Community Experience Distilled

Unity 5.x Game Development Essentials

Third Edition

Gear up to build fully functional 2D and 3D multiplatform games with realistic environments, sound, dynamic effects, and more!

Tommaso Lintrami

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

1
Enter the Third Dimension

Before getting started with any 3D package, it is crucial to understand the environment you'll be working in.

As **Unity** was born a 3D engine / development tool, many concepts throughout this book will assume a certain level of understanding of 3D development in general, and game engines. It is crucial that you equip yourself with at least a basic knowledge of these concepts before diving into the practical elements of the rest of this book.

In this chapter, we'll make sure you're prepared by looking at some important 3D concepts, before moving on to discuss the concepts and interface of Unity itself. You will learn about the following topics:

- Coordinates and vectors
- Polygons, edges, vertices, and meshes
- Shaders, materials, and textures
- Rigidbody physic simulation and collision detection
- GameObjects and scripted or built-in components
- Assets and scenes, the project view
- Unity editor user interface

Getting to grips with 3D

Let's take a look at the crucial elements of the 3D world, and how Unity lets you develop games in the three dimensions.

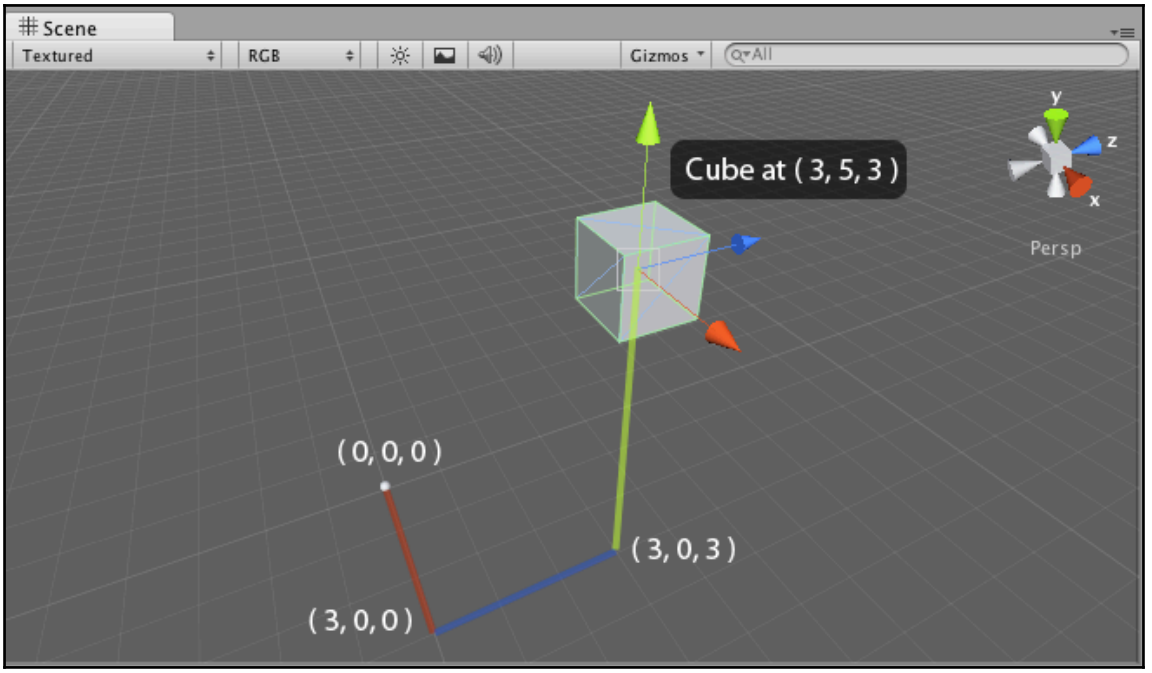
Coordinates

If you have worked with any 3D application before, you'll likely be familiar with the concept of the *Z*-axis. The *Z*-axis, in addition to the existing *X* for horizontal and *Y* for vertical, represents depth. In 3D applications, you'll see information on objects laid out in *X*, *Y*, *Z* format-this is known as the **Cartesian coordinate** method. Dimensions, rotational values, and positions in the 3D world can all be described in this way. In this book, as in other documentation of 3D, you'll see such information written with parenthesis, shown as follows:

```
(3,5,3)
```

This is mostly for neatness, and also due to the fact that in programming, these values must be written in this way. Regardless of their presentation, you can assume that any sets of three values separated by commas will be in *X*, *Y*, *Z* order.

In the following screenshot, a cube is shown at location `(3,5,3)` in the 3D world, meaning it is `3` units from in the *X*-axis, `5` up in the *Y*-axis, and `3` forward in the *Z*-axis:



Local space versus world space

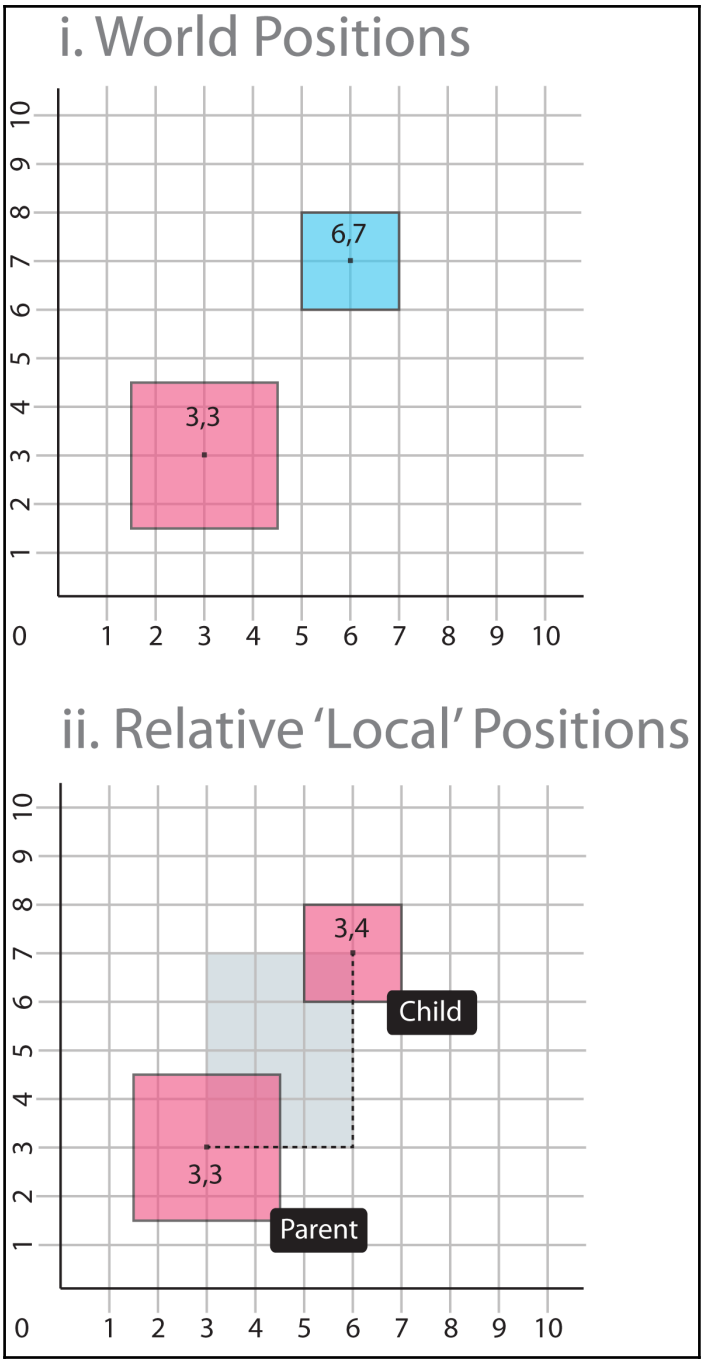
A crucial concept to begin looking at is the difference between local space and world space. In any 3D package, the world you will work in is technically infinite, and it can be difficult to keep track of the location of objects within it. In every 3D world, there is a point of origin, often referred to as the origin or world zero, as it is represented by the position `(0,0,0)`.

All world positions of objects in 3D are relative to world zero. However, to make things simpler, we also use local space (also known as object space) to define object positions in relation to one another. These relationships are known as parent-child relationships. In Unity, parent-child relationships can be established easily by dragging one object onto another in the hierarchy. This causes the dragged object to become a child, and its coordinates from then on are read in terms relative to the parent object. For example, if the child object is exactly at the same world position as the parent object, its position is said to be `(0,0,0)`, even if the parent position is not at world zero.

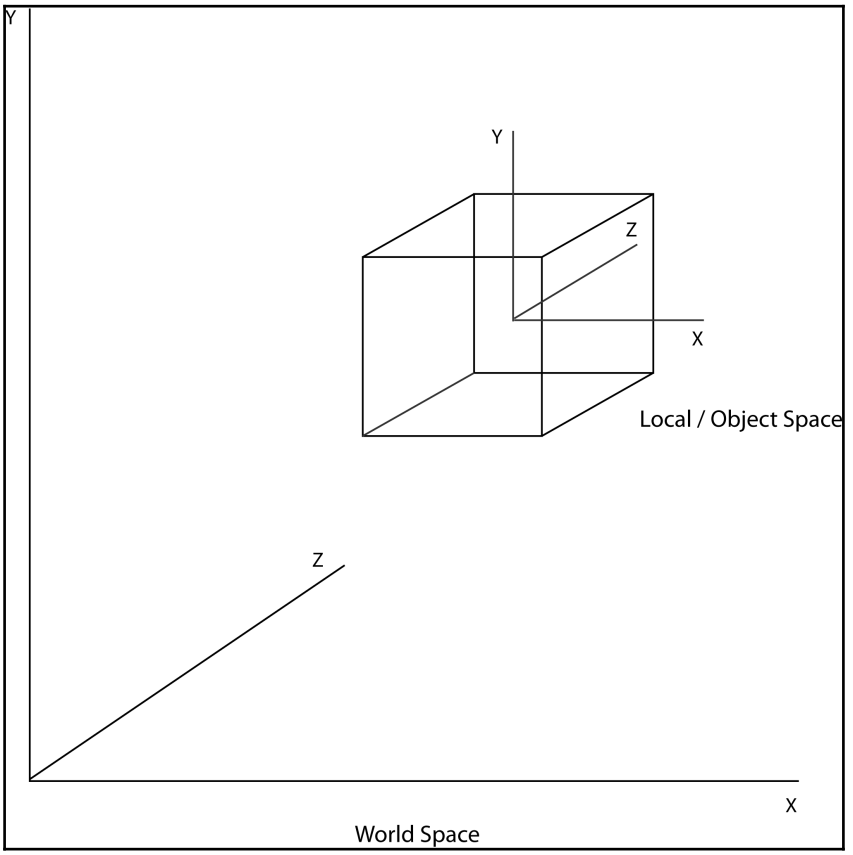
Local space assumes that every object has its own zero point, which is the point from which its axes emerge. This is usually the center of the object, and by creating relationships between objects, we can compare their positions in relation to one another. Such relationships, known as parent-child relationships, mean that we can calculate distances from other objects using local space, with the parent object's position becoming the new zero point for any of its child objects.

We can illustrate this in 2D, as the same conventions will apply to 3D. In the following example:

- The first diagram, **i. World Positions**, shows two objects in world space. A large cube exists at coordinates **(3,3)**, and a smaller one at coordinates **(6,7)**.
- In the second diagram, **ii. Relative 'Local' Positions**, the smaller cube has been made a **Child** object of the larger cube. As such the smaller cube's coordinates are said to be **(3,4)**, because its zero point is the world position of the **Parent**.


i. World Positions
6,7
3,3
ii. Relative 'Local' Positions
3,4
Child
3,3
Parent

To extrapolate this to three dimensions, the same rule applies but with the addition of a *Z*-axis:

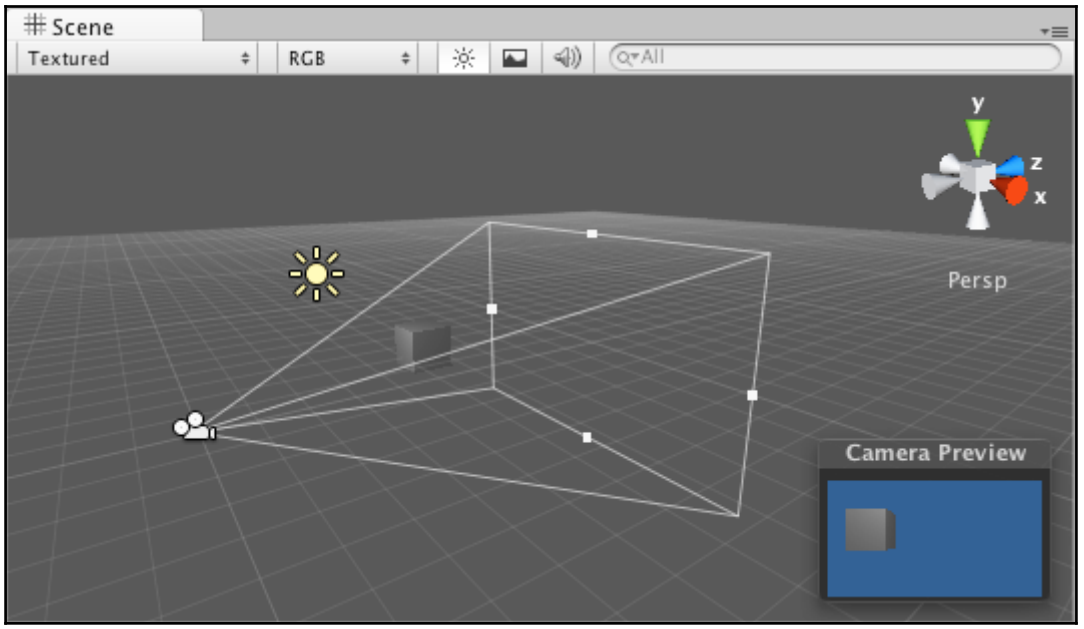


Vectors

You'll also see 3D vectors described in Cartesian coordinates. Like their 2D counterparts, 3D vectors are simply lines drawn in the 3D world that have a direction, and a length. Vectors can be moved in world space, but remain unchanged themselves. Vectors are useful in a game engine context, as they allow us to calculate distances, relative angles between objects, and the direction of objects.

Cameras

Cameras are essential in the 3D world, as they act as the viewport for the screen. Look at the following screenshot:



Cameras can be placed at any point in the world, animated, or attached to characters or objects as part of a game scenario. Many cameras can exist in a particular scene, but it is assumed that a single main camera will always render what the player sees. This is why Unity gives you a main camera object whenever you create a new scene.

Projection mode – 3D versus 2D

The projection mode of a camera states whether it renders in 3D (Perspective) or 2D (Orthographic). Ordinarily, cameras are set to Perspective projection mode, and as such have a pyramid shaped **fieldofview** (**FOV**). A Perspective mode camera renders in 3D and is the default projection mode for a camera in Unity. Cameras can also be set to Orthographic projection mode in order to render in 2D-these have a rectangular FOV. This can be used on a main camera to create complete 2D games, or simply used as a secondary camera to render **HeadsUpDisplay(HUD)** elements such as a map or health bar.

