EXPERT INSIGHT

Unity Game Development

Learn how to design beautiful games from the team at Unity

Anthony M Davis
Travis M W Baptiste
Russell Craig
Ryan Stunkel

Packt

Unity Game Development

Early Access Publication: Unity Game Development

Early Access Production Reference: B17304

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80107-614-2

www.packt.com

Table of Contents

Unity Game Development: Learn how to design beautiful games from the team at Unity

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You'll be notified when a new version is ready.

This title is in development, with more chapters still to be written, which means you have the opportunity to have your say about the content. We want to publish books that provide useful information to you and other customers, so we'll send questionnaires out to you regularly. All feedback is helpful, so please be open about your thoughts and opinions. Our editors will work their magic on the text of the book, so we'd like your input on the technical elements and your experience as a reader. We'll also provide frequent updates on how our authors have changed their chapters based on your feedback.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of

writing a Packt book. Join the exploration of new topics by contributing your ideas and see them come to life in print.

1. Welcome to the Third Dimension
2. Design and Prototype
3. Programming
4. Creating Player Characters
5. Environment and Interactions
6. Prefabs, Collection, and HUD
7. Instantiation and Rigid Bodies
8. Particle Systems
9. Menu Design
10. Adding Finishing Touches
11. Building and Sharing
12. Testing
13. Multiplayer, AR, VR, and ML-Agents

A Primer to the Third Dimension

Welcome to Chapter One! Even if you have some previous knowledge of the systems in this chapter, solidifying the foundation of your knowledge will only help the advanced topics settle in more easily.

In this chapter we will cover topics such as:

- A good look into the parts that make up 3D game development
- Definitions of Unity terminology
- A tour through Unity's interface

Coming around to 3D

We will be going over the basic understanding of the 3D work within this section. From coordinate systems to the makeup of how the 3D model is rendered, we will only go surface level to ensure that the foundations are firmly planted within you as you progress through this journey. By reading through this, you will be able to have a strong understanding of why Unity is displaying the items.

Coordinates Systems

3D coordinate systems are not all the same in each application! Unity is a Left-handed (*Figure 1.1*) world coordinate system with $+Y$ facing upwards. Looking in the image below you can visualize the difference from left-handed and right-handed systems:

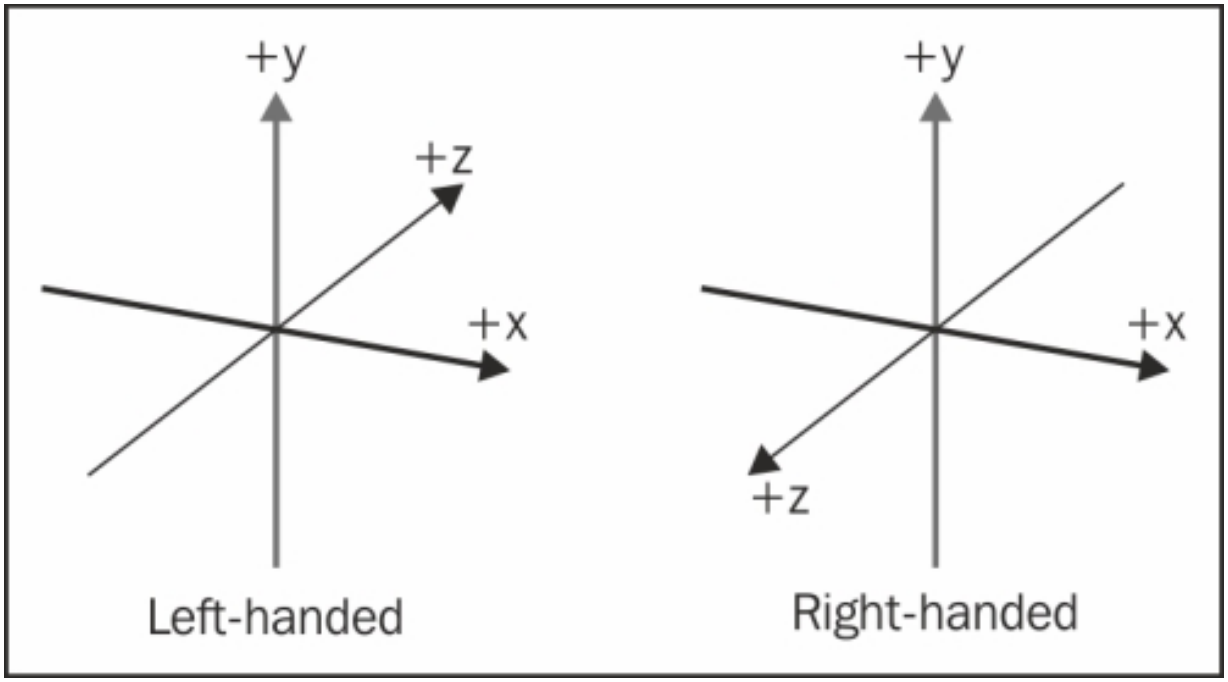


Figure 1.1 Coordinate Systems

While we work within these coordinate systems, you will see positions of objects represented in an array 3 values within parenthesis as follows:

```
(0, 100, 0)
```

This represents (*X, Y, Z*) respectively. This is a good habit to get into as programming utilizes very similar syntax when writing out positions within the scripts. When we are talking about position, it is commonly referred as it's "Transform".

Local Space versus World Space

After understanding the world coordinates *X, Y, Z*, and that the very start of those coordinates start at 0 for each, represented with a (0, 0, 0). In the image below (*Figure 1.2*), where the colored lines meet is that 0,0,0 in the world. The cube has it's own transform, which

encompasses that object's **Transform**, **Rotation**, and **Scale**:

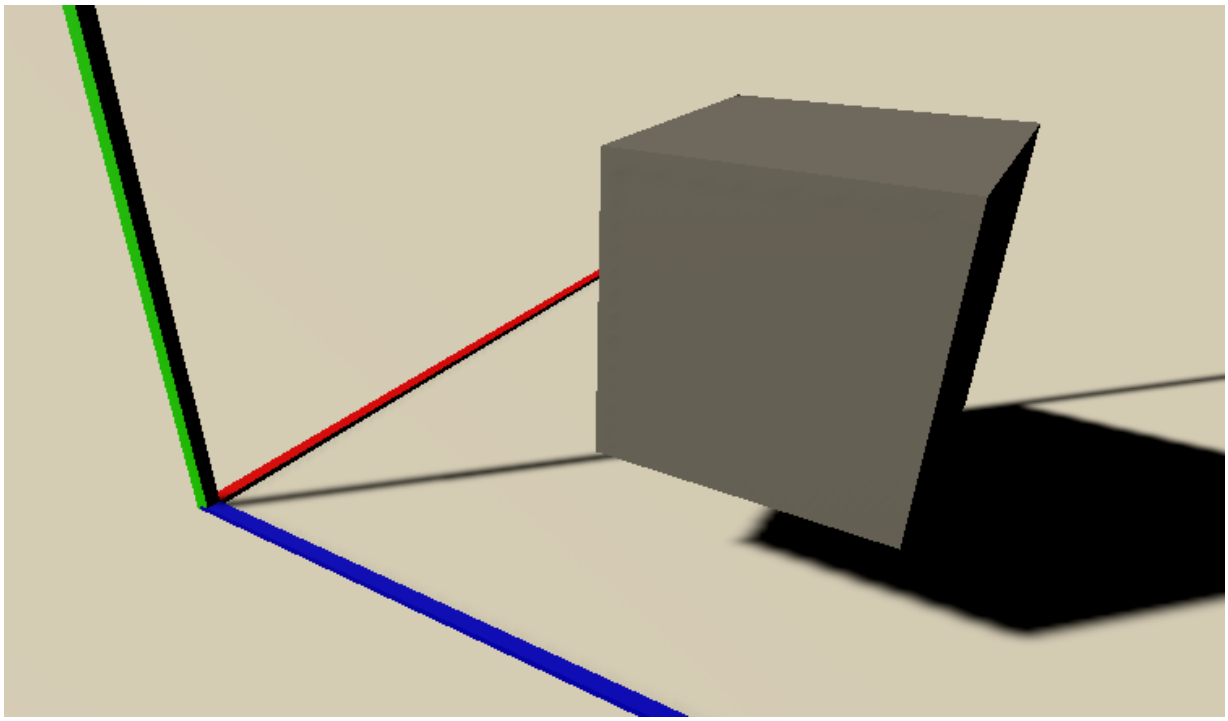


Figure 1.2 3D Coordinate System

The cube in the image (*Figure 1.2*) is at (1, 1.5, 2). This is called world space as the item's Transform is being represented through the world's coordinates starting from (0, 0, 0):

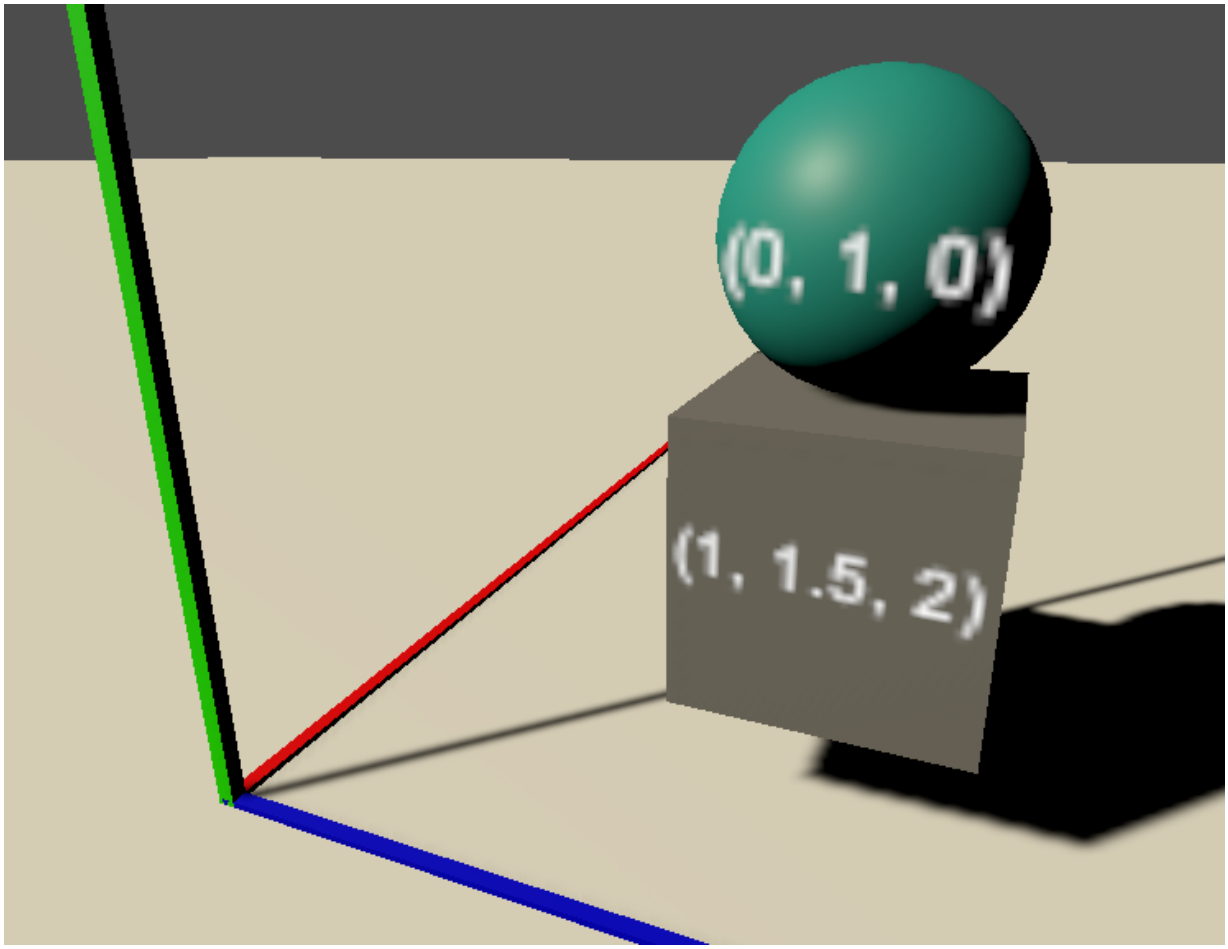


Figure 1.3 World Space vs Local Space

Now that we know the Cube's transform is in relation to the world (0, 0, 0), we will not go over the parent-child relation which describes the local space. In the image above (*Figure 1.3*), the sphere is a child of the cube. The sphere's local position is (0, 1, 0) in relation to the cube. Interestingly, if you now move the cube, the sphere will follow as it's only offset from the cube and it's transforms will remain (0, 1, 0) in relation to the cube.

Vectors

Traditionally, a vector is a unity that has more than one element. In a 3D setting, a vector 3 will look very similar to

what we've worked with currently. (0, 0, 0) is a vector 3! Vectors are used in very many solutions to game elements and logic. Primarily the developer will normalize vectors so that way the magnitude will always equal 1. This allows developer to work with the data very easily as 0 is the start, .5 is halfway, and 1 is the end of the vector.

Cameras

Cameras are incredibly useful Objects! They humbly show us their perspective which allows our players to experience what we are trying to convey to them. As you may have guessed, a camera also has a transform just like all of game objects in the hierarchy. Cameras also have several parameters that can be changed to obtain different visual effects.

Different game elements and genres use cameras in different ways. Resident Evil using static cameras to give a sense of tension not knowing whats outside the window or around the corner, while Tomb Raider pulled the camera in close while she goes through caverns to give a sense of intimacy and emotional understanding with her face looking uncomfortable in tight spaces.

Cameras are essential to the experience you will be creating to your users, take time to play with them and learn compositional concepts to maximize the push of emotions in the players experience.

Faces, Edges, Vertices, and Meshes

3D objects are made up of multiple parts as seen in *Figure 1.4*. Vertices, represented by the green circles, are points in space relative to the world (0, 0, 0). Each object has a list of these vertices and their corresponding connections.

