Introduction to Game Developement

Om Prabhu
19D170018
Undergraduate, Department of Energy Science and Engineering
Indian Institute of Technology Bombay

Last updated July 16, 2020

NOTE: This document is a brief compilation of my notes taken during a course in game design and development. You are free to use it and my project files for your own personal use & modification. You may check out the course here: `https://www.coursera.org/learn/game-development?specialization=game-development`.

Contents

---

1 Introduction

1.1 About myself

Hello. I am Om Prabhu, currently an undergrad at the Department of Energy Science and Engineering, IIT Bombay. If you have gone through my website (`https://omprabhu31.github.io`) earlier, which is probably where you found this document too, you will know that I love playing video games, story-rich titles in particular. I also listen to a lot of music and engage in a little bit of creative writing as and when I get time. With this brief self-introduction, let's get into what actually motivated me to pursue game development.

1.2 Motivation

Most of my motivation for pursuing game development came from playing games itself. I am talking less of titles like *Grand Theft Auto*, generic FPS/RPGs, etc meant purely for self-entertainment and more about games like *Life is Strange*, *When the Darkness Comes*, etc that actually give you some amazing stories and/or simple, powerful messages to be remembered for life. When one has experiences like this, the question naturally hangs at the back of their minds - why not create enriching experiences like this?

Now while playing games (of all genres) is vital to understanding what essentially makes a good game, they are two very different things - it's like comparing movie binging to actually making movies. Making games involves a lot of hardwork at different stages of the development process and there is a reason why good game developers take their time (often more than 10 years) before putting out a game on the market.

Nevertheless, I decided to give it a shot - the worst that could happen is I could end up hating it, but I hate it during quarantine anyway.

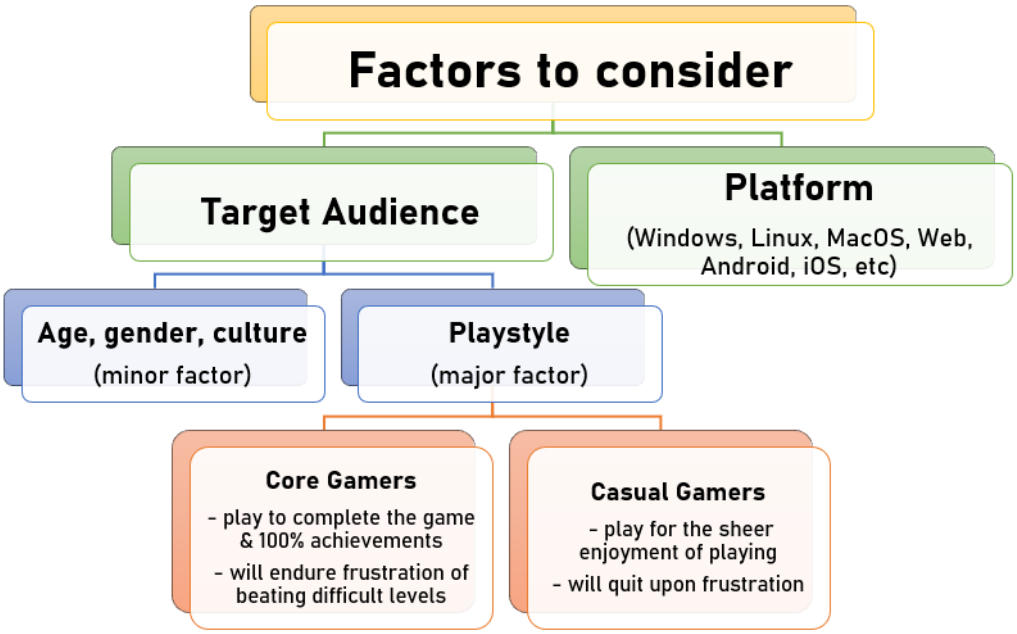
NOTE:

1. This entire course works with Unity3D as the game engine, however the exact same concepts apply to other engines like Unreal, Godot, etc as well.
2. Even if you are using a newer or older version of Unity, the concepts apply just as well. In fact, there is almost negligible difference in functionality between releases.
3. The source files for course projects are available on my website for free use and modification. I will mainly be working with the 2019.4.2f1 and 2017.4.40f1 LTS releases of Unity, so you might need to install these versions of the engine before you try them out.
4. Most of the project builds will be for the WebGL platform. Many browsers do not support running local WebGL content out of the box. You might find this guide handy: `techwiser.com/enable-webgl`
5. If you want to install Standard Assets through Unity Hub, then you might want to install a 2017 LTS release (Unity no longer releases Standard Assets from 2018). Otherwise there is always the option of importing and using 2017 Standard Assets in a 2018 or 2019 release.

---

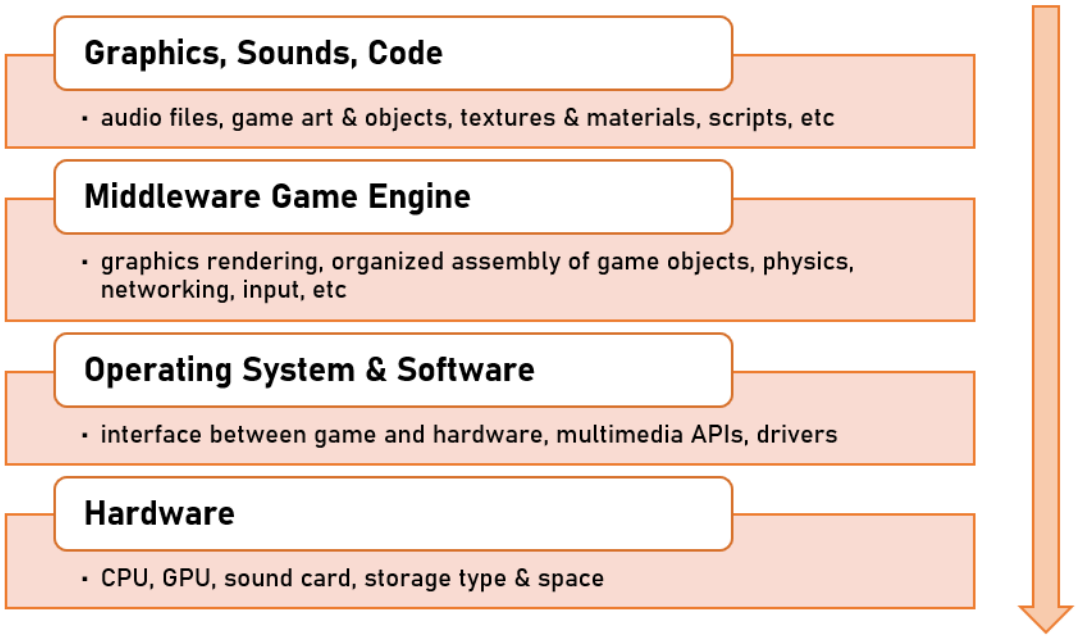
2 Overview of Game Development

2.1 Factors to consider before starting out



2.2 Hierarchy of game structure

Keeping in mind the above factors, we proceed to conceptualize the game structure.



While normally we would construct anything from the ground up, from a game design perspective things work better when conceptualized from the top down.

- We first create everything essential, i.e. 'game assets' - that would mean any art, textures & materials, game object models, audio/music and scripts (basically code that defines interaction between objects)
- Then, we actually construct the levels in the game using the above created assets in an engine like Unity, Unreal, Godot, etc
- We then build the game for the intended platforms
- Finally the game is tested for bugs and to determine the requirements for a system that could hope to run our game

2.3 Team structure and roles

Considering all the stuff from the previous subsection, it would be near impossible for a single person to carry out all the work required and make an end product that ticks all the boxes (individual developers do exist still). This is why there are often teams, ranging in size from as small as 2-3 people to well over 1000 employees, with each member having one or more roles to perform.

- Game Designer: execution of the idea behind the story, designing levels, gameplay mechanics, organizing assets and documents
- Story Designer: character design, story arcs, writing dialogues, etc
- Game Producer: monitoring project budgets, keeping track of deadlines, keeping the team together
- Programmer: writing actual code to determine how game objects interact
- Game Artist: create concept art, textures for game objects to give them unique looks
- Sound Designer: create and edit music to match the mood of various in-game scenes
- Game Tester: not the same as a player; requires sitting on a single game level for hours to identify bugs/inconsistencies

Depending on the size of the team, each person may assume one or more of the above roles based on what they can best do:

- a small team (1-5 people) may be able to make do with just general purpose designers, artists and programmers, and each person will be more or less actively involved in production work
- a medium sized team (6-25 people) may additionally require a manager/producer and specialized designers & artists
- a large team (25+ people) will have supplementary roles like scripters (people who have knowledge of programming as well as design), technical artists (programmers who can also work as artists) and level designers (who make the artwork and game design for separate levels)

---

3 The Unity3D Engine

All games need a few basic features - loading and displaying game assets, playing sound FX. receive player input and react to it, some code to define game rules and mechanics. One way to incorporate all of this is to build everything from scratch i.e. the core game software, renderers, etc. While this can certainly be done, it is much easier to use a game engine to do most of the work for us.

A game engine is a platform that provides basic game functionality, support for integration of game objects of multiple different formats. This allows us to focus on the actual unique gameplay elements.

