
makeuseof
programming a
game with
unity
a beginner's guide
by Andre Infante

by Andre Infante
http://www.petafl0p.com/

Published February 2014

Table Of Contents

1. Introduction

A surprising feature of the Internet economy is the rise of indie videogames. Once the exclusive domain of thousand-man, multi-million dollar triple-A studios, a number of toolsets have been developed that bring modern game development resources into the hands of individuals or small, ad-hoc collections of programmers and designers. These indie game development teams have demonstrated an agility and risk-tolerance that, in many cases, allows them to push gameplay innovation faster than their big budget counterparts. A number of shockingly successful indie titles have premiered in recent years, including Minecraft, Limbo, and Super Meat Boy.

In the rapidly evolving landscape of indie game development, Unity has emerged as something of a de-facto standard: its low cost, ease of use, and broad feature set make it ideal for rapid game development. Even large studios such as CCP (Developers of Eve Online) use it for rapidly prototyping game concepts. Unity provides a "game engine in a box" - a physics and rendering engine with hooks for several scripting languages, adaptable to virtually any genre of videogame.

While Unity does provide a visual editor for manipulating the game environment, Unity is not a 'zero programming game creator' tool. Unity requires the ability to program to produce results, but also gives you a much more flexible and powerful tool than any 'game maker' program possibly could. Unity won't do the work for you, but it does serve to lower the barrier to entry substantially. Starting completely from scratch with C++ and OpenGL, it can take days to get to the point where there's actually something rendered onscreen. Using Unity, it takes about ten seconds. Unity puts the basic elements of game creation into the hands of novice programmers in a fast, intuitive way.

2. Versions of Unity

Unity comes in two basic flavors: the pro version and the free version. There are a number of differences (you can see the full list here), but, broadly speaking, the pro version supports a number of visual improvements (like real-time soft shadows and post-processing), and a large number of relatively minor features that are extremely helpful for more complex games. That said, for most relatively simple games you might want to build, the free version of Unity is perfectly adequate. We'll break down the key differences below in more detail for those interested.

2.1 Pricing

The free version of Unity is, of course, free. However, there are a few limitations: the free version of Unity cannot be licensed to any company with an annual income of more than $100,000. While such organizations are beyond the scope of this guide, if you suspect you might become such an organization, it's probably wise to spring for the Pro version.

The Pro version of Unity is $75 a month, or $1500 for a permanent license, and has no limits on what you can do with the games created with it. There is also a 30-day free trial available, which we'll be using for this guide, in order to give you as complete an overview of the available features as possible. A one-year student license is also available through Studica for $129.

2.2 Features

There are many features absent in the free version of Unity. However, the most important differences are as follows: the free version of Unity lacks a number of rendering options that allow for better-looking, faster-running games (LOD support, screen-space post-processing, advanced shaders, real-time soft shadows, and deferred rendering). It also lacks the full mechanim animation system, and some AI tools. In general, for complex, large-scale projects, or projects where graphical performance is important, the pro version is worthwhile. I use the pro version, because I develop virtual reality games for the Oculus Rift, and the screen-space post-processing support is necessary to correctly interact with the headset.

You can check out an early alpha build of one of my VR games, BeatRunner. It should give you a sense for what Unity makes possible.

3. Installing Unity

Unity is straightforward to install. You can download the executable here (for the OSX installer, click the link that says 'developing on Mac OSX?'). Let it download, run it, and follow the installer instructions. When the installation is finished, a window entitled 'activate your Unity license' will appear. Check the box marked 'activate a free 30-day trial of Unity Pro' and click 'OK.'

Congratulations! You know have a 30-day trial of Unity Pro. When the trial expires, if you don't want to buy the pro version, you can switch to the free version and keep your existing content.

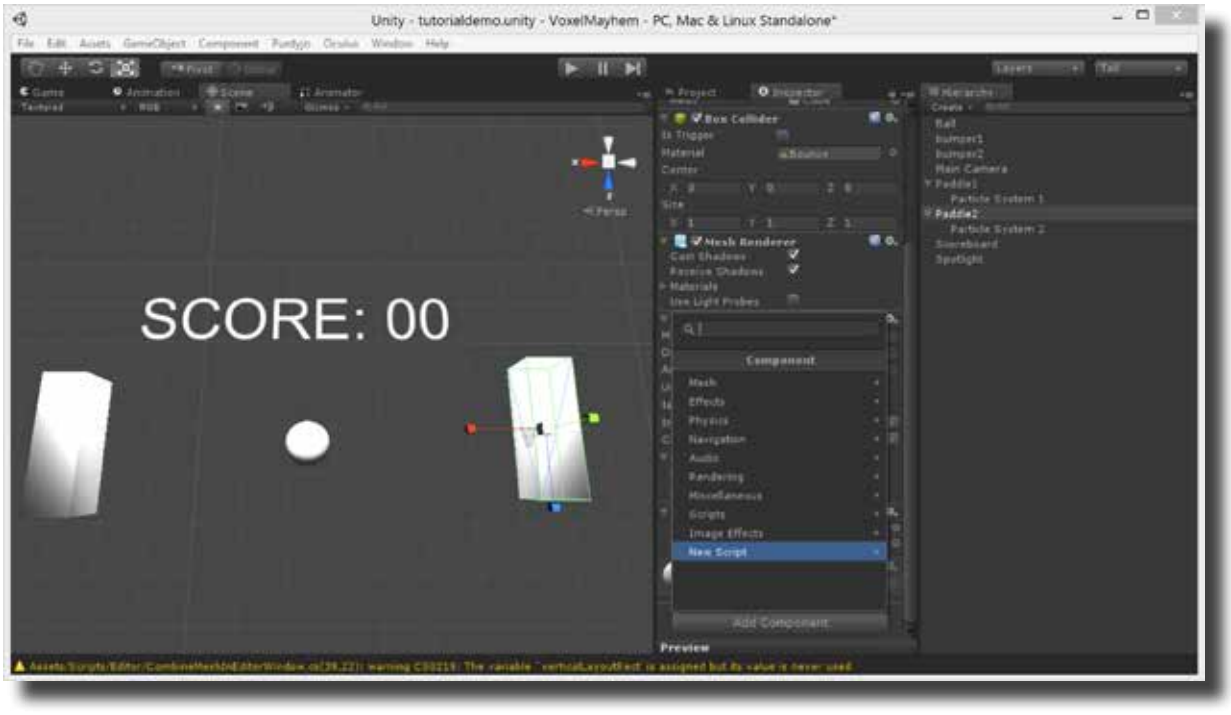
4. Introduction to the Object-Oriented Paradigm

Before we get started with Unity, it's important that we go over the basics a little. Unity supports both C# and Javascript for game programming; we'll be working with C# for this tutorial. First off, if you've never programmed before, put this tutorial aside and spend a few days working through Microsoft's C# Language Primer until you feel comfortable using the language for simple tasks. If you have programmed before in an imperative or object oriented language like C or Java, skim the primer and familiarize yourself with how C# differs from other languages you've used in the past. Either way, don't proceed with the tutorial until you feel comfortable solving simple problems with C# (for example, if I were to ask you to write a program that prints the first hundred prime numbers, you should be able to write that program without consulting Google).

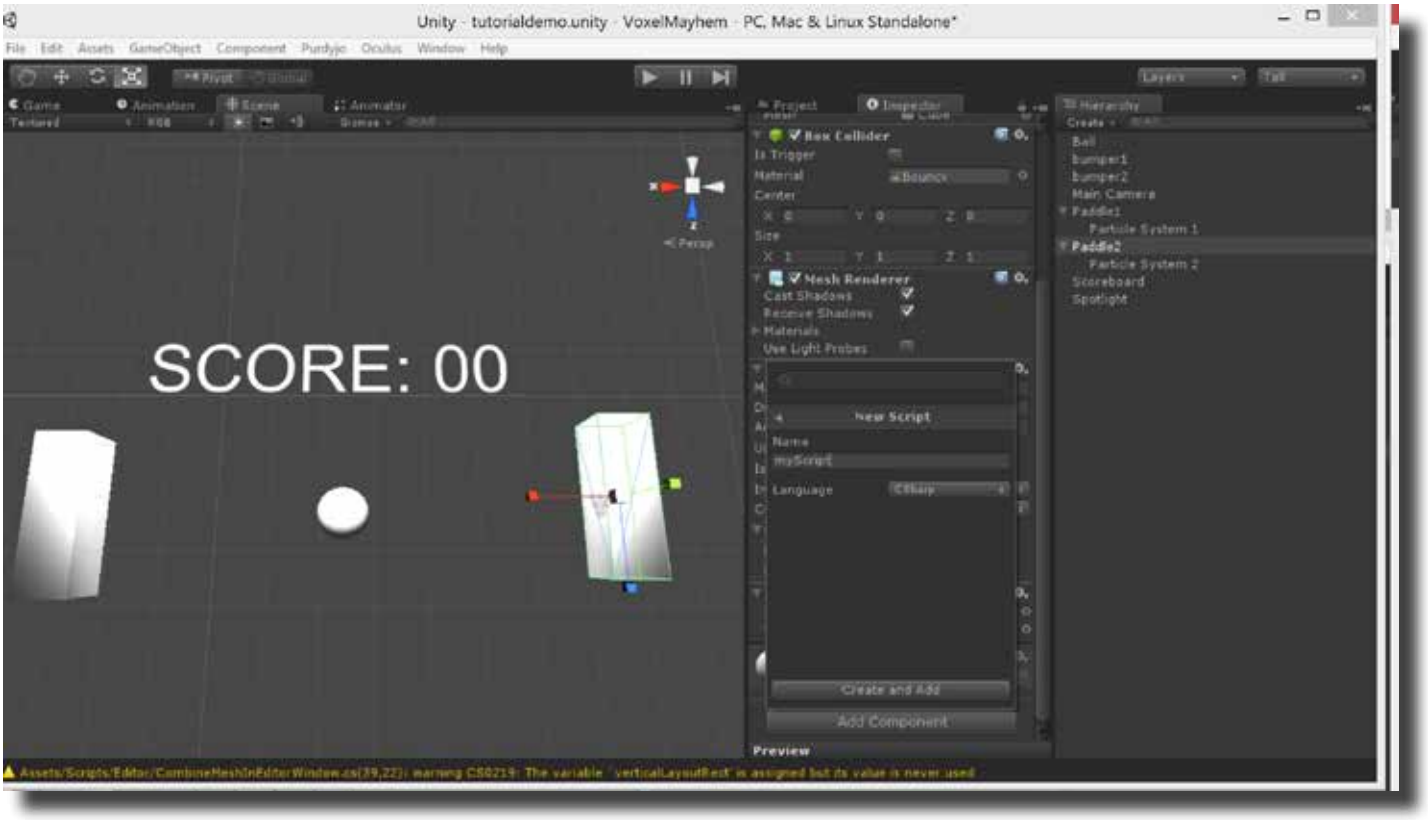
The most important concept to understand here is the object-oriented paradigm (abbreviated as OOP). In object oriented languages, programs are divided into functional units called Objects. Each object has its own private variables and functions. Object-specific functions are called methods. The idea here is modularity: by having each object isolated, and forcing other objects to interact with it through its methods, you can reduce the number of possible unintentional interactions - and, by extension, bugs. You also create objects you can reuse at will later with no modification. In Unity, you'll be building these objects and attaching them to game entities (whose behavior they'll govern).

Objects are instantiated from classes: a class is just a file that lays out the definition of your object. So, if you want a 'Mook' object that handles AI for an enemy in your game, you'd write a 'Mook' class, and then attach that file to every enemy entity. When you run your game, each enemy will be equipped with a copy of the 'Mook' object.