Two vertices connected make an edge, represented with a red line. A face is made when either 3 or 4 edges connect to make a triangle or a quad. Sometimes quads are called a plane when not connected to any other faces. When all of these parts are together, you have a mesh:

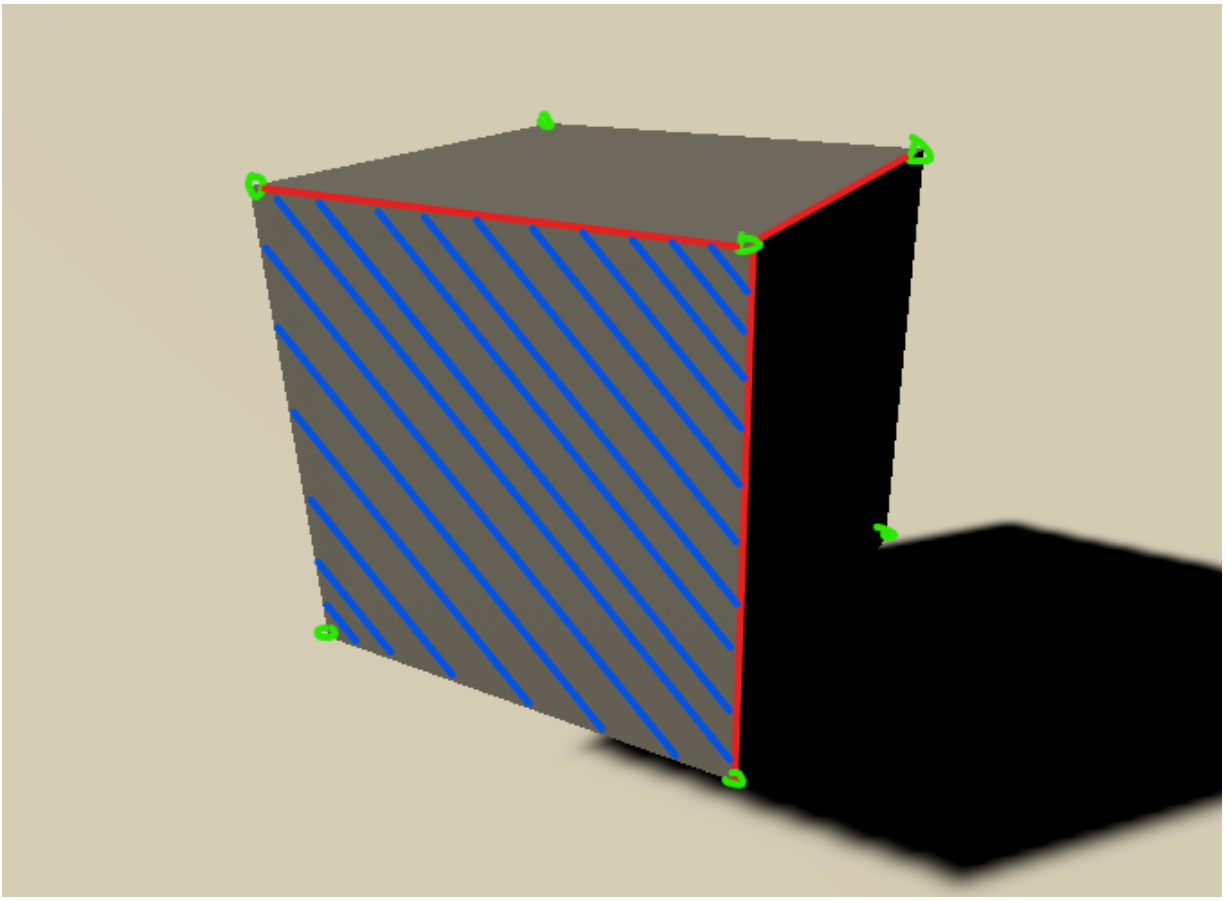


Figure 1.4 Vertices, Edges, Face, and Meshes

Materials, Textures, and Shaders

Now that you know how what a mesh is comprised of in all **Digital Content Creation** (**DCC**) tools, let's look into how Unity displays that mesh to you. At the very base level is a shader. Shaders can be thought of as small programs, which have their own language, that help update the graphics pipeline so Unity can render the objects in your

scene to your screen. You can think of the shader as a large template for materials to be created.

The next level up is materials. A material is a set of attributes which are defined by the shader to be manipulated, which helps show what the object looks like. Each rendering pipeline will have separate shaders. Built-in, **Universal Rendering Pipeline** (**URP**), and High Definition Rendering Pipeline. For this book, we are using the middle and most widely used URP.

Figure 1.5 shows an example of a material using the URP's Standard Lit shader. This allows us to manipulate Surface options, inputs for that surface, and some advanced options. For now, let's just talk about the Base Map, first item in the surface inputs section. The term "Base Map" is being used here as a combination of the "Diffuse/Albedo" and "Tint" together. Diffuse or Albedo is used to define the base color (Red) that will be applied to the surface, in this case, white. If you placed a texture into this map by either dragging a texture onto the square (Green) to the left of the base map or clicking on the circle (Blue) in between the box and the name. After that you can tint the surface with the color if there needs to be any adjustments:

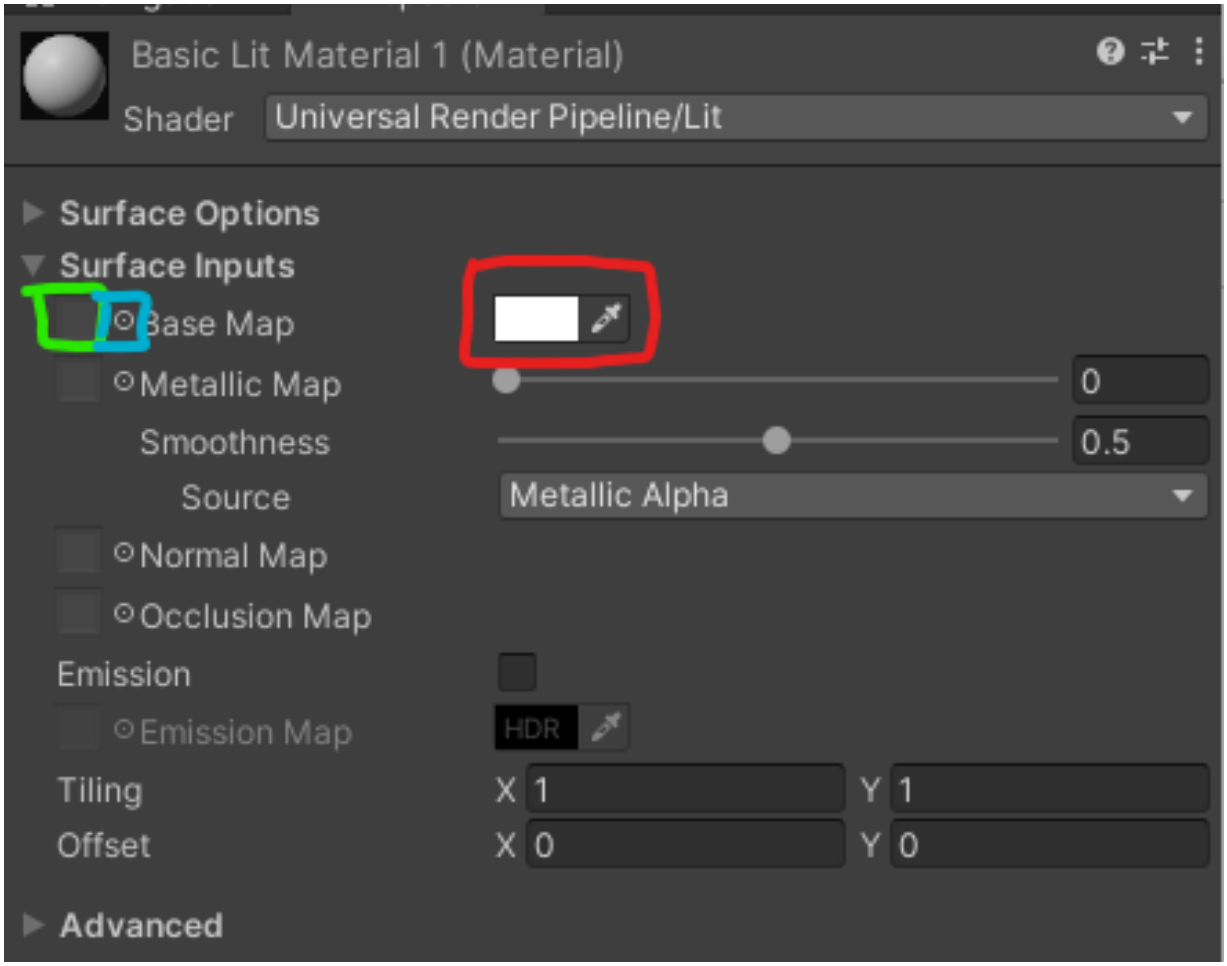


Figure 1.5 Base Material Attributes

Figure 1.6 shows a simple example of what a cube would look like with a tint, texture, and the same texture with the tint changed. As we progress through the book, we will unlock more and more functions of materials, shaders and textures:

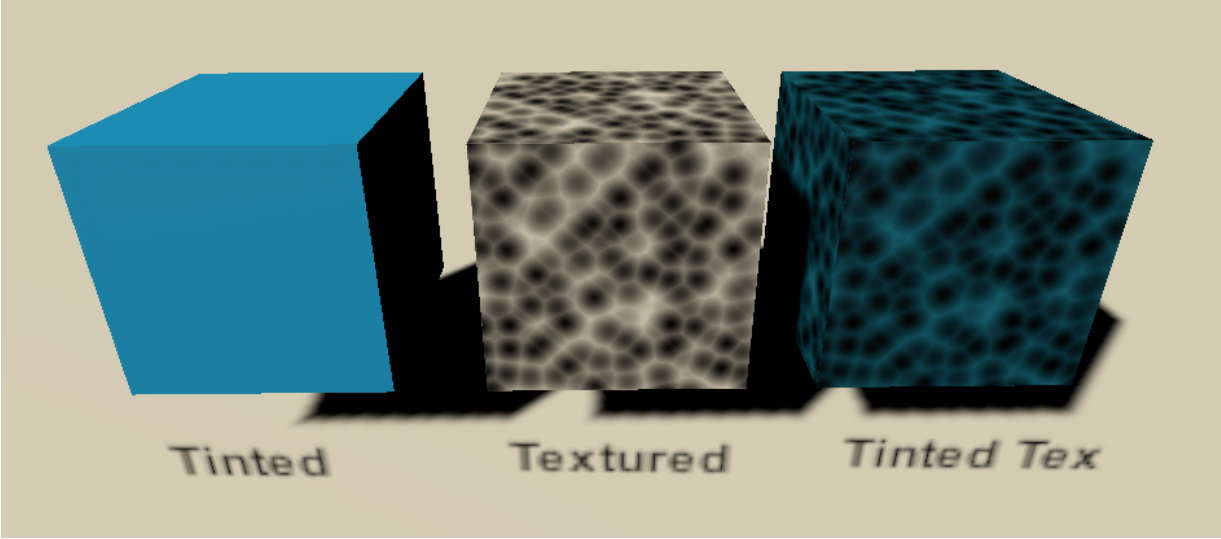


Figure 1.6 Tint and Texture Base Color

Textures can provide incredible detail for your 3D model. When creating a texture, the resolution is an important consideration. The first part of resolution that needs to be understood is “power of 2” sizes. A Power of 2 is as such:

```
2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, etc
```

These numbers represent the pixel size for both width and height. There are cases that you may need to mix the sizes as long as they fit the Power of 2 scale. Examples are:

```
256×256
1024×1024
256×1024 (This is less common to see, but is valid)
```

The second consideration of resolution is the size itself. The easiest way to work through this consideration is thinking about how large the 3D object will be on your screen. If you have a 1920x1080 screen resolution, that is 1920 pixels wide by 1080 pixels tall. If the object in question is only going to be taking up 10% of the screen and rarely closer, you may consider a 256x256 texture. By contrast, if you are making an emotion character driven game where emotions

from facial expressions matter, you may want a 4096x4096 or 4k texture on just the face during those cut scenes.

Rigid Body Physics

Unity assumes that every game object does not need to be evaluated every frame for physics. Unity uses Nvidia's PhysX engine for it's physics calculations. To get any calculated physics responses, the game object needs a Rigid Body component added.

By adding the Rigid Body component to the game object you are then adding some properties to the game object seen in the inspector *Figure 1.7* below":

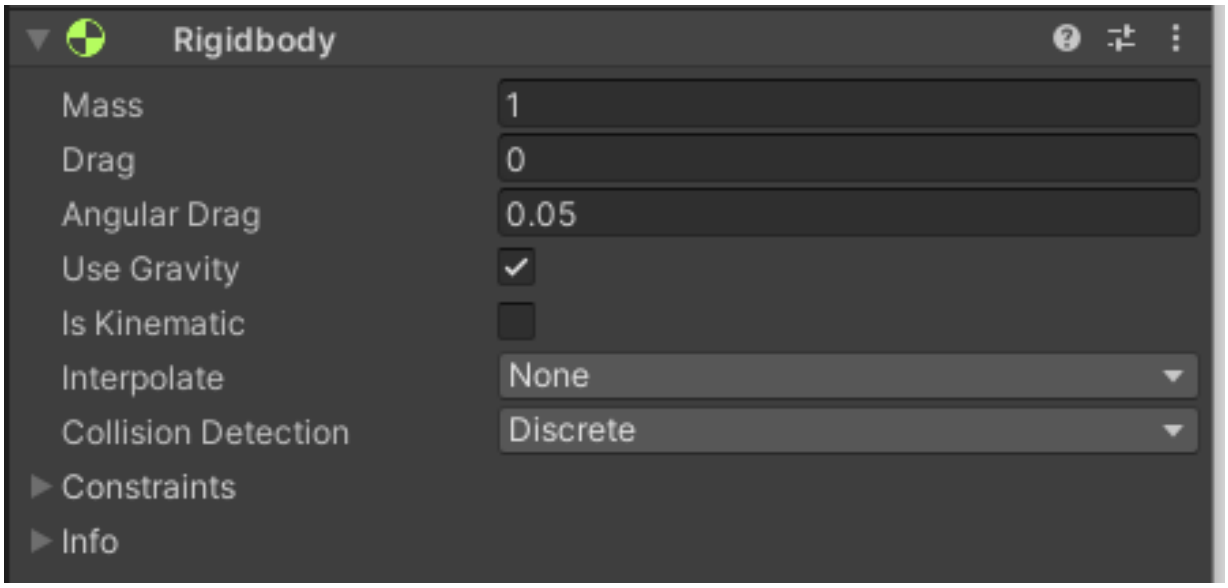


Figure 1.7 Rigid Body

One unity unit of mass is equal to 1 kg of mass. This affects the physics decisions upon collisions. Drag units will attempt to reduce the velocity over time as friction. Angular Drag is similar, but constrained to only rotation speed. Using gravity either turns on or off gravity which is standard Earth gravity so the mass makes sense!