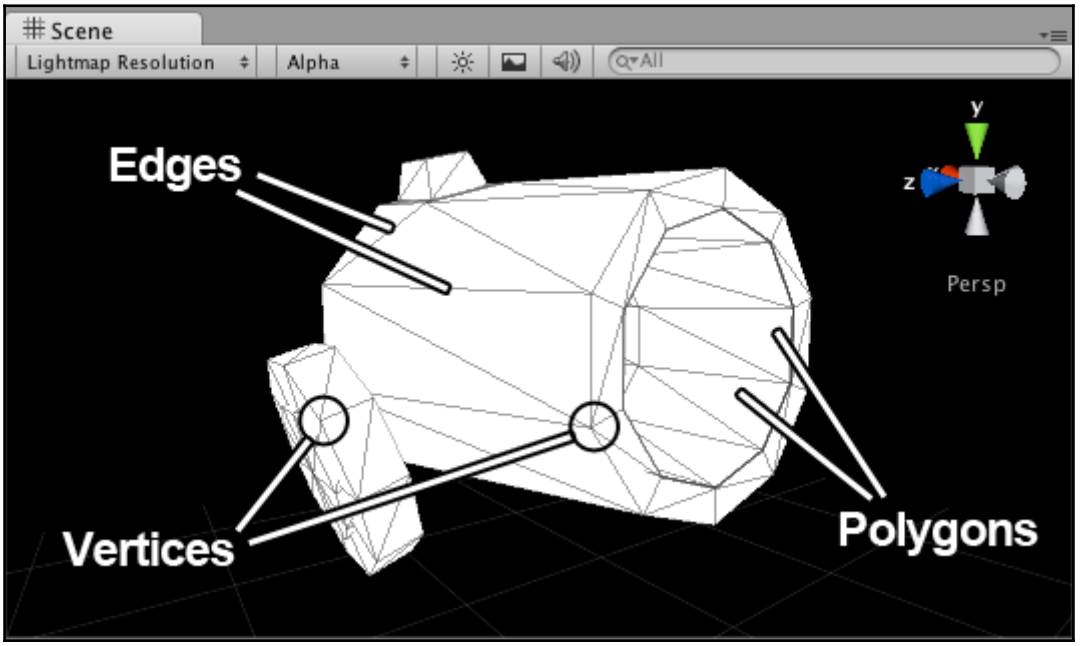
In game engines, you'll notice that effects such as lighting, motion blurs, and other effects

are applied to the camera to help with game simulation of a person's eye view of the world. You can even add a few cinematic effects that the human eye will never experience, such as lens flares when looking at the sun!

Most modern 3D games utilize multiple cameras to show parts of the game world that the character camera is not currently looking at, like a 'cutaway' in cinematic terms. Unity does this with ease by allowing many cameras in a single scene, which can be scripted to act as the main camera at any point during runtime. Multiple cameras can also be used in a game to control the rendering of particular 2D and 3D elements separately as part of the optimization process. For example, objects may be grouped in layers, and cameras may be assigned to render objects in particular layers. This gives us more control over individual renders of certain elements in the game.

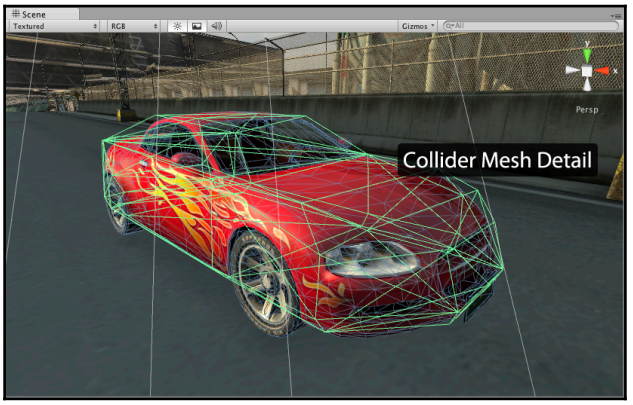
Polygons, edges, vertices, and meshes

In constructing 3D shapes, all objects are ultimately made up of interconnected 2D shapes known as **Polygons**. On importing models from a modeling application, Unity converts all polygons to polygon triangles. Polygon triangles (also referred to as faces) are in turn made up of three connected **Edges**. The locations at which these **Edges** meet are known as points or **Vertices**. Have a look at the following screenshot:



By knowing these locations, game engines are able to make calculations regarding the points of impact, known as collisions, when using complex collision detection with **Mesh Colliders**, such as in shooting games to detect the exact location at which a bullet has hit another object. By combining many linked polygons, 3D modeling applications allow us to build complex shapes, known as meshes. In addition to building 3D shapes that are rendered visibly, mesh data can have many other uses. For example, it can be used to specify a shape for collision that is less detailed than a visible object, but roughly the same shape. This can help save performance as the physics engine needn't check a mesh in detail for collisions. This is seen in the following screenshot from the Unity car tutorial, where the vehicle itself is more detailed than its collision mesh:

In the following screenshot, you can see that the amount of detail in the mesh, used for the Collider, is far less than the visible mesh itself:



In game projects, it is crucial for the developer to understand the importance of the **polygon count**. The polygon count is the total number of polygons, often in reference to the models,

but also in reference to the props, or an entire game level (or in Unity terms, 'Scene'). The higher the number of polygons, the more work your computer must do to render the objects onscreen. This is why, in the past decade or so, we've seen an increase in the level of detail from early 3D games to those of today. Simply compare the visual detail in a game such as id's *Quake* (1996) with the details seen in Epic's *GearsOfWar* (2006) in just a decade. As a result of faster technology, game developers are now able to model 3D characters and worlds, for games that contain a much higher polygon count and resultant level of realism, and this trend will inevitably continue in the years to come. This said, as more platforms emerge such as mobile and online, games previously seen on dedicated consoles can now be played in a web browser thanks to Unity. As such, the hardware constraints are as important now as ever, as low powered devices such as mobile phones and tablets are able to run 3D games. For this reason, when modeling any object to add to your game, you should consider polygonal detail, and where it is necessary.

Shaders, materials and textures

Materials are a common concept to all 3D applications, as they provide the means to set the visual appearance of a 3D model. From basic colors to reflective image-based surfaces, materials handle everything.

Starting with a simple color and the option of using one or more images-known as **textures**. In a single material, the material works with the **shader**, which is a script in charge of the style of rendering. For example, in a reflective shader, the material will render reflections of surrounding objects, but maintain its color or the look of the image applied as its texture.

In Unity, the use of materials is easy. Any materials created in your 3D modeling package will be imported and recreated automatically by the engine and created as assets that are reusable. You can also create your own materials from scratch, assigning images as textures and selecting a shader from a large library that comes built-in. You may also write your own shader scripts or copy-paste those written by fellow developers in the Unity community, giving you more freedom for expansion beyond the included set.

When creating textures, for a game in a graphics package such as Photoshop or **GNU Image Manipulation Program** (**GIMP**), you must be aware of the resolution. Larger textures will give you the chance to add more detail to your textured models, but be more intensive to render. Game textures imported into Unity will be scaled to a power of two resolution. For example:

- 128px x 128px
- 256px x 256px
- 512px x 512px

- 1024px x 1024px
- 2048px x 2048px
- 4096px x 4096px

In the case of rectangular textures, animated textures, or texture atlas (see Chapter 13, *Performance Tweaks and Finishing Touches*) you could also use sizes like:

- 256 x 64px
- 4096 x 512px

Is important for certain hardware, that both width and height of the textures are numbers which are a power of two.

Creating textures of these sizes with content that matches the edges (seamless) will mean that they can be tiled successfully by Unity. In case of graphics imported for the UI instead, Unity will use the original size even for uneven images, eg: 392 x 157 as you will discover over the course of this book.

Rigidbody physics

For developers working with game engines, physics engines provide an accompanying way of simulating real-world responses for objects in games. In Unity, the game engine uses Nvidia's *PhysX* engine, a popular and highly accurate commercial physics engine.

In game engines, there is no assumption that an object should be affected by physics-firstly because it requires a lot of processing power, and secondly because there is simply no need to do so. For example, in a 3D driving game, it makes sense for the cars to be under the influence of the physics engine, but not the track or surrounding objects, such as trees, walls, and so on-they will remain static for the duration of the game. For this reason, when making games in Unity, a Rigidbody component is given to any object that you wish to be under the control of the physics engine.

Physics engines for games uses the Rigidbody dynamics system of creating realistic motion. This simply means that instead of objects being static in the 3D world, they can have properties such as mass, gravity, velocity, and friction.

As the power of hardware and software increases, Rigidbody physics is becoming more widely applied in games, as it offers the potential for more varied and realistic simulation. We'll be utilizing Rigidbody dynamics as part of our prototype in Chapter 1, *Enter to the*

Third Dimension and as part of the main game of the book in Chapter 10, *Instantiation and Rigidbodies*.

Collision detection

More crucial in game engines than in 3D animation, collision detection is the way we analyze our 3D world for inter-object collisions. By giving an object a **Collider** component, we are effectively placing an invisible net around it. This net usually mimics its shape and is in charge of reporting any collisions with other Colliders, making the game engine respond accordingly.

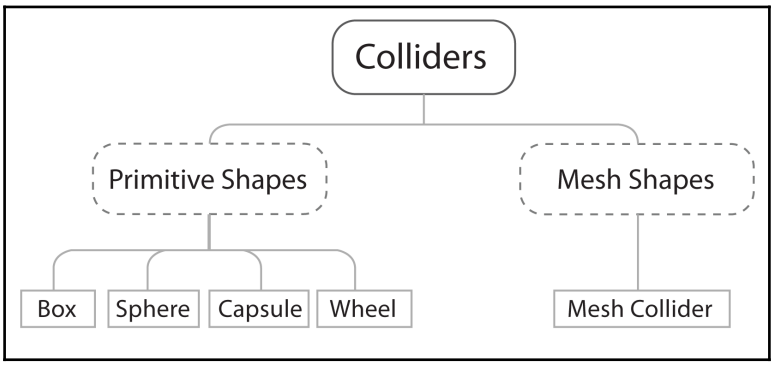
There are two main types of Collider in Unity:

- Primitive
- Mesh

Primitive shapes in 3D terms are simple geometric objects such as **Box**, **Sphere**, and **Capsule**. Therefore, a primitive Collider such as a Box Collider in Unity has that shape, regardless of the visual shape of the 3D object that is applied to it. Often primitive Colliders are used because they are computationally cheaper or because there is no need for precision.

A Mesh Collider is more expensive as it can be based upon the shape of the 3D mesh it is applied to, therefore, the more complex the mesh, the more detailed and precise the Collider will be, and more computationally expensive it will become. However, as shown in the car tutorial example earlier, it is possible to assign a simpler mesh than that which is rendered, in order to create simpler and more efficient mesh Colliders.

The following diagram illustrates the various types and subtypes of Colliders:



Primitive and mesh shape Colliders take a different amount of CPU to be calculated. The Sphere Collider is the faster one, while the Mesh Collider is the one to be calculated slower. The Sphere Collider is just a radius around a point in the world space to be calculated ,while the Mesh Collider has usually an higher poly count hence is the most expensive Collider.

For example, in a ten-pin bowling game, a simple Sphere Collider will surround the ball, while the pins themselves will have either a simple Capsule Collider, or for a more realistic collision, employ a Mesh Collider, as this will be shaped the same as the 3D mesh of the pin. On impact, the Colliders of any affected objects will report to the physics engine, which will dictate their reaction, based on the direction of impact, speed, and other factors.

In this example, employing a **MeshCollider** to fit exactly to the shape of the pin model would be more accurate but is more expensive in processing terms. This simply means that it demands more processing power from the computer, the cost of which is reflected in slower performance, and hence, the term expensive.

Softbody physics

Even though Unity 5 does not have a big set of tools for Softbody physics, it provides a Cloth component that is used to emulate cloth-like physics behavior. This component is used to simulate clothing, flags and other items.

The Cloth component

The Cloth solution used in previous versions of Unity was quite expensive. In the latest version Cloth does not react to all Colliders in a scene. It also does not apply forces back to the world. Instead, we have a faster, multithreaded, more stable character clothing solution.

Therefore you need to create specific Colliders to make sure that the Cloth reacts correctly to the world. But even in that scenario the simulation is still one-way. The Cloth element does not apply forces back to the Colliders.

At the moment, the Cloth component is best used to simulate clothing or flags, but it´s not really prepared to be used as a full Softbody physics solution.

Getting to grips with 2D in 3D

Unity, in its latest versions, started to provide 2D specific tools to create 2D and 2.5D games.

Game developers have been using Unity for lots of years to create 2D games by using a very simple strategy, ignoring the *Z*-axis.

Ignoring one axis

As we discussed previously, Unity uses 3 numbers to represent positions in the 3D world. To work with 2D, we just need to ignore one of them. If we would need to create scripts to handle the physics we would only need to keep the Z number constant. So if you would want an object to change from point `(0,0)` to `(0,10)`, you would basically tell Unity to transform the object from `(0,0,0)` to `(0,10,0)`, where Z is constant as 0.

Since Unity now provides a whole set of tools for 2D games, you can define it as 2D when you create the project. If you create a 2D project, the UI and other elements are already configured to work only in 2D.

You can actually configure a 3D project to be 2D, by selecting the camera as Orthographic and setting the view as 2D. Unity also provides 2D specific physics for 2D games.

Understanding Sprites

Sprites are 2D graphic objects used to retain images. It can be used to hold characters, props and even scene objects. Sprites can also be animated. Unity uses its animation system to exchange Sprites and create the animation. Animation frames can be inserted in a single Sprite and broken up in several images to be animated.

Sprites can also be used to create an atlas. Atlas is a big image that contains several Sprites. These are compressed into one image to lower texture footprint and memory usage. Mobile games can gain a lot by using an atlas to hold all the visual elements and GUI objects.

Using the Canvas

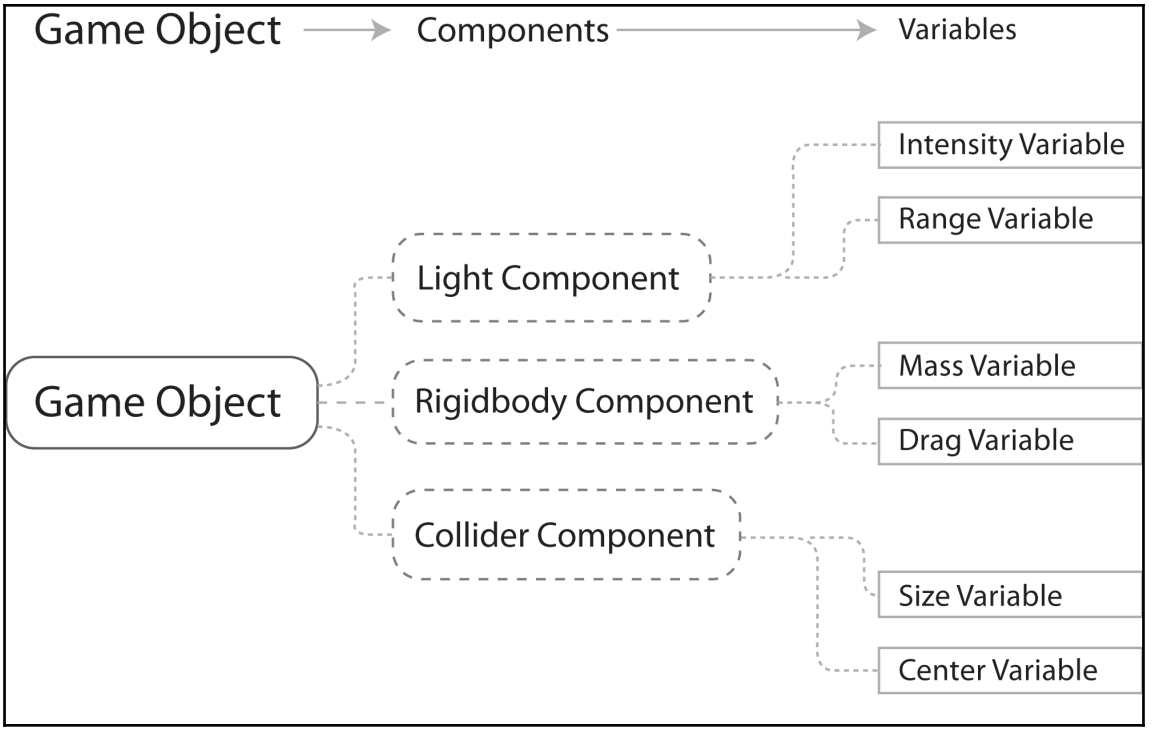
Unity now uses a new UI system and the basis of that system is the Sprite component. Elements can be displayed as Sprites, but to be visualized on top of the main camera, the UI elements need to be inserted into a Canvas.