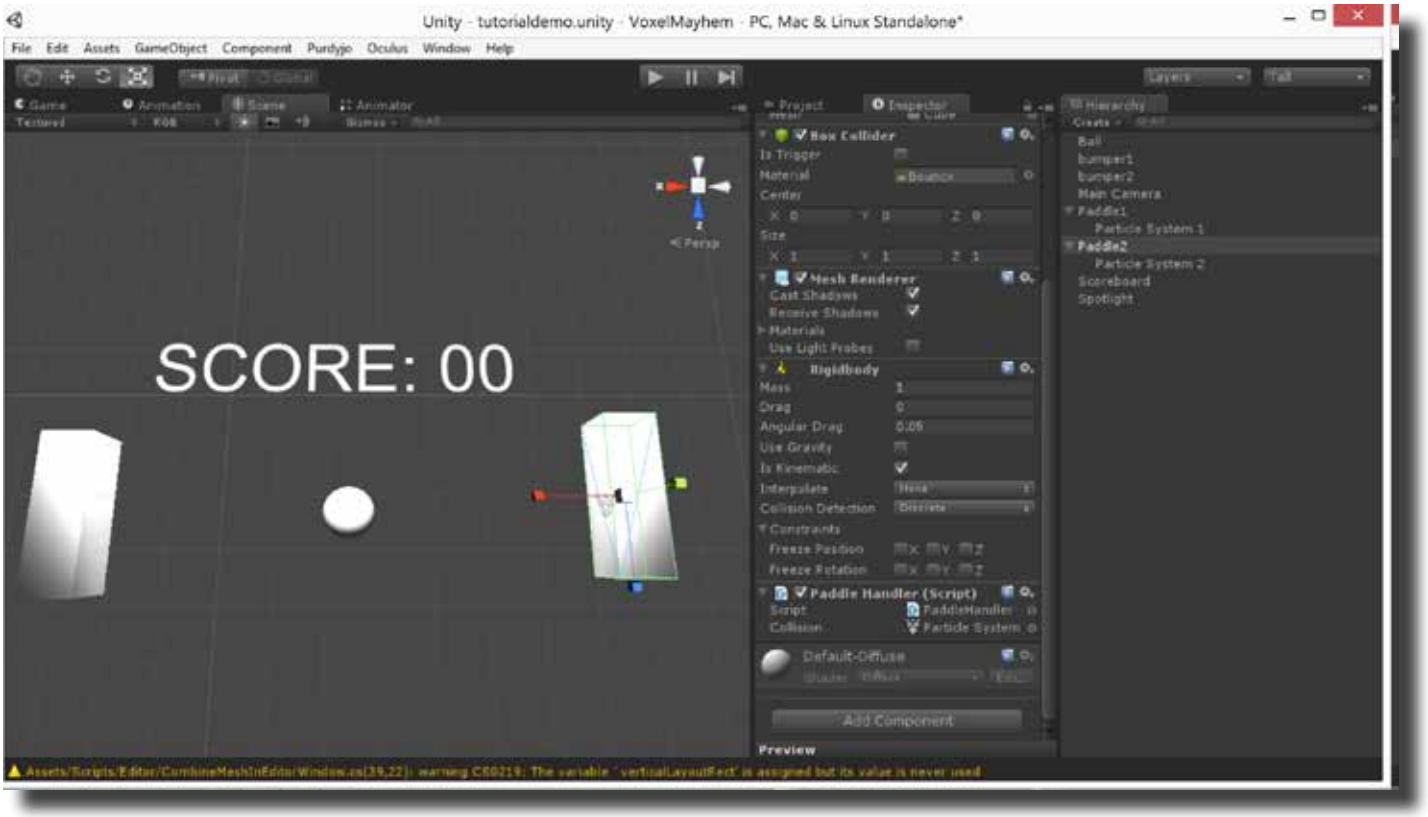
Attaching a new script to an object looks like this:



First, select the object and go to the inspector. Then, click on the 'Add Component' button.



Go to 'new script,' enter the name you want, and click 'create and add.'



Now you have a new script, that you can edit by double-clicking on it!

A class file looks something like this:

```
using UnityEngine; public class Mook : MonoBehaviour { private float health;  void Start () { health = 100;  } void Update(){ if (health > 0){ //search for player //if you encounter the player on the road, kill him //if you get shot, remove a random amount of health } } }
```

Let's break this down:

- *using UnityEngine; -This line tells C# that we want to use Unity's libraries, which allow us to connect to the Unity game engine.*
- *public class Mook :* MonoBehaviour *{ -This line actually declared the class and its name ("Mook");*
- *private float health; -This declares a private class variable (which can only be changed from inside the class). The variable is given a value in Start().*
- *void Start () { -This declares a method called 'Start.' Start is a special method that runs only once, when the game initially launches.*
- *void Update(){ -Update is another special method, which runs on every frame. Most of your game logic will go here.*
- *//if you encounter the player on the road, kill him -This line is a comment (any line starting with a double slash is ignored by C#). Comments are used to remind yourself of what particular bits of code do. In this case, this comment is being used to stand in for a more complicated block of code that actually does what the comment describes.*

Along with 'Start' and 'Update,' you can instantiate your own methods with almost any name. However, methods that you create won't run unless they're called. Let's declare a method for a hypothetical class called myClass that adds two numbers together.

```
public float addTwoNumbers(float a, float b){ return a+b; }
```

This declares a public (accessible to other objects) method that returns a float, called "addTwoNumbers," which takes two floats as input (called a and b). It then returns the sum of the two values as its output.

Calling this method from within the same class (say, from inside Update) looks like this:

```
float result = addTwoNumbers(1,2);
```

Calling the method from another class is similar:

```
 myClass instance; float result = instance.addTwoNumbers(1, 2);
```

Again, this just creates an instance of our class, accesses the appropriate method and feeds it the numbers we want to add, then stores the result in 'result.' Simple.

If your script is attached to an object that has special properties (like a particle emitter) that can't be accessed under the normal set of GameObject parameters, you can choose to treat it as a different kind of game entity by using the GetComponent method.

The syntax for that looks like this:

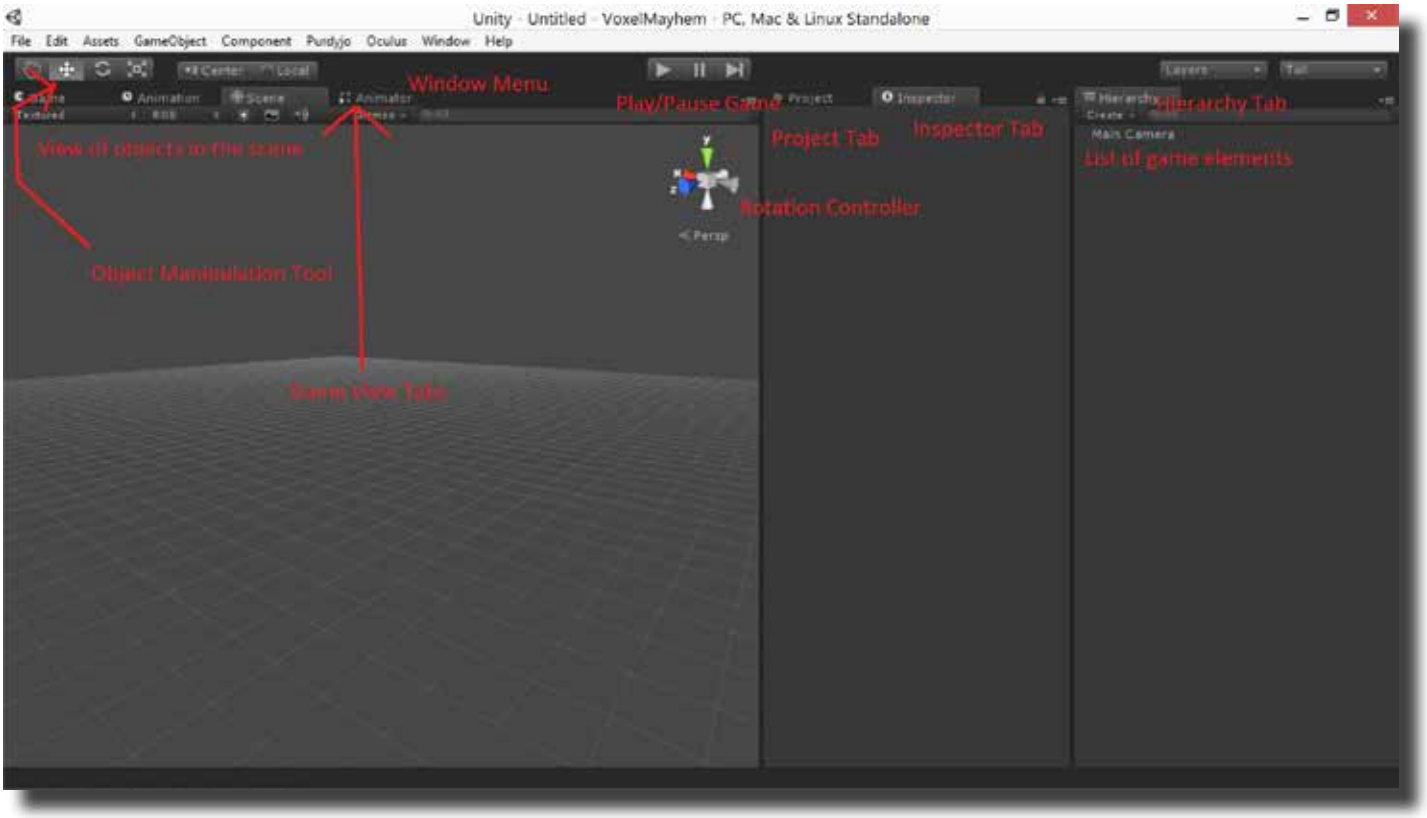
```
 GetComponent<ParticleSystem>().Play();
```

If any of this is unfamiliar to you, go back and go through the C# primer. It'll save you a lot of frustration as we proceed.

5. Unity Basics

In this section, we're going to work our way through the basic mechanics of the Unity engine. The workflow in Unity goes something like this: create an entity to serve a role in the game (blank GameObjects can be used for abstract logical tasks). Then, either write or find a class file, and add it to the entity as a script (using the 'add component' button in the 'inspector' view. Then run, test, debug, repeat until it works and move on to the next element of the game.

Unity comes with a number of basic view tabs that can be laid out in various ways to the taste of the user. The big five are the 'game' tab, the 'scene' tab, the 'inspector' tab, the 'project' tab, and the 'hierarchy tab. The game tab, when the 'play' button is depressed, displays a running instance of the game that the user can interact with and test. The 'scene' tab provides a static, editable version of the gameworld. The 'inspector' tab allows the user to modify individual entities in the game world by selecting them in the 'editor' tab. The 'project' tab allows the user to browse through the project's files and drag models, materials, and other resources into the 'editor' tab to place them in the gameworld. Finally, the 'hierarchy' tab shows all objects in the world, allowing you to find distant objects in the scene, and parent entities to one another by clicking and dragging. See the diagram below for the locations of all these things.



5.1 Unity Entities

5.1.1 Meshes

Meshes are the way 3D geometry is represented in Unity. The user can either use Unity's built-in 'primitive' objects (cubes, spheres, cylinders, etc), or import their own 3D models from a modelling package like Blender or Maya. Unity supports a variety of 3D formats, including Collada (.fbx), and .3ds.

The basic tools for manipulating meshes are the scaling, rotation, and translation buttons in the upper left corner of the interface. These buttons add control icons to the models in the editor view, which can then be used to manipulate them in space. To alter the texture or physics properties of an object, select them and use the 'inspector' view to analyze the 'material' and 'rigidbody' elements.

5.1.2 GUI Elements

Traditional GUI sprites and text can be displayed using the 'GUI Text' and the 'GUI Texture' GameObjects in the editor.

However, a more robust and realistic way to handle UI elements is to use the 3D text and Quad GameObjects (with transparent textures and an unlit transparent shader) to place HUD elements into the gameworld as entities. In the 'hierarchy' view, these gameplay elements can be dragged onto the main camera to make them children, ensuring that they move and rotate with the camera.

GUI elements (text and textures) can have their size and scale adjusted using the relevant fields in the inspector tab.

5.1.3 Materials

Materials are combinations of textures and shaders, and can be dragged directly onto game objects from the project tab. A large number of shaders come with Unity Pro, and you can adjust the texture attached to them using the inspector tab for an object that they're applied to.