A thorough explanation of rigid body will be worked through in future chapters.

Collision Detection

A gameobject with a rigid body without any collision will not be fully utilizing the physics and with gravity turned on will just fall through the world. There are quite a few colliders to play with to best suit your games' needs:

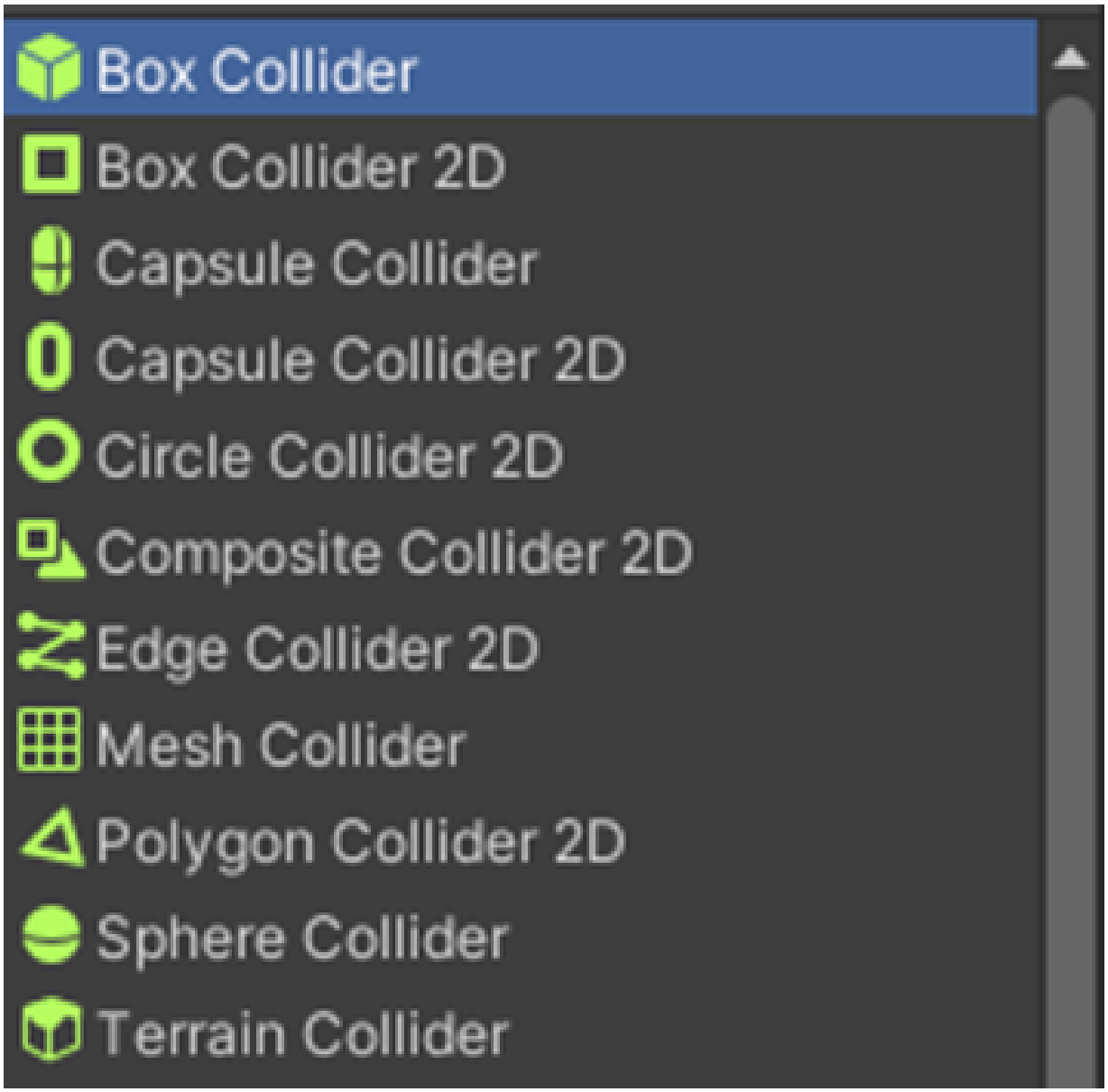


Figure 1.8 Collider Component Options

You are also welcome to add multiple colliders, basic options seen in *Figure 1.8* above, to an object to best suit the shape of the gameobject. It is very common to see colliders on empty gameobjects that are children of the primary object to allow easy transformation of the colliders. We will see this in practice later in this book.

Essential Unity Concepts

In the first section that you previously read we went over some Unity concepts already. We will go over them in a bit more detail here as you've read previously where several of these might be used. Unity houses a very modular focus to the items that are housed within the game development environment.

Assets

Unity treats every file as an asset. This is from 3D Model, Texture file, sprite, particle system, and so on. In your project you will have an Assets folder as the base folder to house all of your in project items. This could be textures, 3D models, particles systems, materials, shaders, animations, sprites, and the list goes on. As we add more onto our project, the assets folder should be organized and ready to grow. It is strongly recommended to keep your folder structure organized so that you or your team aren't wasting time trying to find that one texture item that was left in a random folder on accident.

Scenes

A scene houses all of the gameplay logic, game objects, cinematics, and everything else else which your game will be referencing to render or interact with.

Scenes are also used to cut up gameplay sections to bring down the load times. If you could imaging trying to load every single asset on a modern game every time you loaded it up. It would take way too much precious gaming time.

Game Objects

Most assets that are being referenced in a scene will be a **GameObject** (**GO**). There are some instances in which an assets can only be a component of a GO. The one common factor that you will see with all gameobjects is that they have the **Transform** component. Remembering back in the 3D Primer unity above we know that this is the world or local position, rotation, or scale of that game object. GO's can have a long list of components connected to give functionality or data to be used in scripts for mechanics to grow.

Components

Game Objects have the ability to house multiple functionality attached as "components". Each component has it's own unique properties. The entire list of components you can add is fairly extensive as you can see in *Figure 1.9* below:

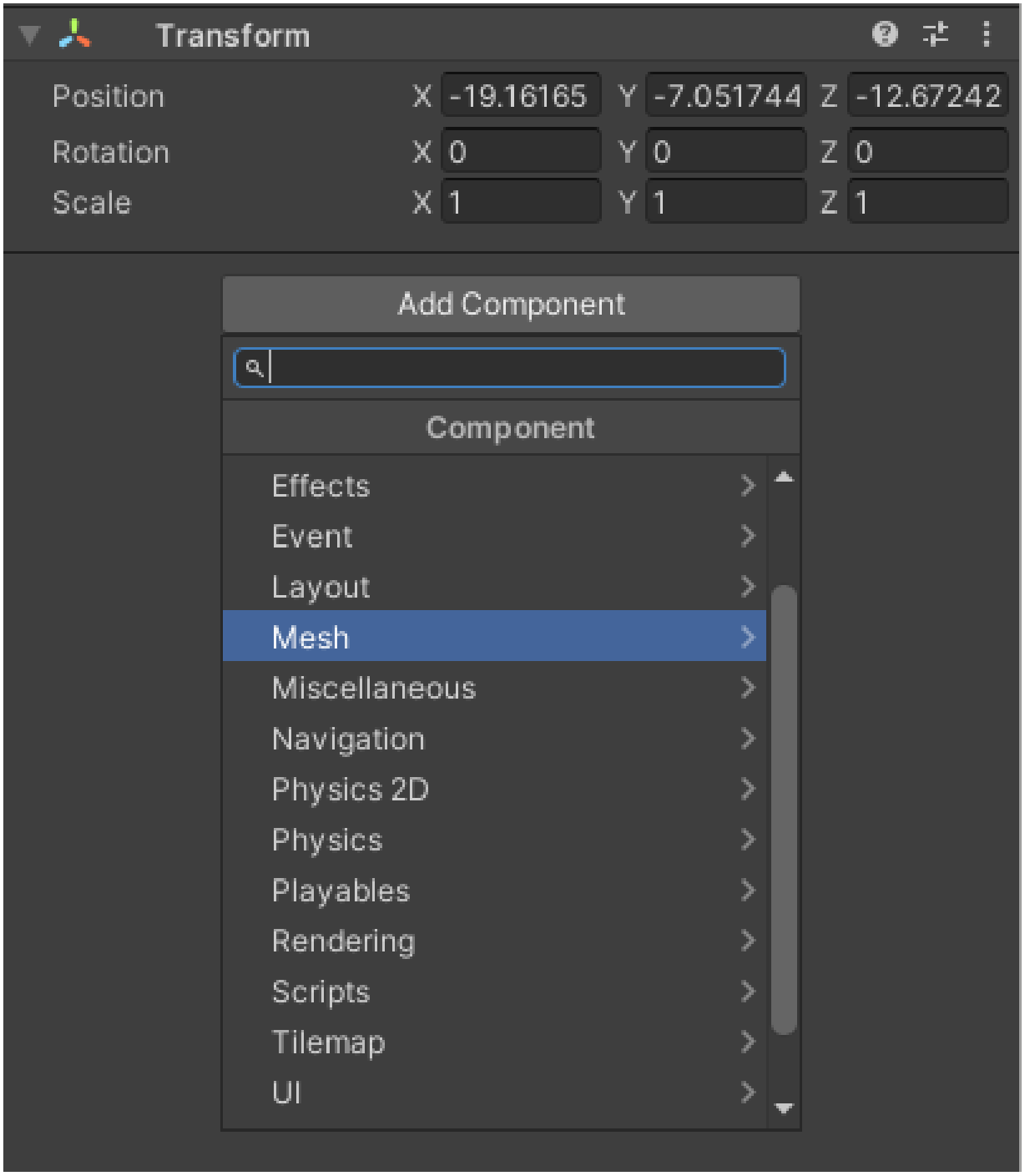


Figure 1.9 Component List

Each of these sections has smaller subsections. We will go over quite a few of them throughout this book. When you add an asset to the scene hierarchy which may require

components, Unity will add them by default. An example of this default action happening is when you drag a 3D mesh into the hierarchy, the gameobject will have a mesh renderer component attached to the object automatically.

Scripts

One component that is often used on game objects is scripts. This is where all of the logic and mechanics will be built onto your gameobjects. Whether you want to change the color, jump, change the time of day, collect an item, and so on, you will need to add that logic into a script on the object.

In Unity the primary language is C# (pronounced “C-Sharp”). This is a type strong programming language, meaning that there must be a type assigned to any variable that is being manipulated.

We will be using scripts in a multitude of ways and I know you are excited to get right into coding, but first we need to get into other Unity standard processes.

Prefabs

Utilizing the modular and strong object oriented nature of Unity, we can put together a grouping of items with default values set on their components which can be instanced in the scene at any time and house their own values.

To make a prefab, you drag a game object from the hierarchy in the scene to the asset browser. It will create a new prefab as well as turn that gameobject into the newly created prefab. It will also turn blue by default in the hierarchy as seen in *Figure 1.10*:

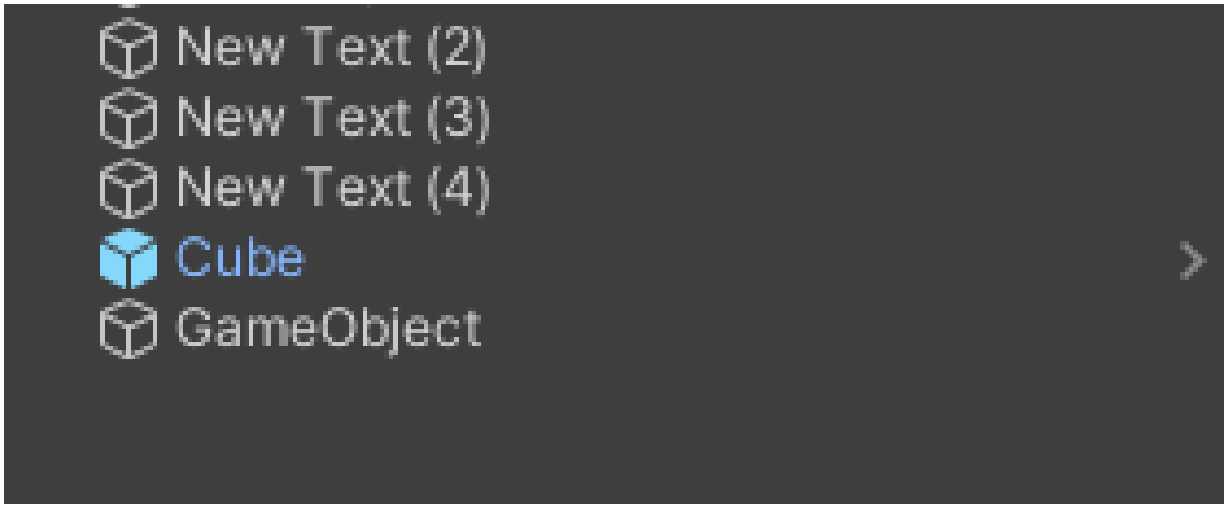


Figure 1.10 Prefab in Hierarchy

Packages

To take the modular components to a whole new level, Unity can take a package with all of it's dependencies and export them out so you can bring them into other projects! Even better, you can sell your packages to other game developers from the Unity Asset Store!

Now that you have a solid foundation in 3D and Unity terms, let's open it up and go over the interface itself. Next section will be a look into all of the most common interface pieces of Unity.