The Canvas is used to hold all the UI elements and to give them specific properties. The canvas can be ordered, so you can have lots of layered elements. You can also change the render mode to be in screen or world space. In screen space the objects are placed on top of the camera view. In world space the objects are positioned within the 3D/2D world. The

world space render mode is great to create text elements and energy bars that can be

Essential Unity concepts

Unity makes the game production process simple by giving you a set of logical steps to build any conceivable game scenario. Renowned for being non-game-type specific, Unity offers you a blank canvas and a set of consistent procedures to let your imagination be the limit of your creativity. By establishing the use of the **GameObject** concept, you are able to break down parts of your game into easily manageable objects, which are made of many individual **Component** parts. By making individual objects within the game-introducing functionality to them with each component you add, you are able to infinitely expand your game in a logical progressive manner. Component parts in turn have **Variables**-essentially properties of the component, or settings to control them with. By adjusting these Variables, you'll have complete control over the effect that Component has on your object. The following diagram illustrates this:



Structure scheme of a Unity GameObject

In the following screenshot we can see a GameObject with a **Light** component, as seen in the Unity interface:

Now let's look at how this approach would be used in a simple gameplay context.

The Unity way – an example

If we wished to have a bouncing ball as part of a game, then we would begin with a Sphere. This can quickly be created from the Unity menus, and will give you a new GameObject with a Sphere mesh (the 3D shape itself). Unity will automatically add a Renderer component to make it visible. Having created this, we can then add a Rigidbody component. A Rigidbody (Unity refers to most two-word phrases as a single word term) is a component which tells Unity to apply its physics engine to an object. With this comes properties such as mass, gravity, drag, and also the ability to apply forces to the object, either when the player commands it or simply when it collides with another object.

Our sphere will now fall to the ground when the game runs, but how do we make it bounce? This is simple! The Collider component has a variable called **Physic Material**, which is a setting for the physics engine, defining how it will react to other objects' surfaces. Here we can select **Bouncy**-a ready-made Physic Material provided by Unity as part of an importable package and voila! Our bouncing ball is complete in only a few clicks.

This streamlined approach for the most basic tasks, such as the previous example, seems pedestrian at first. However, you'll soon find that by applying this approach to more complex tasks, they become very simple to achieve. Here is an overview of some further key Unity concepts you'll need to know as you get started.

Assets

These are the building blocks of all Unity projects. From textures in the form of image files, through 3D models for meshes, and sound files for effects, Unity refers to the files you'll use to create your game as assets. This is why in any Unity project folder all files used are stored in a child folder named `Assets`. This `Assets` folder is mirrored in the **Project** panel of the Unity interface, see *Theinterface* section in this chapter.

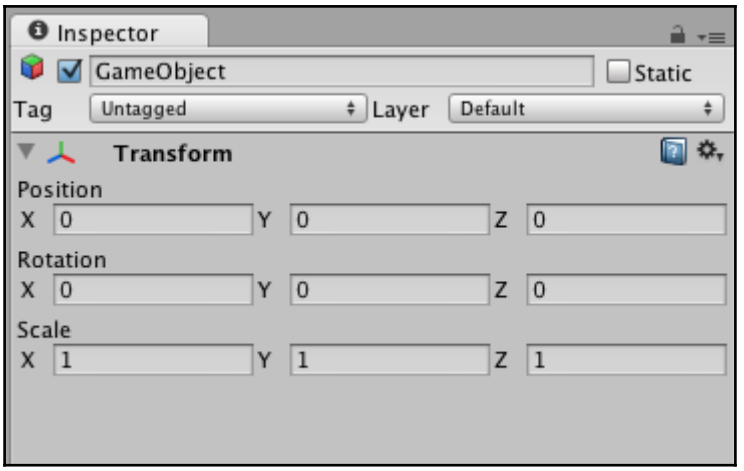
Scenes

In Unity, you should think of scenes as individual levels, or areas of game content. However some developers create entire games in a single scene, such as puzzle games, by dynamically loading content through the code. By constructing your game with many scenes, you'll be able to distribute loading times and test different parts of your game individually. New scenes are often used separately to a game scene you may be working on, in order to prototype or test a piece of potential gameplay.

Any currently open scene is what you are working on, as no two scenes can be worked on simultaneously. Scenes can be manipulated and constructed by using the **Hierarchy** and **Scene** views.

GameObjects

Any active object in the currently open scene is called a GameObject. Certain assets taken from the **Project** panel such as `Models` and `Prefabs` become assets when placed (or 'instantiated') into the current scene. Other objects such as particle systems and primitives can be placed into the scene by using the **Create** button on the **Hierarchy** or by using the **GameObject** menu at the top of the interface. All GameObjects contain at least one component to begin with, that is, the **Transform** component. **Transform** simply tells the Unity engine the **Position**, **Rotation**, and **Scale** of an object, all described in **X**, **Y**, **Z** coordinate (or in the case of scale, dimensionally) order. In turn, the component can then be addressed in scripting in order to set an object's position, rotation, or scale. From this initial component, you will build upon GameObjects with further components adding required functionality to build every part of any game scenario you can imagine. In the following screenshot, you can see the most basic form of a **GameObject**, as shown in the **Inspector** panel:



GameObjects can also be nested in the **Hierarchy**, in order to create the parent-child relationships mentioned previously.

Components

Components come in various forms. They can be for creating behavior, defining appearance, and influencing other aspects of an object's function in the game. By attaching components to an object, you can immediately apply new parts of the game engine to your object. Common components of game production come built-in with Unity, such as the Rigidbody component mentioned earlier, down to simpler elements such as lights, cameras, particle emitters, and more. To build further interactive elements of the game, you'll write scripts, which are also treated as components in Unity. Try to think of scripts as something that extends or modifies the existing functionality available in Unity or creates behavior with the Unity scripting classes provided.

Scripts

While being considered by Unity to be components, scripts are an essential part of game production, and deserve a mention as a key concept. In this book, we will write our scripts in both C Sharp (More often written as C#) and JavaScript. You should also be aware that Unity offers you the opportunity to write in Boo (a derivative of the Python language).We have chosen to primarily focus on C# and JavaScript as these are the two main languages used by Unity developers, and Boo is not supported for scripting on mobile devices, due to which it is not advised to begin learning Unity scripting with Boo.

Unity does not require you to learn how the coding of its own engine works or how to modify it, but you will be utilizing scripting in almost every game scenario you develop. The beauty of using Unity scripting is that any script you write for your game will be straightforward enough after a few examples, as Unity has its own built-in Behaviour class called `MonoBehaviour`-a set of scripting instructions for you to call upon. For many new developers, getting to grips with scripting can be a daunting prospect, and one that threatens to put off new Unity users who are more accustomed to design. If this is your first attempt at getting into game development, or you have no experience in writing code, do not worry. We will introduce scripting one step at a time, with a mind to showing you not only the importance, but also the power of effective scripting for your Unity games.

To write scripts, you'll use Unity's standalone script editor, MonoDevelop. This separate application can be found in the Unity application folder on your PC or Mac and will be launched every time you edit a new script or an existing one. Amending and saving scripts in the script editor will immediately update the script in Unity as soon as you switch back to Unity. You may also designate your own script editor in the Unity preferences if you wish to, such as Visual Studio. MonoDevelop is recommended however, as it offers autocompletion of code as you type and is natively developed and updated by Unity Technologies.

Prefabs

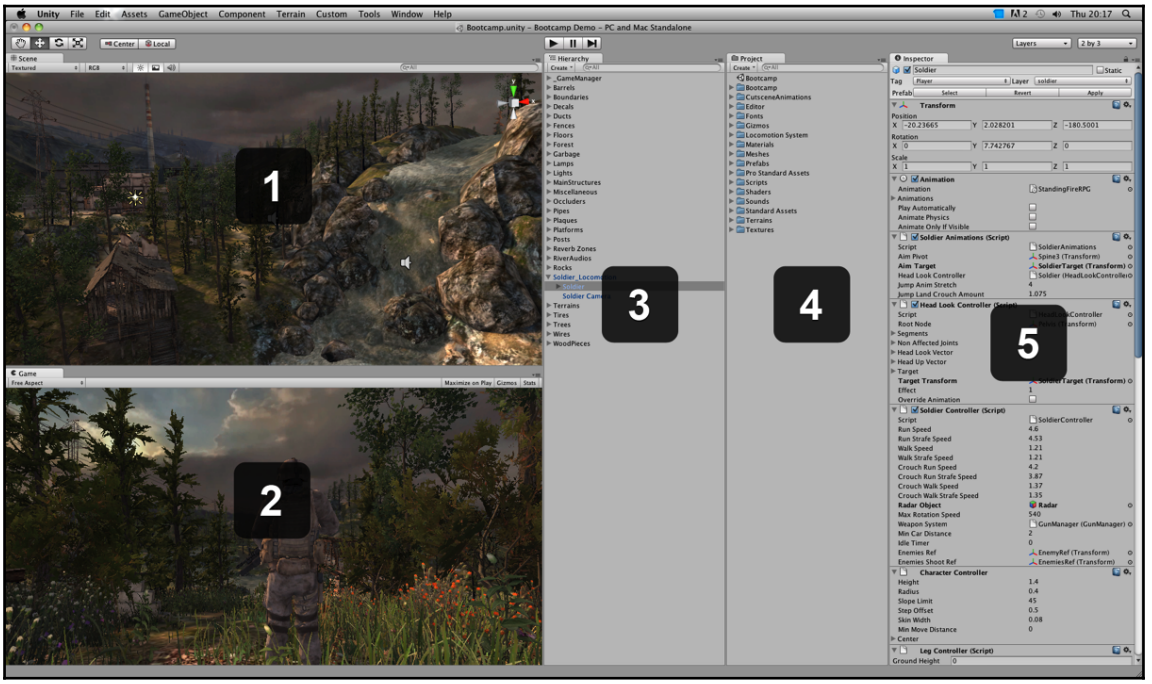
Unity's development approach hinges around the **GameObject** concept, but it also has a clever way to store objects as assets to be reused in different parts of your game, and then instantiated (also known as 'spawning' or 'cloning') at any time. By creating complex objects with various components and settings, you'll be effectively building a template for something you may want to spawn multiple instances of (hence 'instantiate'), with each instance then being individually modifiable.

Consider a crate as an example. You may have given the object in the game a mass, and written scripted behaviors for its destruction. Chances are you'll want to use this object more than once in a game, and perhaps even in games other than the one it was designed for.

Prefabs allow you to store the object, complete with components and current configuration. Comparable to the *MovieClip* concept in Adobe Flash, think of prefabs simply as empty containers that you can fill with objects to form a data template you'll likely recycle.

The interface

The Unity interface, like many other working environments, has a customizable layout. Consisting of several dockable spaces, you can pick which parts of the interface appear where. Let's take a look at a typical Unity layout:



This layout can be achieved by going to **Window** | **Layouts** | **2 by 3 in Unity**.

As the preceding screenshot demonstrates (Mac version shown), there are five different panels or views you'll be dealing with which are as follows:

- **Scene [1]**: Where the game scene is constructed.
- **Game [2]**: The preview window, active only in play mode.
- **Hierarchy [3]**: A list of GameObjects in the scene.
- **Project [4]**: A list of your project's assets, acts as a library.
- **Inspector [5]**: Settings for currently selected asset/object/setting.

The Scene view and Hierarchy

The **Scene** view is where you will build the entirety of your game project in Unity. This window offers a perspective (full 3D) view, which is switchable to Orthographic (top-down, side-on, and front-on) views. When working in one of the Orthographic views, rotating the view will display the scene isometrically. The **Scene** view acts as a fully rendered 'Editor' view of the game world you build. Dragging an asset to this window (or the Hierarchy) will create an instance of it as a **GameObject** in the Scene.

The **Scene** view is tied to the **Hierarchy**, which lists all GameObjects in the currently open scene in ascending alphabetical order.

Control tools

The **Scene** window is also accompanied by four useful control tools, as shown in the following screenshot.

Accessible from the keyboard using keys Q, W, E, and R, these keys perform the following operations:

- **The Hand tool (**Q**)**: This tool allows navigation of the **Scene** window. It allows you to drag around in the **Scene** window with the left mouse button to pan your view. Holding down Alt with this tool selected will allow you to orbit your view around a central point you are looking at, and holding the Ctrl key will allow you to zoom, as will scrolling the mouse wheel. Holding the Shift key down will speed up both of these functions.
- **The Translate tool [**W**]**: This is your active selection tool. As you can completely interact with the **Scene** window, selecting objects either in the **Hierarchy** or **Scene** means you'll be able to drag the object's axis handle in order to reposition them.
- **The Rotate tool [**E**]**: This works in the same way as Translate, using visual 'handles' to allow you to rotate your object around each axis.
- **The Scale tool [**R**]**: This tool also works as the Translate and Rotate tools. It adjusts the size or scale of an object using visual handles.

Having selected objects in either the **Scene** or **Hierarchy**, they immediately get selected in both. Selection of objects in this way will also show the properties of the object in the **Inspector**. Given that you may not be able to see an object you've selected in the **Hierarchy**

in the **Scene** window, Unity also provides the use of the F key, to focus your **Scene** view on that object. Simply select an object from the **Hierarchy**, hover your mouse cursor over the **Scene** window, and press F.

Flythrough scene navigation

To move around your Scene view using the mouse and keys you can use Flythrough mode. Simply hold down the right mouse button and drag to look around in first-person style. You can use W, A, S and D to move and Q and E to descend and ascend respectively.

Control bar

In addition to the control tools, there is also a bar of additional options to help you work with your Unity scenes which is shown as follows:

Known as the Scene view's control bar, this bar allows you to adjust (left to right):

- **Draw mode**: The default is set to **Textured**.
- **Render mode**: The default is set to **RGB**.
- **Toggle scene lighting**: This can be switched ON/OFF by clicking on it.
- **Toggle overlays**: Shows and hides GUI elements and skyboxes and toggles the 3D grid.
- **Toggle audition mode**: Previews audio sources in the current scene.
- **Gizmos**: Use this pop-out menu to show or hide gizmos, the 2D icons of cameras, lights, and other components shown in the Scene view to help you instantly visualize the different type of objects.

Search box

While the **Scene** view is intrinsically linked with the **Hierarchy**, often you may need to locate an item or type of item in the **Scene** view itself by searching. Simply type the name or data type (in other words, an attached component) of an object into the search, and the **Scene** view will *grey* out other objects in order to highlight the item you have searched for. This becomes very useful when dealing with more complex scenes, and should be used in conjunction with F on the keyboard to focus on the highlighted object in the **Scene** window itself.

Create button

As many of the game assets you'll use in Unity will be created by the editor itself, the **Hierarchy** has a **Create** button that allows you to create objects that are also located within the top **GameObject** menu. Similar to the **Create** button on the **Project** panel, this drop-down menu creates items and immediately selects them so that you may rename or begin working with them in the **Scene** or **Inspector**.