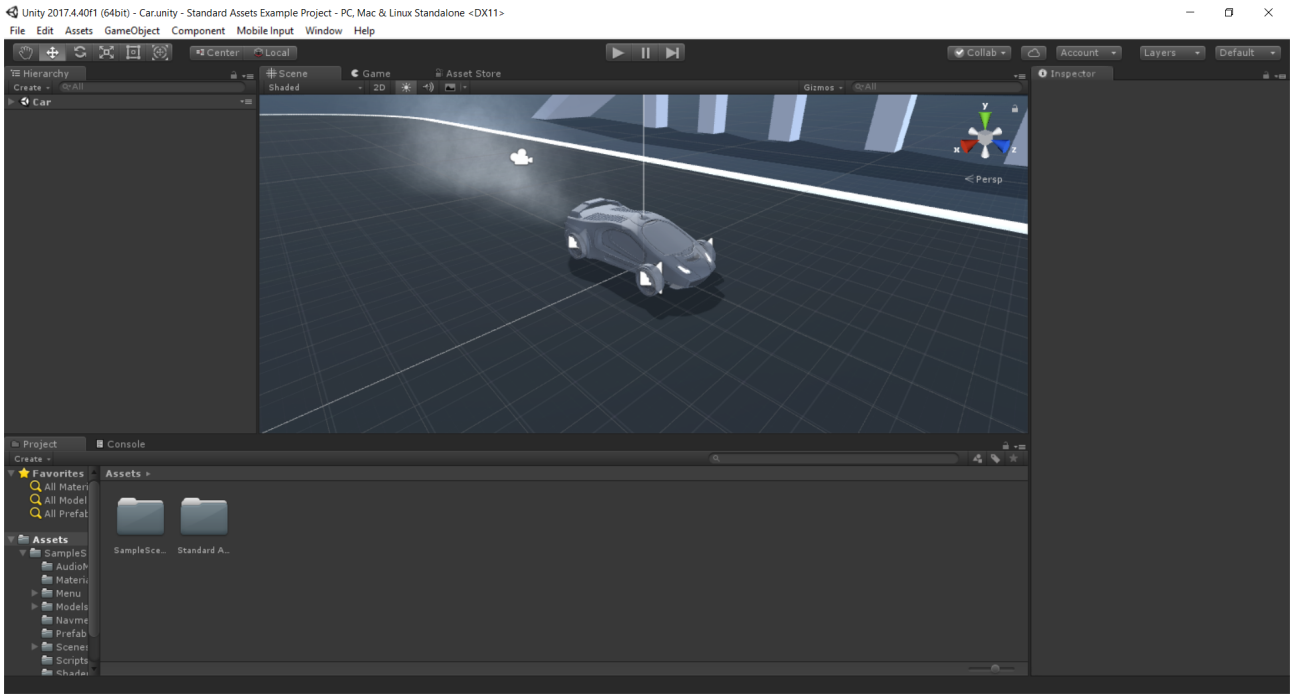
Some advantages of using the Unity3D game engine are:

- great documentation (by which I mean insanely great) and a very large dev community
- very fast build times and support for many platforms
- great asset pipeline (mechanism to import and use game assets)
- not as processor intensive as other engines like Unreal
- can actively switch between different editor versions using Unity Hub without needing to rebuild the entire project

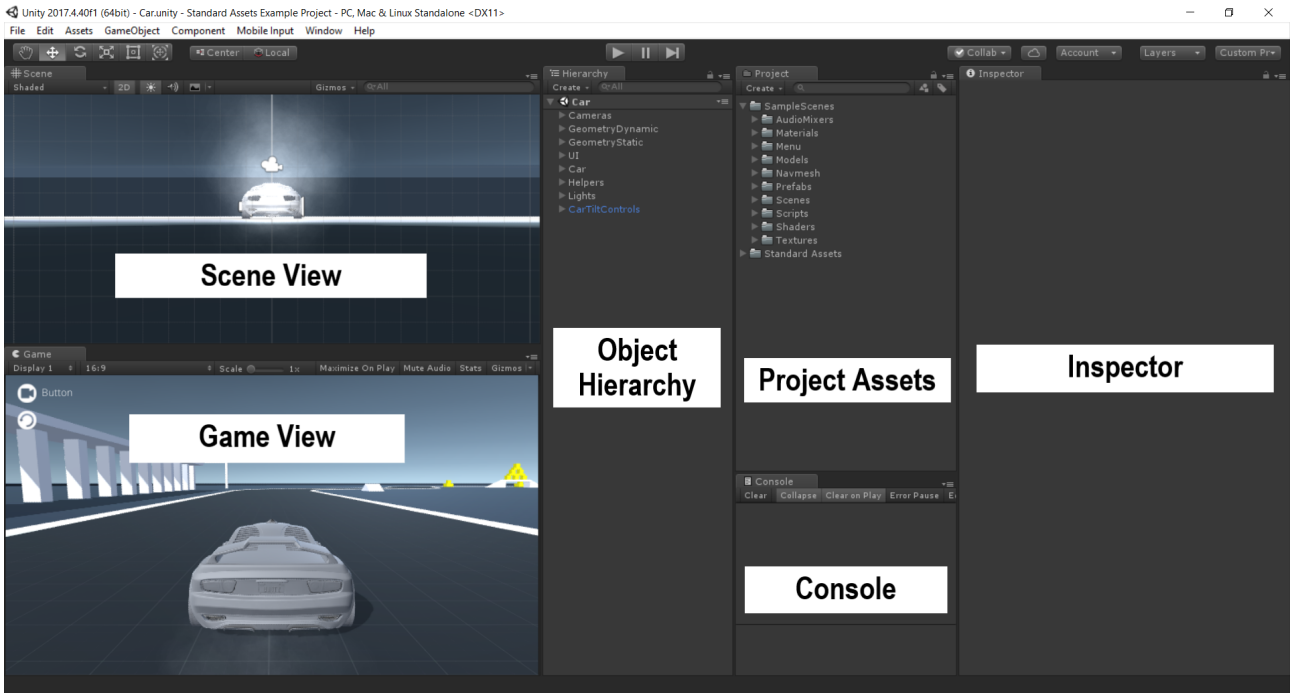
NOTE: If you take up this course, the instructor will mention that Unity3D offers flexibility in programming languages between C# and JavaScript. This is no longer valid after the 2019 LTS releases. However, many .NET compatible languages like C++ can be used if they are first compiled into a DLL (for which you can refer this - `https://docs.unity3d.com/Manual/UsingDLL.html`).

3.1 Unity3D interface and configuration

The process of downloading and installing Unity is very intuitive and I won't go over it. However, note that Unity has stopped distributing the Standard Assets Example Project from 2018 onwards, so you might need to use the 2017 releases if you want to try stuff out with the example project. On launching Unity, the editor window might look something similar to this:



I personally hate this layout (for the main reason that you can only see 4 panels at any given time which means you have to keep on switching between tabs). You can play around with this layout by simply dragging various tabs around the window to get the window configuration you like. I personally prefer this editor configuration, since now you can see all tabs within a single window:



Let us now take a brief look at the various elements in the Unity3D interface:

- Scene View: visual level editor for the currently open scene (edit mode)
- Game View: shows the view through the active camera object (the play button can be used to switch to play mode for playing and testing the game within the Unity editor)
- Object Hierarchy: hierarchical list of all game objects being used in the current scene along with parent-child relationships (is linked directly to the scene view)
- Project Assets: all the files and folders that make up the project (including the ones which are not being used in the current scene, but can be used later)
- Console: displays errors while building the scene (eg: lighting needs to be rebaked)
- Inspector: shows details of all game objects and components (like position, scale, scripts, materials, audio, etc) attached to them

You might now want to explore the Unity editor to look at various features. There are also many quick shortcuts in Unity that you can refer to here: `https://docs.unity3d.com/Manual/UnityHotkeys.html`.