To import a texture, just convert it to a jpg, png, or bmp, and drag it into the 'assets' folder under the Unity project directory (which appears in 'My Documents' by default). After a few seconds, a loading bar will appear in the editor. When it finishes, you'll be able to find the image as a texture under the 'project' tab.

5.1.5 Lights

Lights are GameObjects which project radiance onto the world. If there are no lights in your scene, all polygons are drawn at the same brightness level, giving the world a 'flat' look.

Lights can be positioned, rotated, and have several internal characteristics that you can customize. The 'intensity' slider controls the brightness of the light, and the 'range' controls how quickly it fades out. The guidelines in the scene view show you the maximum range of the illumination. Play with both settings to achieve the desired effect. You can also adjust the color of the light, the pattern ("cookie" displayed on the surface the light is pointed at, and what kind of flare appears onscreen when looking directly at the light. The cookie can be used to fake more realistic light patterns, create dramatic false shadows, and simulate projectors.

The three main kinds of light are 'spot', 'point', and 'directional.' Spot lights have a location in 3D space and project light only in one direction in a cone of variable angle. These are good for flashlights, searchlights, and, in general, give you more precise control of lighting. Spot lights can cast shadows. Point lights have a location in 3D space, and cast light evenly in all directions. Point lights do not cast shadows. Directional lights, finally, are used to simulate sunlight: they project light in a direction as though from infinitely far away. Directional lights affect every object in the scene, and can produce shadows.

5.1.6 Particle Systems

Particle systems are the term for Unity GameObjects that generate and control hundreds or thousands of particles simultaneously. Particles are small, optimized 2D objects displayed in 3D space. Particle systems use simplified rendering and physics, but can display thousands of entities in real time without stuttering, making them ideal for smoke, fire, rain, sparks, magic effects, and more.

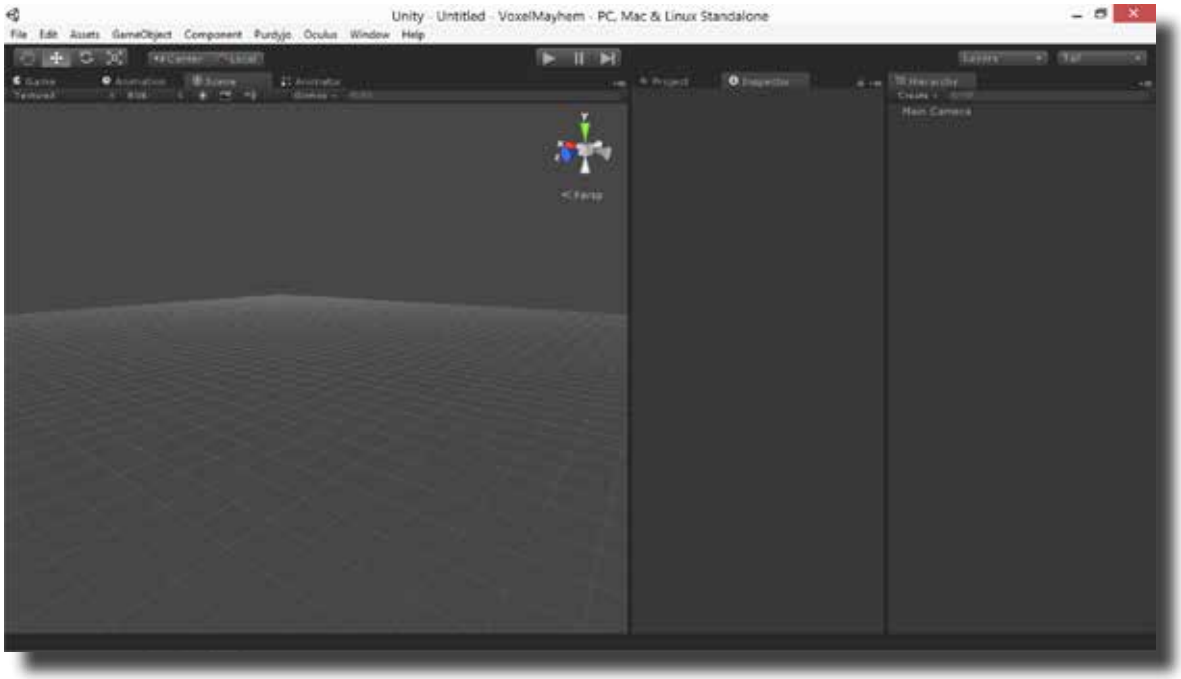
There are a lot of parameters that you can tweak to achieve these effects, and you can access them by spawning a particle system under the component editor, selecting the particle system, and then opening the 'inspector' tab. You can change the size, speed, direction, rotation, color, and texture of each particle, and set most of those parameters to change over time as well. Under the 'collision' attribute, if you enable it and set the simulation space to 'world' you'll get particles that will collide with objects in the world, which can be used for a number of realistic particle effects, including rain, moving water, and sparks.

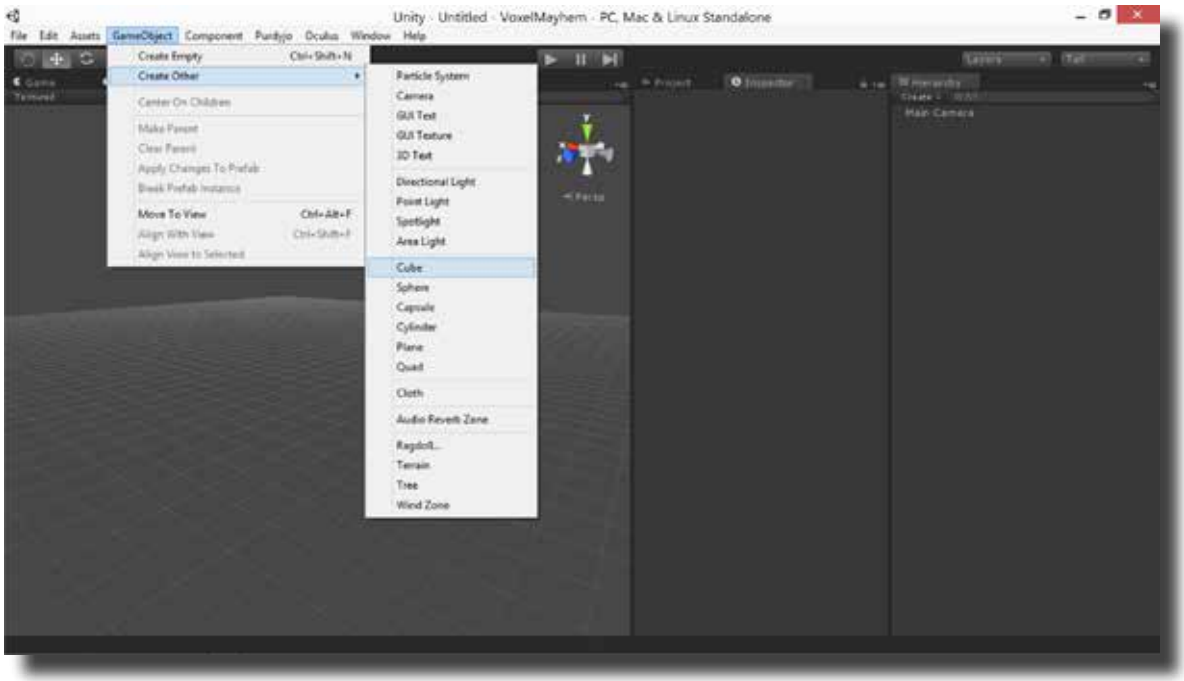
6. Example: Basic Elements of a Game

For this tutorial, we're going to make a simple game of Pong. In this section, we'll just go over arranging the core elements – the scripting tutorial will come later.

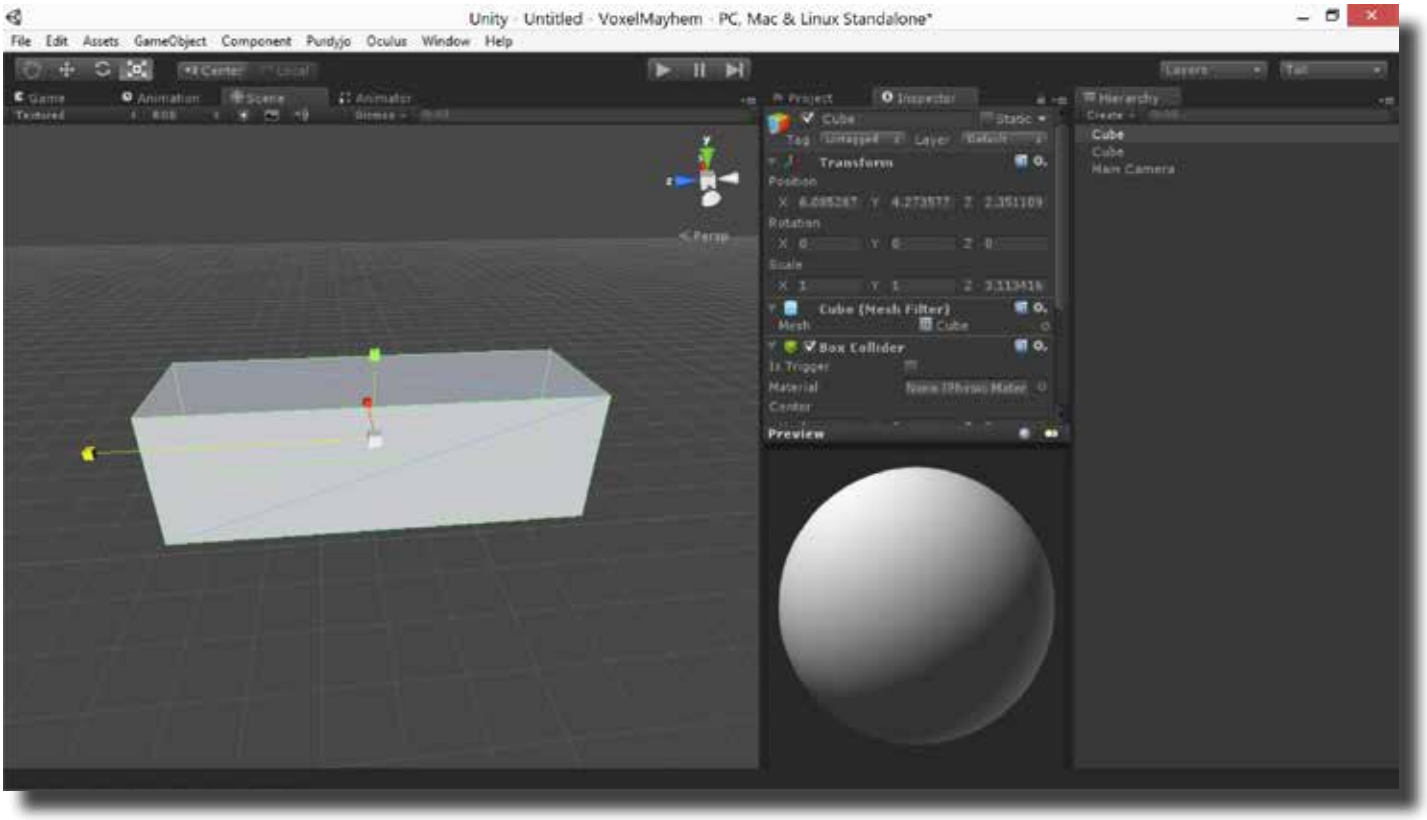
First, let's break down the game of Pong into its basic components. First, we need two paddles, and a ball. The ball flies offscreen, so we'll want a mechanism to reset it. We also want text to display the current score, and, for the sake of showing you all the core elements of Unity, we'll want a fancy particle effect when you hit the ball, and the whole game will need to be dramatically lit.