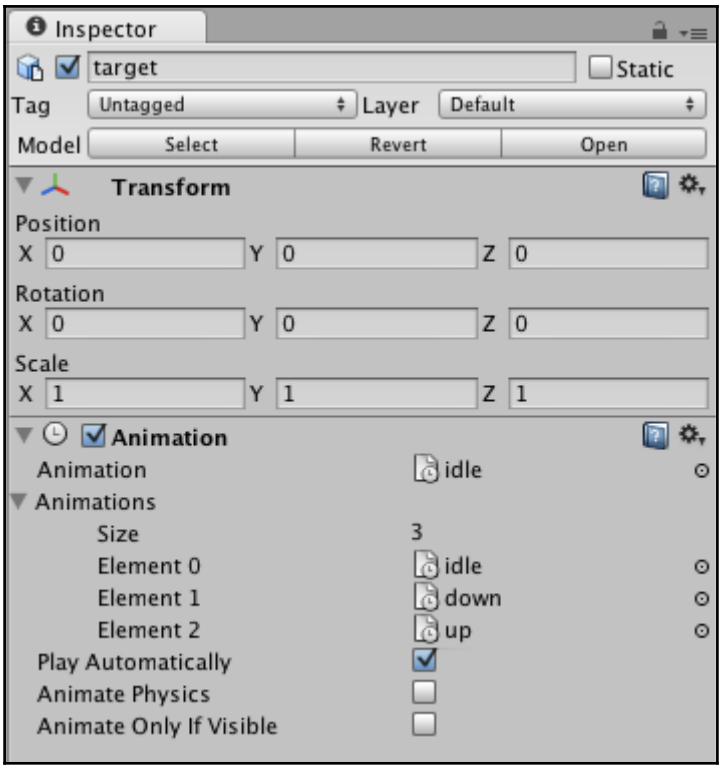
You can create object also from the main menu

The Inspector

Think of the **Inspector** as your personal toolkit to adjust every element of any GameObject or asset in your project. Much like the *Property Inspector* concept utilized by Adobe in Flash and Dreamweaver, this is a context-sensitive window. All this means is that whatever you select, the **Inspector** will change to show its relevant properties, which makes it sensitive to the context in which you are working.

The **Inspector** will show every component part of anything you select, and allow you to adjust the variables of these components, using simple form elements such as text input boxes, slider scales, buttons, and drop-down menus. Many of these variables are tied into Unity's drag-and-drop system, which means that rather than selecting from a drop-down menu, if it is more convenient, you can drag and drop to choose settings or assign properties.

This window is not only for inspecting objects. It will also change to show the various options for your project when choosing them from the **Edit** menu, as it acts as an ideal space to show you preferences-changing back to showing component properties as soon as you reselect an object or asset. The following screenshot shows the **Inspector** window:



In the preceding screenshot, the **Inspector** is showing properties for a **target** object in the game. The object itself features two components, namely, **Transform** and **Animation**. The **Inspector** will allow you to make changes to settings in either of them. Also note that in order to temporarily disable any component at any time, which will become very useful for testing and experimentation, you can simply deselect the checkbox to the left of the component's name. Likewise, if you wish to switch off an entire object at a time, then you may deselect the checkbox next to its name at the top of the **Inspector** window.

The Project window

The Project window is a direct view of the `Assets` folder of your project. Every Unity project is made up of a parent folder, containing three subfolders-`Assets`, `Library`, and while the Unity Editor is running, a `Temp` folder. Placing assets into the `Assets` folder means you'll immediately be able to see them in the **Project** window, and they'll also be automatically imported into your Unity project. Likewise, changing any asset located in the `Assets` folder, and resaving it from a third-party application, such as Photoshop, will cause

Unity to reimport the asset, reflecting your changes immediately in your project and any active scenes that use that particular asset.

Asset management

It is important to remember that you should only alter asset locations and names using the **Project** window-using Finder (Mac) or Windows Explorer (PC) to do so may break connections in your Unity project. Therefore, to relocate or rename objects in your `Assets` folder, use Unity's **Project** window instead of your operating system.

The **Project** window, like the **Hierarchy** is accompanied by a **Create** button. This allows the creation of any assets that can be made within Unity, for example, scripts, prefabs, and materials.

The Game View

The Game View is invoked by pressing the **Play** button and acts as a realistic test of your game. It also has settings for screen ratio, which will come in handy when testing how much of the player's view will be restricted in certain ratios, such as 4:3 (as opposed to wide) screen resolutions. Having pressed **Play**, it is crucial that you bear in mind the following advice:

Play mode – testing only!

In play mode, the adjustments you make to any parts of your game scene are merely temporary. It is meant as a testing mode only, and when you press **Play** again to stop the game, all changes made to active GameObjects during Play Mode will be undone. This can often trip up new users, so don't forget about it!

The Game View can also be set to **Maximize** when you invoke play mode, giving you a better view of the game at nearly fullscreen-the window expands to fill the interface. It is worth noting that you can expand any part of the interface in this way, simply by hovering over the part you wish to expand and pressing the space bar.

In addition to using **Play** to preview your game, the live game mode can also be paused by pressing the **Pause** button at the top of the interface, and play can be advanced a frame at a time using the third **AdvanceFrame** button next to **Pause**. This is useful when Debugging-the process of finding and solving problems or 'bugs' with your game development.

Summary

Here we have looked at the key concepts you will need to understand and complete the exercises in this book. However, 3D is a detailed discipline that you will continue to learn not only with Unity, but in other areas as well. With this in mind, you are recommended to continue to read more on the topics discussed in this chapter, in order to supplement your study of 3D development. Each individual piece of software you encounter will have its own dedicated tutorials and resources dedicated to learning it. If you wish to learn 3D artwork to complement your work in Unity, you are recommended to familiarize yourself with your chosen package, after researching the list of tools that work with the Unity pipeline and choosing which one suits you the best.

Now that we've taken a brief look at 3D concepts and the processes used by Unity to create games, we'll begin by completing a simple exercise before getting started on the larger game element of this book.

In the following chapter, we'll begin with a short exercise in which you will prototype a simple game mechanic using primitive shapes and some basic coding to get you started in C#/JavaScript. It is important to kick start your Unity learning with a simple example, using primitives as you will often find yourself prototypinggame ideas in this manner, once you feel more comfortable in using Unity.

Let's get started!

2
Prototyping and Scripting Basics

When starting out in game development, one of the best ways to learn the various parts of the discipline is to prototype your idea. Unity excels in assisting you with this, with its visual scene editor and public member variables that form settings in the **Inspector**. To get to grips with working in the Unity editor, we'll begin by prototyping a simple game mechanic using primitive shapes and basic coding.

In this chapter, you will learn about:

- Creating a **New Project** in Unity
- Importing asset packages
- Working with GameObjects in the **Scene** view and **Hierarchy**
- Adding materials
- Writing C# and JavaScript (UnityScript)
- Variables, functions and commands
- Using the `Translate()` command to move objects
- Using prefabs to store objects
- Using the `Instantiate()` command to spawn objects

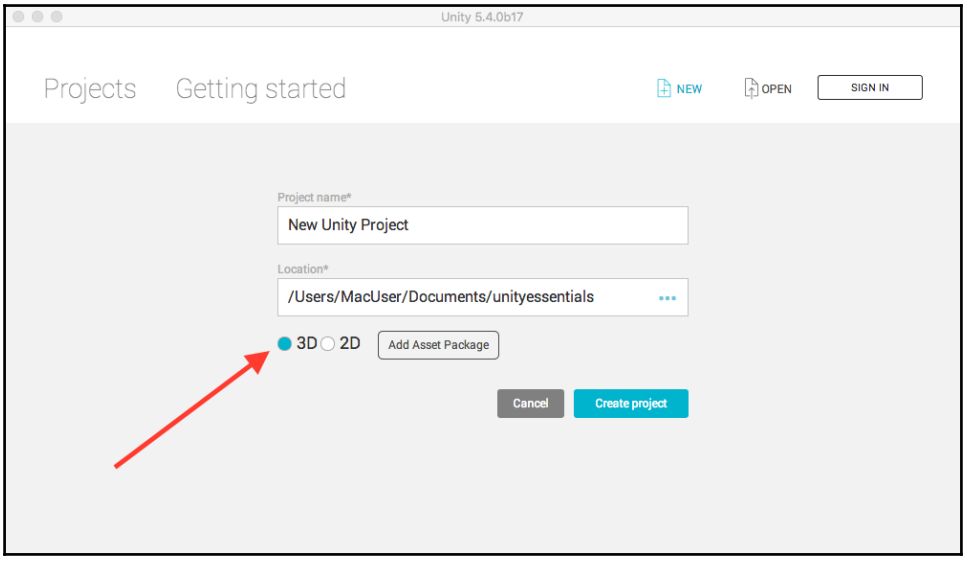
Your first Unity project

As Unity comes in two main forms-a standard, free download and a paid Pro developer license. We'll stick to using features that users of the standard free edition will have access to.

If you're launching Unity for the very first time, you'll be presented with a Unitydemonstration project. While this is useful to look into the best practices for the development of high-end projects, if you're starting out, looking over some of the assets and scripting may feel daunting, so we'll leave this behind and start from scratch!

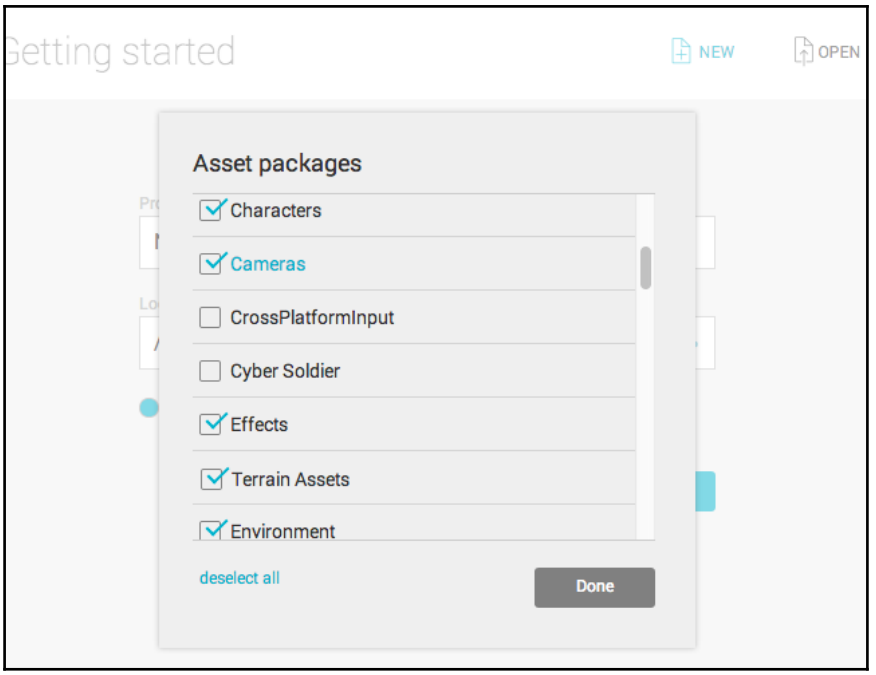
Take a look at the following steps for setting up your Unity project:

1. In Unity go to **File** | **NewProject** and you will be presented with the **ProjectWizard.** The following screenshot is a Mac version shown:



From here select the **NEW** tab and **3D** type of project.

Be aware that if at any time you wish to launch Unity and be taken directly to the **Project Wizard**, then simply launch the Unity Editor application and immediately hold the Alt key (Mac and Windows). This can be set to the default behavior for launch in the Unity preferences.



2. Click the **Set** button and choose where you would like to save your new Unity project folder on your hard drive. The new project has been named **UGDE** after this book, and chosen to store it on my desktop for easy access.

   The **Project Wizard** also offers the ability to import many **Assetpackages** into your new project which are provided free to use in your game development by Unity Technologies. Comprising scripts, ready-made objects, and other artwork, these packages are a useful way to get started in various types of new project. You can also import these packages at any time from the **Assets** menu within Unity, by selecting **ImportPackage**, and choosing from the list of available packages. You can also import a package from anywhere on your hard drive by choosing the **CustomPackage** option here. This import method is also used to share assets with others, and when receiving assets you have downloaded through the **AssetStore**-see **Window** | **Asset Store** to view this part of Unity later.

3. From the list of packages to be imported, select the following (as shown in the previous image)
   - **Characters**

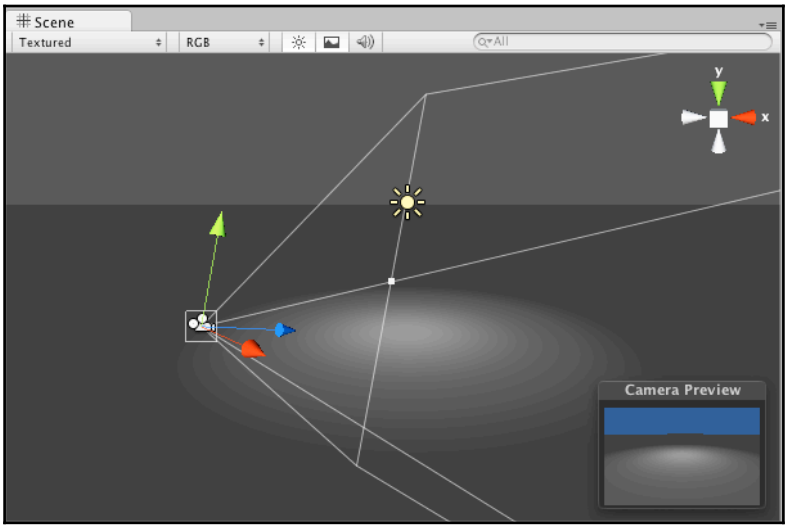
- **Cameras**
- **Effects**
- **Terrain Assets**
- **Environment**

4. When you are happy with your selection, simply choose **Create Project** at the bottom of this dialog window. Unity will then create your new project and you will see progress bars representing the import of the four packages.

A basic prototyping environment

To create a simple environment, in which to prototype some game mechanics, we'll begin with a basic series of objects with which to introduce gameplay that allows the player to aim and shoot at a wall of primitive cubes.

When complete, your prototyping environment will feature a floor comprised of a cube primitive, a main camera through which to view the 3D world and a Point Light setup to highlight the area where our gameplay will be introduced. It will look like something as follows:

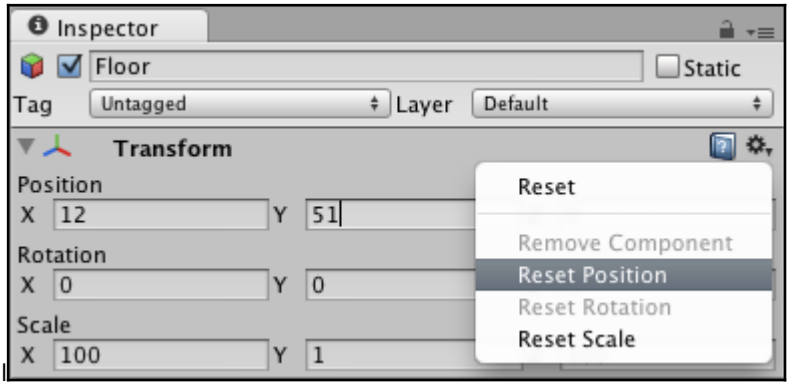


Setting the scene

As all new scenes come with a Main Camera object by default, we'll begin by adding a floor for our prototyping environment.