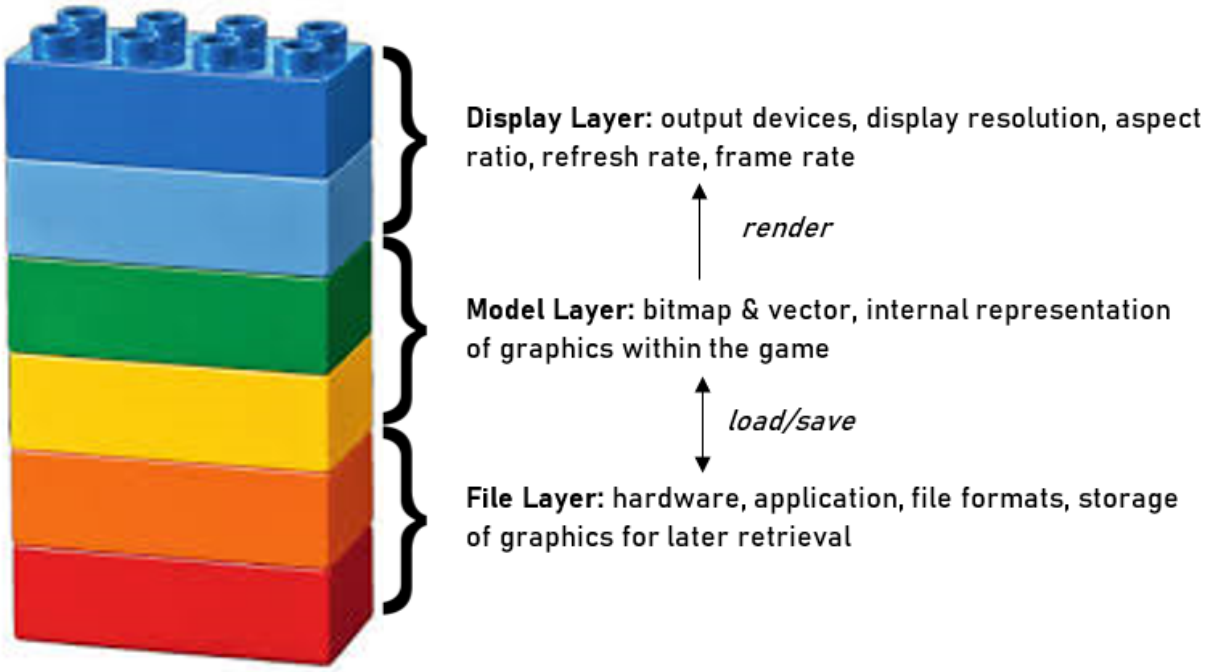
NOTE: If you ever need to move/rename/delete asset files, do it in the Unity editor itself (in the Project panel) and *NOT* in the system file explorer - doing this may affect some connections between game objects and break your project.

---

4 Basic Concepts

4.1 Game graphics concepts

Before we start using Unity, let us briefly discuss the basics of computer graphics using the graphics display model:



1. Display layer:
    - display devices: laptop, smartphone, tablet, VR glasses
    - display resolution: measure of how accurately a display approximates continuous images using finite pixels (pixel dimensions like 1920×1080, dots/pixels per inch)
    - aspect ratio: proportion of pixel map width vs height (eg: 1920×1080 and 1280×720 both have aspect ratio of 16:9)
    - refresh rate: maximum rate at which the hardware can refresh the display image (expressed in hertz)
    - frame rate: actual rate at which the hardware updates the display images (eg: a complex game might run at 60 FPS even though the display has a refresh rate of 240 Hz)
2. Model layer:
    - bitmap image: stores color information about each individual pixel (zooming degrades image quality since we are not creating new pixels, but rather zooming in on the available pixels)
    - vector image: stored internally as mathematical equations representing certain geometric aspects of the image (can be scaled up without loss of quality)
    - vector images take longer to load (additional step of converting vector information to the corresponding pixel map before rendering it to the display), hence usually give lower frame rates
    - 3D graphics: essentially vector graphics in 3 dimensions (i.e. 3D model defined by geometric polygons, usually triangles, to create a 'mesh' of the model) - for realtime rendering at high FPS, we use low polygon modeling (i.e. limit number of polygons)
3. File layer:
    - Unity supports nearly all file formats (support for 2D vector graphics also added recently)
    - for 3D formats (eg: native Maya, 3ds Max, Blender files), Unity converts to FBX files while importing

NOTE: As a general rule of thumb, try to maintain highest achievable image quality for as long as possible - reduce it only if it is adversely affecting the gameplay.

4.2 Game audio concepts

We can apply the analogy of the graphics display model here as well:

- instead of the display we have a device which converts digital data signals back into analogue waves (7.1 surround system, headphones, etc)
- instead of image storage algorithms we have a process called sampling (approximation of continuous analogue input to discrete digital pulses)
- sample rate: number of times per second the sound wave is 'sampled' (typically 44.1 kHz)
- sample size: amount of data stored per sample (typically 8, 16 or 32 bits)
- typical file formats include WAV, AIFF, OGG, MP3, etc (software like Adobe Audition or Audacity have their native file formats)

Audio is very important to games, and also one of the tougher things to actually make or synthesize digitally. There are 3 main types of game audio:

- voice: easiest to come by - voice actors need not be professionals, they can be friends, family, etc (pro tip: keep background noise as low as possible, otherwise noise removal is a very difficult task in production)
- sound FX:
  - reactive FX: effects that occur as things happen in the game (royalty free, creative commons licensed effects can be easily found in places like the Unity Asset Store)
  - ambient FX: ambient noises that add a feeling of immersion (natural sounds often work better than digitized ambient effects)
- music: adds emotional impact and complements the atmosphere (again free resources exist; commercially produced music will need you to buy a license in order that you do not violate copyright law)

END OF WEEK 1

This is the end of the documentation from Week 1 of the course. Continue reading forward, or head over into Unity and try out some stuff yourself. In subsequent weeks, we will be actually making stuff in Unity:

- Week 2 - Solar System Simulation
- Week 3 - Roller Madness
- Week 4 - Box Shooter

---

5 Solar System Simulation

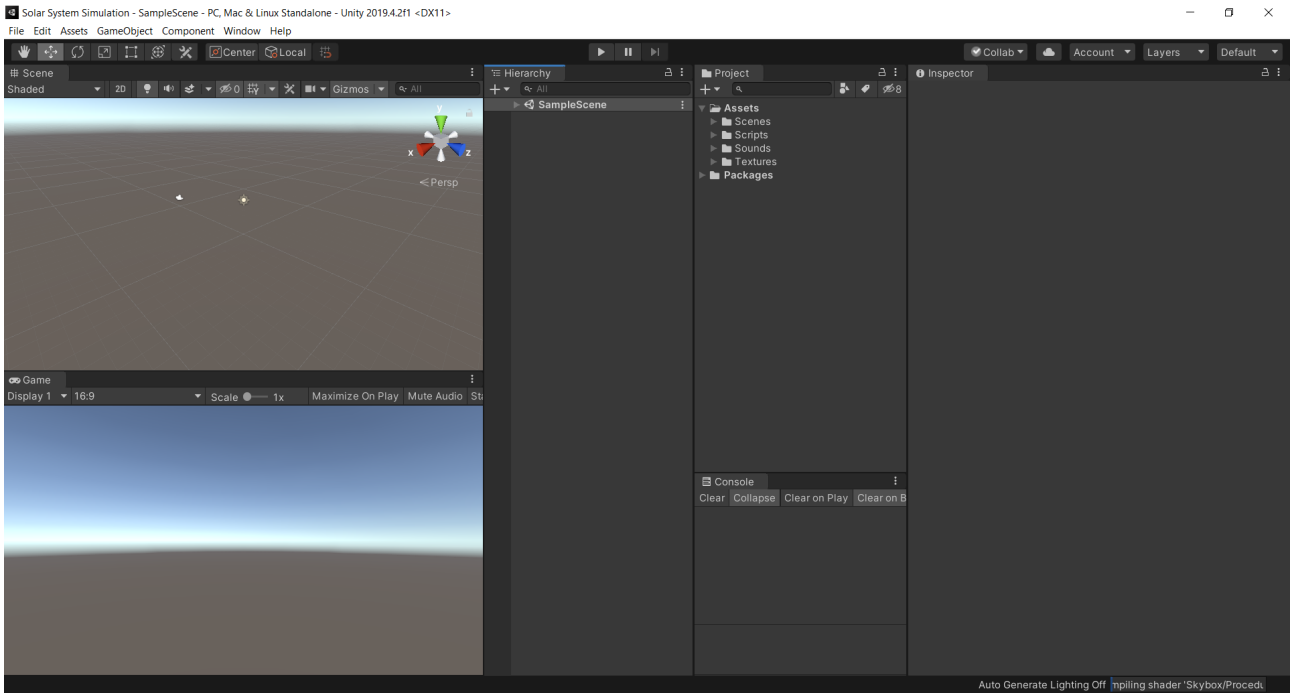
While a solar system model is not technically a game, it is useful to start small and understand the basic workflow that goes into making a game. Sadly due to copyright issues, I am not sure if I can directly provide you the link to the project assets. In case you wish to follow along as I discuss the concepts, you might need to dig a little bit into my website github repository: `https://github.com/omprabhu31/omprabhu31.github.io`.

5.1 Importing assets into Unity

To get started, launch Unity and create a new project (set the template as 3D and any location you prefer; no need to import standard assets) and extract the assets to a suitable location. To import the assets there are 2 methods:

- Drag and drop the asset folders from the file explorer into the 'Project' panel
- Click on the 'Assets' tab > Import New Asset... > Browse each asset and add it

The first method is easier, especially in a big project where we have hundreds of individual asset files. Note that changes to assets within Unity will not affect the assets in the location you extracted them to. By now, you should have a screen that looks like this:



We will be dealing with various game objects in a 3D cartesian system (pro-tip: to remember which axis is which colour, **RGB=XYZ**).