That breaks down into a ball object (a sphere), a spawner, two paddle props with particle emitters attached, a 3D-text entity, and a spot light. For this tutorial, we'll be using the default physic material 'bounce,' with bounce combine set to 'multiply'. Here's what the setup looks like, in ten screenshots:

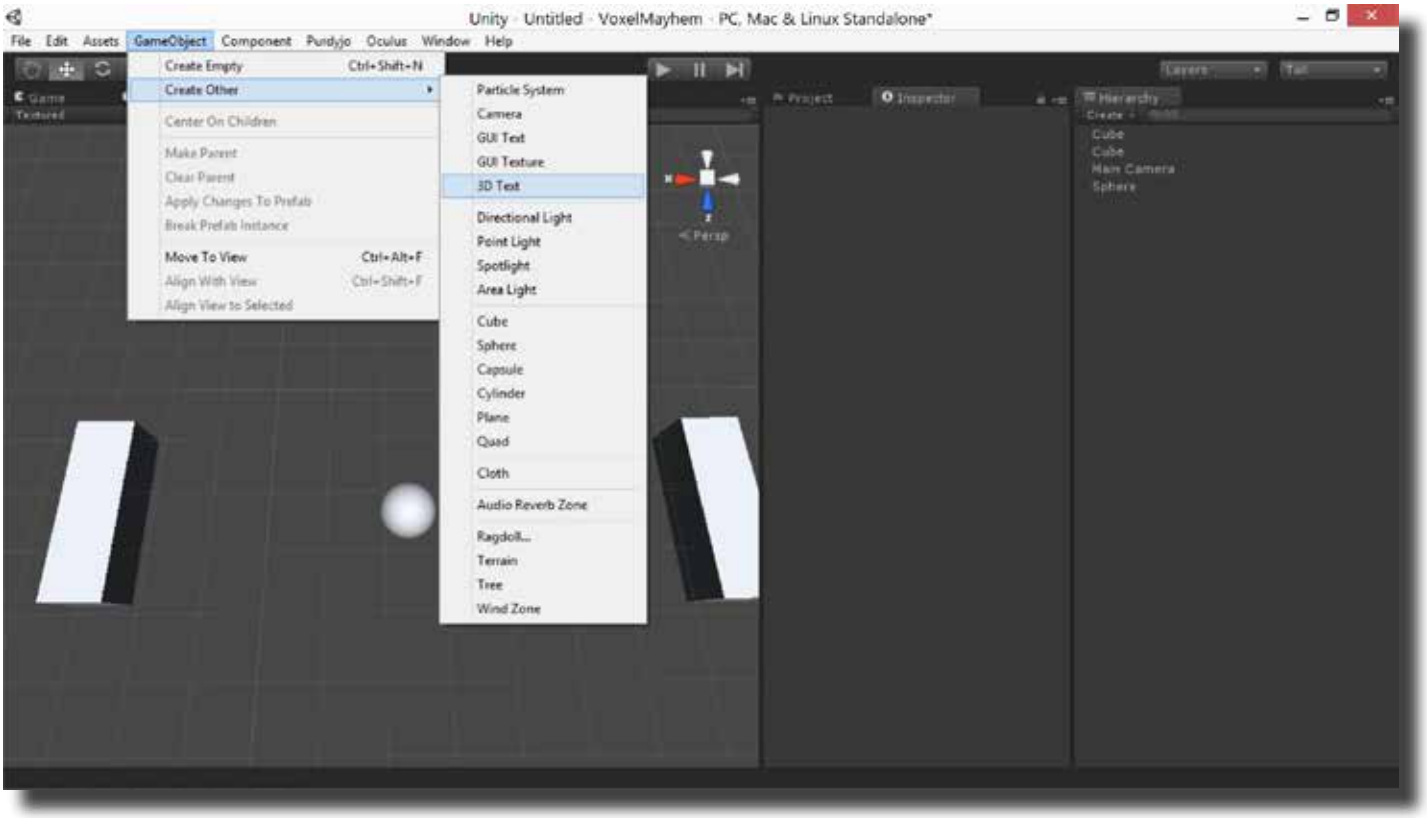




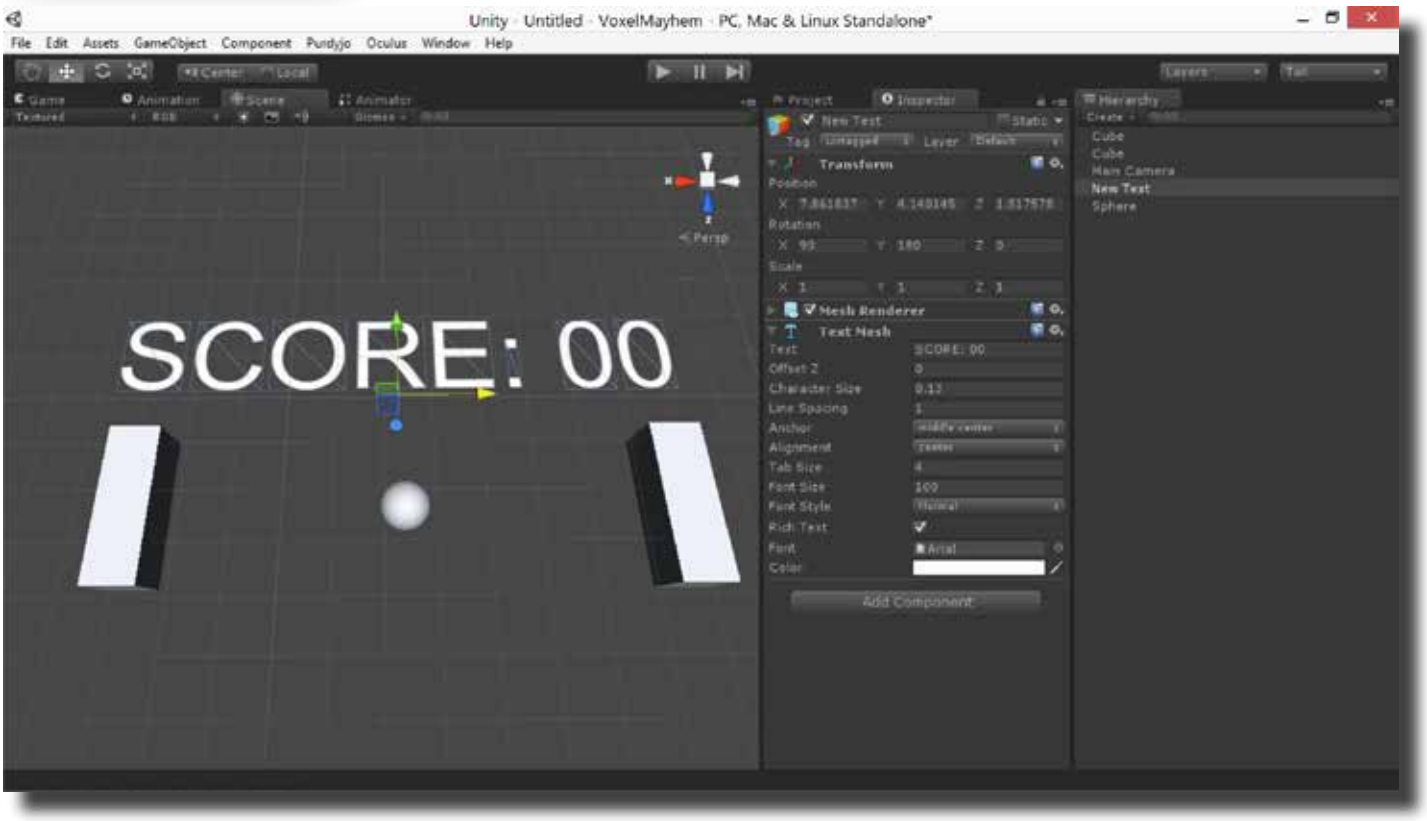
First, create a cube prop for the paddle.



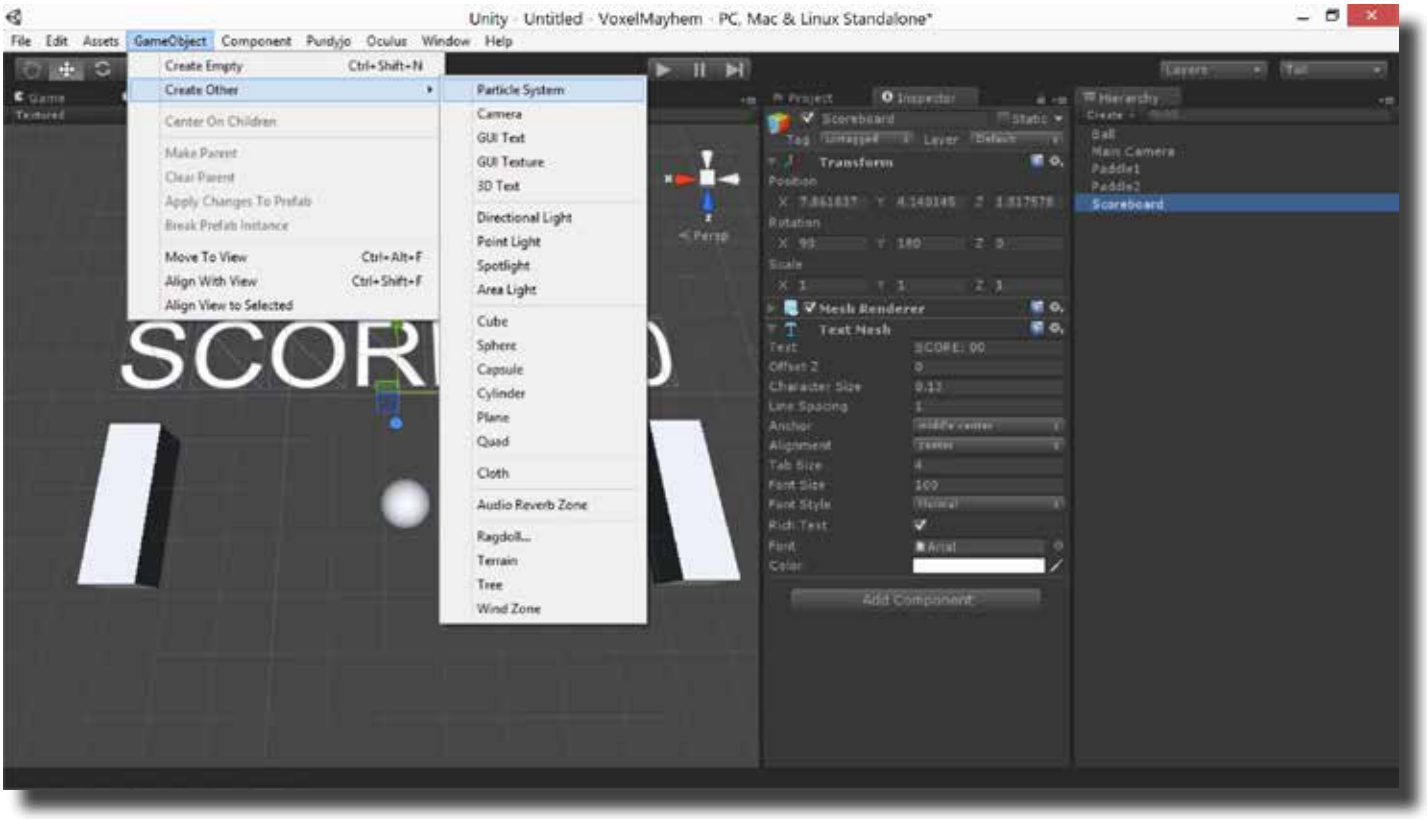
Scale it appropriately, duplicate it, and put a sphere between the paddles for the ball.

