
private var hasWon : boolean;
```

To understand what the first variable is for, you need look no further than [Application.LoadLevel\(\)](#). What this nifty little function does is... load a level! The level it loads is based on the string you feed it. Which is why I also told you to name your scene 'Level'. Otherwise, your level wouldn't load. Of course, you're smart enough to figure out why and fix it, but consistency is awesome. Moving on, the purpose of the 'nextLevel' variable is to provide a level for *Application* to load when the player wins. More on that later.

'loadDelay' is just a variable to determine how long it takes for us to load the level after the player wins. This is in case you want to play some fireworks or something. Without this variable, the next level would start to load instantly, and there would be no fireworks.

'winObject' is once again for the fireworks. If you assign this variable, the script we're writing will create a copy of the gameobject you assigned.

The private variable, 'hasWon' is a boolean that determines if we've already one. This is to prevent code from being executed twice.

Alright, we've got our variables, let's use them.

```
function OnTriggerEnter(col : Collider){
    if(col.gameObject.CompareTag("Player")){
        if(!hasWon) Win(); //don't call Win() if we've already won!
    }
}

function Win(){
    hasWon = true; //notify the rest of the class
    /*Create a winObject if it exists.*/
    if(winObject) Instantiate(winObject, transform.position, Quaternion.identity.
    yield WaitForSeconds(loadDelay);
    Application.LoadLevel(nextLevel);
}
```

There are a couple new methods here that are pretty self-explanatory, but it boosts my ego to explain things, so:

1. *Instantiate()* creates a copy of the gameobject you pass in(*winObject*), at the position you pass in(*Vector3.zero*), at the rotation you pass in(*Quaternion.Identity*).
2. *WaitForSeconds()* delays execution of the function it's in until the seconds have passed. Note that it can't be run in the *Update()* methods because you can't delay execution of *Update()*.

So what we're doing in the *OnTriggerEnter()* function is:

1. Checking if the ball(the gameobject tagged 'Player') has collided with us
2. If it has, and we haven't already won, call the *Win()* function.

And in the *Win()* function:

1. Telling the rest of the class we've won, so we don't get called again
2. Creating the fireworks(or whatever you put in there)

3. Waiting a while, so the fireworks can explode.
4. Loading the next level based on the string that we have.

Pretty simple, huh? Now before we go back to Unity, let's go ahead and add a GUI button. The purpose of this button is to restart the game in case the ball gets stuck.

We will accomplish using the Unity GUI system. Add some code to your file:

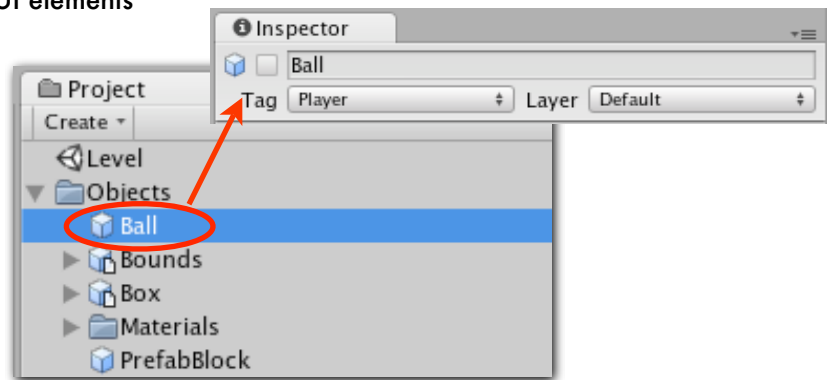
```
function OnGUI(){
    if(GUILayout.Button("Restart")){
        Restart();
    }
}

function Restart(){
    Application.LoadLevel(Application.loadedLevelName);
}
```

The `OnGUI` function is a method that you override to place GUI elements. `GUILayout` is a class that creates GUI elements and assigns their position automatically based on the string that you pass in. I won't be explaining the GUI system in depth here - if you want to read more about that, visit [the Unity site](http://unity3d.com/learn/tutorials/topics/gui).

We're finished with scripting for now - save your file and get back to Unity.

Now there are a couple things we need to do before we can run our game and see the level loading in action. The first thing to do is select the 'Ball' prefab, and in the inspector, give it the tag 'Player'. The copy of the ball that



is in your scene should update to reflect your changes. Now at the moment, the game is kind of boring, because the ball moves pretty slowly. We're going to add a script that will allow the player to give the ball a "kick" so to speak. It will also constrain the Z position of the ball to keep it from going off into outer space. So, create a new script, and name it 'Ball'. Add this code:

```
var jumpMultiplier : float = 20;
var maxVelocity : float = 20.0;
private var collisionCount : int = 0;

function FixedUpdate () {
    transform.position.z = 0;
    if(Input.GetButton("Jump") && (collisionCount > 0) && (rigidbody.velocity.y < maxVelocity)){
        rigidbody.AddForce(Vector3.up * jumpMultiplier, ForceMode.VelocityChange);
    }
}
```

```
function OnCollisionEnter(col : Collision){
    collisionCount++;
}

function OnCollisionExit(col : Collision){
    collisionCount--;
}
```

The 'jumpMultiplier' variable determines how high the ball will go when the user presses the space bar. The 'maxVelocity' variable will make sure that we don't get the ball going so fast that it goes through things.

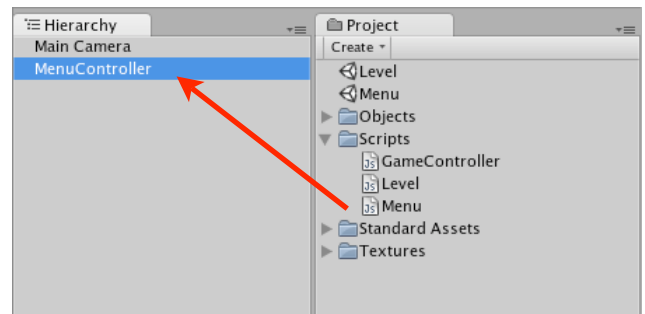
The purpose of the 'collisionCount' variable is to make sure that the player cannot jump without the ball sitting on a collider. Without the 'collisionCount' the player would be able to fly - which would be kind of cool, but...

Make sure you save your scene before continuing.

Creating a Menu

Now what we're going to do is create a main menu for our game. This will consist of adding some text and a couple of buttons. Create a new scene - call it 'Menu'. Create an empty gameobject and call it 'MenuController'. Add a new script to your 'Scripts' folder and call it 'Menu'. Add the 'Menu' script to your 'MenuController' object.

Add this code:



```
var firstLevel : String; //1

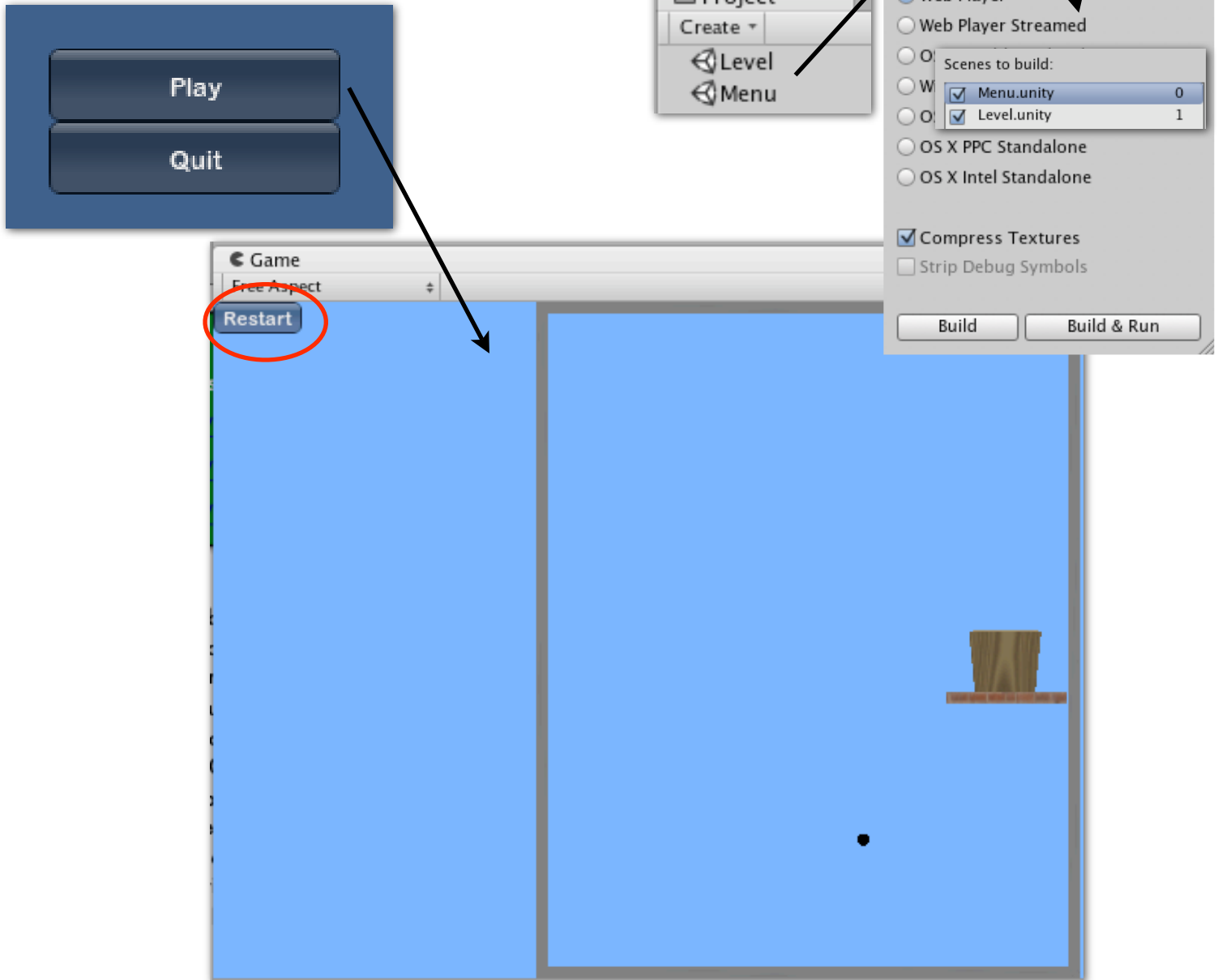
function OnGUI () {
    if(GUI.Button(Rect(Screen.width / 2 - 64, Screen.height / 2, 128, 32), "Play")){ //2
        Application.LoadLevel(firstLevel); //3
    }
    if(GUI.Button(Rect(Screen.width / 2 - 64, Screen.height / 2 + 32, 128, 32), "Quit")){ //4
        Application.Quit(); //5
    }
}
```

If it's a bit impenetrable, don't worry. We'll walk through it line by line.

1. This variable is used to determine what level to load when the player presses the 'Play' button.
2. This line will create a button labeled "Play", and tell us whether or not the button is being clicked by the player.
3. If the "Play" button is pressed, we load the first level(based on the variable we defined earlier).
4. 5. Create a button labeled "Quit" and determine whether or not it is pressed.
5. If it is pressed, then call the Application function *Quit()*.

We are now finished with the scripting side of things. Switch back to Unity and run your game. You should see your buttons. At this point, you will probably get some kind of warning saying that 'the level can't be loaded because

it isn't added to the build settings'. Let's fix that. Go to the menu bar and click "File > Build Settings". A dialog similar to the one at the right should pop up. Drag your two scenes ("Menu" and "Level") into the 'Scenes to build' box. Make sure that the "Menu" level is at index 0 and the "Level" scene is at index 1. At this point, you probably still can't load the level because you haven't told the 'MenuController' object which level to load. Select the 'MenuController' object and in the Unity Inspector, look for the 'Menu' script. For the 'firstLevel' variable, type in "Level"(the name of your ball/box scene).



Now, you should be able to run your game and load the level. Click the 'Play' button and the level loads. Also note that you can restart the game with the 'Restart' button.

Back to the menu. Create a new **GUI Text** object (GameObject > Create > GUI Text). Put your game name in there, and position it using the 'Pixel Offset' variables as you see fit. Now if you look at your new **GUI Text** in the 'Game' pane, it looks pretty small. To make it bigger, look in the 'Project' pane. In the 'Standard Assets' folder, there is a 'Fonts' folder. Inside this folder, you will see a font called 'Courier'. Drag this font onto the 'Font' variable of the **GUI Text** you've created. Then, select the 'Courier' font back in the 'Project' pane. Look at the "Import Settings" in the Unity inspector, and change the font size to a larger value. When you're done, be sure to click the 'Apply' button. I've set my font to size '48'.