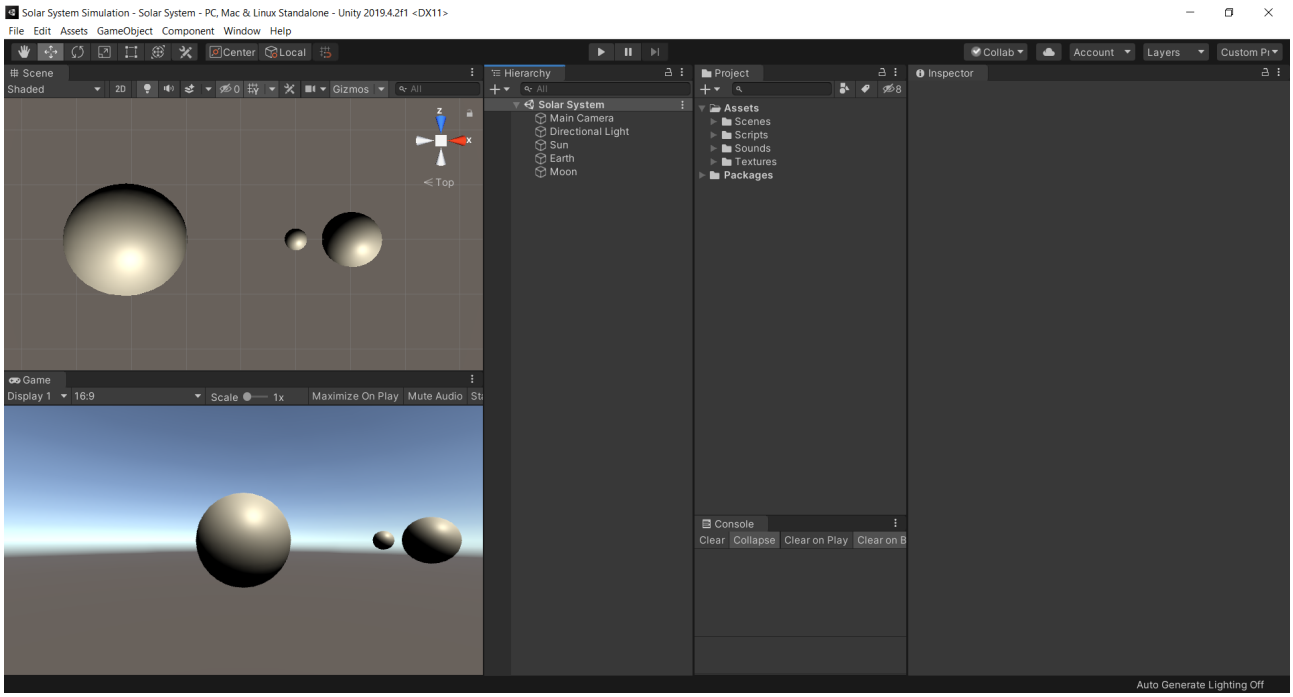
5.2 Game objects and the transform component

Now that we have a basic scene to work with, let's start out by adding the Sun, Earth and Moon as the first game objects. To do so, you might want to follow these steps (or maybe try to explore yourself how to do it):

1. Click on the 'GameObject' tab > 3D Object > Sphere
2. In the 'Inspector' panel, change the name of the object to 'Sun'
3. In the 'Inspector' panel, adjust the position and scale parameters of the transform component
4. Add the other 2 spheres, change their name, position and scale to suit them

(If your game object initially appears in a weird position on your scene view, you can reset its transform by going to the 'Inspector' panel > right-click 'Transform' > Reset)

If you followed along these lines, you should have a screen that looks somewhat like this (could look slightly different based on the transform values you set):



As you can see, our objects have successfully been created in the 'Hierarchy' panel and our scene and game views have been updated (Note: I have changed the position values in the transform component of the 'Main Camera' object to focus the game view, however this is up to you).

5.3 Adding behaviour to game objects

Up till now, we have added the Sun, Earth and Moon as game objects. However, go into play mode and you will observe that they do not move. It is time to get them to rotate around each other appropriately.

We do this by 'adding behaviour' to our game objects. We do this through the use of scripts, which are nothing but files that contain code. This code tells Unity exactly how to get the game objects to behave with each other. Traditionally we would do this by writing the code ourselves, but as of now we have ready-made programs in the 'Scripts' folder. From the names of the scripts, you might be able to (correctly) guess that we need to use the `RotateAround.cs` script here.

5.3.1 Adding scripts

Let us add the script to the Sun, Earth and Moon. Again there are multiple methods to add the script:

- Click on the object in the hierarchy panel > Click on 'Add Component' in the inspector panel > Click on 'Scripts' > `RotateAround.cs`
- Click `RotateAround.cs` in the projects panel > drag and drop it on the desired game object

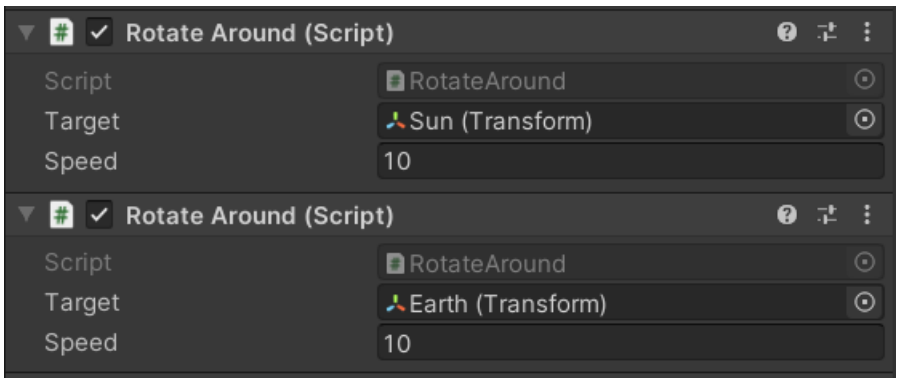
You will notice that a new 'Rotate Around' component has been added into the inspector panel for the particular game object. It also contains two parameters - target (the object around which the selected game object will rotate) and speed (angular speed of rotation in degrees per second).

Note that the Earth rotates around itself as well as the Sun. This means we need to add the same script twice (similarly for the Moon as well).

5.3.2 Adjusting script parameters

Merely adding the script usually won't do anything. It is simply a component, and updating its parameter(s) is what will actually define the behaviour. I will elaborate the process for the 'Earth' object, since there are 2 scripts we have to deal with:

1. Target: drag and drop the 'Sun' object onto target parameter of the first script in the inspector panel
2. Speed: type in a suitable speed and test it in your game view and make further edits if any
3. Similarly set up the second script by setting the target to the 'Earth' object this time



Complete these steps for the Sun and Moon as well (the Sun will need only one script, since it only rotates around itself), and please do not go into play mode yet. Note that there will not be any apparent effect of the objects rotating around themselves, but this is because we haven't added textures to them yet.

5.3.3 Parent-child relationships

Did you complete the above steps for the moon? Now go into play mode - I'm guessing you're having a pretty weird experience now. You're probably seeing the Moon chase the Earth and the distance between them increase. The reason for this delves a little bit into Mathematics and Physics, but I will discuss it in brief:

- the `RotateAround.cs` script instructs the Moon to rotate around the 'instantaneous center' of the Earth at any given point in time
- since the Earth is itself rotating around the Sun, the 'instantaneous center' of the Earth is changing with time in the frame of reference of the scene view and so is the distance between the centers of the Earth and the Moon
- however, if we look from the frame of reference of the Earth (i.e. try to imagine that you're inside the Earth and at its center), what do you see? You will see apparently the Sun rotate around you, and the center of the Earth will appear stationary for you

If you still feel lost or get it partially, don't worry. You can look up the concept of 'instantaneous center of rotation' on the internet and refer to some pretty good articles for more knowledge.

Let's get back on track now. To fix this, we need to attach the Moon to the frame of reference to the Earth. In Unity language, we have to make the Moon a 'child' of the Earth - this will make any motion of the Moon relative to the frame of the reference of the Earth. To do this, we simply click on the Moon in the hierarchy view and drag & drop it onto the Earth.

Note that on setting the Moon as a child of the Earth, the transform components also change since the coordinates are now relative to the center of the Earth.