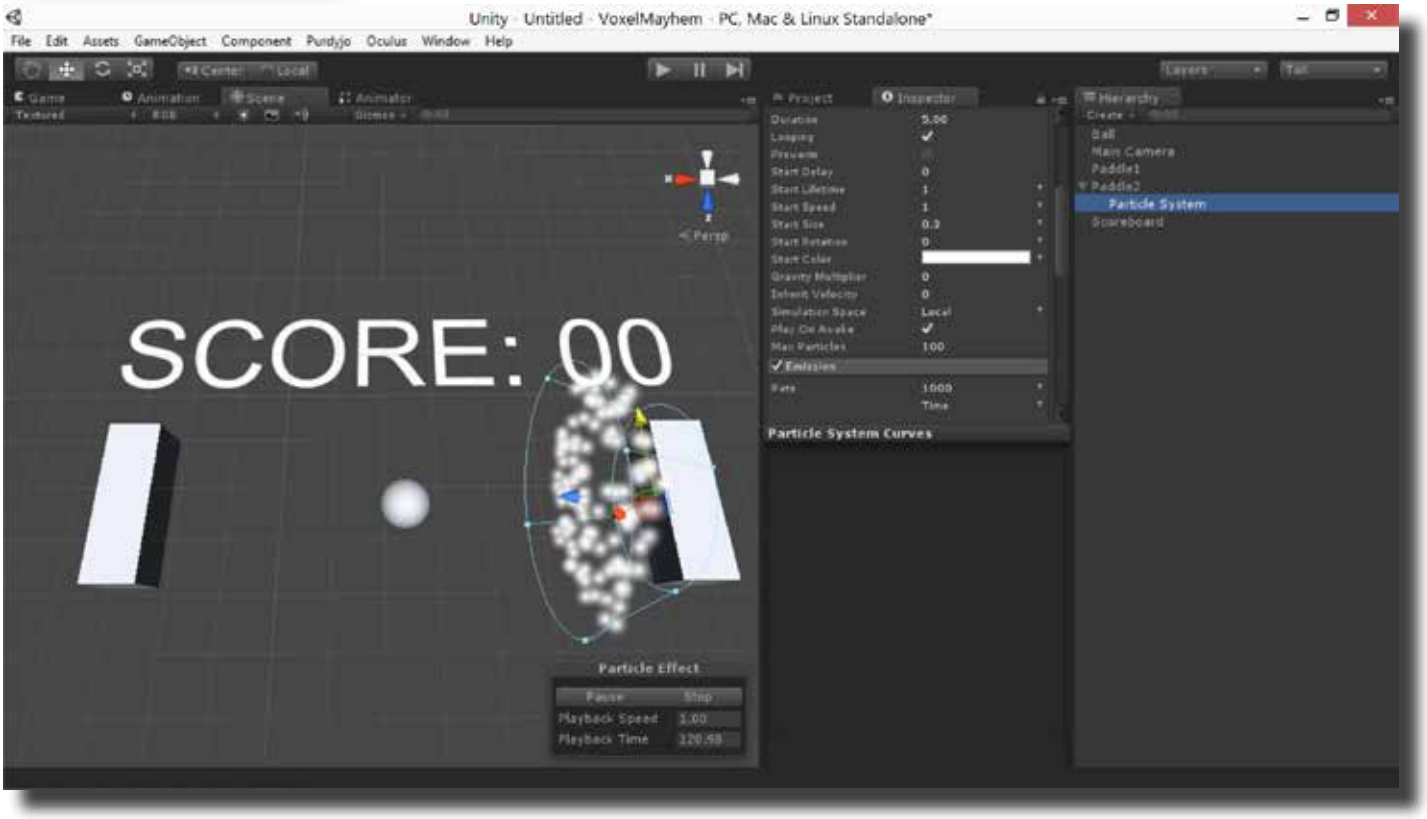


Then, create a 3DText object and scale and position it correctly, changing the ‘font size’ attribute to get a less pixelated image.

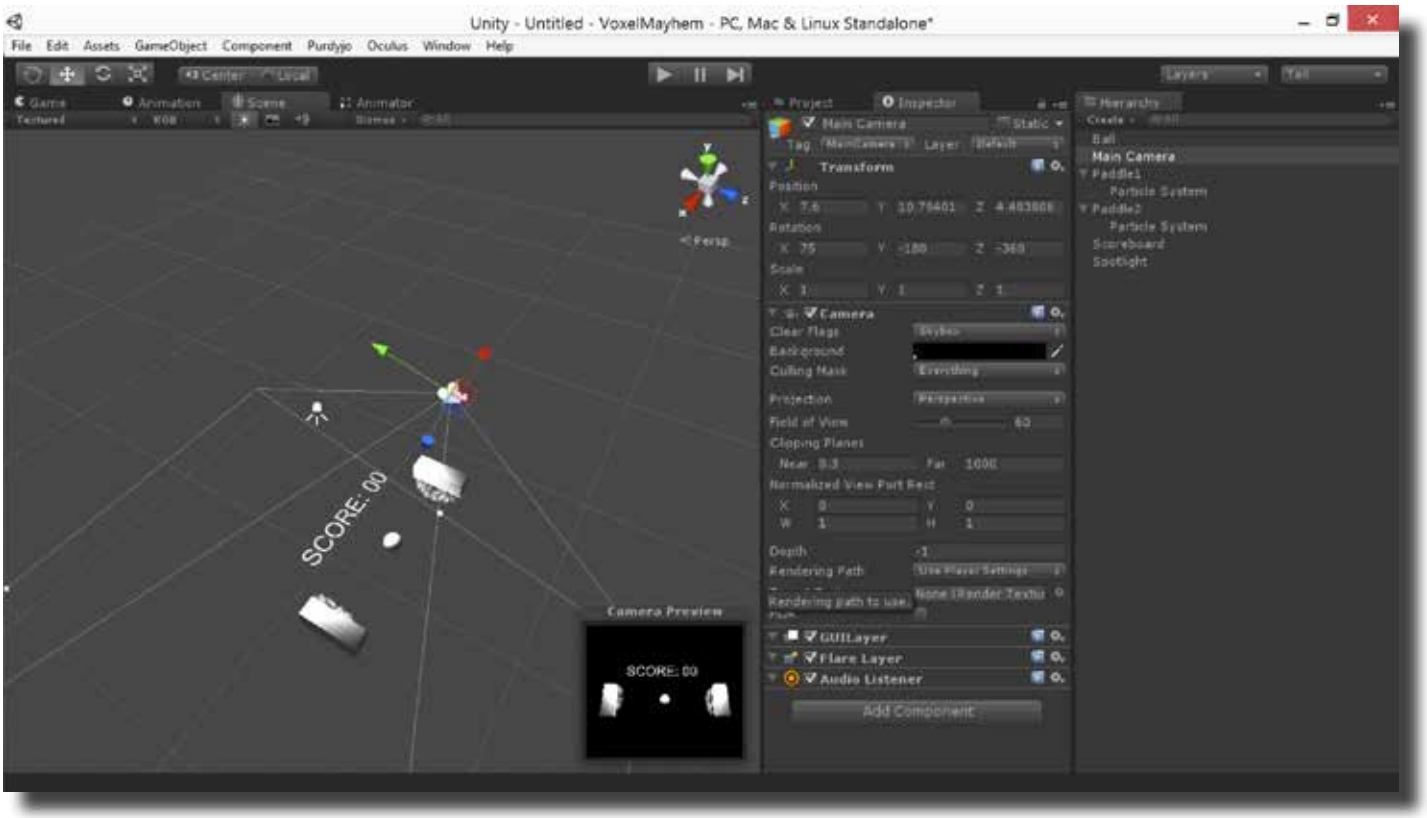


Next, create two particle systems, pick the characteristics you want, and attach them to the paddles.