In the GUI Text settings, I've set the 'Pixel Offset X' to -160 and the 'Pixel Offset Y' to 80.

You also might want to change the background color of the camera.

Now we will add some instructions for the player. Create another GUI Text, and set its text to something similar to:

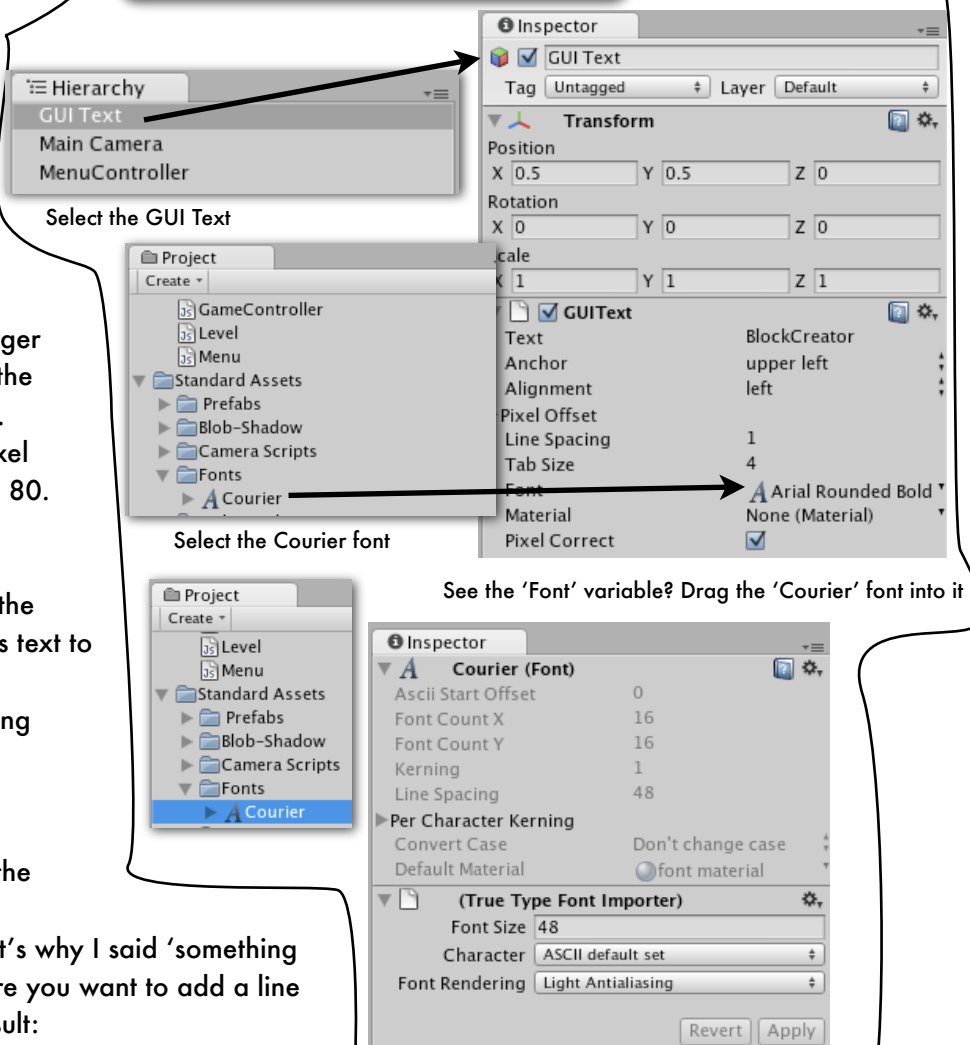
"Try to get the ball in the box by creating ledges.

Create ledges by dragging on the background.

If your ball stops moving, you can use the space bar to jump".

Obviously, that's pretty boring, but that's why I said 'something similar'. To add the line breaks, select where you want to add a line break and hit 'Option(Alt) + Enter'. The result:

If you don't care to use the 'Courier' font, you can just drop in your own font and follow the steps.



Polishing it off

There are a couple more things in the "Level" scene that we need to fix. Save the "Menu" scene and open up the "Level" scene. The first thing to do is make the level winnable. Select the 'Finish' object(it's a child of the Box), and in the inspector, set the 'Next Level' variable to "Menu". This will result in the game loading the menu when the player wins. You can also create a prefab to use for the 'winObject'. I've used the 'explosion' particle emitter from the 'Standard Assets' folder. You can also set the 'loadDelay' variable.

Fixing the Bugs

I congratulate you! You have come far. Again. What you have here could be the start of a game. Add some decent art, add lots of cool levels, powerups, and physic puzzles, figure out a way to make the ball move faster, and... and... and...

Well, I won't help you out with that stuff, but before we close this tutorial, we should probably fix some of the bugs in the system. In case you haven't discovered them yourself, a couple of the big ones are:

1. Dragging right to left rather than left to right doesn't do anything
2. Dragging outside of the play area makes our line go wacky.

The reason for the first is that dragging from right to left messes up our *CreateBox()* function, and so the triangles are drawn backwards. The second problem is due to us using raycasting to capture the mouse position. When you drag outside the play area, the raycast doesn't hit any colliders, so we can't get the mouse position.

Fixing the first problem is fairly simple. Open up the "Level" script and navigate to the *OnMouseDown()* function. The current code simply puts the 'mouseHit' variable in slot 1, and the *MousePoint()* vector into slot 2. To fix our bug, we will change this function to put the variable with the smallest x position into slot 1, and the variable with the largest x position into slot 2. Change your *OnMouseDown()* function to:

```
function OnMouseDown(){
    dragging = false;
    var currentMousePos : Vector3 = MousePoint();
    if(mouseHit.x > currentMousePos.x){
        CreateBox(mouseHit, currentMousePos);
    } else if(mouseHit.x < currentMousePos.x){
        CreateBox(currentMousePos, mouseHit);
    }
}
```

The second problem is a wee bit more complex, and requires the use of another variable. Back at the top of your file, where you defined your private variables, add another:

```
private var lastMousePosition : Vector3; !
```

This variable will store the position of the mouse the last time we could get it. So when the mouse moves out of the playing area, this variable will store the position it was at before it moved out of the area.

```
function MousePoint() : Vector3 {
    var ray = GameObject.FindWithTag("MainCamera").camera.ScreenPointToRay(Input.mousePosition);
    var hit : RaycastHit;
```

```

    if(Physics.Raycast(ray, hit, 100, kDefaultRaycastLayers)){
        lastMousePosition = hit.point;
    }
    return lastMousePosition;
}

```

The code logic:

1. If we can cast against the collider, update the mouse position.
2. Otherwise, do nothing.
3. Return the mouse position

Extending the Game

Don't worry, this tutorial is nearly over. But before we end, I should mention the topic of extensibility. Currently, the scripts we have created are pretty general purpose and reusable throughout the game. To create a level, all you have to do is drag the 'Ball' prefab into your scene, create a gameobject with your 'Level' script and a collider of some sort, and add drop the 'Box' prefab somewhere in the scene. Easy enough, right? Right.

However, one thing that is not easily workable with our setup is level layouts that don't fit inside the current camera viewport. So we're going to do one last thing here: write a camera script for our game. This script will enable our camera to zoom in and out and track the ball. Note that unless you make your playing area bigger, this script will have no purpose in the level you've already created, because you can already see everything. I will not walk you through the creation of a new level, so if you're going to use this script, you will have to create your own level that is big enough to warrant the usage of this script. Alternatively, you can download the example project (check page 23 of this tutorial), which includes a larger level.

Let's get started. You know the drill: Create a new script (Assets > Create > JavaScript) and put it in your scripts folder. I named my script "SmoothFollowXY" to reflect the 2D nature of the script. The first thing I do is define some variables:

```

var target : Transform;
var speed : float = 3.0; //how fast should we move to catch up with the target
var zoomSpeed : float = 1.5; //how fast is zooming
var maxZoom : float = 1.0;
var minZoom : float = 35.0;
var threshold : float = 20.0;
private var wantedPosition : Vector3;

```

How will this script work? What we'll do is track the position of the target. Zoom will be controlled by the scroll wheel, and to accomplish it, we will edit the 'Orthographic Size' variable of the camera.

Now for the actual code:

```

function LateUpdate () {
    if(!target) return;
    camera.orthographicSize -= Input.GetAxis("Mouse ScrollWheel") * zoomSpeed; //read the input
    from the scrollwheel and multiply it by the zoomSpeed variable.
    camera.orthographicSize = Mathf.Max(camera.orthographicSize, maxZoom);
    camera.orthographicSize = Mathf.Min(camera.orthographicSize, minZoom);
}

```



```

wantedPosition = target.position;
wantedPosition.z = transform.position.z; //Make sure we're not messing with the z-axis...

        if(Vector3.Distance(transform.position, wantedPosition) > threshold){ //if the
target moved out of range

                transform.position = Vector3.Slerp(transform.position,
wantedPosition, speed * Time.deltaTime);

        }
}

```

In the first line, we make sure we have a target. If we don't, then we simply return without doing anything. After all, if there's no target, then it would be kind of useless to execute our code. The next three lines handle zooming. We read the input from the scrollwheel and multiply it by our 'zoomSpeed' variable. The `Mathf.Max()` and `Mathf.Min()` return the larger and smaller of two values, respectively. This will prevent the camera's orthographic size variable from getting above or below the max and min zoom variables.

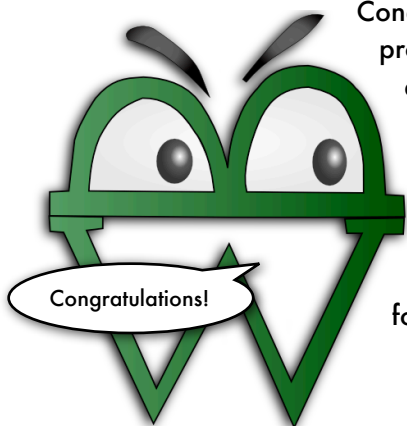
It is customary to use *LateUpdate()* in camera scripts because, as the name implies, it is executed after *Update()*.

The next two lines will calculate the position we're trying to get to. We perform the operation because we don't want to change the Z position of the camera.

The last line checks to see if the distance between us and our target is bigger than the threshold, and, if it is, we smoothly move the camera toward the target. Note the multiplication by `Time.deltaTime`. Without this multiplication, our movement would be based on how fast the player's computer is, rather than time. For a more detailed explanation, take a look at the [Unity docs](#).

Let's try it out! Save your scene and hop back over to Unity. Place the script on your 'Main Camera' object, set the 'Ball' object as the target, and run the game. You'll have to do a bit of toying with the variables in the Unity inspector, and you probably won't even want to use the camera script for this level, but you now have a script that you can use on a bigger level.

The End



Congratulations! You have reached the end of the first BinkWorks Unity3D tutorial. Our project is downloadable from <http://www.binkworks.com/files/BlockCreator.zip>. The example project includes the complete and commented source files, art, and scenes. It also includes a couple of extra levels, to give you something of an idea of what a full-blown game might be like.

This being the first edition of a first tutorial, it is probably full of mistakes, typos, and unclarities. If you are so unfortunate as to discover one of the above, please email me at lincolngreen@binkworks.com and let me know. Thanks go out to the Unity community, for their help in the writing of this piece.