5.4 Adding materials, lighting and audio

We have set up the basic rotation scripts, but it is still just a couple of spheres rotating around each other. We can enhance the look of our solar system by:

- making the Sun, Earth and Moon actually look like their solar system counterparts through the use of textures and materials
- removing the directional light & adding a suitable background and lighting to the Sun
- adding basic audio to the simulation

5.4.1 Adding materials

A game object is made up of a 3D model. It can be thought of as a wire frame, and its material as the skin which wraps around this framework. Let us get into some theory about materials before we actually apply them to our solar system:

- Material: all visible game objects have a material - however, it is only a set of parameters that specifies how the object should respond to light (it does not actually define the look of the object)

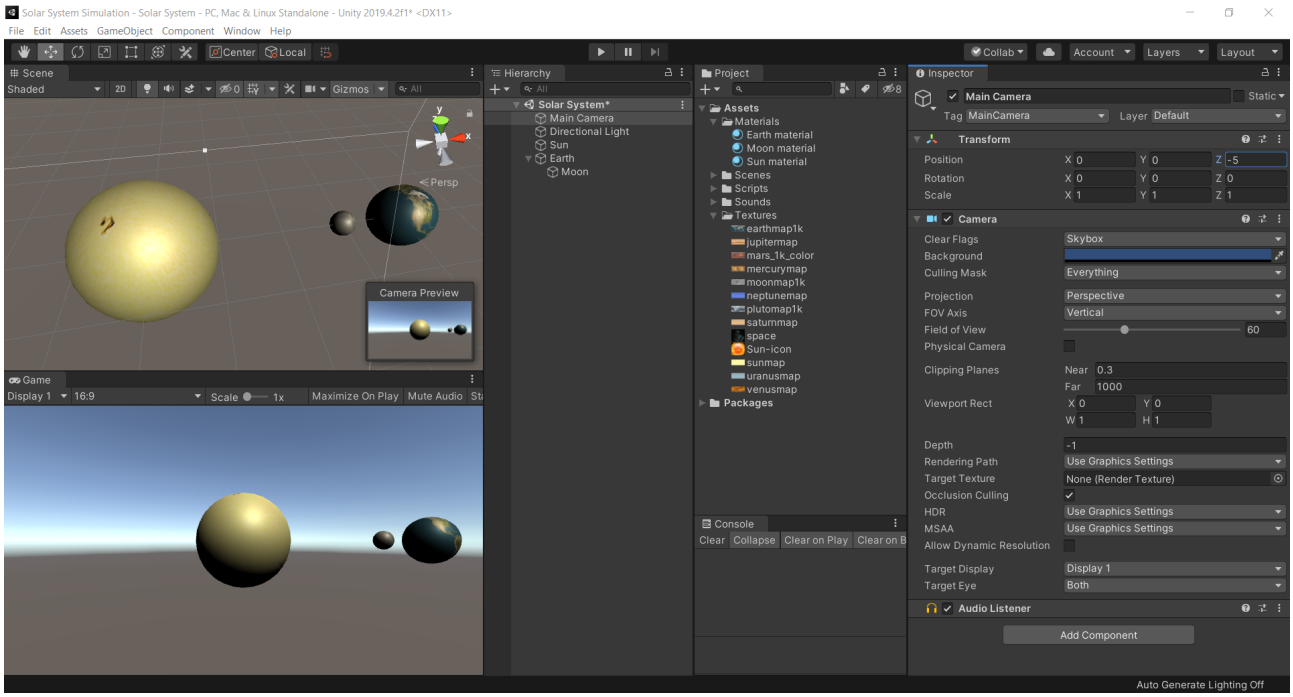
- Shader: this is the formula that takes the above parameters (position of individual triangles, relative position of light, material color values for each triangle) and defines the look of the object
- the shader has properties for texture maps (like emission, albedo, occlusion, etc), which in turn is the property of a material

If you're still confused, head over to this link to learn more about materials and shaders:
`https://gamedev.stackexchange.com/questions/45842/difference-between-material-and-shader`

With the theory covered up, it is now time to actually add materials to our game objects:

1. Go into the 'Project' panel > Assets > Textures
2. Drag and drop the 'sunmap' texture on the Sun in the hierarchy view
3. Notice how Unity automatically created a new 'Materials' folder under the assets category - for organizational purposes rename the material to 'Sun material'
4. Repeat the process for the Earth and Moon

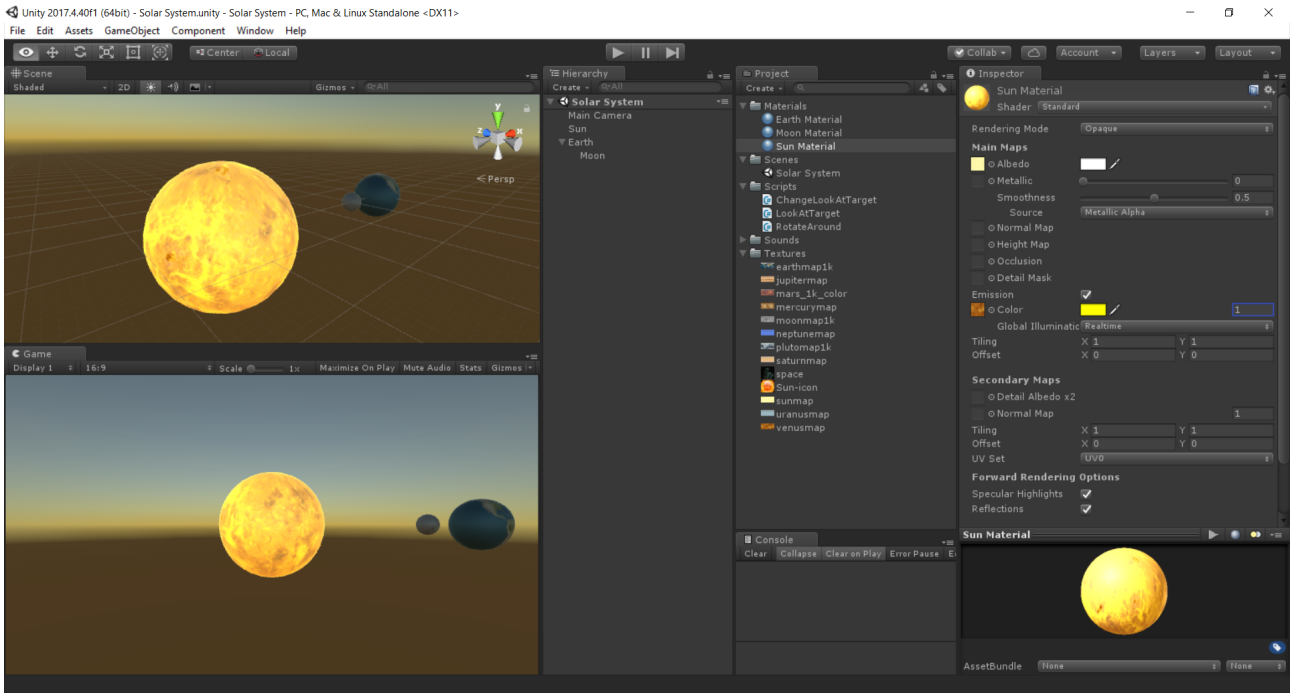
Note that over here, we have only changed one property of the shader for each material. The texture map is what makes these objects look like entities in the solar system. By the end of this section, your window should look similar to this:



5.4.2 Adding lighting

We have 2 major lighting issues in this project - the Sun is not emitting any light of its own and the background doesn't look like space. Let us start by fixing the lighting for the Sun:

1. Remove the 'Directional Light' component from the hierarchy view (right-click > Delete)
2. Browse to 'Sun material' in the projects panel and check the 'Emission' property box
3. It is possible to set a texture map for the emission color (which will essentially form a layer over the existing texture) - Go to the 'Color' parameter under 'Emission' > Click the little dot next to 'Color' > Select a suitable texture map
4. Set a suitable emission color (anything between yellow and orange should work for the Sun)
5. (Optional, since 2019+ releases do not have this feature) Set the emission color brightness to an value to your liking

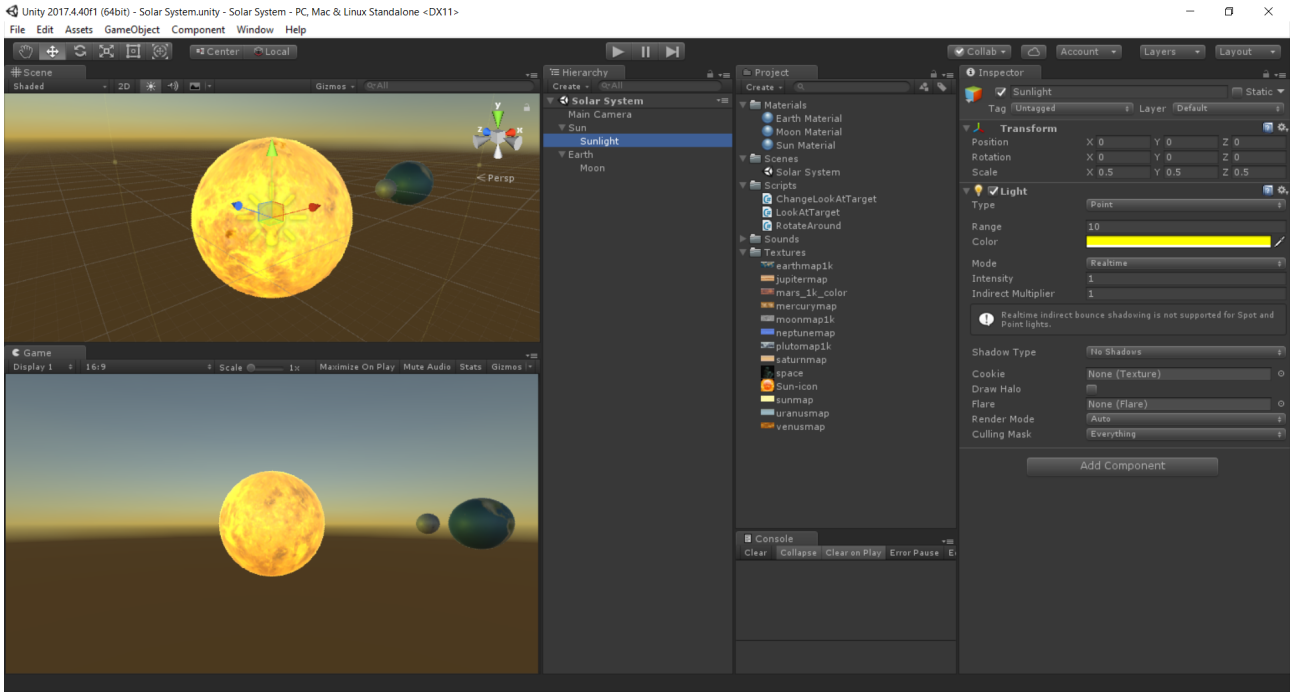


Now the sun has a little emission characteristic, but it still does not emit its own light onto the Earth and Moon. To do this, let us first understand the basic types of light objects in Unity:

- ambient light: used for basic illumination (will be discussed later)
- directional light: used for illumination and establish the time of day if required
- point light: sends out light in all directions
- spot light: acts like the lights on a stage; illuminates a particular region

You will probably have guessed by now that we need to add a point light object as a child of the Sun. To do this, go to the 'GameObject' tab > Light > Point Light. Reset its transform component if needed and set it as a child of the Sun. Play around with the parameters a little to get an appropriate lighting effect. You might want to rename this to something like 'Sunlight'.