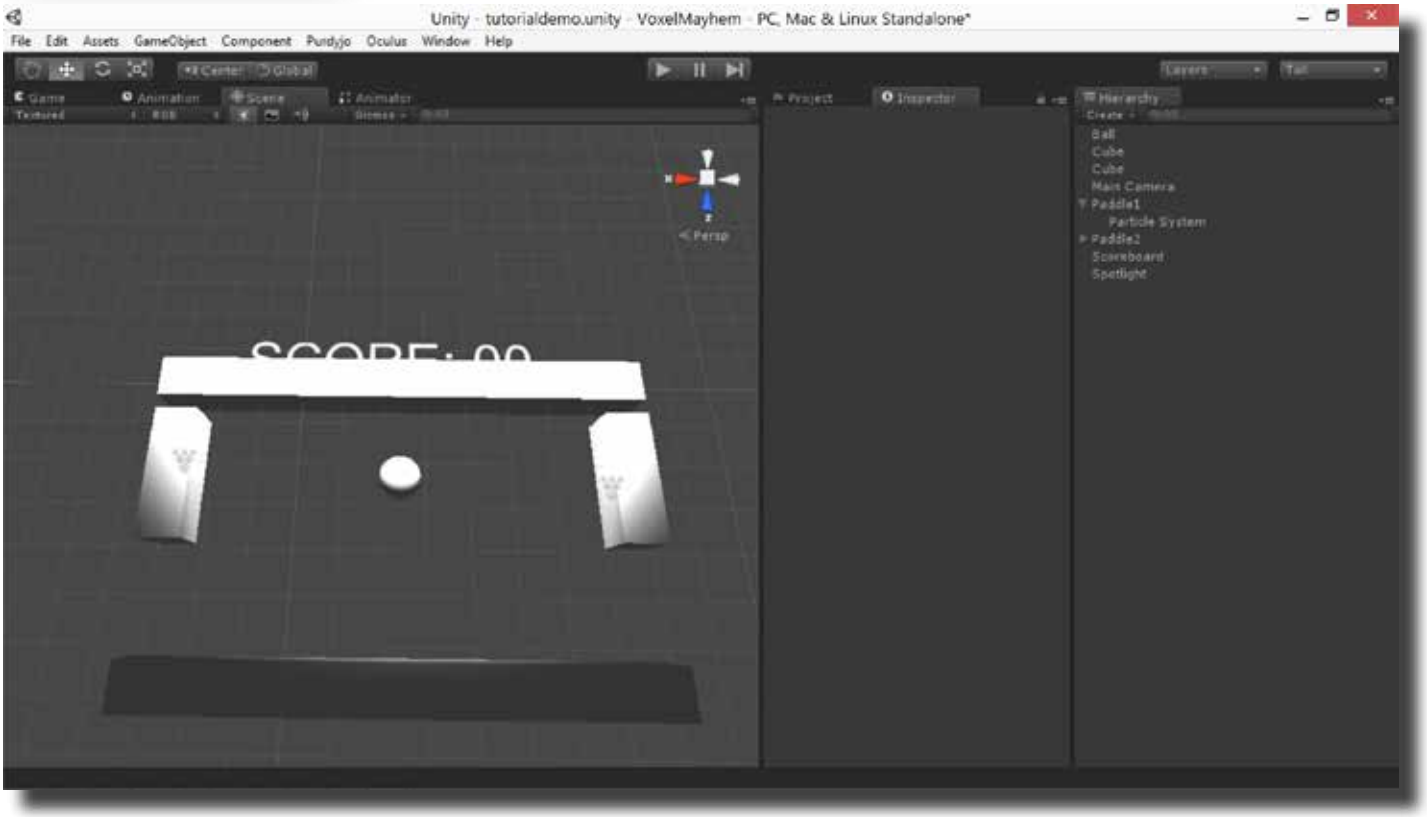


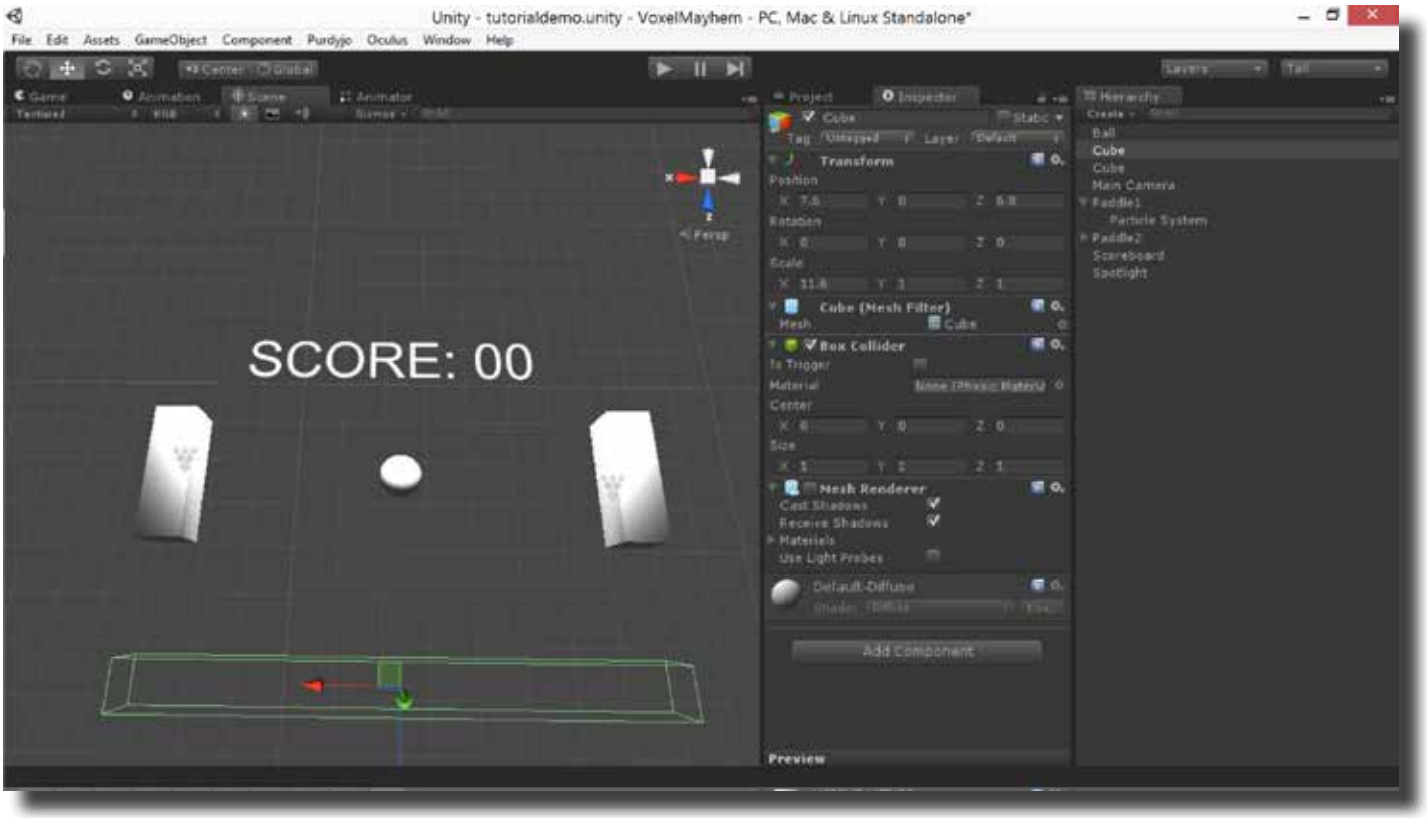


Next, you'll want to position and rotate the camera so that it frames the scene correctly. While the camera is selected, you can see a small preview of the camera's view in the lower right hand corner.

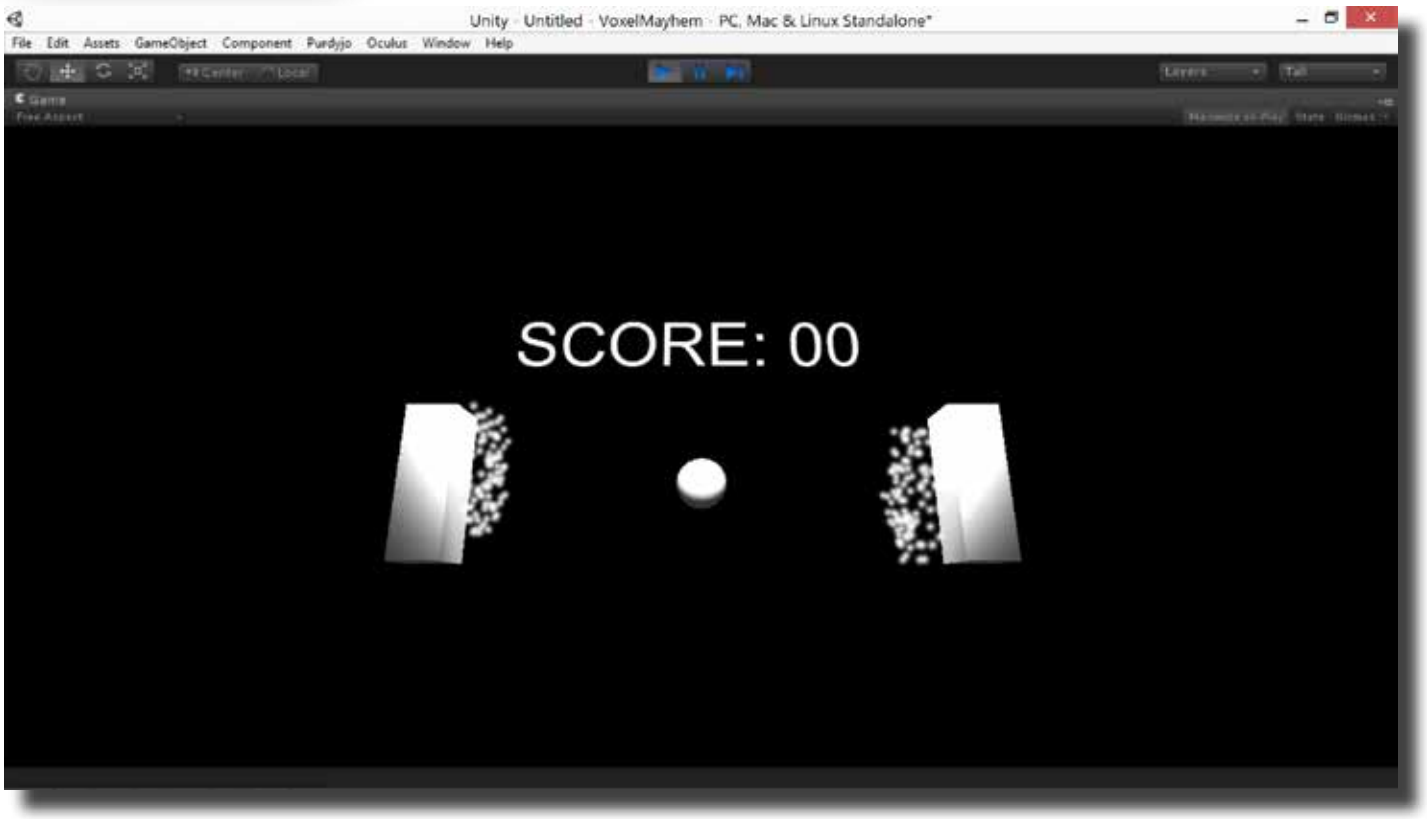


Before we finish, we need to create two additional cubes to be bumpers, to prevent the ball from bouncing out of the game area. We can make them invisible by unchecking the 'mesh renderer' in the inspector tab.





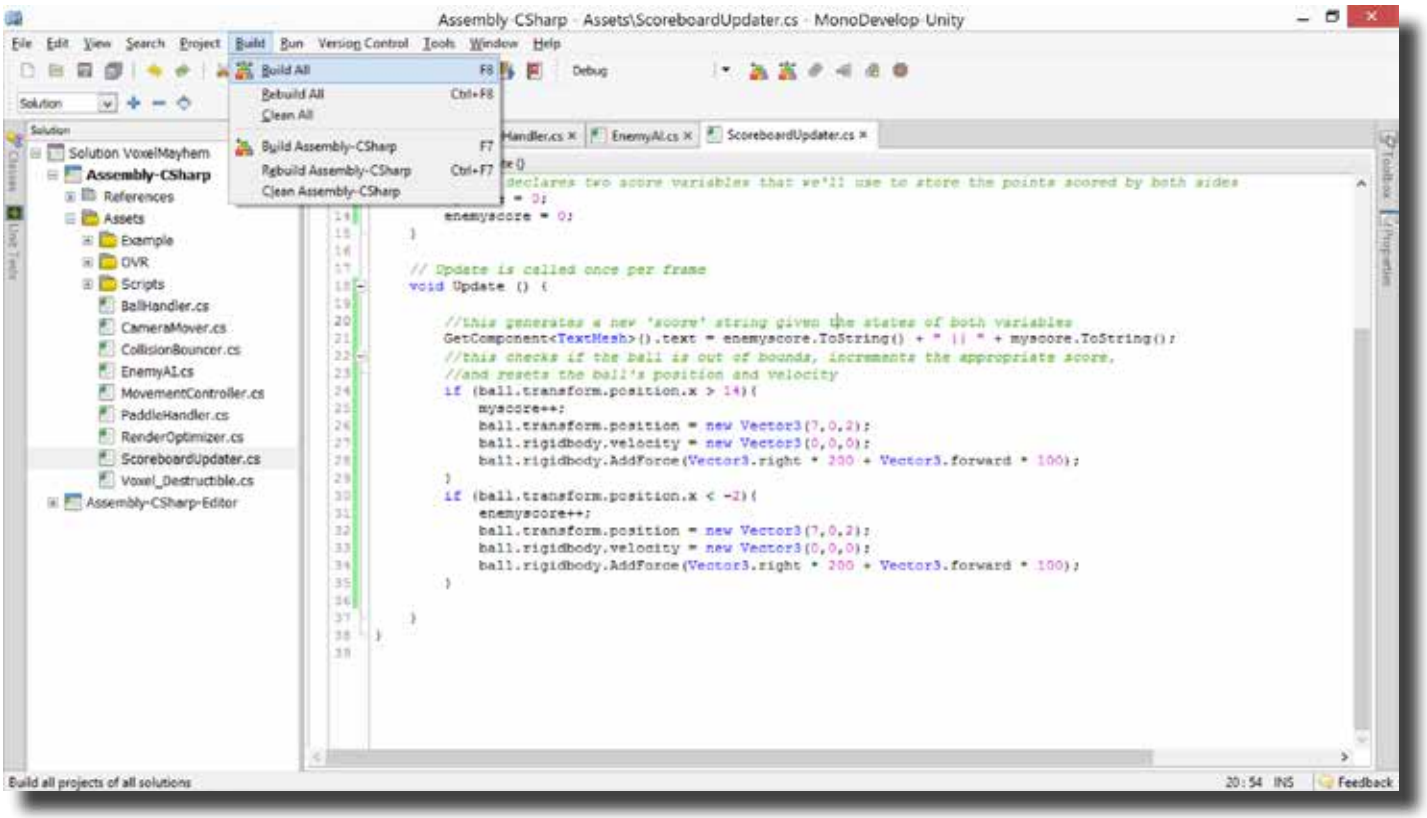
If you hit play, you can now see the basic elements of our game laid out. They won't do anything yet, but we'll get to that!



Now that we've got that setup, we're going to talk about what's involved in scripting these elements to make a game.

7. Scripting in Unity

Once you have a script attached to an object, you can revise it by double clicking on it in the 'inspector.' This opens MonoDevelop, the default development environment for Unity. In essence, Monodevelop is a text editor with features specifically optimized toward programming. Keywords and comments are highlighted in blue and green, and numerical values and strings appear in red. If you've used Eclipse or other IDE's, MonoDevelop is very similar. You can 'build' your scripts from inside the editor, to check for syntax errors, like so:



In general, to get your script to interact with Unity, you'll need to reference elements that the object holding the script possesses (you can see a list of these elements under the 'inspector' tab when the relevant object is selected). You can then call methods or set variables for each of these elements to enact the changes you want.

If you want a script on an object to affect the properties of a different object, you can create an empty GameObject variable in your script, and use the inspector to assign it to another object in the scene. The screenshots below show what that looks like.

A list of the elements an object might have is as follows (taken from the inspector view of one of our paddles in the above example):

- *Transform*
- *Cube (Mesh Filter)*
- *Box Collider*
- *Mesh Renderer*

Each of these aspects of the object can be influenced from within a script. Next, we'll look at exactly how.