Unity Interface

The interface for Unity is separated into several major components. We will go over the scene (*Figure 1.11*, Red) and the items within it's interface as well as how to manipulate their properties in the Inspector (*Figure 1.11*, Orange). Then we will go into items which aren't active in the scene, but available to add in the project window (*Figure 1.11*, Yellow). Finally, we will go over game view

(*Figure 1.11*, Green) and the package manager (separate from the image below):

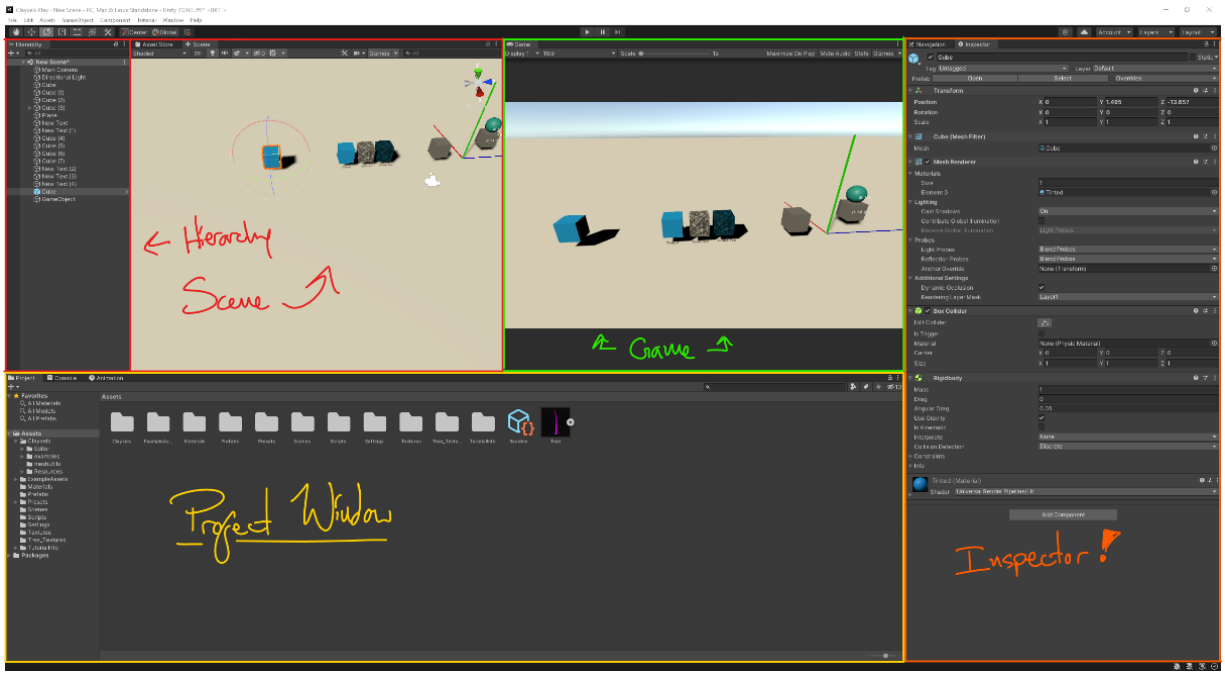


Figure. 1.11 Overall Interface

Scene View and Hierarchy

Scene view and hierarchy work in tandem. The hierarchy is how the scene will be rendering when the game plays. The scene view allows you to manipulate the game objects and their values in real time:

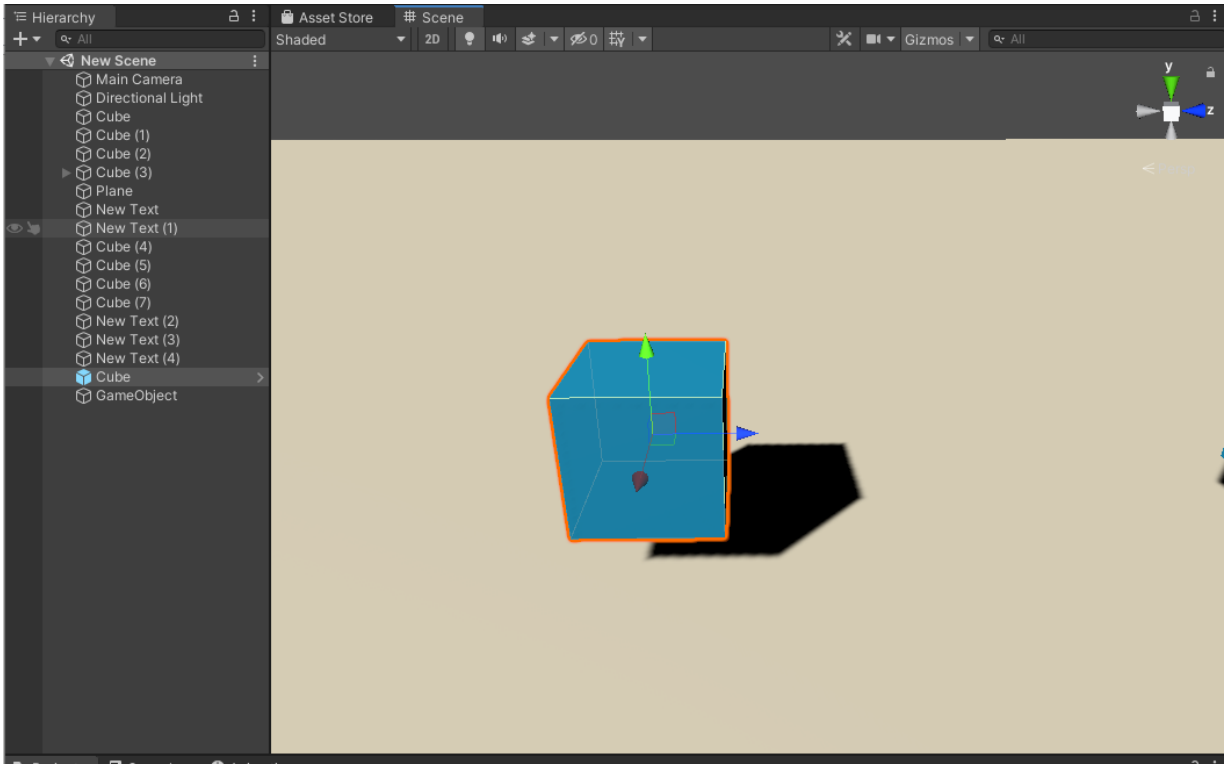


Figure 1.12 Scene and Hierarchy

In *Figure 1.12* above, there is a lot of information that can be seen right away. On the left, the hierarchy, you can see that there are objects in the scene. These Objects all have a transform which places them in the world. If you double click on an item or click on an item, put your mouse in the scene view then press ‘f’, you will then focus on that game object which puts the item centered on the scene’s viewport.

When you have an item selected, you can see at the object’s pivot point, usually center of the object, there is a tool showing colored arrows. The tool allows you to position the gameobject in space. You can also position the object on a plane by selecting the little square in between to axis.

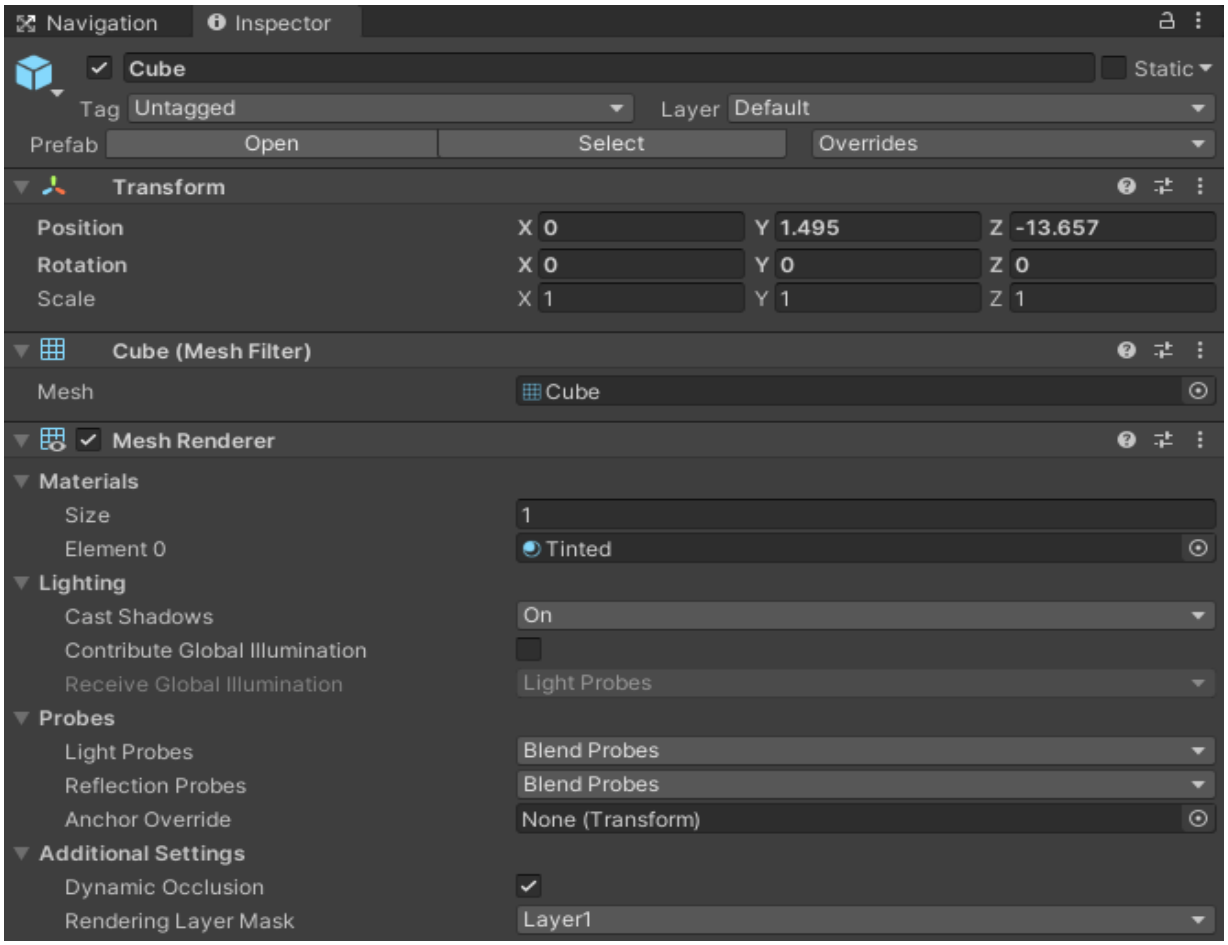
In the upper right of *Figure 1.12*, you will see a camera gizmo, this little gizmo will allow you easily orient the

viewport camera in to the front, sides, top, bottom, or change to an isometric camera or perspective with a single click.

Now that you have seen the item in scene, selected by left clicking in the scene or the hierarchy, you may want to change some properties or add components to that gameobject. This is where the inspector comes into play.

Inspector

To manipulate those game object's value, when you select the game object in the scene or hierarchy the inspector will update to show you the viable options to change per game object:



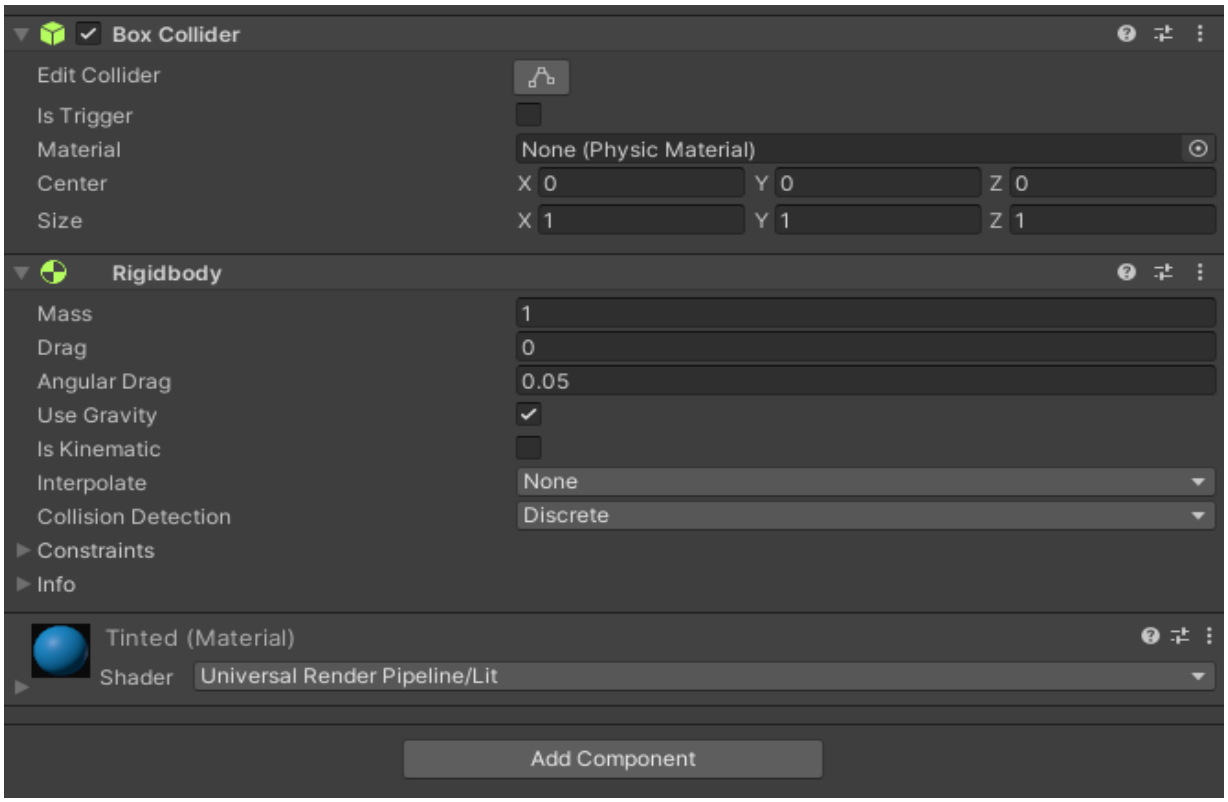


Figure 1.13 Inspector window

The inspector window in *Figure 1.13* shows a good amount of this item that has been chosen. From the top, the name is Cube and the blue cube to the left denotes a prefab data type. You are able to make changes to the prefab itself by pressing the open button just below the name. This will create a new scene view which shows the prefab only. When you make changes to the prefab it will make a change to all instanced prefabs in any scene that is referencing it.

The transform component is showing position, rotate, and scale of the prefab in the scene.

The mesh filter is the makeup of the vertices, edges, and faces that make up that polygon.

Below that is the mesh renderer. This component will allow the rendering of the mesh filter. We can set the material here and other options that pretain to this items specific lighting and probes which we will cover in lighting section of this book.

Now below this is a collider and a rigid body. These work in tandem and are helping this object to react to physics in real time to the settings available on the components.

We’ve talked a lot about items in the scene and their properties, but where are they housed outside of the scene if their only referenced items? The project window will answer this question.