On the Hierarchy panel, click the **Create** button, and from the drop-down menu, choose **Cube**. The items listed in this drop-down menu can also be found in the **GameObject** | **CreateOther** top menu. You will now see an object in the Hierarchy panel called cube. Select this and press Return (Mac) /F2 (PC) or double-click the object name slowly (both platforms) to rename this object, type in **Floor** and press Return (both platforms) to confirm this change.

For consistency's sake, we will begin our creation at world zero-the center of the 3D environment we are working in. To ensure that the floor cube you just added is at this position, ensure it is still selected in the **Hierarchy** and then check the **Transform** component on the **Inspector** panel, ensuring that the position values for **X**, **Y**, and **Z** are all at 0, if not, change them all to zero either by typing them in or by clicking the **Cog** icon to the right of the component, and selecting **ResetPosition** from the pop-out menu. Check out the following screenshot:



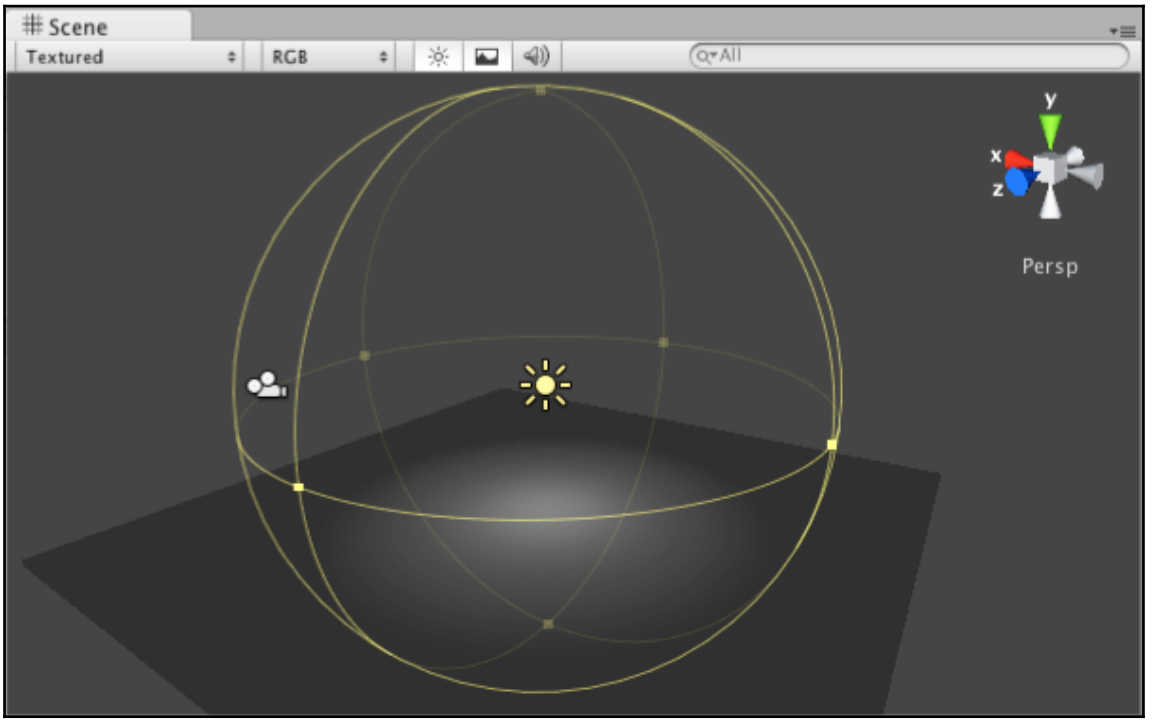
Next, we'll turn the cube into a floor, by stretching it out in the X and Z axes. Into the **X** and **Z** values under **Scale** in the **Transform** component, type a value of **100**, leaving **Y** at a value of **1**.

Adding simple lighting

Now we will highlight part of our prototyping floor by adding a Point Light. Select the **Create** button on the **Hierarchy** (or go to **Game Object** | **Create Other**) and choose **Point**

Light. Position the new Point Light at (**0, 20, 0**) using the **Position** values in the **Transform** component, so that it is **20** units above the floor.

You will notice that this means that the floor is out of range of the light, so expand the **Range** by dragging on the yellow dot handles that intersect the outline of the Point Light in the **Scene** view, until the value for **Range** shown in the **Light** component in the **Inspector** reaches something around a value of **40**, and the light is creating a lit part of the floor object.



Bear in mind that most components and visual editing tools in the **Scene** view are inextricably linked, so altering values such as **Range** in the Inspector **Light** component will update the visual display in the **Scene** view as you type, and stay constant as soon as you press **Return** to confirm the values entered.

Another brick in the wall

Now let's make a wall of cubes that we can launch a projectile at. We'll do this by creating a single master brick, adding components as necessary, and then duplicating this until our

wall is complete.

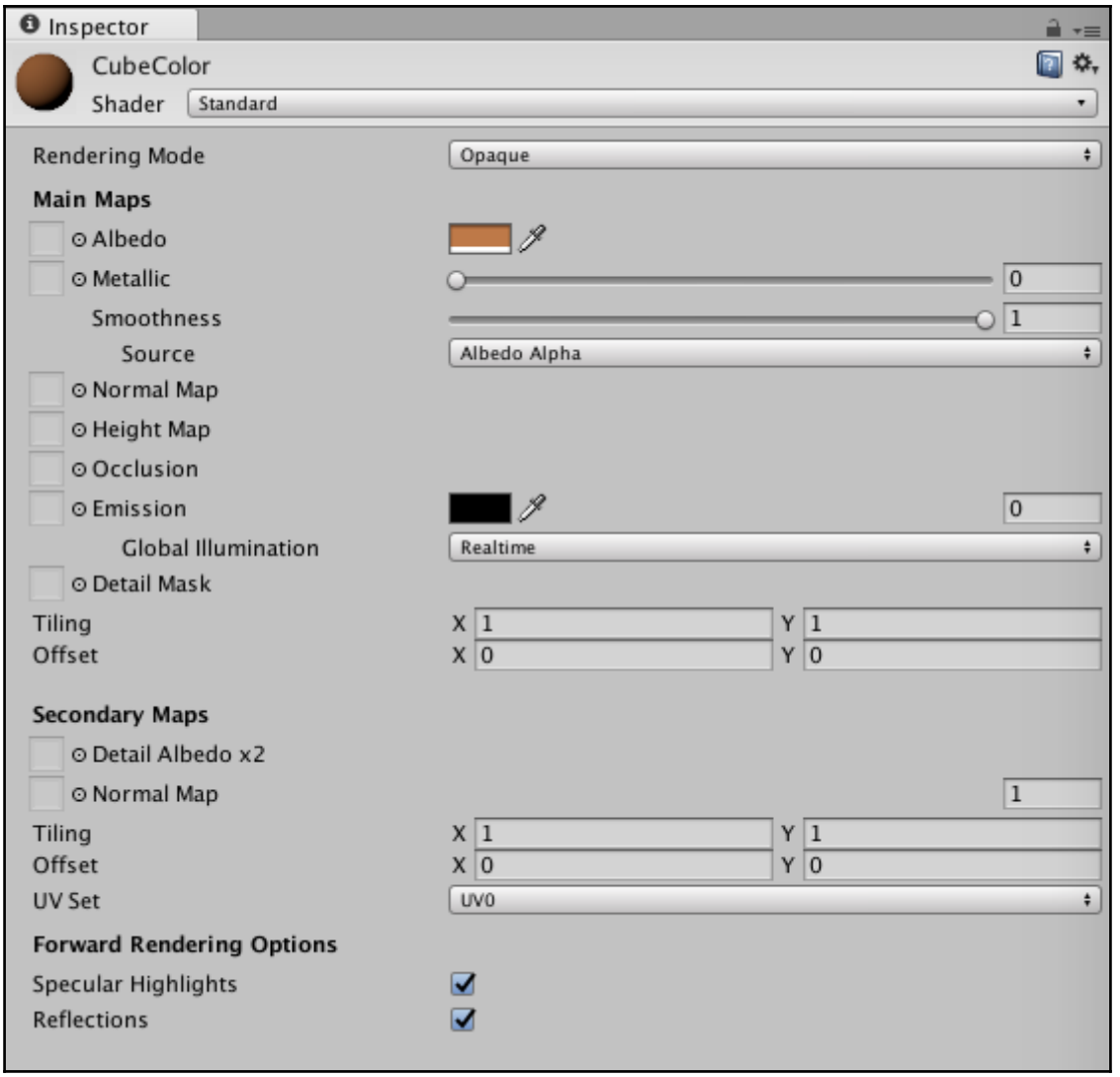
Building the master brick

In order to create a template for all of our bricks, we'll start by creating a master object, something to create clones of. This is done as follows:

1. Click the **Create** button at the top of the **Hierarchy**, and select **Cube**. Position this at **(0, 1, 0)** using the **Position** values in the **Transform** component on the **Inspector**. Then, focus your view on this object by ensuring it is still selectedin the **Hierarchy**, hovering your cursor over the **Scene** view, and pressing *F*.
2. Add physics to your **Cube** object by choosing **Component** | **Physics** | **Rigidbody** from the top menu. This means that your object is now a Rigidbody-it has mass, gravity, and is affected by other objects using the physics engine for realistic reactions in the 3D world.
3. Finally, we'll color this object by creating a **Material**. Materials are a way of applying color and imagery to our 3D geometry. To make a new one, go to the **Create** button on the **Project** panel and choose **Material** from the drop-down menu. Press Return (Mac) or F2 (PC) to rename this asset to **Red** instead of the default name **New Material**.

You can also right-click in the "Materials" Project folder and **Create** | **Material** or alternatively you can use the editor main menu: **Assets** | **Create** | **Material**

4. With this material selected, the **Inspector** shows its properties. Click on the color block to the right of **MainColor** [see image label 1] to open the **Color Picker**[see image label 2]; this will differ in appearance depending uponwhether you are using Mac or PC. Simply choose a shade of red, and then close the window. The **Main Color** block should now have been updated.



5. To apply this material, drag it from the **Project** panel and drop it onto either the cube as seen in the **Scene** view, or onto the name of the object in the Hierarchy. The material is then applied to the Mesh Renderer component of this object and immediately seen following the other components of the object in the **Inspector**. Most importantly, your cube should now be red! Adjusting settings using the **Preview** of this material on any object will edit the original asset, as this preview is simply a link to the asset itself, not a newly editable instance.
6. Now that our cube has a color and physics applied through the **Rigidbody**

component, it is ready to be duplicated and act as one brick in a wall of many, before we do that however, lets have a quick look at the physics in action. With the cube still selected, set the **Y Position** value to **15** and the **X Rotation** value to **40** in the **Transform** component in the **Inspector**. Press **Play** at the top of the Unity interface and you should see the cube fall and then settle, having fallen at an angle.

The shortcut for **Play** is Ctrl+P [PC] Command+P [Mac])

7. Press **Play** again to stop testing. Do not press **Pause** as this will only temporarily halt the test, and changes made thereafter to the scene will not be saved.
8. Set the **Y Position** value for the cube back to **1**, and set the **X Rotation** back to .

Now that we know our brick behaves correctly, let's start creating a row of bricks to form our wall.

And snap!-It's a row

To help you position objects, Unity allows you to snap to specific increments when dragging-these increments can be redefined by going to **Edit** | **Snap Settings**.

To use snapping, hold down Command (Mac) or Control (PC) when using the **Translate**tool (*W*) to move objects in the **Scene** view. So in order to start building thewall, duplicate the cube brick we already have using the shortcut Command+D (Mac) or Control+D (PC), then drag the red axis handle while holding the snapping key. This will snap by one unit at a time by default, so snap-move your cube one unit in the **X** axis to that it sits next to the original cube, shown as follows:

Repeat this procedure of duplication and snap-dragging until you have a row of 10 cubes in a line. This is the first row of bricks, and to simplify building the rest of the bricks we will now group this row under an empty object, and then duplicate the parent empty object.

Vertex snapping

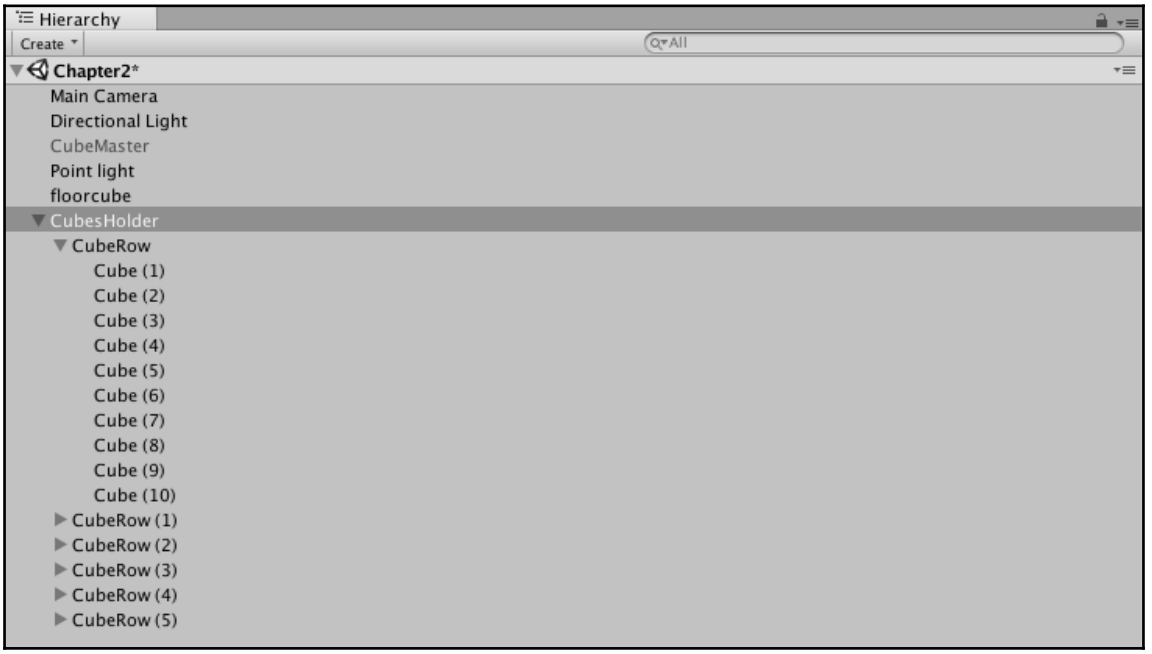
The basic snapping technique used here works well as our cubes are a generic scale of 1, but when scaling more detailed shaped objects, you should use vertex snapping instead. To do this, ensure that the **Translate** tool is selected and hold down *V* on the keyboard; now hover your cursor over a vertex point on your selected object and drag to any other vertex of

another object to snap to it.

Grouping and duplicating with empty objects

Create an empty object by choosing **GameObject** | **Create Empty** from the top menu, then position this at **(4.5, 0.5, -1)** using the **Transform** component in the **Inspector**. Rename this from the default name**GameObject**to**CubeHolder**.

Now select all of the cube objects in the **Hierarchy** by selecting the top one, holding the Shift key, and then selecting the last. Now drag this list of cubes in the Hierarchy onto the empty object named **CubeHolder** in the **Hierarchy** in order to make this their parent object; the **Hierarchy** should now look like this:



You'll notice that the parent empty object now has an arrow to the left of its object title, meaning you can expand and collapse it. To save space in the **Hierarchy**, click the arrow now to hide all of the child objects, and then re-select the **CubeHolder**.