7.1 Transform

The transform functions for a GameObject in Unity control the physical parameters of that object: its scale, its position, and its orientation. You can access them from within a script like this:

```
transform.position = newPositionVector3; transform.rotation = newRotationQuaternion; transform.localScale = newScaleVector3;
```

In the above examples, the named variables are of the types specified in the names. There are a few key details here: position and scale are, as you'd expect, stored as Vector3s. You can access the x, y, and z components of each (for example, transform.position.y gives you the distance of an object above the zero plane). However, to avoid gimbal lock, rotations are handled as Quaternions (four-component vectors). Because hand-manipulating quaternions is unintuitive, you can manipulate rotations using Eulerian angles by using the Quaternion.Euler method like so:

```
transform.rotation = Quaternion.Euler(pitch, yaw, roll);
```

If you wish to move objects smoothly from one place to another, you'll find the Slerp method for quaternions and vector3s helpful. Slerp takes in three arguments - the current state, the final state, and the speed of change, and smoothly interpolates between them at the given speed. The syntax looks like this:

```
transform.position = Vector3.Slerp(startPositionVector3, newDestinationVector3, 1);
```

7.2 Renderer

The renderer functions in Unity allow you to control the way the surfaces of props are rendered on-screen. You can reassign the texture, change the color, and change the shader and visibility of the object. The syntax looks like this:

```
renderer.enabled = false;  renderer.material.color = new Color(0, 255, 0);  renderer.material.mainTexture = myTexture;  renderer.material.shader = newShader;
```

Most of these have pretty clear functions. The first example makes the object in question invisible: a useful trick in a number of situations. The second example assigns a new RGB color (namely, green) to the object in question. The third assigns the main diffuse texture to a new Texture variable. The last example changes the shader of the object's material to a newly defined shader variable.

7.3 Physics

Unity comes with an integrated physics engine that allows you to assign the physical properties of objects and let the details of their simulation be handled for you. In general, rather than trying to implement your own physics using a textbook and the transform system, it is simpler and more robust to use Unity's physics engine to the greatest extent possible.

All physics props require colliders. However, the actual simulation itself is handled by a rigidbody, which can be added in the inspector view. Rigidbodies can be kinematic or nonkinematic. Kinematic physics props collide with (and effect) nonkinematic physics props around them, but are unaffected by collision themselves. Static kinematic props are the proverbial immoveable objects, and moving kinematic objects are the proverbial unstoppable force (for the record, when they collide, they simply pass through each other). Beyond that, you can adjust the angular drag of the object (how much energy it takes to spin it), change its mass, dictate whether or not it's affected by gravity, and apply forces to it.

Examples:

```
rigidbody.angularDrag = 0.1f; rigidbody.mass = 100; rigidbody.isKinematic = false; rigidbody.useGravity = true; rigidbody.AddForce(transform.forward * 100);
```

These are all pretty self-explanatory. The only thing to note here is the use of 'transform.forward.' Vector3's all have three components(.forward, .up, and .right) associated with them, which can be accessed and rotates with them ("forward" is the direction of the blue arrow in the editor). "transform.forward" is simply the forward vector for the current object with magnitude 1. It can be multiplied by a float to create more force on the object. You can also reference "transform.up" and "transform.right," and negate them to get their reverses.

7.4 Collision

Often, when building a game, you'd like a collision to result in some change-of-state in your code, beyond just physics simulation. For this, you'll need a collision detection method.

There's a certain amount of prep work needed to detect collisions in Unity. First, at least one of the objects in the collision needs a non-kinematic rigidbody attached to it. Both objects must have correct colliders, set to be non-triggers. The total speed of both objects must be low enough that they actually collide, instead of simply skipping through one another.

If you've got all that taken care of, you can check for collision by placing a special collision detection method in a script attached to the object you'd like to check collision with. The method will look like this:

```
void OnCollisionEnter(Collision other){ //do things here }
```

This method will automatically run during the first frame that another object touches your object. The collision entity 'other' is a reference to the object that you hit. You can, for example, reference its 'gameobject,' 'rigidbody,' and 'transform' characteristics to manipulate it in various ways. While 'OnCollisionEnter' is probably the most common function you'll be using, you can also use 'OnCollisionExit' and 'OnCollisionStay' (with otherwise identical syntax and usage), which activate during the first frame that you stop colliding with an object and during every frame that you're colliding with an object, respectively.

Sometimes, it can also be useful to do what's called raycasting. In raycasting, an infinitely thin line (a 'ray') is cast through the world from some origin, along some vector, and, when it hits something, the position and other details of the first collision are returned. The code for a raycast looks like this:

```
RaycastHit hit; if (Physics.Raycast(transform.position, -Vector3.up, out hit)){ float distanc-
eToGround = hit.distance; }
```

This casts a ray from the position of the current object along -Vector3.up (straight down), and links the variable 'hit' to the first object it collides with. Once your ray has hit something, you can access hit.distance to determine how far away it is, or hit.GameObject to manipulate the object you hit.

Raycasts like this can be used for shooters to determine what the gun's pointed at, or to select objects when the camera looks at them, or for certain styles of movement mechanic.

7.5 Time Correction

One important factor to keep in mind when you're manipulating objects in this way has to do with framerate. No matter how carefully you optimize, framerates will always vary, and you don't want your game speed to vary accordingly. If someone else runs your game on a faster computer than you developed it on, you don't want the game to run at double speed.

The way you correct for this is by multiplying the values you're using by the time it took to render the last frame. This is done by using 'Time.deltaTime. This effectively changes the speed of any variable you're incrementing every frame from 'change per frame' to 'change per second,' and you should probably make this change to any value you're incrementing or decrementing every frame.

7.6 Audio Sources and Listeners

Now that we've covered how to create, render, and control objects, let's talk about the other sense that computer games can serve: namely, sound. Unity supports two kinds of sounds: 2D and 3D sounds. 3D sounds vary their volume based on distance, and distort as they move relative to the camera; 2D sounds do not. 2D sounds are appropriate for voice-overs and background music, and 3D sounds apply to sounds generated by events in the world. In order to change whether or not a sound is 3D, select it in the 'project' view, switch to the 'inspector' view and select the appropriate option from the dropdown menu, then press the 'reimport' button.

In order to actually play the sound, you'll need to attach an 'audiosource' to a prop (the prop you want the sound to originate from, in the case of a 3D sound. Then you'll need to open the 'audioclip' field and select your sound file.

You can use myAudioSource.Pause() and myAudioSource.Play() to control those sound files. You can adjust the falloff behaviors, volume, and doppler shifting of the sounds under the 'inspector' tab for the audiosource.

7.7 Input

A game that doesn't take any input from the user isn't much of a game. There are a lot of different kinds of input you can read in, and almost all of them are accessible through the Input and KeyCode objects. Some sample input state-

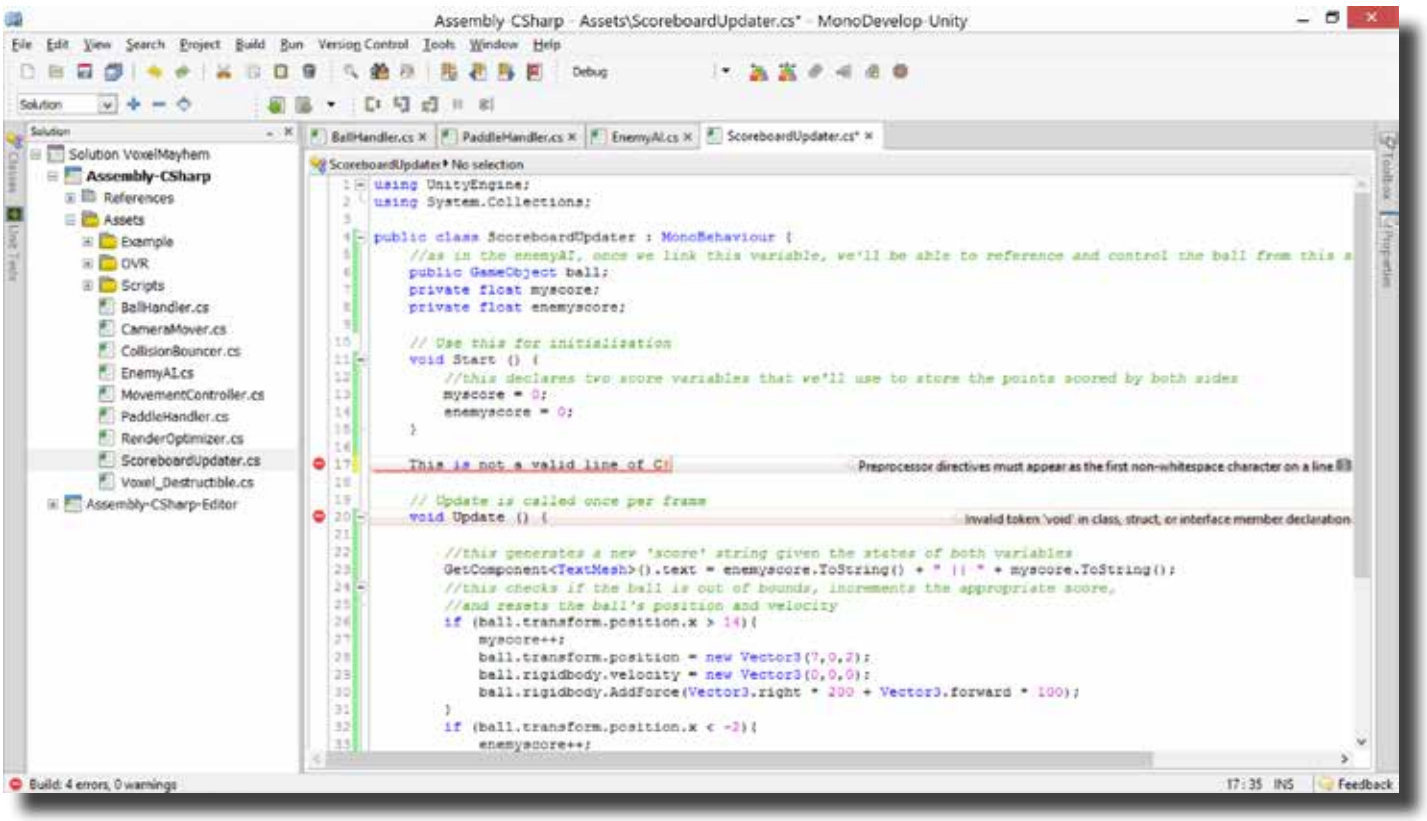
ments (which have a values evaluated every frame) are below.

```
Vector3 mousePos = Input.mousePosition; bool isLeftClicking = Input.GetMouseButton(0); bool isPressingSpace = Input.GetKey(KeyCode.Space);
```

The functions of these lines is mostly self explanatory. Using these three kinds of input reference, you can reconstruct the control schemes of most modern 3D computer games.

7.8 Debugging a Script

Let's say a script doesn't work. As the good doctor says, bangups and hangups can happen to you. If there are outright syntax errors with your C#, the game will generally refuse to run when you hit play, and some fairly useful error messages are provided if you 'build' the scripts from within the editor. See below:



These bugs are typically not the most difficult to fix. What can be more problematic are subtle semantic errors, in which you have successfully written a file full of valid C# – just not one that does what you thought it would. If you have one of these errors, and you're having trouble tracking it down, there are a few things you can try to improve the situation.

The first is to pause the execution of the game, and check the console. You can pause the game by clicking on the 'pause' icon in the upper middle portion of the editor, and then selecting 'console' from the bottom of the 'window' menu (or pressing Ctrl-Shift-C). Even if there are no errors, warnings can still help to give some clues as to what might be going wrong.

If this doesn't work, you can also try to get some idea about the state of your script by printing the state of internal variables to validate that the program is doing what you think it's doing. You can use Debug.Log(String); to print the contents of a string to the console when the program execution hits that line. In general, if you work backwards from what you think should be happening through the things that should be making it happen, eventually you will reach a point where your debug prints don't do what you expect them to do. That's where your error is.

8. Example: Scripting Pong

To build Pong, let's break the game down into its core elements: we need a ball that ricochets back and forth between the paddles at increasing speed, we need a scoreboard that knows when the balls have passed the paddles, and we need a mechanism for restarting the ball when that happens. A good first step would be to add a non-kinematic rigid-body to the ball, two kinematic rigidbodies to the paddles, disable gravity for all of them, and assign an appropriate physic material from the standard assets ('bounce' with 'bounce combine' set to 'multiply).