If you have followed until now, your window should look similar to this (notice the difference in lighting between the 2 screenshots on this page):

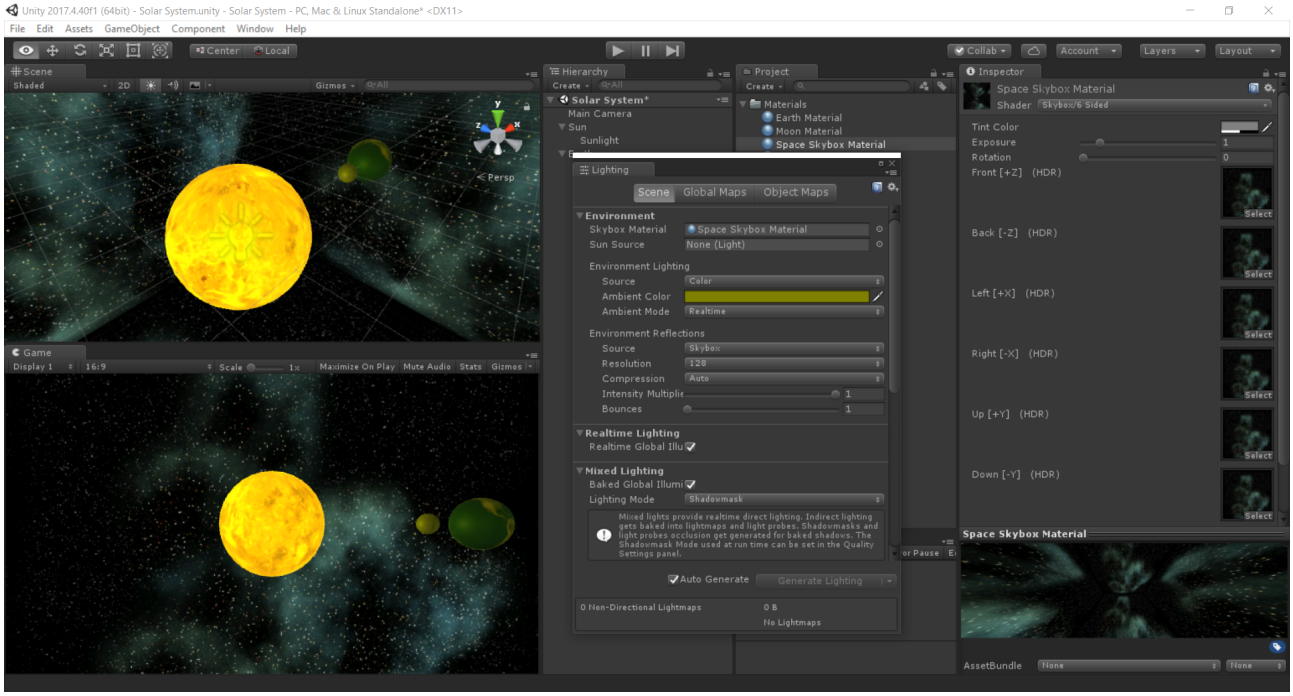


Now that we have set up the lighting of the Sun, the only thing left to do is change the background and add ambient light. In Unity, the background is referred to as the Skybox. To setup the Skybox material:

1. Go to the 'Project' panel > Click on 'Create' > Material
2. Rename it to something appropriate
3. In the 'Inspector' panel > Click the shader dropdown > Skybox > 6 Sided
4. Set all 6 texture maps to the 'space' texture (you could have multiple space textures and set them accordingly, but here we have only one to work with)

Now it is time to add this material to the Skybox. To do this, go to the 'Window' tab > Lighting > Settings (In some versions of Unity, you will have to go to 'Window' > Rendering > Lighting Settings). Drag and drop the skybox material onto the corresponding property in the pop-up window.

We don't really need to add any ambient light, but we can do it by playing around with the parameters within the 'Enviromment Lighting' section of the same window to add a little more flair to the scene. Once we are done with this, our scene should start looking a lot more like the solar system.



NOTE: While adding a lot of lights into a scene might make it look somewhat better, it is important to understand that more lights make the game more processor intensive and can adversely affect frame rates. It is hence advisable to work with as little light objects as possible.

5.4.3 Adding audio

While a solar system simulation doesn't necessarily need any audio, many other projects do. We will add some basic audio to illustrate some basic audio concepts in Unity. We might want to add a burning sound effect to the Sun and some sort of humming sound to the Earth which fades in and out as the distance of the Earth from the camera changes.

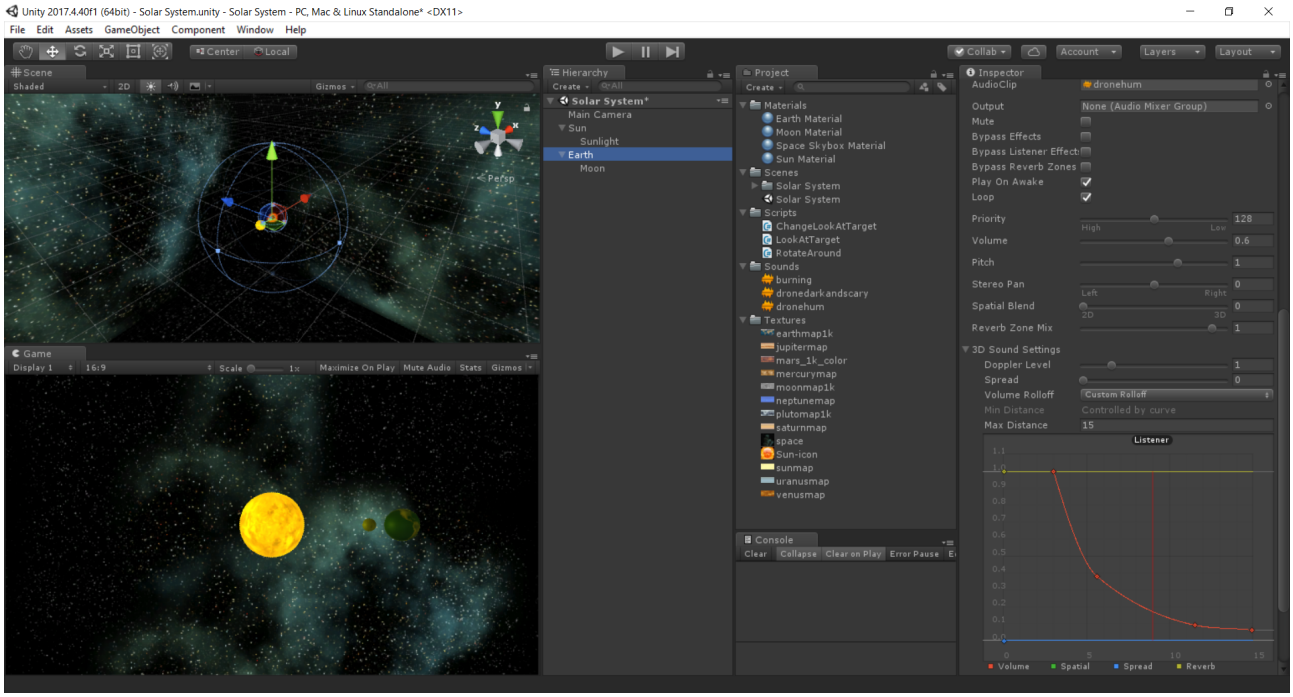
There are 2 audio components that we will work with. The audio source component has to be attached to the game object which is meant to emit sound, while the audio listener is often attached to a camera object. As the name suggests, the audio listener picks up the sound emitted from the audio sources (intensity depends on the distance between the source and listener). The listener is a simple component - it has no properties and can only be turned on or off.