Now that we have a complete row made and parented, we can simply duplicate the parent object, and use snap-dragging to lift a whole new row up in the Y axis. Use the duplicate shortcut (Command/Control + D) as before, then select the **Translate** tool (*W*) and use the

snap-drag technique (hold Command on Mac, Control on PC) outlined earlier to lift by **1** unit in the **Y** axis by pulling the green axis handle.

Repeat this procedure to create eight rows of bricks in all, one on top of the other. It should look something like the following screenshot. Note that in the image all **CubeHolder** row objects are selected in the **Hierarchy**.

Build it up, knock it down!

Now that we have built a wall, let's make a simple game mechanic where the player can maneuver the camera and shoot projectiles at the wall to knock it down.

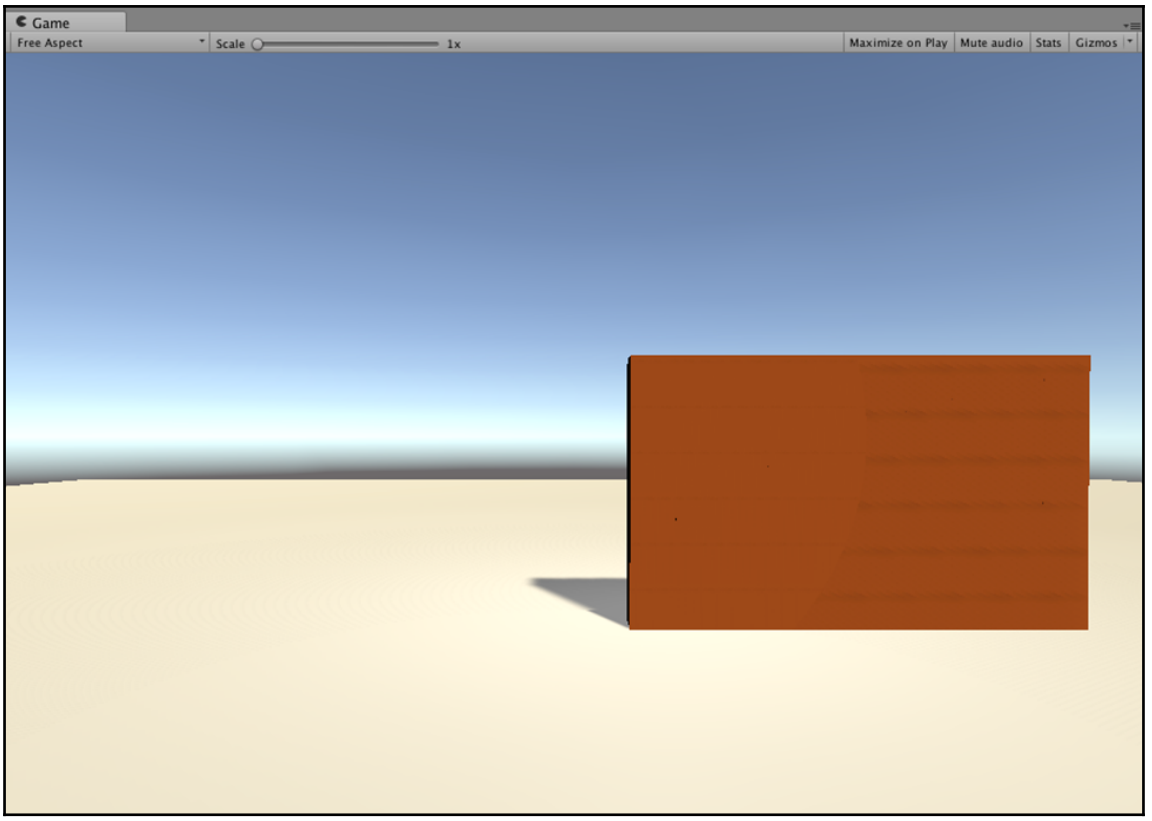
Setting the viewpoint

Set up the camera facing the wall by selecting the **Main Camera** object in the Hierarchy, and positioning it at **(4, 3, -15)** in the **Transform** component. Also ensure that it has no rotation values; they should all be set to .

Introducing scripting

To take your first steps into programming, we will look at a simple example of the same functionality in both C Sharp(C#) and JavaScript, the two main programming languages used by Unity developers.

It is also possible to write Boo based scripts, but these are rarely used outside of those with existing experience in that language.



To follow the next steps you may choose either JavaScript or C#, then in the rest of this book continue with the chosen language your prefer.

To begin, click the **Create** button on the Project panel, then choose either JavaScript or C# Script, or simply press the Add Component button on the Main Camera inspector panel.

Your new script will be placed into the Project panel named **NewBehaviourScript**, and show an icon of a page with either JS or C# written on it. When selecting your new script, Unity offers a preview of what is in the script already, in the view of the Inspector, and an accompanying **Edit** button that when clicked will launch the script into the default script editor-Monodevelop. You can also launch a script in your script editor at any time by double-clicking on its icon in the Project panel.

A new behaviour script or class

Whether choosing C# or JavaScript, it is recommended that you read through both parts of the ensuing section of the book as it contains overall information about scripting and may also help you decide as to which language you want to choose.

New scripts can be thought of as a new class in Unity terms. If you are new to programming, think of a class as a set of actions, properties, and other stored information that can be accessed under the heading of its name.

For example, a class called `Dog` may contain properties such as `color`, `breed`, `size`, or `gender` and have actions such as `roll over` or `fetch stick`. These properties can be described as **variables**, while the actions can be written in **functions**, also known as methods.

In this example, to refer to the `breed` variable-a property of the `Dog` class, we might refer to the class it is in, `Dog`, and use a period (full stop) to refer to this variable in the following way:

```
Dog.breed;
```

If calling a function within the `Dog` class, we might say for example:

```
Dog.fetchStick();
```

We can also add arguments into functions-these aren't the everyday arguments we have with one another! Think of them as more like modifying the behavior of a function, for example,with our `fetchStick` function, we might build in an argument that defines how quickly our dog will fetch the stick. This might be called as follows:

```
Dog.fetchStick(25);
```

While these are abstract examples, often it can help to transpose coding into commonplace examples in order to make sense of them. As we continue in this book, think back to this example or come up with some examples of your own, to help train yourself to understand classes of information and their properties.

When you write a script in C# or JavaScript, you are writing a new class or classes with their own properties (variables) and instructions (functions) that you can call into play at the desired moment in your games.

What's inside a new C# behaviour

When you begin with a new C# script, Unity gives you the following code to get started:

```
using UnityEngine;
using System.Collections;

public class NewBehaviourScript : MonoBehaviour {
  // Use this for initialization
  void Start () {
                }

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

This begins with the necessary two calls to the Unity Engine itself:

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

It goes on to establish the class named after the script. With C# you'll be required to name your scripts with matching names to the class declared inside of the script itself. This is why you will see:

`public class NewBehaviourScript : MonoBehaviour {` at the start of a new C# document, as NewBehaviourScript is the default name that Unity gives to newly generated scripts. If you rename your script in the Project panel when it is created, Unity will rewrite the class name in your C# script.

Code in classes

When writing code, most of your functions, variables, and other scripting elements will be placed within the class of a script in C#. *Within*-in this context-means that it must occur after the classdeclaration, and following the corresponding closing '`}`' of that, at the bottom of the script. So unless told otherwise, while following the instructions in this book, assume that your code should be placed within the class established in the script. In JavaScript this is less relevant as the entire script is the class; it is not explicitly established. See the following section *What's inside a new JavaScript behavior?*

Basic functions

Unity as an engine has many of its own functions that can be used to call different features of the game engine, and it includes two important ones when you create a new script in C#.

Functions or methods as they are also known, most often start with the term `void` in C#. This is the function's return type-which is the kind of data a function may result in. As most functions are simply there to carry out instructions rather than return information, often you will see `void` at the beginning of their declaration, which simply means that a certain type of data will not be returned.

Some basic functions are explained as follows:

- `Start()`: This is called when the scene first launches, so is often used as it is suggested in the code, for initialization. For example, you may have a core variable that must be set to when the game scene begins or perhaps a function that spawns your player character in the correct place at the start of a level.
- `Update()`: This is called in every frame that the game runs, and is crucial for checking the state of various parts of your game during this time, as many different conditions of game objects may change while the game is running.

Variables in C#

To store information in a variable in C#, you will use the following syntax:

```
typeOfData nameOfVariable = value;
```

For example:

```
int currentScore = 5;
```

Or:

```
float currentVelocity = 5.86f;
```

Note that the examples here show numerical data, with `int` meaning **integer**-a whole number, and `float` meaning **floating point**-a number with a decimal place, which in C# requires a letter f to be placed at the end of the value. This syntax is somewhat different from JavaScript. See the following section, *Variables in JavaScript*.

What's inside a new JavaScript behaviour?

While fulfilling the same functions as a C# file, a new empty JavaScript file shows you less as the entire script itself is considered to be the class, and the empty space in the script is considered to be within the opening and closing of the class, as the class declaration itself is hidden.

You will also notice that the lines using UnityEngine; and using System. Collections; are also hidden in JavaScript, so in a new JavaScript you will simply be shown the Update() function:

```
function Update () {

                    }
```

You will notice that in JavaScript, you declare functions differently, using the

term `function` before the name. You will also need to write declaration of variables, and various other scripted elements with a slightly different syntax; we will look at examples of this as we progress.

Variables in JavaScript

The syntax for variables in JavaScript works as follows, and is always preceded by the prefix var:

```
var variableName : TypeOfData = value;
```

For example:

```
var currentScore : int = 0;
```

Or

```
var currentVelocity : float = 5.86;
```

As you will likely have noticed, the `float` value does not require a letter f following its value as it does in C#. You will notice as you see further scripts written in the two different languages that C# often has stricter rules about how scripts are written, especially regarding implicitly stating types of data that are being used.

Comments

In both C# and JavaScript in Unity, you can write in comments using:

```
// two forward slashes symbols for a single line comment
```

Or:

```
/* forward-slash, star to open a multi line comments and at
the end of it,star, forward-slash to close */
```

You may write comments in the code to help you remember what each part does as you progress through the book. Remember that because comments are not executed code, you can write whatever you like, including pieces of code; as long as they are contained within a comment they will never be treated as working code.

Wall attack

Now let's put some of your new scripting knowledge into action and turn our existing scene into an interactive gameplay prototype. In the **Project** panel in Unity, rename your newly created script **Shooter** by selecting it, pressing *Return* (Mac) or *F2* (PC), and typing in the new name.

If you are using C#, remember to ensure that your class declaration inside the script matches this name of the script:

```
public class Shooter : MonoBehaviour {
```

As mentioned previously, JavaScript users will not need to do this. To kick-start your knowledge of using scripting in Unity, we will write a script to control the camera and allow shooting of a projectile at the wall that we have built.

To begin with, we will establish three variables:

- `bullet`: This is a variable of type Rigidbody, as it will hold a reference to a physics controlled object we will make
- `power`: This is a floating point variable number we will use to set the power of shooting
- `moveSpeed`: This is another floating point variable number we will use to define the speed of movement of the camera using the arrow keys

These variables must be **Public Member Variables**, in order for them to display as

adjustable settings in the Inspector. You'll see this in action very shortly!

Declaring public variables

Public variables are important to understand as they allow you to create variables that will be accessible from other scripts-an important part of game development as it allows for simpler inter-object communication. Public variables are also really useful as they appear as settings you can adjust visually in the Inspector once your script is attached to an object. Private variables are the opposite-designed to be only accessible within the scope of the script, class, or function they are defined within, and do not appear as settings in the Inspector.

C#

Before we begin, as we will not be using it, remove the `Start()` function from this script by deleting `void Start(){}`. To establish the required variables, put the following code snippet into your script after the opening of the class, shown as follows:

```
using UnityEngine;

using System.Collections;

public class Shooter : MonoBehaviour {

public Rigidbody bullet; public float power = 1500f;
public float moveSpeed = 2f;

void Update () {

                }
    }
```

Note that in this example, the default explanatory comments and the Start() function have been removed in order to save space.

JavaScript

In order to establish Public Member Variables in JavaScript, you will need to simply ensure that your variables are declared outside of any existing function. This is usually done at the

top of the script, so to declare the three variables we need, add the following to the top of your new **Shooter** script so that it looks like this:

```
var bullet : Rigidbody;
var power : float = 1500;
var moveSpeed : float = 5;