Project Window

Here you will find assets that will be instanced in the scene or used as a component to fully realize the game you are building:

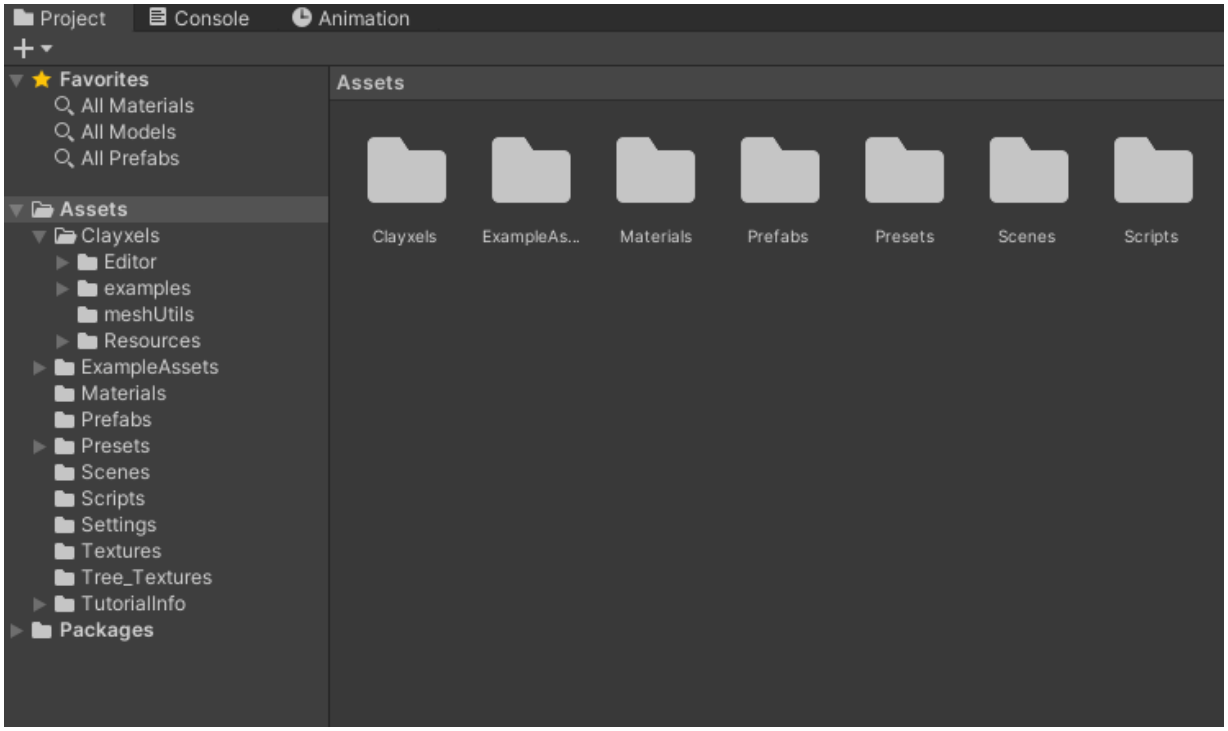


Figure 1.14 Project Window

This window is the physical representation of the game objects that are referenced from. All of the items are in the assets folder seen in *Figure 1.14* are physically on your hard drive. Unity makes meta files that are housing all of the properties of the items.

The interesting thing about having the raw files in the project window is that you can make changes to the items and when you focus on the Unity project (click on unity app), it will readjust the meta files and reload the items in the scene. This makes it so that way you can iterate on scripts and art faster!

We’ve look at the gameobjects in scene, placed them by manipulating the transforms, and know where the game object was referenced from. Now we should look at the game view to know how the game itself looks.

Game View

The Game View is similar to the scene view, however it's following the rules that are build in scene view. The game will automatically look through the Main camera in the scene or else wise defined:

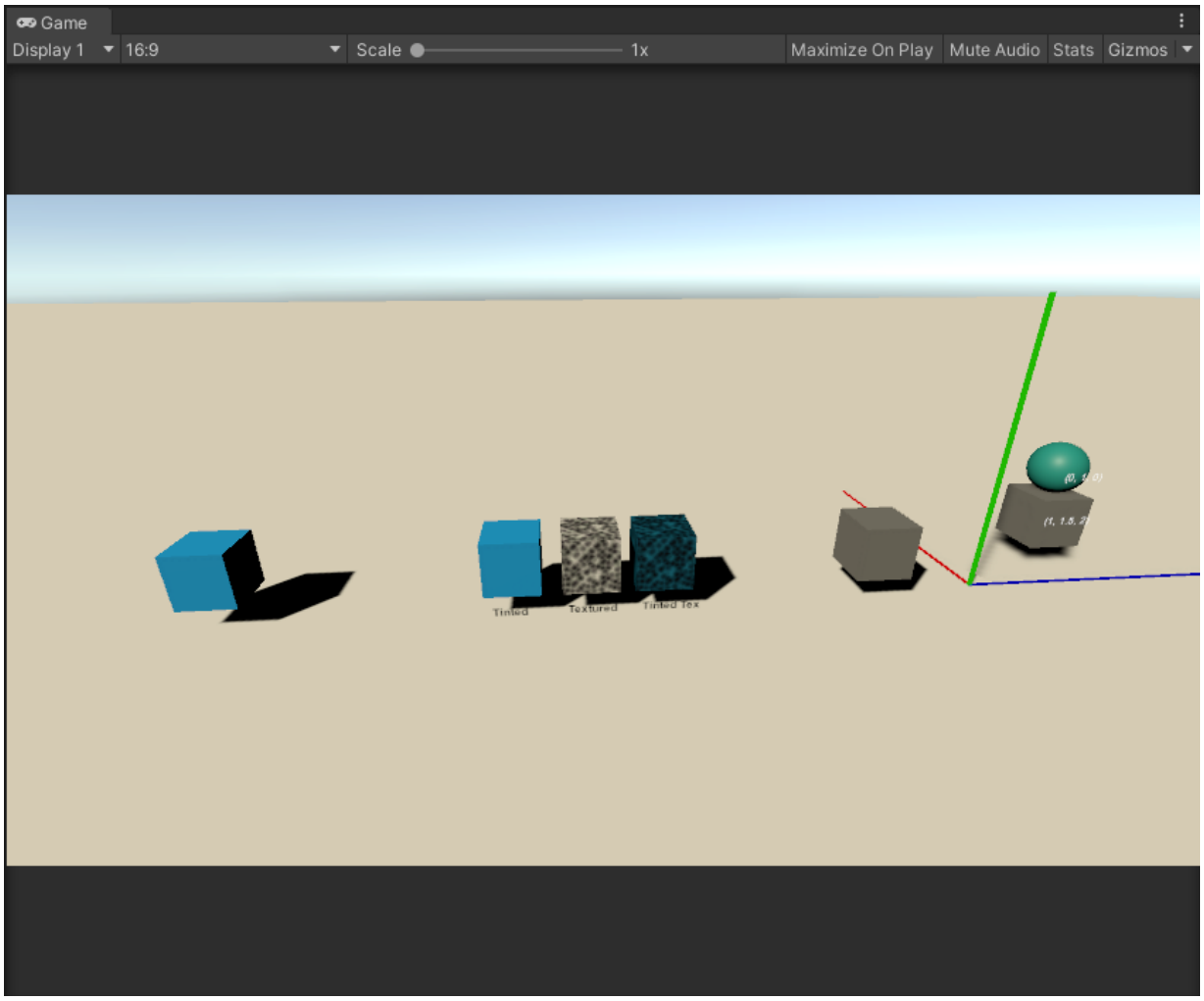


Figure 1.15 Game View

You can see that this looks very similar to the scene window, but the top has different options. On the top left we can see the Display drop down. This allows us to change cameras if we have multiple in the scene. The ratio is to the right of that, which is helpful to look at so you can target

certain devices. Scale, to the right of the screen ratio, is helpful to quickly make the window larger or zoom in for debugging.

Maximize on play will maximize the screen on play to take advantage of full screen. Mute audio means to mute the games audio.

Stats will give a small overview of the stats in the game view. Later on during optimization we will go through profiling to get a much more in depth way to look at what may be causing issues within the gameplay for memory usage and other optimization opportunities:

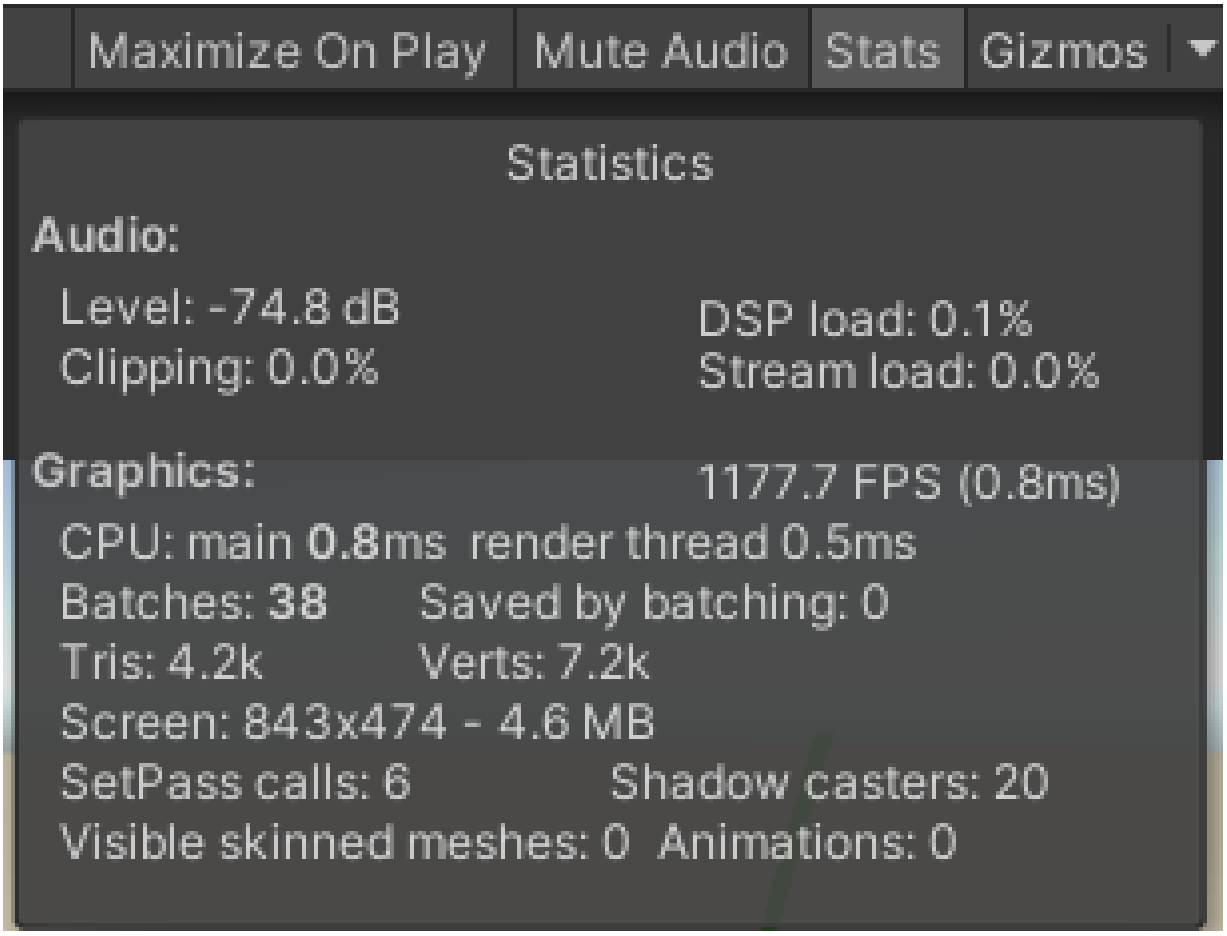


Figure 1.16 Game Statistics

Continuing on to the right is gizmos. This is a set of items that are showing in the game view in *Figure 1.16* which you might now want to see. In this menu, you are able to turn them off or on depending on your needs.

Package Manager

Your Unity ID will house the packages you've bought from the Unity Asset store as well as the packages you may have on your hard drive or GitHub! You can use the package manager to import the packages into your project. You can get to this packages under **Window → Package Manager** as seen on *Figure 1.17* below:

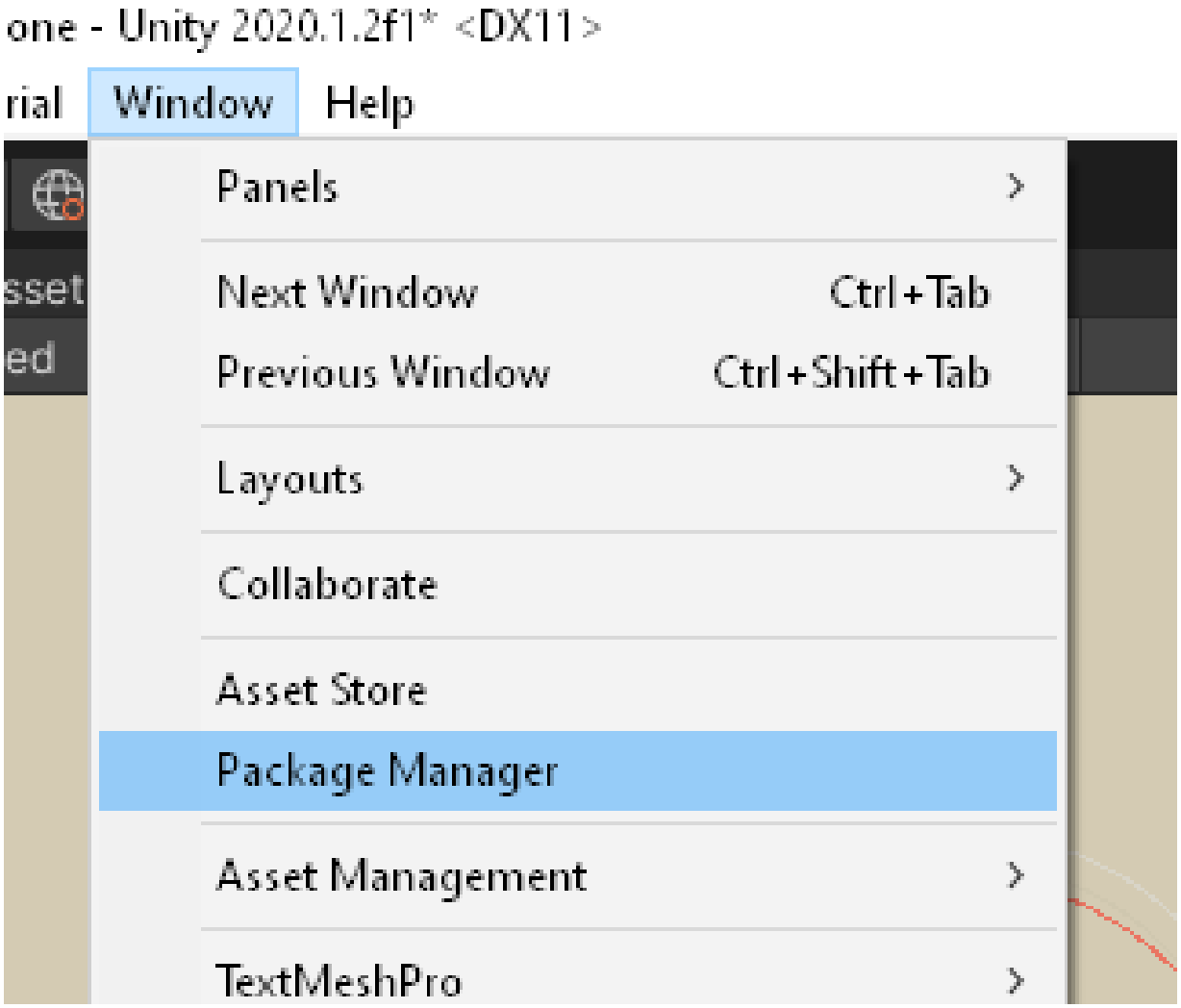


Figure 1.17 Package Manager Path

After you open the package manager, you will initially be shown what packages are in the project. You can change the top left dropdown to see what is standard in Unity or what packages you have bought in the asset store:

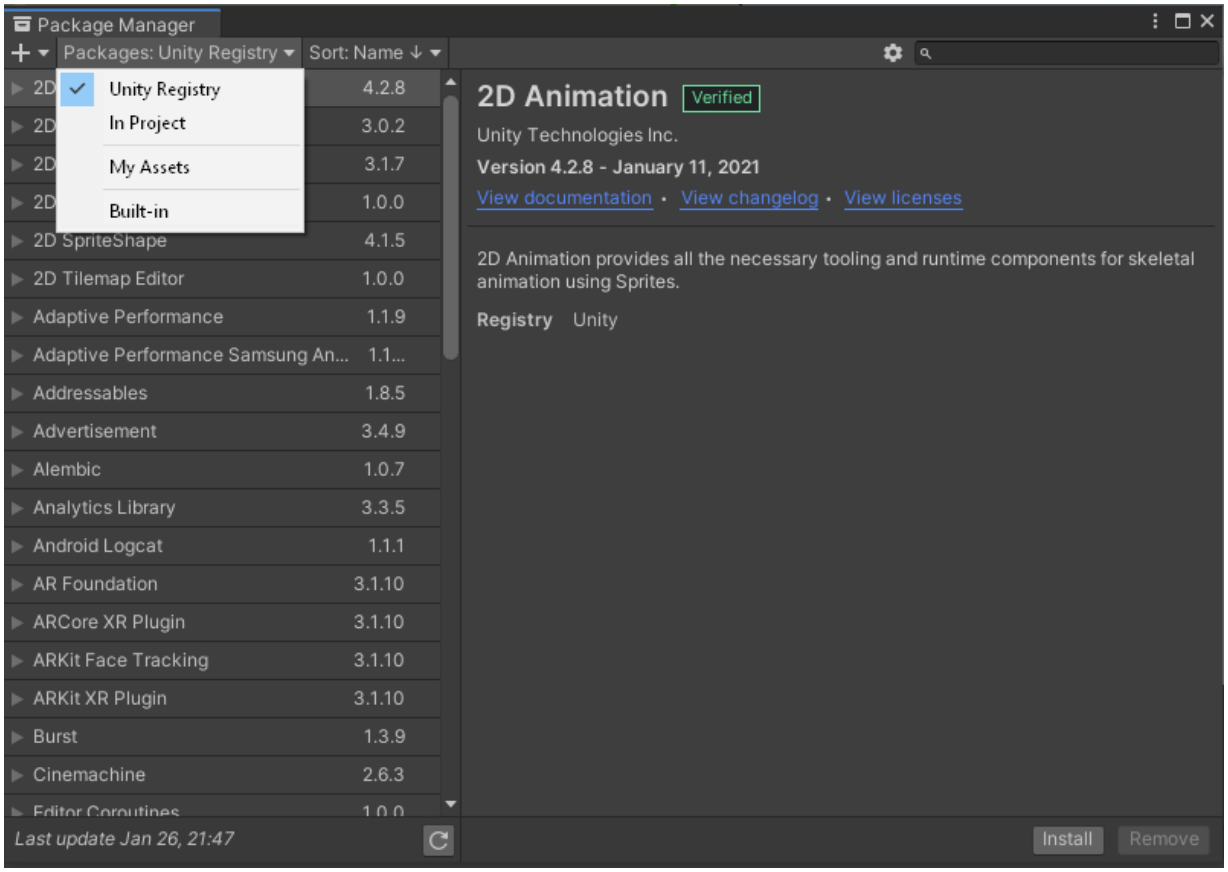


Figure 1.18 Package Manager

By choosing **Unity Registry**, this is a list of the Unity tested packages that come free and are part of the Unity platform available if you need them. You can read up on every package in the documents that are provided by the link in the right hand side, labeled **View documentation** when you click on a package on the left.

If you select the **In Project**, if will show you what packages are already installed with the current project that is loaded. This is helpful when you want to possible uninstall a package that may not be needed.

My Assets are the assets that you've bought or the project you are on and associated to your Unity ID has paid for previously.

Built in are standard with any project. You may need to enable or disable a built in package depending on what your needs will be. Explore them and disable what is not needed! Tidy projects now leads to less optimizations later!

Summary

Together we went over several key areas to begin your journey to game development. In this chapter, we laid the foundation for what is to come by going over some key fundamentals features of three primary topics:

Third Dimension

We went over the coordinate system which led into world and local space. This allowed us to talk about vectors, which is used to denote position, rotation, scale, and many other data points in development. We talked about the role of cameras and how they allow you to be the storyteller their their lens. We went through the facets of 3D meshes and how polygons are made up of vertices, edges, faces. These meshes can then be colored by materials which are driven by shaders and informed by textures. We then ended on the basics of rigid body physics and the collision detection which accompanies it. This was enough of the basics to allow us to get into Unity concepts.

Unity Concepts

To get into how Unity works, we needed to talk about the object oriented nature of the application itself and understand common terminology. We started with the most fundamental of items, the asset which is all things in Unity. One primary asset is the Scene, which ends up housing and has reference to all gameobjects for the game to run with.

Speaking of gameobjects, we dove into this heavily with defining what could be a gameobject and why it matters! Following this, we explained how gameobjects could house components with multiple properties and one of which you can script in C# to make your own properties and logic with them to create mechanics. After this, we brought up the basics of prefabs which are referenced containers with predefined properties on each gameobject. Finally, there is a package which is a unity file which houses multiple items you can place on the asset store as well as import into other projects.

Unity Interface

To end this chapter, we went through a virtual tour of the Unity interface. We looked at the scene and hierarchy view and how to view and understand the hierarchy of the parent child relationships of the game objects. Then we went into the inspector to manipulate the gameobjects that needed to be manipulated. After this we went into the project window to look at all the assets we have in the project. After that, we needed to talk about the game view to show where the game logic would take place and how it was different from the scene window. Finally, we talked over the package manager which has all the asset packages from Unity as well as the ones you've bought from Unity Asset Store.

Design and Prototype

Now that we have worked through all of the primary lingo of game development and have a stronger understanding of 3D spaces, we need to talk through the game itself. In this book we are building a portion of a game that could be. For this chapter, we're going to go into the beginnings of getting the project going. Major topics include:

- Game Design Fundamentals
- Your First Unity Project
- Prototyping

Game Design Fundamentals

Game design is a young art. With any art, there are some very basic fundamentals that must be thoroughly thought about before exploration can be done. We will go through ways that developers like to capture their thoughts into a "document". Then we will start with a micro lecture on how each decision should be deliberate as granular as you can go. Then we will build on those decisions with iteration. Then finally go through a concepting explanation. Let's get started with a discussion on design documents.

Game Design Document

No matter how small your game is looking to be, there will need to be some sort of documentation that will need to take place. There are strong organizational reasons to making a document, but the strongest reason is that to put something down on paper or drawn out in a collaborative space tends to be pause to an idea. This pause is sometimes

referred to as heuristic design. This can be done alone or a collaboration.

Some designers wish to make a beautifully written, well outlined, document in a word processor or an online collaboration tool. This gives a neat outline and ability to write into each detail strictly. When the scope of the game is going to be larger, this works out well. The writers tend to be technical writers and well versed into the art of documenting processes. The means of this is to have a single source of truth for any part of the game which anyone can refer while developing. To go to this extent may not be the best method for you or your team.

Another option for game design documents is done through collaborative brainstorming software. This allows the users working together to make flowcharts, draw, and outline in a creative manner. This manner is the direct opposite of the curated form of the written document form above, but serves a different need. The creative form tends to be more intimate and art focused. Some pre concept art sketches are done, flowcharts are drawn up quickly to draw out questions in the gameplay elements, and overall ideas can be quickly drawn upon and tossed or kept. This way of designing wouldn't work well for a large-scale team as there is no real organization of sorts. New members would have a hard time onboarding in these situations.

Neither of these options are the magic pill to make a game design document, but rest assured, your team needs to have some sort of document to keep your ideas written down. The mind is fleeting on ideas and some of the best ideas slip into the ether as they aren’t written down to keep. Experiment with your team to find the best option for your team. There is a saying in design groups. The best tool is the tool your team will actually use.

Now that we have started on documenting our game design, we need to take our thoughts and make deliberate choices with them to make them concrete.

Deliberate Decisions

Though this part of the chapter may be slightly smaller than others, but take this section to heart. Being a designer means building an immersive world which makes sense, even when it doesn't make sense. The player subconsciously makes observations at an alarming rate. The more that the player can see noncongruent pieces to the puzzle that is the game environment or character, the immersion is broken. The winning medal to fix any immersion breaking issue is deliberate decisions.

To give a very simple explanation of this is door handles. You've seen them your whole life and used them intuitively. In fact, when you had to deal with a badly design door handle is when your actual real-life immersion breaking happens. If you have ever grabbed a handle to a door and tried to pull inwards only to find out the door was designed to be pushed, you've ran into this issue. If the door was designed to only be allowed to move in one direction, the correct design for an exit is a flat panel where the door handle would be. This immediately implies "push".

Every level, mesh, texture, concept, or feeling needs to be deliberately thought about with an attempt to implement. Only after you have a strong reason to place something in a certain way without giving into cliché is when you then can explore other unique aspects to build something truly unique.

The project you will be making and playing with in this book has undergone heavy deliberate thought. To make a

strong point of this, each section you will see a set of questions that are answered in as much detail as needed in a concise matter.

Iterative Production

Game development has an interesting need for immersion to be at the forefront of play. To get this in a complete manner, the development team needs to continuously ask if the direction it is going in works well. Very often the game that you began developing is not what you intended in the beginning. This cycle is called an iterative design or production.

The pattern of design can be in a multitude of ways. The way that will be described here is not a definitive only way to complete a design, but is a good thought process to start from and branch off as your team sees fit.

Iteration needs to happen often and early for a game to grow its strength of understanding. There is a concept called **MVP**, or **Minimum Viable Product**, where the game developers make the minimum needed amount of gameplay elements to give the game to testers. This should take very little time and the feedback is invaluable. The combat may not have been as snappy as you thought. The testers may have not understood what to do from the start! This feedback forces you and your team to iterate on the design and possibly cut or add mechanics to answer to the major testing feedback.

After having iterations resolving major holes in your design, you then move into a vertical slice of the game. This should be an iteration where you are comfortable with the basics of the movement and primary game mechanics. Your team will want to make a full game loop from start to finish

with a single level that houses a win and lose condition. Then you guessed it, test again. This time with new testers that have never seen this game. Ask similar questions and some new ones that you have surfaced while internally playtesting.

The loop for development should seem repetitive and it is:

- First think and test.
- Create and Test.
- Update and Test.

Then continue this until you are at X number of iterations with a shippable product. Now that we all know the cycle, let's get into the starting portion which is concepting.

Concepting

You've got need to make a game and you have a group ready to go. You are comfortable with deliberately making granular decisions and know the iterative process. Now you need to get a concept started.

The first step to get a project going is to explore what emotion you and your team want to have your players experience. With our art form being so young and malleable, we can pursue this emotion is any way that we feel. This is the power of the game developer. After you know what emotion your focused on for the players experience, start thinking about how it may come about as an experience.

If the emotion is fear, you could have the player deal with spaces that are dark with just a flashlight as their primary defense tool. This may lead you to exploring sound design

being your development focus since vision will not be the primary tool of experience.

If the emotion is grieving, then you may work through narrative focus, where you play a child who has lost a family member and the players works through a narrative driven gameplay in a dream world. This pushes storytelling and pacing with a tight understanding of color theory as well as the stages of grief through a child’s perspective.

We could go on for a while on concepts as there are an infinite number of scenarios. Make a choice on what your primary goal is then work through it. After this, you may want to put some of these ideas to paper to get an idea of what the feelings of the immersion will be artistically. This could potentially be silhouettes of character concepts. It could also be **Architectural designs**. It could also be a collection of pictures you’ve saved that gave you a feeling of the emotion you want to evoke that you can draw ideas from.

Either way, the prominent action is to visually get the ideas started. After you have some art panels drawn up and you have a visual idea on how this may be built, we would then make a Unity project.

Your First Unity Project

Choosing a Version

Unity runs multiple versions at once. There are Alpha, Beta, Official, and LTS releases. Alpha versions have experimental features that may not be fully complete or production ready and are not recommended for builds as there may be features causing build breaking bugs. Studios and enthusiasts may use this for mechanics testing or new

packages testing. They are generally one release ahead of official releases. Beta is similar to Alpha versions; however, they are experimental versions of the most current official release. Official releases are stable current releases. LTS means Long Term Support. These are the final releases of a version with minor hotfixes if bugs are found.