Below, you can view the script for the ball with explanatory comments. The ball needs to accomplish some basic goals: it should bounce in a complicated pattern, always maintaining movement on both axes, and it should accelerate at a challenging but not impossible pace in the horizontal direction.

BallHandler.cs

Next, we need to script our paddle, which you can, again, view below. The paddle needs to move up and down in response to key presses (but not outside certain bounds). It also needs to trigger the particle system when it collides with something.

PaddleHandler.cs

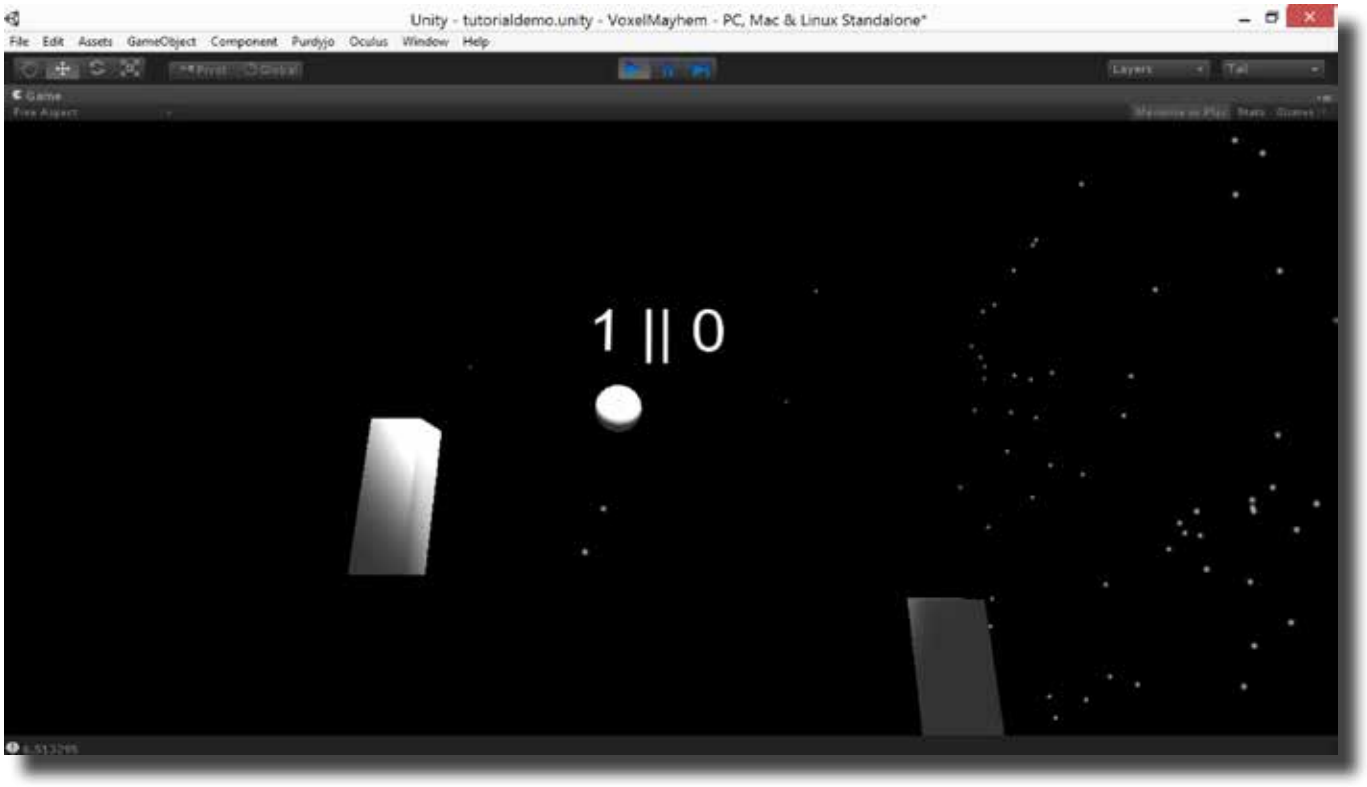
Next, we need enemy AI: something that will cause the enemy's paddle to track towards the ball at a fixed rate. For that, we'll be using Vector3.Slerp for maximum simplicity. We'd also like the same particle behavior that we see on our own paddle.

EnemyAI.cs

Finally, we need a script to update the scoreboard and reset the ball when it goes out of bounds.

ScoreboardUpdater.cs

With those scripts attached and the references filled in, when we run our game of Pong, we experience gameplay!



You can download my Pong demo, if you'd like to see everything I've outlined in action. It runs on Windows, Mac and Linux systems.

9. Exploring Documentation / Learning More

Unity is a complex engine with many more features than could feasibly be covered in a guide of this style, and that's before you include the wide swathe of (free and commercial) Unity extensions available on the Internet. This guide will give you a strong starting place for developing a game, but self-education is an important skill in any endeavor, and doubly so here.

A crucial resource here is the Unity ScriptReference. The ScriptReference is a searchable database, available for both C# and Javascript, which has a list of every Unity command and feature, with descriptions of their functions and brief examples of syntax.

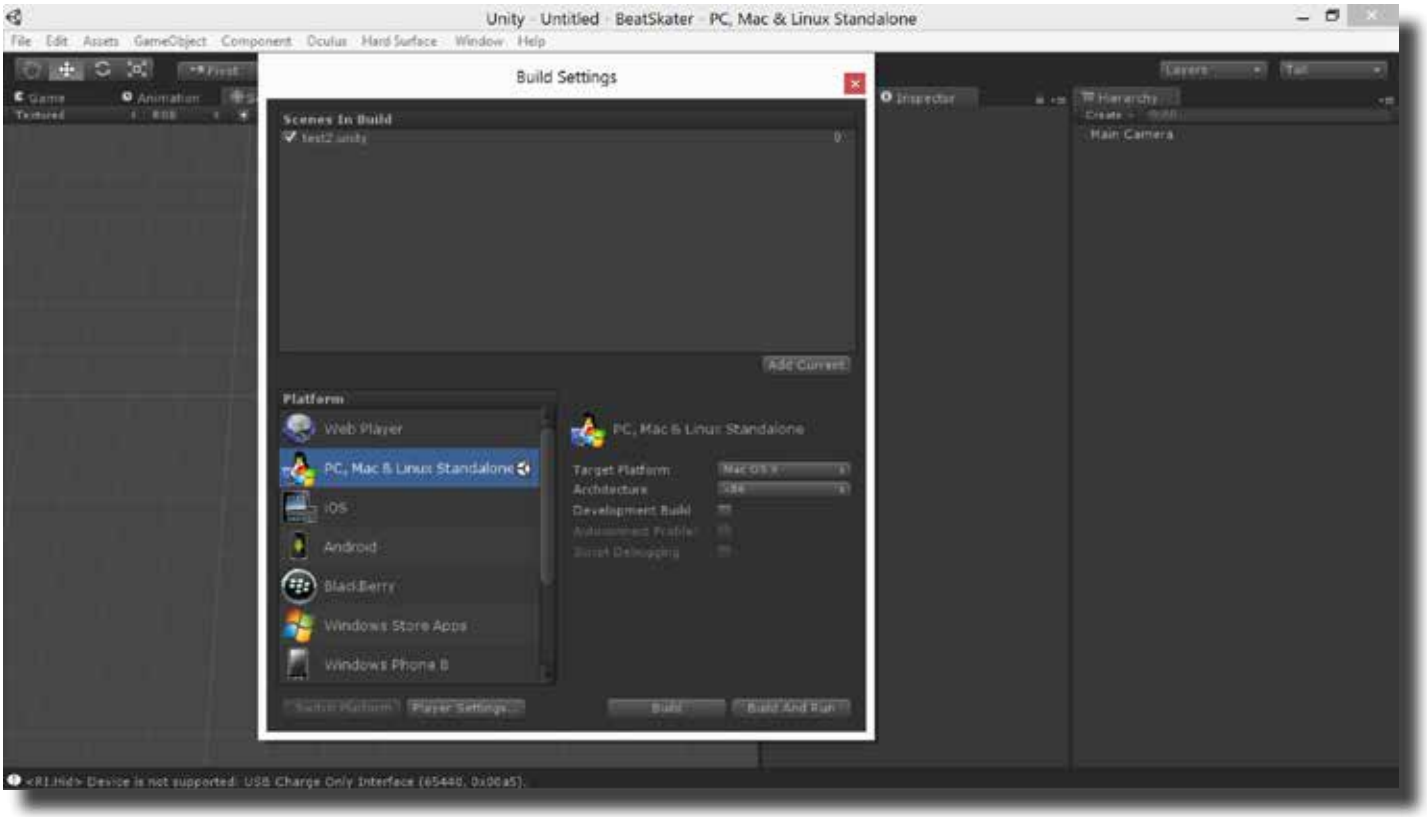
If you're having trouble with the editor and interface of Unity, or just like video tutorials as a matter of preference, there is a long list of high-quality Unity video tutorials available. More extensive (but less broad) text tutorials for Unity are also available from CatLikeCoding.

Finally, if you have questions beyond the scope of documentation or tutorials, you can ask specific questions at answers.Unity3d.com. Remember that answers are provided by volunteers, so respect their time and search the database first to make sure your question hasn't already been answered.

10. Building Your Game / Compiling

When you've built something you're proud of (or you've finished cloning our slightly dodgy Pong example for practice), it's time to move your game from the editor and turn it into something that you can post on the Internet and force your friends and family to play. In order to do that, you'll need to build a standalone application. The good news is that in Unity, this is very, very easy. There are, however, a few potential hiccoughs that you'll want to be careful of.

For starters, know that you can only build an error-free project. To that end, make sure you have the console open as you build: there are some error conditions that the game will ignore in the editor, but will still abort an attempted build. This only dumps error messages to the console, with no visible results onscreen, which can be frustrating if you forget to check. Once you've got your game compiling error-free, though, you can select 'Build Settings' under the 'File' menu, or press Ctrl-Shift-B. This will bring up a simple dialog that allows you to build your game for several platforms.



The process from there is self explanatory: select your options, and hit 'build;' the game will prompt you for a directory to install to, and will place both the executable and data directory there. These two files can be zipped together and distributed (just make sure you aren't charging for a game built in the Unity demo, as this violates the terms of service).

11. Closing Notes

As with any game development tool, the key to success with Unity is iterative development. You have to build in manageable increments - be ambitious, by all means, but be ambitious in small chunks, and arrange those chunks such that, even if you fall short of your ultimate ambition, you'll at least wind up with a coherent product. Get the most crucial elements in first: have an idea in mind of your 'minimum viable product,' the simplest, most bare-bones thing you could possibly create and still feel as though you achieved something worthwhile. Get to that minimum viable project before moving on to larger ambitions.

This tutorial gives you a strong starting place, but the best way to learn Unity is while building a game. Start building a game, fill gaps in your knowledge as they come up, and the gradual flow of knowledge will erode away the things you don't know surprisingly quickly. Unity is a powerful tool, and with a bit of exploration, you can be building impressive projects with it quicker than you might expect. Good luck, and enjoy Unity!

Guide Published: February 2014

Did you like this PDF Guide? Then why not visit MakeUseOf.com for daily posts on cool websites, free software and internet tips?

If you want more great guides like this, why not subscribe to MakeUseOf and receive instant access to 50+ PDF Guides like this one covering wide range of topics. Moreover, you will be able to download free Cheat Sheets, Free Giveaways and other cool things.

Home:	**http://www.makeuseof.com**
MakeUseOf Answers:	**http://www.makeuseof.com/answers**
PDF Guides:	**http://www.makeuseof.com/pages/**
Tech Deals:	**http://www.makeuseof.com/pages/hot-tech-deals**

Follow MakeUseOf:

RSS Feed:	**http://feedproxy.google.com/Makeuseof**
Newsletter:	**http://www.makeuseof.com/subscribe/**
Facebook:	**http://www.facebook.com/makeuseof**
Twitter:	**http://www.twitter.com/Makeuseof**

Think you've got what it takes to write a manual for MakeUseOf.com? We're always willing to hear a pitch! Send your ideas to justinpot@makeuseof.com.