function Update () {

                    }
```

Note that JavaScript (UnityScript) is much less declarative and needs less typing to start

Assigning scripts to objects

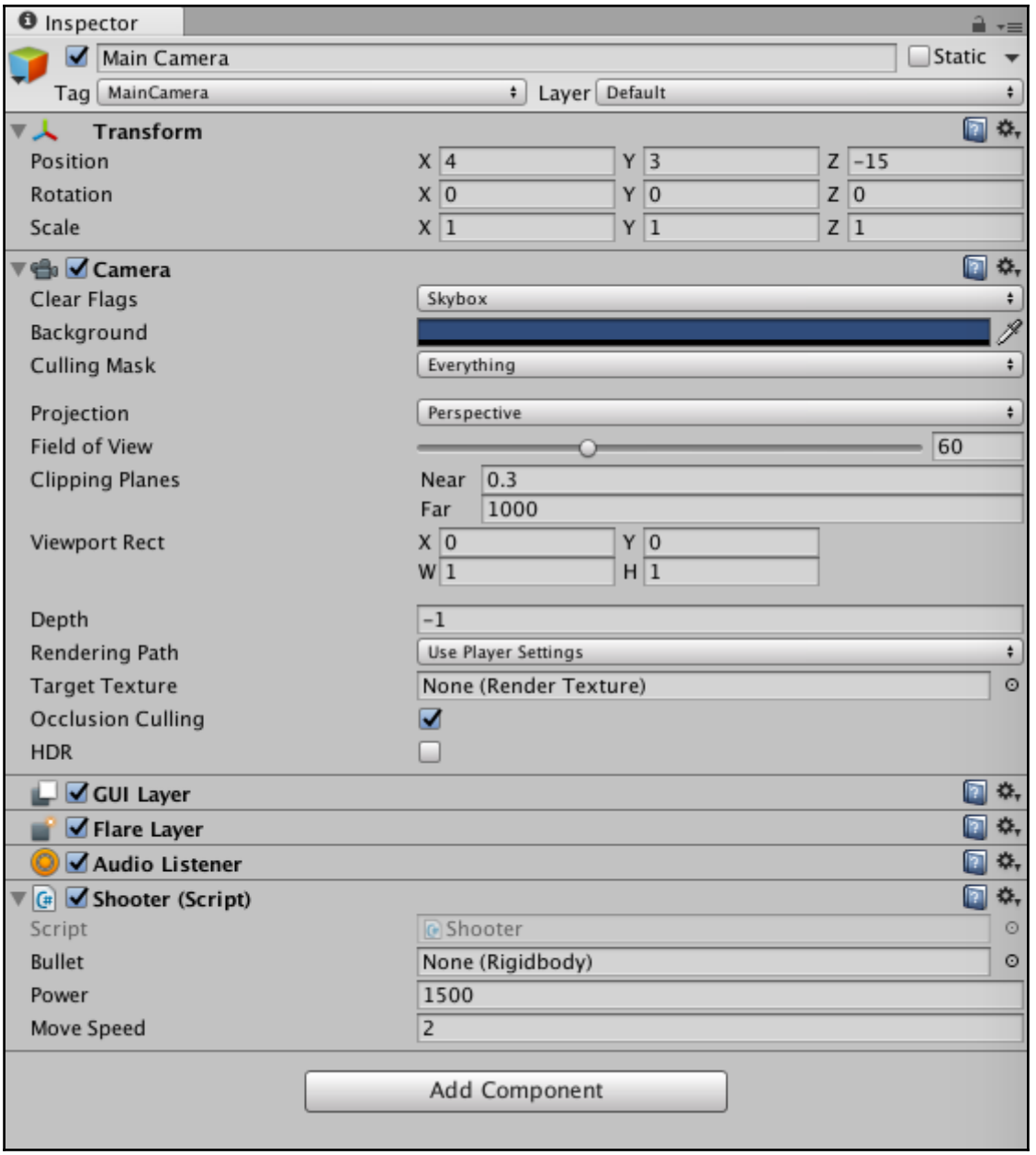
In order for this script to be used within our game it must be attached as a component of one of the game objects within the existing scene.

Save your script by choosing **File** | **Save** from the top menu of your script editor and return to Unity. There are several ways to assign a script to an object in Unity:

1. Drag it from the **Project** panel and drop it onto the name of an object in the **Hierarchy**panel.
2. Drag it from the **Project** panel and drop it onto the visual representation of the object in the **Scene** panel.
3. Select the object you wish to apply the script to and then drag and drop the script to empty space at the bottom of the **Inspector** view for that object.
4. Select the object you wish to apply the script to and then choose **Component** | **Scripts** | and then the name of your script from the top menu.

The most common method is the first approach, and this would be most appropriate as trying to drag to the camera in the **Scene** view, for example, would be difficult, as the camera itself doesn't have a tangible surface to drag to.

For this reason, drag your new **Shooter** script from the **Project** panel and drop it onto the name of **Main Camera** in the **Hierarchy** to assign it, and you should see your script appear as a new component, following the existing Audio Listener component. You will also see its three Public Variables, **Bullet**, **Power**, and **Move Speed,** show in the**Inspector**, shown as follows:


Inspector
Main Camera
Static
Tag MainCamera
Layer Default
Transform
Position X 4 Y 3 Z -15
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Perspective
Field of View 60
Clipping Planes Near 0.3 Far 1000
Viewport Rect X 0 Y 0 W 1 H 1
Depth -1
Rendering Path Use Player Settings
Target Texture None (Render Texture)
Occlusion Culling
HDR
GUI Layer
Flare Layer
Audio Listener
Shooter (Script)
Script Shooter
Bullet None (Rigidbody)
Power 1500
Move Speed 2
Add Component

You can alternatively act in the **Inspector**, directly, press the "**Add Component**" button, and look for Shooter by typing in the search box.

Note: This is valid if you didn't add the component in this way initially. In that case the Shooter component will be already attached to the camera gameobject.

As you will see, Unity has taken the variable names and given them capital letters, and in the case of our `moveSpeed` variable, it takes a capital letter in the middle of the phrase to signify the start of a new word in the **Inspector**, placing a space between the two words when seen as a public variable.

You can also see here that the **Bullet** variable is not yet set, but it is expecting an object to be assigned to it that has a Rigidbody attached-this is often referred to as being a Rigidbody object. Despite the fact that in Unity all objects in the scene can be referred to as game objects, when describing an object as a `Rigidbody` object in scripting, we will only be able to refer to properties and functions of the `Rigidbody` class. This is not a problem however; it simply makes our script more efficient than referring to the entire `GameObject` class. For more on this, take a look at the script reference documentation for both the classes:

GameObject:

`http://unity3d.com/support/documentation/ScriptReference/GameObject.html`

Rigidbody:

`http://unity3d.com/support/documentation/ScriptReference/Rigidbody.html`

Be aware that when adjusting values of public variables in the **Inspector**, any values changed will simply override those written in the script, rather than replace them.

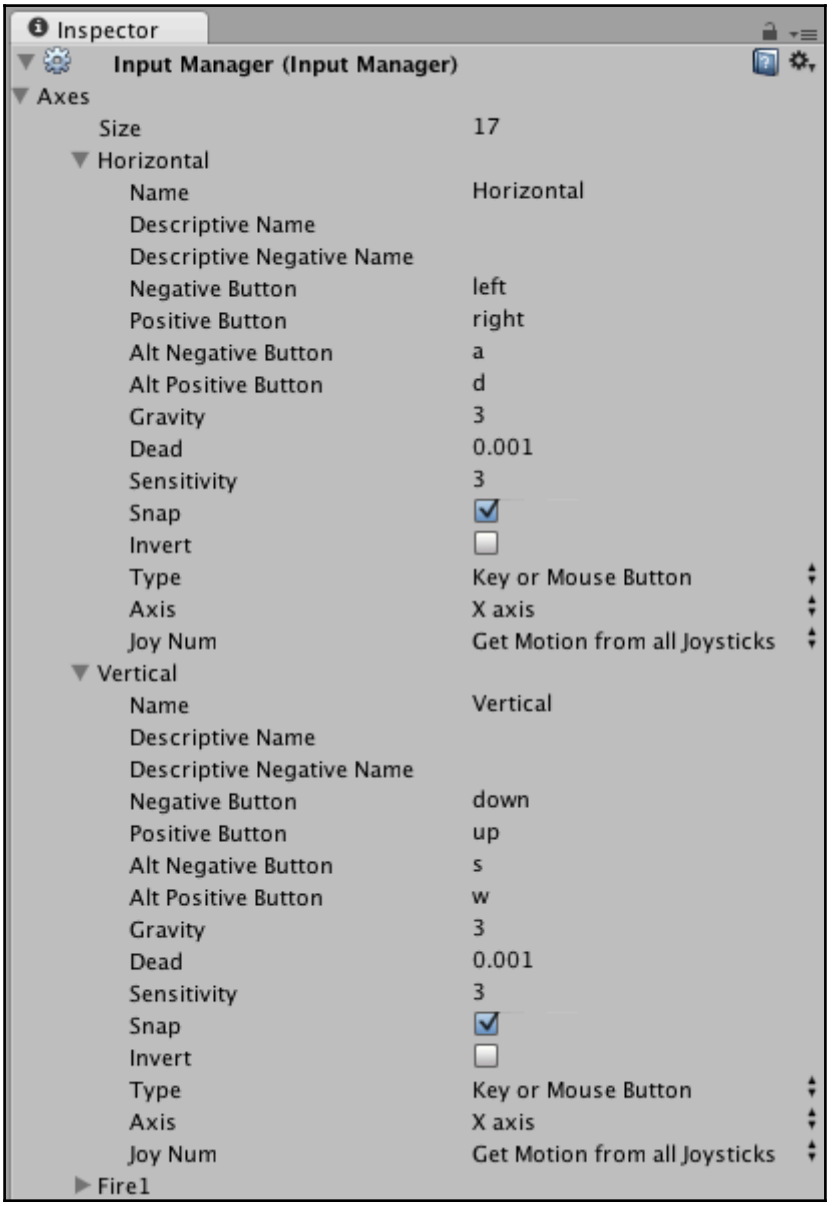
Let's continue working on our script and add some interactivity, so return to your script editor now.

Moving the camera

Next we will make use of the `moveSpeed` variable, combined with keyboard input in order to move the camera, and effectively create a primitive aiming of our shot, as we will use the

camera as the point at which to shoot from.

As we want to use the arrow keys on the keyboard we need to be aware of how to address them in code first. Unity has many inputs that can be viewed and adjusted using the **Input Manager**-see **Edit** | **Project Settings** | **Input**.



As seen in this screenshot, two of the default settings for **Input** are **Horizontal** and **Vertical**. These rely on an axis based input that when holding the **Positive Button** builds to a value of 1, and when holding the **Negative Button** builds to a value of -1. Releasing either button means that the **Input's** value springs back to 0, as it would if using a sprung analog joystick on a gamepad.

Because **Input** is also the name of a class, and all named elements in the **InputManager** are axes or buttons, in scripting terms we can simply use:

```
Input.GetAxis("Horizontal");
```

This receives the current value of the horizontal keys-a value between -1 and 1 depending upon what the user is pressing. Let's put that into practice in our script now, using local variables to represent our axes.

By doing this we can modify the value of this variable later using multiplication, taking it from a maximum value of 1 to a higher number-allowing us to move the camera faster than 1 unit at a time.

This variable is not something we will ever need to set inside the **Inspector**, as Unity is assigning values based upon our key input. As such, these values can be established as **Local** variables.

Local, private, and public variables

Before we continue, let's take an overview of local, private, and public variables in order to cement your understanding:

- Local variables: These are variables established inside a function; they will not be shown in the Inspector, and are only accessible to the function they are within.
- Private variables: These are established outside of a function, and therefore, accessible to any function within your class – however they are also not visible in the Inspector.
- Public variables: These are established outside of a function, accessible to any function in their class, and also to other scripts as well as being visible for editing in the Inspector.

Local variables and receiving input

The local variables in C# and JavaScript are shown as follows:

C#

```
void Update() {
float h = Input.GetAxis("Horizontal") * Time.deltaTime * moveSpeed;
float v = Input.GetAxis("Vertical") * Time.deltaTime * moveSpeed;
```

JavaScript

function Update () { var h : float = Input.GetAxis(“Horizontal”) * Time.deltaTime * moveSpeed; var v : float = Input.GetAxis(“Vertical”) * Time.deltaTime * moveSpeed;

The variables declared here-`h` for `Horizontal` and `v` for `Vertical`, could be named anything we like; it is simply quicker to write single letters. Generally speaking, we would normally give these a name, because some letters cannot be used as variable names, for example, x, y, and z because they are used for coordinate values and therefore reserved for use as such.

As these axes' values can be anything from -1 to 1, they are likely to be a number with a decimal place, and as such we must declare them as floating point type variables. They are then multiplied using the `*` symbol by `Time.deltaTime`-this simply means that the value is divided by the number of frames per second (the `deltaTime` is the time it takes from one frame to the next or the time taken since the `Update()` function last ran), meaning that the value adds up to a consistent amount, per second, regardless of the framerate.

The resultant value is then increased by multiplying it by the public variable we made earlier-`moveSpeed`. This means that although the values of `h` and `v` are local variables, we can still affect them by adjusting public `moveSpeed` in the **Inspector,** as it is part of the equation that those variables represent. This is common practice in scripting as it takes advantage of the use of publicly accessible settings combined with specific values generated by a function.

Understanding Translate

To actually use these variables to move an object, we will use the Translate command. When implementing any piece of scripting, you should make sure you know how to use it first.

Translate is a command which is part of the Transform class: `http://unity3d.com/support/documentation/ScriptReference/Transform.html`

This is a class of information that stores the position, rotation, and scale properties of an object, and also functions that can be used to move and rotate the object.

The expected usage of Translate is as follows:

```
Transform.Translate(Vector3);
```

The use of Vector3 here means that Translate is expecting a piece of Vector3 data as it's the main argument-Vector3 data is simply information that contains a value for the X, Y, and Z coordinates; in this case coordinates to move the object by.

Implementing Translate

Now let's implement the Translate command by taking the h and v input values that we have established, placing them into Vector3 within the command.

C# and JavaScript

Place the given line within the `Update()` function in your script; – meaning after the opening curly brace of `Update(){` and before the function closes with a right curly brace `}`. Note that this does not differ between languages:

```
transform.Translate(h, v, 0);
```

We can use just the word `transform` here because we know that any object we attach this script to will have a Transform component. Attached components of an object can be addressed using the lowercase version of their name, whereas accessing components on other objects requires use of the `GetComponent` command and their uppercase equivalent, for example:

```
GameObject.Find("OtherObjectName").GetComponent
<Transform>().Translate(h,v,0);
```

We do not need that in this instance. Accessing components on other objects is covered later in this book in *Chapter 4*, under *Inter-script communication and dot syntax*.

Here we are using the current value of `h` to affect the X axis, `v` to affect the Y axis, and simply passing a value of to the Z axis, as we do not wish to move back and forth.

Save your script now by going to **File** | **Save** in your script editor and return to Unity. Save the scene we have worked on thus far also by going to **File** | **Save SceneAs,** and name this scene **Prototype**.

Unity will prompt you to save into the `Assets` folder of your project by default and you should always ensure that you do not save outside of this folder as you will not be able to

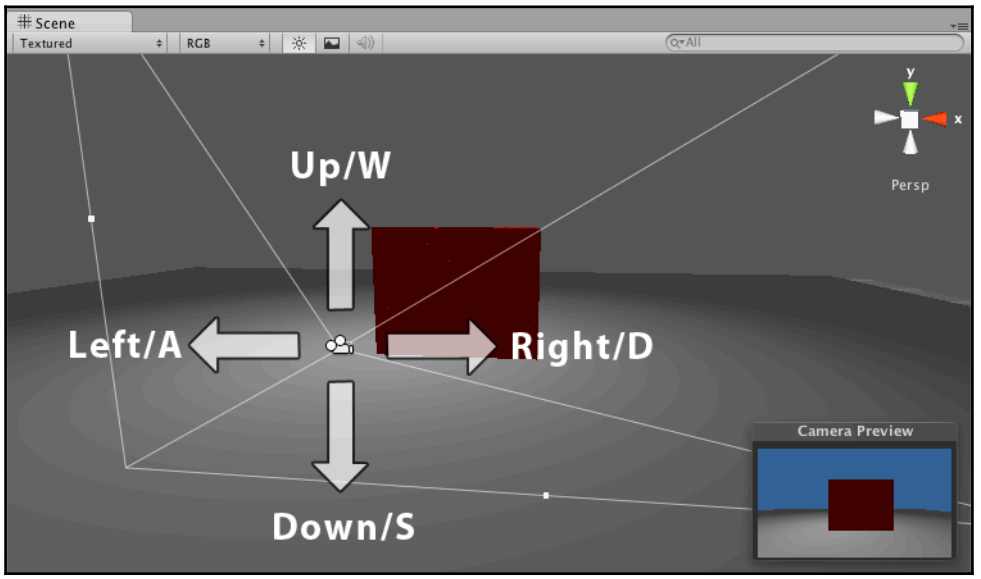
access the scene through the **Project** panel otherwise. You may also create a sub folder within `Assets` in which to keep your scenes if you wish to be extra tidy, this is not necessary but is generally considered to be good practice.

Testing the game so far

In Unity you can play test at any time, provided there are no errors in your scripts. If there are, Unity will ask you to fix all errors before allowing you to enter the Play Mode.

Once all errors are fixed-this will be signified by an empty or cleared **Console** bar at the bottom of the Unity interface. The **Console** bar represents the most recent entry into the Unity console. You can check this by choosing **Window** | **Console** (shortcut Ctrl + Shift + C[PC]Command-Shift + C[Mac]) from the top menu. All the errors willbe listed in red, and double-clicking on the error will take you to the part of the script that will be causing the issue described. Most errors are often a forgotten character or simple misspelling, so always double check what you have written as you go.

If your game is free of errors, click the **Play** button at the top of the screen to enter the Play Mode. You will now be able to move the Main Camera object around by using the arrow keys-*Up*, *Down*, *Left*, and *Right* or their alternates *W*, *A*, *S*, and *D* as shown in the following image:



Once you have tested and confirmed that this works, press the **Play** button at the top of the interface again to switch off the Play Mode.

Switching off the Play Mode before continuing to work is important because settings of components and objects in the current scene that are adjusted during Play Mode will be discarded as soon as the Play Mode is switched off, so leaving Unity in Play Mode as you continue to work will mean you lose work.

Now let's finish our game mechanic prototype by adding the ability to shoot projectiles at the wall to knock it down.

Making a projectile

In order to shoot a projectile at the wall, we will need to first create it within the scene, and then store it as a **prefab**.

A prefab is a GameObject, stored as an asset in the project that can be instantiated-created during runtime and then manipulated, all through code.

Creating the projectile prefab

Begin by clicking the **Create** button at the top of the **Hierarchy**, and then select **Sphere**from the drop-down menu that appears. As mentioned previously, you canalso access primitive creation from the **GameObject** | **Create Other** top menu.

Now ensure that the Sphere object is selected in the **Hierarchy** and hover your cursor over the **Scene** view and press F on the keyboard to focus your view on the Sphere.

If your Sphere has been created at the same position as one of your other objects, then simply change to the **Translate tool** (W) and drag the relevant axis handle until your Sphere is out of the way of the object blocking your view, then re-focus your view by pressing F again.

Taking a look at the **Inspector** panel, you will notice that when introducing new primitive objects to the scene, Unity automatically assigns them three new components in addition to the existing Transform component, which are as follows:

- **Mesh Filter**: This component is to handle the shape

- **Renderer**: This component is to handle the appearance
- **Collider**: This component is to manage interactions (known as collisions) with other objects

Creating and applying a material

We'll begin with the visual appearance of the projectile, and alter this by creating a material to apply to the renderer. Whenever you need to adjust the appearance of an object, you'll likely look to alter settings in some kind of Renderer component.

For 3D objects it will be the Mesh Renderer, on Particle systems it will be a Particle Renderer, and so on.

To keep things neat, we'll make a new folder within our `Assets` folder to store all materials that we may create in this project. On the **Project** panel click the **Create** button and choose **Folder** from the drop-down menu. Rename this folder **Materials** by pressing Return (Mac) or F2 (PC). Take the time now to place the Red brick material you made earlier inside this new folder.

To create any new asset inside of an existing folder in the **Project** panel, simply select the folder first and then create using the **Create** Button.

Now we will create the material we need and apply it to our object:

1. With the new `Materials` folder still selected click the **Create** button on the **Project** panel once again and this time choose **Material**. This creates a NewMaterial asset that you should rename **bulletColor**-or something of your own choosing that reminds you that this asset is to be applied to the projectile.
2. With your new material still selected, click on this block to open the **ColorPicker** window, select a shade of blue, and then close this window when youare happy with your selection.
3. Now that you have chosen your color for the material, drag and drop the **bulletColor** material from the **Project** panel and drop it onto the name of theSphere object in the Hierarchy to assign it.

Note that, if you want to test how a material will look when applied to a 3D object in Unity, you can drag the material to the **Scene** view, hovering the cursor over meshes-Unity will show you a preview of what it will look like and you can then move the mouse away or press Esc if you wish to

 cancel, or release the mouse to apply.

Adding physics with a Rigidbody

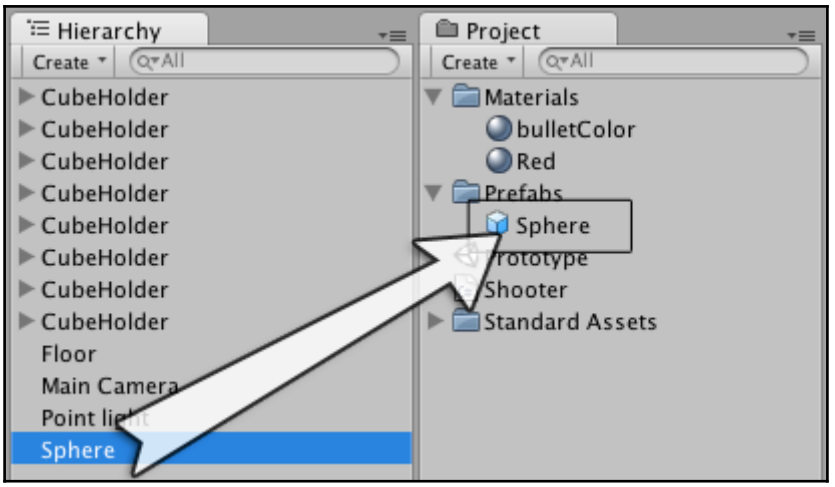
Next we'll ensure that the physics engine has control of the projectile **Sphere** by adding a **Rigidbody** component. Select the **Sphere** in the Hierarchy panel and choose **Component** | **Physics** | **Rigidbody** from the top menu.

Your **Rigidbody** component is now added, with settings available to adjust in the **Inspector**. For the purpose of this prototype however, we needn't adjust any settingsfrom the default.

Storing with prefabs

As we wish to fire this projectile when the player presses a key, we do not want the projectile to be in the scene by default, but instead want it to be stored and created when the key is pressed. For this reason we will store the object as a prefab, and use our script to instantiate (that is, create an instance of) it at the precise moment a key is pressed.

Prefabs are Unity's way of storing GameObjects that have been set up in a particular way; for example, you may have configured an enemy soldier with particular scripts and properties that behaves a certain way. You can store this object as a prefab and instantiate it when necessary. Similarly you might have a differing soldier that behaves differently, this might be a different prefab, or you might create an instance of the first, and adjust settings in the soldier's components, making him faster or slower upon instantiation for example; the Prefab system gives you a lot of flexibility in this regard.



Click the **Create** button at the top of the **Project** panel, and choose **Folder**, then rename this to **Prefabs**. Now drag the **Sphere** from the **Hierarchy** and drop it onto the **Prefabs** folder in the **Project** panel as show in the following screenshot. Dragging a GameObject such as this anywhere into the **Project** panel will save it as a prefab; we simply created this folder for neatness and good practice. Rename this new

Prefab from Sphere to Projectile.

You can now delete the original Sphere object from the **Hierarchy** by selecting it, and pressing *Command+Backspace* (Mac) or *Delete* (PC); alternatively, you can right-click the object in the **Hierarchy** and choose **Delete** from the pop-out menu that appears.

You can also rename the Sphere in Projectile in the hierarchy view, then, drag into the Project view will create a prefab with that name.

Firing the projectile

Return to the **Shooter** script we have been working on so far by double-clicking its icon in the **Project** panel, or by selecting it in the **Project** panel and clicking the **Open** button at the top of the **Inspector**.

Now we will make use of the `bullet` variable we declared earlier, using it as a reference to the particular object we wish to instantiate. As soon as the object is created from our stored

prefab, we will apply a force to it, in order to fire it at the wall in our scene.

After the line:

```
transform.Translate(h, v, 0);
```

Within the Update() function in your script, add the following code, regardless of whether you are using C# or JavaScript:

```
if(Input.GetButtonUp("Fire1")){

   }
```

This IF statement listens for the key applied to the Input button named **Fire1** to be released. By default, this is mapped to the *Left Ctrl* key or left mouse button, but you can change this to a different key by adjust settings in the **Input Manager** (**Edit** | **Project Settings** | **Input**).

Using Instantiate() to spawn objects

Now within this IF statement-meaning after the opening { and before the closing }, put the following line:

C#

For the code in C# use the following line:

```
Rigidbody instance = Instantiate(bullet, transform.position,
transform.rotation) as Rigidbody;
```

JavaScript

For the code in JavaScript we will insert the following line:

```
var instance: Rigidbody = Instantiate(bullet, transform.position,
transform.rotation);
```

Here we are creating a new variable named instance. Into this variable we are storing a reference to the creation of a new object that is of type Rigidbody.

The Instantiate commands requires three pieces of information namely,

```
Instantiate(What to make, Where to make it, a rotation to give it);
```

So in our example, we are telling our script to create an instance of whatever object or prefab is assigned to the bullet public member variable and that we would like it to be created using the values of position and rotation from the transform component of the object this script is attached to-the **Main Camera**. This is why you will often see transform.position written in scripts as it refers to the transform component's position settings of the object the script is attached to.

Note that in C# you must add as `Rigidbody` to specifically state the data type after the Instantiate command.

Adding a force to the Rigidbody

Having now created our object, we need it to be immediately fired forward using the AddForce() command. This commands works as follows:

```
Rigidbody.AddForce(Direction and amount of force expressed
as a Vector3);
```

So before we add the force, we will create a reference to the direction we wish to shoot in. The camera is facing the brick wall so it makes sense to shoot objects at the wall in the camera's forward direction. Following the instantiate line you just added, still within the IF statement, add the given code:

C#

To add force, we must insert the following line in the C# code:

```
Vector3 fwd = transform.TransformDirection(Vector3.forward);
```

JavaScript

In the JavaScript code, we must add the following line:

```
var fwd: Vector3 = transform.TransformDirection(Vector3.forward);
```

Here we have created a Vector3 type variable called fwd, and told it to represent the forward direction of the current transform this script is attached to.

The TransformDirection command can be used to convert a local direction-that of the

forward direction of the camera, to a world direction-as objects and the world each have their coordinate system, and the forward direction of an object may not necessarily match that of the world, so conversion is crucial. Vector3.forward in this context is simply a shortcut to writing Vector3(0,0,1). It is one unit in length on the Z axis.

Finally, we will apply the force by first referring to our variable that represents the newly created object-instance, then using the AddForce() command to add a force in the direction of the fwd variable-multiplied by the public variable named power that we created earlier. Add the following line to your code beneath the last line you added:

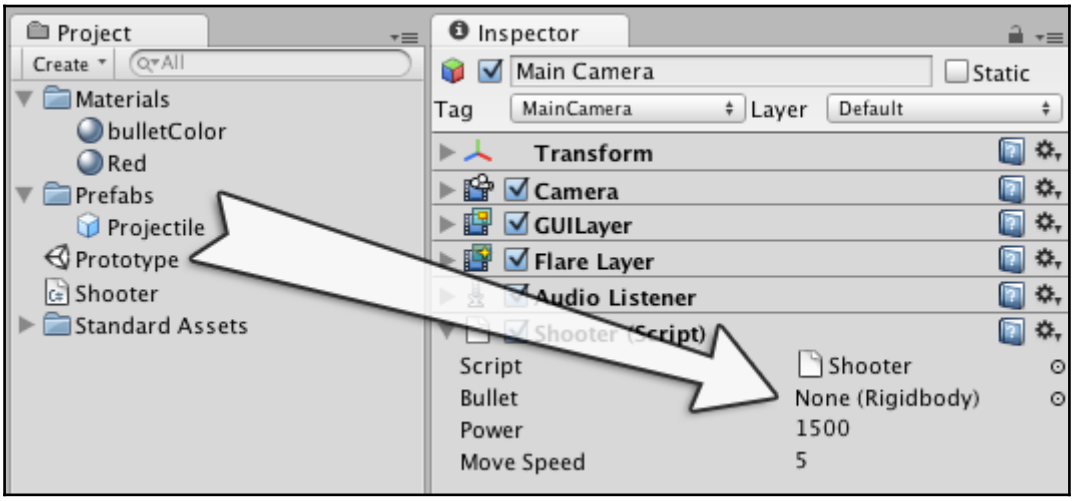
C# and JavaScript:

```
instance.AddForce(fwd * power);
```

Save your script and return to Unity.

Now before we can test play the finished game mechanic, we need to assign the **Projectile** prefab to the **Bullet** public variable. To do this, select the **Main Camera** in the **Hierarchy**, in order to see the **Shooter** script as a component in the **Inspector**.

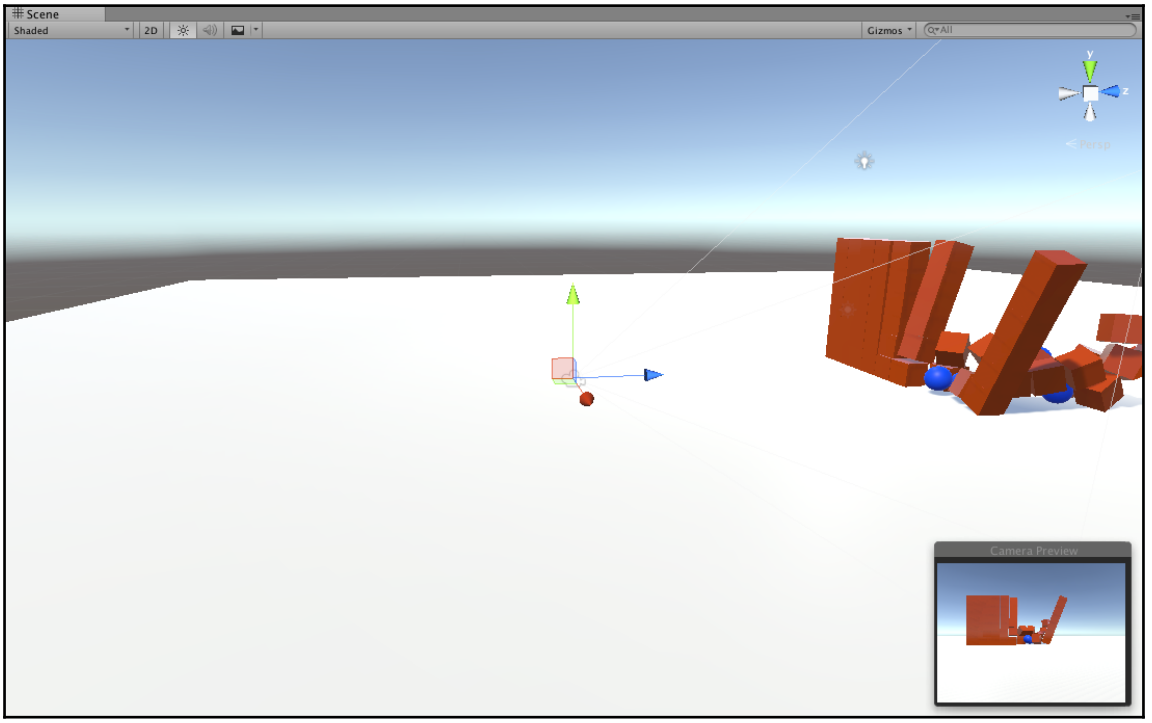
Now drag the **Projectile** prefab from the **Project** panel, and drop it onto the **Bullet** variable in the **Inspector**, where it currently says **None (Rigidbody),** as shown in the following screenshot:



Once applied correctly, you will see the name of the projectile in this variable slot. Save the scene now (**File** | **Save Scene**) and test the game by pressing the **Play** button at the top of

the interface.

You will now be able to move the camera around and fire the projectiles we created earlier using the *Left Ctrl* on the keyboard. If you wish to adjust how much power you give to the projectiles when they are fired, simply adjust the number in the **Power**public variable by selecting the**Main Camera**, and adjusting the value of**1500**currently assigned to it in the **Shooter (Script)** component, increasing or decreasing as you see fit. Remember to always press the **Play** button again to stop testing.



Reseting the wall to initial state and clearing the projectiles.

There might be a lot of way to implement this, for this chapter, we will do it in the simplest way possible for this chapter, we will basically use the SceneManager.LoadScene API to reload the level.

In the next chapters we will discover how to use gameobject layers and tags to choose

among objects in a complex scene, and how to `Destroy()` the objects individually or per group.

If using C#, after the first line of the C# script add:

```
using UnityEngine.SceneManagement;
```

Within the `Update()` function in your script, add the following code, regardless of whether you are using C# or JavaScript:

```
if(Input.GetButtonUp("Cancel")){
  SceneManager.LoadScene("Prototype");
 }
```

Test the game and when you have destroyed part of the wall by shooting some bullets, press the Esc key.

Summary

Congratulations! You have just created your first Unity prototype.

In this chapter, you should have become familiar with with the basics of using the Unity interface, working with Game Objects, components, and basic scripting. This will hopefully act as a solid foundation upon which to build further experience in Unity game development.

Now you might want to relax a little and take time to play your prototype or even create one of your own based on what you have learned. Or you may just be eager to learn more; if so, keep reading!

Let's move on to the main game of this book, now that you are a little more prepared on some of the basic operations of the Unity development.