An easy way to see the versions is through the Unity Hub. An example of what this may look like is in the figure below:

Recommended Release

- (●) **Unity 2019.4.21f1 (LTS)**

Official Releases

- ( ) **Unity 2020.2.7f1**
- ( ) **Unity 2020.1.17f1**
- ( ) **Unity 2018.4.32f1 (LTS)**

Pre-Releases

- ( ) **Unity 2021.2.0a6 (Alpha)**
- ( ) **Unity 2021.1.0b10 (Beta)**

It is recommended for a production application the LTS Releases. If your team is experimenting or looking to prototype with new features, it would only be possible in prerelease versions. After choosing a version for your project to be built in, you need to choose a template from Unity's options as you make a new project.

Choosing a Template

Unity presents you with a few template choices when you press the **New** button on the projects tab. This gives you an option of 2D, 3D, **Universal rendering Pipeline** (**URP**), and **High-Definition Rendering Pipeline** (**HDRP**). Underneath all of these templates are large rendering differences as well as some functionality that might be interesting to you and your team to work with. This split of differences came when the **scriptable rendering pipeline** (**SRP**) came to life!

Scriptable Render Pipeline

Rendering and computer graphics are a deep subject that you could get a PhD in, so we will scratch the surface of what is possible through a rendering pipeline. The top level of the pipeline is three tasks: **Culling**, **Rendering**, and **Post-Processing**. In each of these categories there are many tasks that are taking place in certain orders and to certain degrees of accuracy. The primary function of all of this is to optimize the view to the end user for high framerate as well as maintaining the art style that is required for the experience you are wanting for the user.

With the advent of the SRP, these templates splits into three main categories. **Built-in**, **Universal**, and **High Definition**. To get an idea of those three templates let's

break them out into their respective groups and dive in a little bit further.

Built-in Rendering

This is an older pipeline which doesn’t use scriptable pipelines. There are a great many applications for the built-in renderer. Both of the 2D and 3D templates are running built-in rendering systems. This is also the standard in which most of the assets in the asset store were built for prior to the SRP coming out. You can think of the built in as the base experience in Unity. There are several reasons you may not want to use the built-in renderer. If you aren’t well versed in shader language, you want to use GPU particles, or use raytracing or pathtracing.

Universal Rendering

The Universal Rendering pipeline is aptly named as it has the most features available with a scriptable render pipeline available. If you are looking to make a 2D game, this is the best option to choose as it has built in pixel perfect rendering, 2D lights, and 2D shadows. For 3D option, this is also a fantastic choice. There are two graphs available to both URP and HDRP, which are **Shadergraph** and **VFXGraph**. Shadergraph is a visual shader creation tool that allows for complex shaders to be written in a visual manner. VFXGraph’s primary function is to be a particle system focused on GPU particles allowing you to create millions of particles on screen at the same time for stunning visuals.

If you are looking for more of a physically accurate rendering system with raytracing and volumetric clouds, then HDRP is what you are looking for.

High-Definition Rendering

This rendering pipeline has one major purpose. Give the best looking output while remaining as optimized as possible. Whether to use HDRP or not is a widely discussed topic. There are several main reasons why HDRP would be the option for you. This is if you are looking for physical based sky with cloud layers, volumetric clouds, multiple directional lights, highly customizable shadow options, raytracing to include raytraced reflections, volumetrics, and multiple high level shader outputs. There are many other high level rendering options that HDRP can only provide. These concepts are deep topics in the computer graphics world and we implore you to look them up to see the beautiful work of what real time rendering is becoming.

Prototyping

Now that you have a project, you can start putting together the assets that will create the game. In this book we have worked through how we are going to build this game out so that way we can section off each major chunk into chapters. Prototyping can happen in a multitude of ways. We can’t go over every way any studio will prototype as each business has their own manner of creation. We will speak to major progression patterns that are common within the industry as a whole. Breaking down the life cycle of any task that is built upon iteration, there needs to be a cycle to pull out the impurities. Commonly seen as Analyze, Design, Implement, Test, and then iterate again until complete. The prototyping phase goes through all of these steps as well. Take a look at them all and work through each portion that makes sense to yourself or your group building your game.

Wireframing or Paper Creation

In this form of prototyping, the creator takes video game into steps in a physical or digital system to go through each game loop or experience the player will feel throughout your game. Sometimes this could be creating a paper board game to run through the rules. Sometimes this is digitally drawing literal game wireframes through the user interface to feel the gameplay out.

Greyboxing

The name is as you may think! A bunch of untextured shapes, generally boxes, that line out your environment to ensure the storytelling of the environment through its silhouette can be defined. This version of prototyping is particularly useful if you need a very direct camera angle you need to display and do not have assets to set up the environment to prove it out. This can be useful as well to develop concept art as you can push the composition to the concept artists for quicker turnarounds.

Proof of Concept (POC)

The naming is fairly accurate. This is where you are getting very specific about your testing. Maybe you need to tune a camera to get a very specific feel of the gameplay. This might take several iterations in itself and multiple people taking a crack at it if you have a team.

Minimum Viable Product (MVP)

This is as the name implies. A boiled down version of the game. In a platformer, there must be jumping. Maybe you have swinging as your game demands that as a mechanic

that will not get cut no matter how much funding you have? You do not need polished art assets or even animations. The purpose of a MVP is to prove out the gameplay features to an acceptable range to ensure anything on top has the foundation of the MVP’s mechanics working properly.

Vertical Slice

Sometimes you have a good idea about the art direction, the primary mechanics, and narrative, but need to gather some feedback or possibly funding. A vertical slice is when you take a very thin slice of the game and polish it to gain hype and a sense of the end product. Demos are a similar concept of a vertical slice. This is more complex than the MVP as there is a level of polish to the art, animations, mechanics, lighting, etc. That MVPs aren’t expecting to need which can take significant amount of time as well as understanding of the final product which may not be available when the MVP could’ve been made.

While making prototypes, your game may need to go through all of these to get to a proper gameplay that feels good to play. You may also not need all of them. This varies greatly within each development group.

Summary

Within this chapter we went over some deep topics. Game design, what options to choose for your first project, and prototyping fundamentals. Let's take a little bit of time to go over them in a conclusive manner.

Game Design Fundamentals

Game design is a wide field of study. We covered some strong topics to consider on your further study throughout this book. The game design document is a topic that was a great place to start. We went over how you and your team could collaborate together in a word document or more visual flowchart manner. The tool doesn't matter as much as the willingness of each team member use of it. After you work through which tools work best for you, you will then need to look at the decisions you make with your team and ensure that they are being deliberate. Even when your decisions feel right at the time, putting that thought to someone else and seeing how it feels overall, revising, then repeating the process is called iterative design. Iteration is fundamental to game development. After the design is starting to come together, concepts need to start forming. Whether this is through storyboarding, environmental concepts to get the feel of the style, or character concepts to start the journey of the characters development, these are here to get the design's ideas forming into reality. Now you've come this far, let's get into Unity and talk about the project.

Your First Unity Project

Unity has several starting options and versions. This will continue to evolve over time as optimizations and functionality grows. Versions going up means growth. Growth in bug fixes as well as new features. Keeping an eye on the documentation and version notes shows the roadmap to what is available in each version. Choosing which template to work with can seem daunting as there is a lot of information that feels like it's needing to know. If you are first starting out, work with URP as it fits the most available applications for 2D and 3D with features that are only available to SRP templates, such as ShaderGraph and

VFXGraph. Now that we've worked through basic game design, we need to talk about prototyping.

Prototyping

There are several ways to get into prototyping. In our project we will be prototyping as a vertical slice of a larger project. In this section we went over prototyping that didn't include Unity as a tool. This is to include making a board game if it's a combat driven game or wireframing if it's a narrative driven game. These could be done outside of Unity. In some instances, this is considered a proof of concept as well. The point of a prototype is to get your project started and get a sense of if it is conveying the experience to the user that you want it to convey. If you are in Unity, then you will most likely use a greybox method to begin. In some cases, greyboxing might even be part of the iterative design portion and concepting game design as Unity provides a very simple platform to just start building!

Programming

Welcome to Chapter 3! We are going to cover all of the basics of C# and how to use it in Unity. We're going over the primary chunk of programming knowledge needed to be used in most projects. This chapter should serve you as a referenceable chapter throughout the rest of the book when we get into scripting for each future chapter. We will first need to begin with ensuring that your computer's environment is set up to start programming with Unity.

Environment

Programming environment is specifically talking about the **Integrated Development Environment** (**IDE**) that you will use and the dependencies associated with it. C# is part of the .Net framework from Microsoft which needs to be installed on your machine to work. Luckily for us, the majority of IDEs in the world will install that for you when you start working in C#.

Unity Environment

Microsoft Visual Studio is free and directly connects to Unity and comes with tools to help you right off the bat! This is like needing to work on your car and someone just hands you the right tool for what you need as you put your hand in the engine bay.

There are some steps to ensure that every application is talking to each other. Let's walk through them together so we can ensure we're on the same page moving through the book as well as the rest of this chapter as we will work through small snippets of code right away. No need to check the water's temperature, let's dive in!

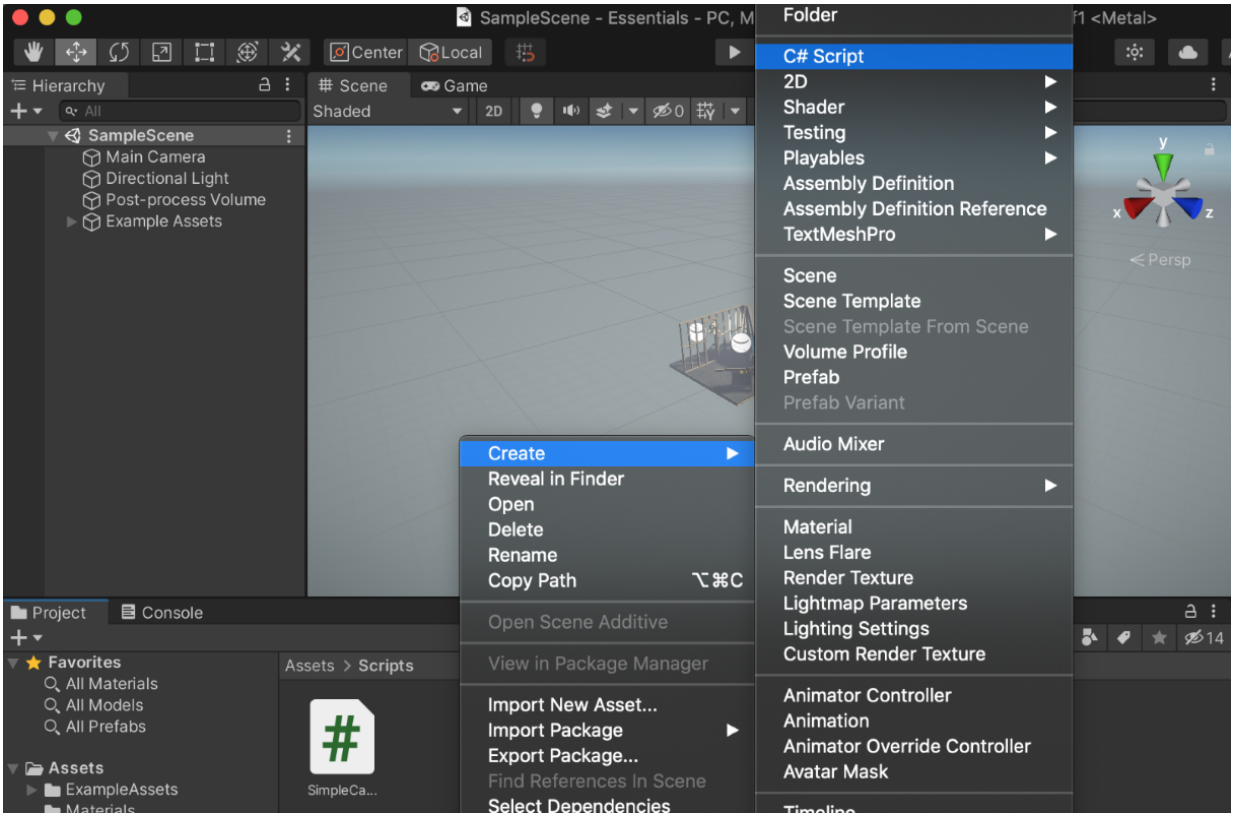
Step 1. Install Microsoft's Visual Studio (Community version, it's free). If you search for "visual studio community download" it will be the first link. After it is downloaded and you start to install it, there are Unity specific extensions to help with your development. If you are working on a Mac, the tools will be installed by default. If you are on a PC, when the installer starts the installation process you will need to scroll down and there is a gaming section with "Game Development with Unity" as an option, select the checkbox and continue installation.

Step 2. Now that we have Visual Studio installed, let's connect Unity to it and start learning. First, close Visual Studio and open up the Unity Project we created in the previous chapter. If you didn't make one, no better time than now.

Navigate to these respective menus to connect Visual Studio to Unity:

- Mac: **Unity** (Top left of the screen) -> **Preferences** -> **External Tools** tab
- PC: **Edit** -> **Preferences** -> **External Tools** tab

In the external scripting tool dropdown choose Visual Studio. After this has been selected, go to your assets folder in the project window and right click in the grey open space. Choose **Create** -> **C# Script**. Name it "`ScriptingLesson`" and double click it to open it. This should open Visual Studio with the proper hooks from Unity. This means that Visual studio will read in the project file and Unity will keep it up to date with any changes. Now we can get started on scripting!



Fundamentals

With Visual Studio installed and connected it to Unity's editor, we should go over the basics. In this section we will talk about data types, variables, logic or code flow, functions, classes, and MonoBehaviour. There is a lot of knowledge to be held in this section of this chapter, but it is meant to be referenced. If you have a sticky note, it might be a good idea to place it in these chapters to easily be referenced. When you open the file, there will be autopopulated C# that we will not need for this part. For now, delete the extra parts so that way it looks like this:

```
using UnityEngine;

public class ScriptingLesson : MonoBehaviour
{
// Data and Variables
// Logic and Flow
// Functions
}
```

The code here is doing two primary tasks. The first line imports the library of `UnityEngine` so that we can use classes from `UnityEngine` for our game. It also allows us to create the next portion which is a class named "`ScriptingLesson`" that inherits from `MonoBehaviour`. We will explain all of this in Classes and `MonoBehaviour` sections of this chapter.

The "`//`" means comment. Anything on that line will not be considered by the IDE. You can use this to help yourself out by pseudocode or adding some defining words on your code to help other programmers that may work with your code. We're using it as some organization.

After you make the change to the script, save by pressing *cmd* + *s* or *ctrl* + *s*.

If you then go back to Unity, you will see that Unity will compile the scripts. Every time we make a significant change to the script, we will go back to Unity and check on how we are doing. There will be times that Unity editor will not like the code we are working with, but Visual Studio doesn't get these editor warnings or errors.

In the Scene, add an empty gameobject in the scene. Name it "`Scripting Lesson`" and then select it. In the inspector, click "**Add Component**" and type "`scriptinglesson`" in the search bar. Left click the script to add it to the empty gameobject. Now when we make changes you will see it on this game object. Before we get into any data types, we should have a small discussion about variables.

Variables

Just like in Algebra, variables are named containers for something. C# is a type strong programming language. This means that each variable does have to have its own data type associated to it when it is declared. There are guidelines on how to name your variables and which data type to use for certain situations. We will go into detail of this within the next few sections.

Data Types

There are 10 unity data types that are utilized in C#, however in Unity we will primarily need to know 4 really well. They are bool, int, float, and string. We will be creating each of these data types in the "`ScriptingLesson.cs`" file that we created.

Bool

This stands for Boolean data type which is designed for either true / false variables. These values are also represented by a 1 (True) / 0 (False). As an example, this could be used when your character entered an area that they shouldn't have to trigger something to happen, like a SPIKE TRAP!

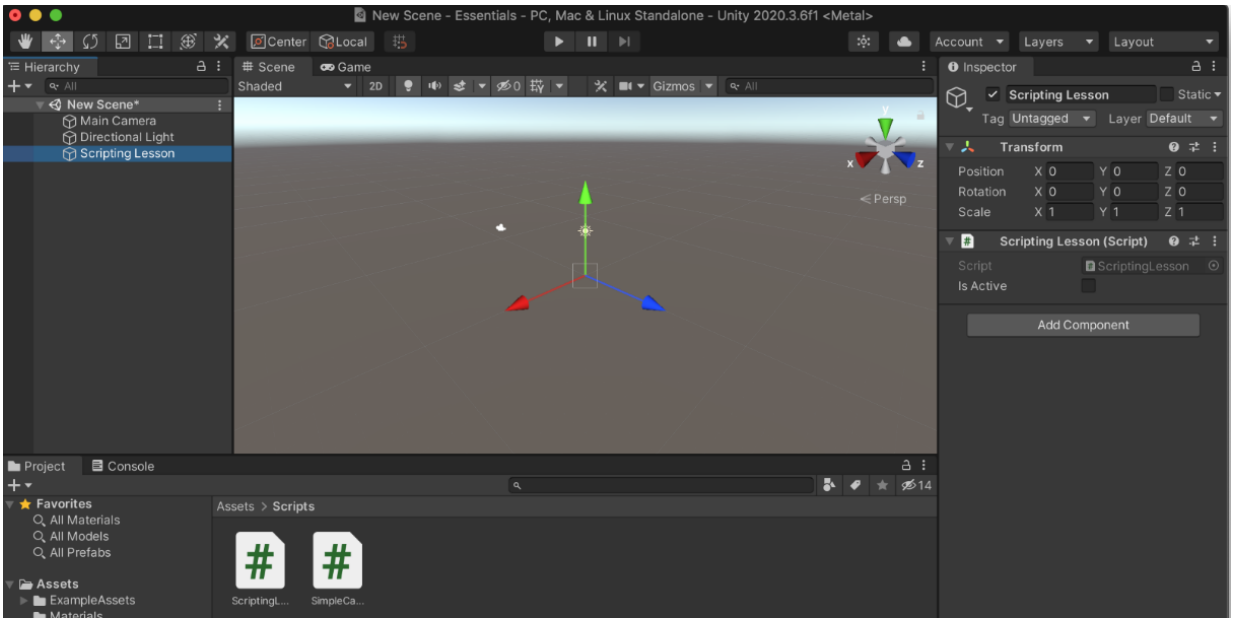
On line 5 add:

```
public bool IsActive;
```

There are 4 parts to this line and all of them have specific purposes.

- “ `public` ” allows Unity to access the item we are creating for use in the editor.
- “ `bool` ” is the data type of the item we are creating.
- “ `IsActive` ” is the name of the bool data we are creating. It will default to false as a value.
- The semicolon (;) is being used here to denote that it’s the end of the instruction.

If you save and go back into the Unity editor, you will now see there is a checkbox in the inspector named “ `Is Active` ”. It should look similar to this:



Int

An Integer is a whole number such as 1, 100, 200, -234571, or 0. It cannot house decimal places. This is used, for example, if you have a discrete value

needed to count up or down. How many points are gathered in a play session is a great place to use ints.

On line 6 add:

```
public int MyInt;
```

This is very similar to the bool. We are declaring that “`MyInt`” is a publicly accessible integer data type variable. When you save and go back to the Unity editor you will now notice a text input to the right of a variable named “`My Int`”. Since it is an integer, you cannot put a (.) in there to make a decimal as it’s an int and only whole numbers are allowed.

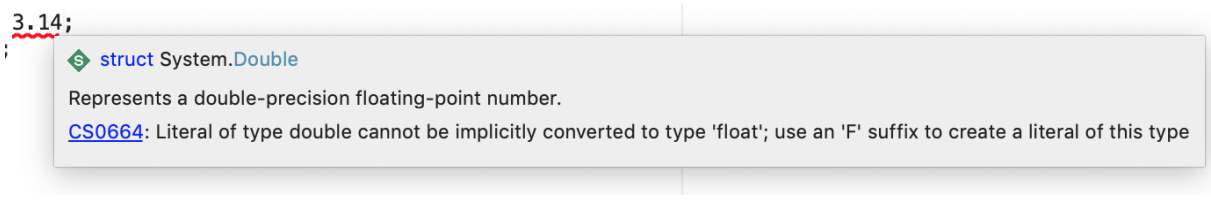
Float

You might be asking; how can I get decimal places in my numbers? The answer you seek is the almighty float! With a float you can get decimal places such as 1.3 or 100.454. There is a small unique factor to the float. When you are scripting you must put a small f behind the value to help the compiler know the value is a float. C# assumes that any number without ‘f’ at the end, such as 3.14 is assumed a type of ‘`double`’. We won’t be using a double for our scripting, so we need to remember adding the little after our floats.

On line 7 add:

```
public float MyFloat = 3.14;
```

When you tried to enter this, there was an issue with the number and there was a red line underneath the “`3.14`” right? If you hover over the red underlined area, you will get an error. Might look something like this:



Visual Studio is trying to tell you that the number you entered will be thought of as a double., so let’s make a bit of change to help out the IDE.

Change line 7 to:

```
public float MyFloat = 3.14f;
```

There we go, now we have a float declared and initialized. We declared it as `MyFloat` and initialized the value to be 3.14. The default float is 0, but when you tell it to assign a value as you are declaring it, the IDE overwrites that default 0

with the value you set it to. When you go into Unity and look at the inspector you will now see the value starts out as 3.14.

String

We've gone after numbers this whole time. Now it's time for letters to shine. Strings hold the values of letters. An example of this could be the character's display name.

On line 8 add:

```
public string Mystring = "Myvari";
```

This looks the same as well, but now you can add letters! The interesting point to be made here is that these public values' inputs all look the same, so we need make sure our variable names are unique.

Gameobject

This is an interesting data type as it's unique to Unity. Its reference is to a gameobject that you place in it from the scene or prefab. This is extremely powerful as the item you place in here has components and we can access them from the script to do a great many things.

On line 8 add:

```
public GameObject MyGameObject;
```

Save your code and head back into the editor. You will notice this time instead of an input field, it is wanting a gameobject to be placed here. In our scene there is a directional light. Let's drag that light from the hierarchy and into that spot. You now have a reference to a scene gameobject! Let's continue a bit further to logic and flow to see what we can do with our initial variables.

Logic or Flow

We've created some amazing variables filled with such fantastic data. The only problem is that we aren't doing anything with it. Why don't we start working in some logic into our little script to start making some connection as to why we would need to program in the first place. For this we are going to go into if statements and while loops.

Before we get to doing some runtime work, we need to add `MonoBehaviour` to our class. The term for the actions we're about to take is inheritance. We will derive out class from `MonoBehaviour` which will give us access to its class by inheriting them! To do this is very simple.

On line 3 change it to read:

```
public class ScriptingLesson : MonoBehaviour
```

All we did was add “ `: MonoBehaviour` ” after our class name, and now we have access to any of the methods inside the `MonoBehaviour` class that are available to us. There are quite a few methods we are able to use from `MonoBehaviour`. For now, we will use about the `Start()` and `Update()` methods inside out if, for, and while loops.

If Statements

We’re now going to set up some simple code to turn a game object off and on according to the state of the `isActive` bool we’ve defined. To do this we will need to check it on an update function which is part of `MonoBehaviour`.

Starting with Line 13, we made these changes:

```
private void Start()
{
    isActive = true;
}

private void Update()
{
        if (MyGameObject != null)
        {
            if (isActive)
            {
                MyGameObject.SetActive(isActive);
            }
            else
            {
                MyGameObject.SetActive(isActive);
            }
        }
}
```

Inside the Start function from `MonoBehaviour`, we are setting `isActive` to true. This was added in here to set the boolean to be considered on regardless of what was set in the editor.

After that there is an update function. The update function from `MonoBehaviour` will make a check of the entire code inside the curly braces every frame. Initially we check out the gameobject we defined by comparing it to a ‘null’. This is validity a validity check. Null is a special keyword which denotes the lack of type or data. If you do not perform these checks and your editor will not be able to play as there will be a null exception.

Inside the validity check, we have an if/else statement. It’s currently saying, if the `isActive` variable is true then set `MyGameObject` active. Anything else than true set `MyGameObject` to inactive.

If you save and press play, you will be able to select the scripting lesson game object and then uncheck the `isActive` boolean checkbox. It will turn toff the light. Since this is checking every frame, you can do this forever and it will turn on and off until your hearts content. Next, we will go over a while loop.

While Loops

An if statement is a simple branching pattern checking for a true or false to execute something. While loops will continually run code until the statement is true or false. This can cause issues as you could imagine some tasks could go on forever. This is called an infinite loop and can hang your application indefinitely or until it's forced to be closed. For the most part we catch infinite loops fairly quickly and don't cause as much of a fuss. We should still pay attention to our criteria when creating a while loop.

On Line 32 add these lines to the update function:

```
while (MyInt > 0)
        {
            Debug.Log($"MyInt should be less than zero. It's currently at: {MyInt}");
            MyInt--;
        }
```

There are several new things were doing here in this while loop. We are doing a debug log, string interpolation, and a decrementer. Let's go through those.

`Debug.Log` allows us to pass in a string and it will be written out in the console inside Unity. This is very helpful if there are strange things happening and you want to get some runtime information out to console.

Inside the log, we are performing an action called string interpolation. This is a very handy way to add your variables into a string. This is started off with a $ followed by double quotes. Inside this is a string that you want to write out. It is a literal string to include spaces. The interesting thing is that there are curly braces {} inside the string! If you place the variable name inside the curly braces, you will get the data passed into the string.

The next line is a decrementer. This is an efficient way to write this line:

```
MyInt = Myint – 1;
```

If you save this and then run it, there will be nothing in the console. This is because we haven't set `MyInt` to anything, so it defaults to 0, the while loop will no run because `MyInt` is not greater than 0, it is 0. Let's make this change:

On Line 16 add this:

```
MyInt = 10;
```

Save and run the game. Now if you look at the console it will quickly decrement `MyInt` down to 0 and print out the line letting you know it's current value. If you look in the inspector, you will see that `MyInt` is showing 0 as well.

Now that we have an idea of logic and flow or the logic, let's add some functionality.

For Loop

Similar to while loops, for loops are iteration over a set number of items or scale. For loops are most used if there is an idea of how many times the iteration will need to run. We will do a simple for loop to show the syntax of it and go over it below:

First, let's comment out line 35 as we don't need all the while loop debug log cluttering up our for loop debug lines.

Then, on line 39 add this block of code:

```
for (int i = 0; i < 10; ++i)
        {
            Debug.Log($"For Loop number: {i}");
        }
```

Choosing between For and While

Functions

If logic is the butter of programming, this is the bread. The purpose of functions is to perform actions in a concise manner. A very simple example of a function is a basic calculator function call "`Add`". To perform this function, we will do three things. Make some public variables to add with, have a way to trigger the function, and the function itself. Previously we had asked you to insert into a specific line. We will now just ask you to insert it into the specific sections.

In the class's variable section add these lines:

```
public int AddA;
public int AddB;
public int TotalAdd;
```

We're making them public so that way we can make changes to them and see that they are able to be changed during runtime when we run the function.

Next in the update function add these lines:

```
if (Input.GetKeyDown(KeyCode.A))
{
    Debug.Log(IntAdd(AddA, AddB));
}
```

Using the if statement to check if the letter '*a*' is pressed, we will then log out the `IntAdd` function passing in `AddA` and `AddB` as arguments. We've used a few new terms here; they will be more apparent after we add the next lines in.

Finally, below and outside the update function let's add our new function.

```
private int IntAdd(int a, int b)
{
    TotalAdd = a + b;
    return TotalAdd;
}
```

This is a function that is private, which means outside of this class we can't access this function. This is going to return an int and its name is `IntAdd`. Inside the parenthesis after the name is arguments for the function. In our case, we have two ints *a* and *b*. We need to define their data types and their names. If you look into the update function, we are asking the `IntAdd` function to run using `AddA` and `AddB`. When the function is running our function makes two ints with the values at that time and assigns them to variable *a* and *b*. We made `TotalAdd` equal to it so we could show the values change in the inspector as well as in the console to debug it out even further.

Click on the script in the hierarchy and then put some values in the `AddA` and `AddB` variables in the inspector. Start the game and press '*a*'. You should see the `TotalAdd` change as well as the console print out the number.

We will be using all of these features when programming later on as well as adding new libraries and different implementations of classes. This is the foundations to programming; we will grow upon it thoroughly throughout the book. If you ever get lost, refer to the github as there will be these scripts in their completed form which you can reference from if something isn't working correctly.

Feedback

Have your say! Help our authors to provide useful information to customers just like you. Fill out our quick surveys to give your feedback. We'd love to know more about your experience as a reader.

- Chapter 1: https://forms.office.com/r/S2zEdzLU9T
- Chapter 2: https://forms.office.com/r/9Hx6LY1vY2
- Chapter 3: https://forms.office.com/r/9tRZCjU9zm

Remember that Early Access chapters are first drafts, so they haven't received a final polish from our editors yet.