To add the audio source component to the game objects, follow these steps (I will illustrate it for the Earth object, since we will also need to edit 3D sound settings for it):

1. In the 'Inspector' panel > Add Component > Audio > Audio Source
2. From the project panel, drag and drop the 'dronehum.aif' audio file onto the AudioClip parameter in the inspector panel (if you go into play mode, you will hardly notice the fading effect of the sound)

3. To make the fade effect more prominent, go to the Audio Source component > 3D Sound Settings > set the Min and Max distances to appropriate values and adjust the volume rolloff to your liking

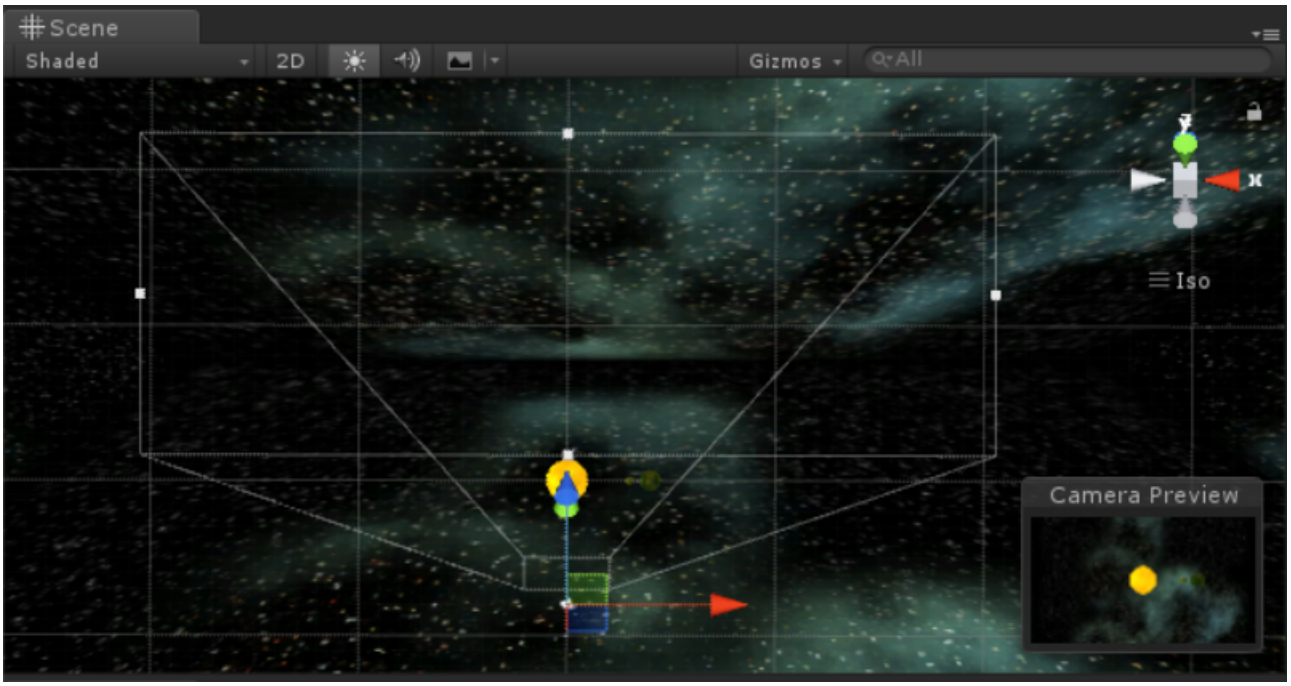
Repeat the same process for the Sun (since it is at a fixed position, no need to change any 3D sound settings. If you followed along, your audio source component settings should look similar to this:



5.5 Adding cameras and adjusting views

A camera object in Unity is far from its real life counterpart. What a camera in Unity essentially does is project the 3D geometric information onto a 2D screen. We can choose to have multiple cameras in a single scene and show them simultaneously on screen.

There are 2 types of 3D projections - perspective and orthographic. The perspective projection is what we have in our camera now, since the Earth appears bigger as it moves closer and vice-versa - it renders everything within its 'view frustum'. Orthographic projections can probably be found in case of top-down or isometric views - the camera maintains the same scale of the object irrespective of the distance (a good example of this is *SimCity*).



The view frustum is better described by the above image. The white outlines determine the field of view and the near and far clipping planes (set right now at 2 and 20 metres respectively). If, for instance, the Earth were to come within 2m distance or go farther than 20m distance from the camera, the object would not be rendered.

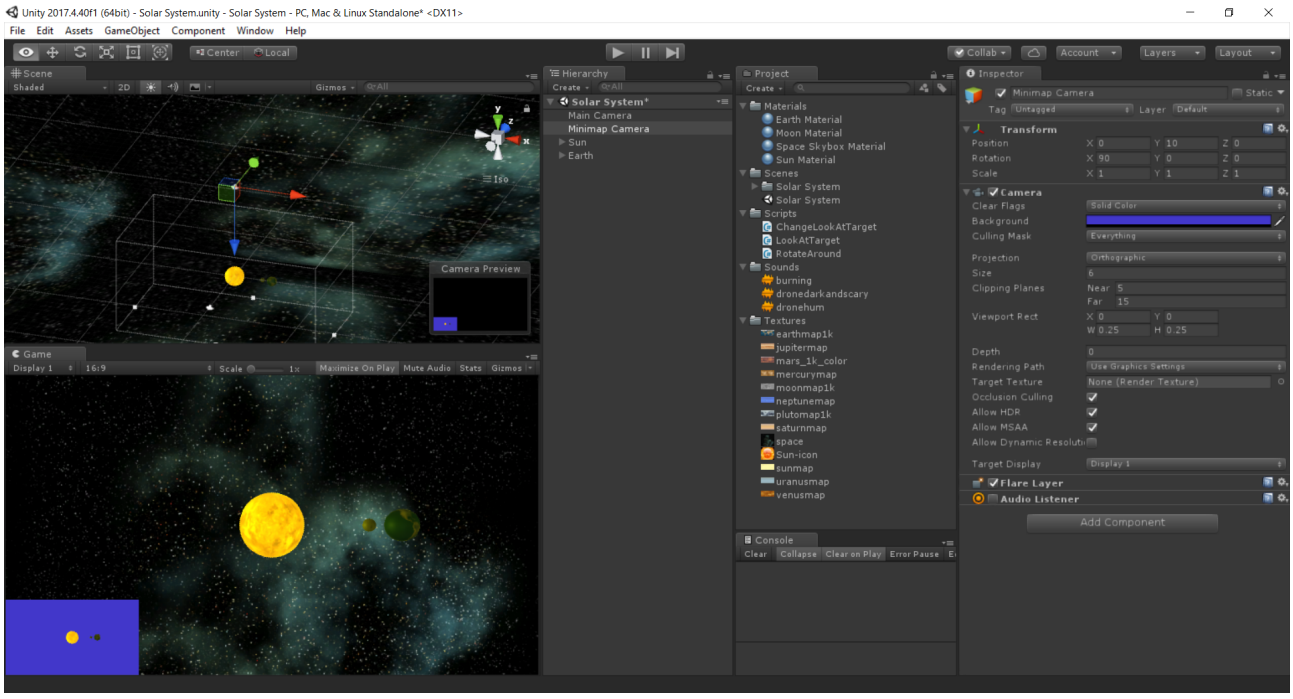
For the purpose of applying these camera concepts, we will create a second camera to show a top-down view of our solar system and position it in a little 'mini-window'. To add and position the camera suitably, follow these steps:

1. Go to the 'Hierarchy' panel > Create > Camera (the game view will suddenly change, since Unity reads this as the new active camera)
2. Reset its transform if needed and rename it to something suitable
3. Position and angle the camera so that it provides a top-down view of our solar system model
4. Go to the 'Camera' component in the inspector > Projection > Orthographic
5. Play around with further properties till you get a view that you like

So we have added our second camera. However the game view shows only the view from this new camera object. Since this is going to be positioned in a 'mini-window', we probably need to update some further settings within the inspector. Follow these steps to set up the minimap camera window:

1. In the 'Inspector' panel > Camera > Viewport Rect (this is basically a set of origin coordinates size of the rectangle on the screen to which the camera outputs)
2. Set suitable values for width and height (width of 0.25 means that the rectangle will have one-fourth the width of the screen)
3. (Optional) To make the background color of the minimap different, in the same component > Clear Flags > Solid Color (set any color you like)

Note how the main camera view appeared again. By now you should have a screen looking like this:



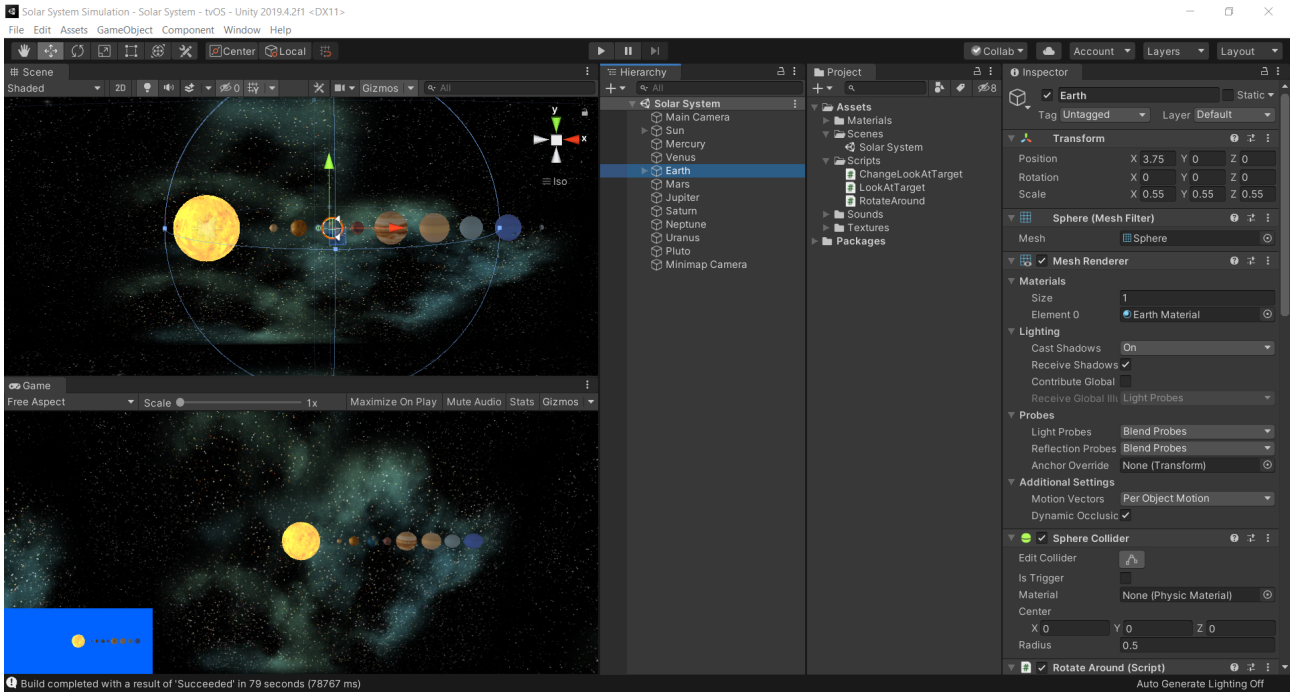
NOTE: We now have 2 camera objects and hence 2 audio listeners. It is highly advised to have only one audio listener component switched on at any given point of time - otherwise Unity will pick one of those two, which may not be the one we want. To avoid this, switch off the audio listener component on the second camera (or remove it altogether since we won't be using it anyway).

5.6 Finishing, building and deploying a project

We have essentially completed our simulation of the solar system. We could do some more finish work like (this is all essentially based on the concepts covered earlier so I will not discuss this in detail):

- add other planetary bodies to complete the solar system
- figure out how the `LookAtTarget.cs` and `ChangeLookAtTarget.cs` scripts work (hint: they are meant to work together so that clicking on the Earth will focus the main camera on the Earth and so on)

Take some time out to test your final project to test for any bugs. It is highly unlikely that this project will have any, but it is still a good idea to do so. By the end you should have something that looks like this:



Once you're done testing, it is time to finally build it so that it can be launched as an application from any specific platform. This can be done in the following manner:

1. File > Build Settings
2. Drag the 'Solar System' scene from the projects panel into the 'Build Settings' window
3. Select the platform for which you want to build the application. If needed, click on 'Switch Platform'
4. Click on 'Player Settings ...' and play around with the settings to your liking (do not mess around too much, the default settings work well for the most part)
5. Click on 'Build', select a destination and wait for the build to be complete
6. Launch the application to see if it works successfully

NOTE: Again for organizational purposes, it is better to save all your builds in the Unity project folder itself in a folder called 'Builds' (eg: for a Windows build, save it in Builds > Windows and so on).

END OF WEEK 2

This is the end of the documentation from Week 2 of the course. In the next project, will be creating an actual game called 'Roller Madness' which will put into action all the previously discussed concepts and more. Continue reading forward, or head over into Unity and try out some stuff yourself.

---

6 Roller Madness

(This section is in the works along with notes from Week 4 as well. I will probably be finishing Week 4 first since I am already going through the Week 4 course material at this time. However, there shouldn't be any major loss if you do Week 4 before Week 3. Expect it to be up in about 2 weeks.)

---

7 Box Shooter

Up till now we have only used ready-made scripts. However it is important to be able to write our own code and modify existing code as well, irrespective of our role in the dev team. With a working knowledge of programming, we can go much further in Unity as well as other tools like Maya, Blender, etc. One must strive to gain at least an intermediate level of understanding in programming which will allow them to grasp the core language, write code and modify code from an example source.

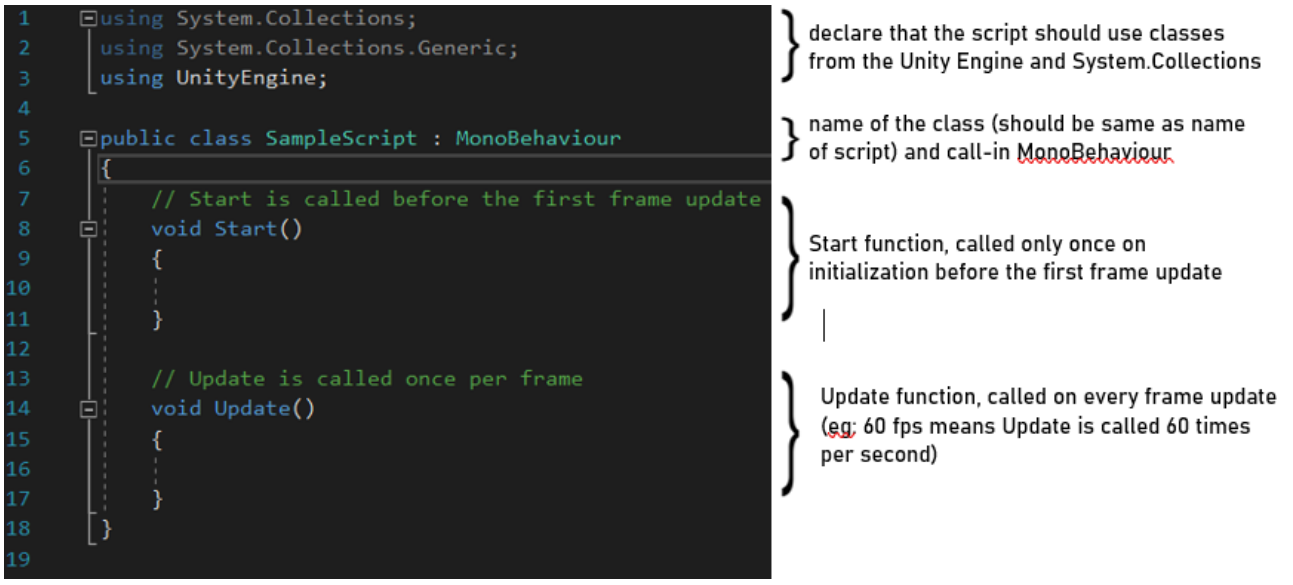
7.1 Unity programming concepts

Unity follows an object-oriented approach to programming. Much like the real world, everything here is considered an object - it has properties (variable type, size, etc) and methods (functions which enable performing actions using code) associated with it. Most of the assets we have worked with up till now were visual objects. These objects have their properties (i.e. components) and methods (parameter values).

One important concept in object oriented programming is classes. Classes are templates for creating objects, much like prefabs which we used in the previous project. For example there could be a class for a 'clock' object that defines how a general clock should function, but individual clocks generated from this class will have their own properties like color, timezone, etc.

Let us now discuss more about programming in Unity:

- the language used to write code for Unity is C#
- a C# script can either be created from scratch by 'Project' panel > Create > C# Script, or be imported from an existing source and later modified
- a script is just another type of asset like sounds, textures, etc
- scripts can be attached as components to game objects in a scene

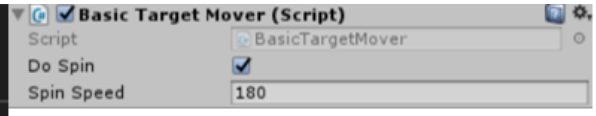


A typical script template generated by Unity is as above:

- part 1: instructs the scripts to use classes from `UnityEngine` and `Systems.Collections` (these are namespaces which define various types of objects like lists, queues, arrays, etc)
- part 2: `public class` indicates that this class will be visible as a component in Unity (public classes & variables are generally seen as components in Unity)
- part 2 (continued): `MonoBehaviour` is the base class from which every script in Unity derives (in essence, `SampleScript` is our custom class which uses stuff from the broader `MonoBehaviour` class) - Mono is an open-source implementation of Microsoft's .NET framework
- part 3: variables & functions that make up the details of the `SampleScript` class

Let us now look at a more complete C# script:

```
using System.Collections;
using UnityEngine;

public class BasicTargetMover : MonoBehaviour {

    public bool doSpin = true;
    public float spinSpeed = 180.0f;

    private Transform mover;

    void Start () {
        //get a reference to this gameObject's transform component
        mover = gameObject.transform;
    }

    void Update () {
        if (doSpin) {
            //rotate around the up axis of the gameObject
            mover.Rotate(Vector3.up * spinSpeed * Time.deltaTime);
        }
    }
}
```



- only public variables are visible as parameters in the script component
- Start is called only on initialization (sets `mover` to the initial transform values of the gameObject)
- Update is called on every frame update (changes the `Rotate` parameter of `mover` based on a pre-defined rule) – `Vector3.up` is another way of writing 3D vector (0,1,0); `Time.deltaTime` is the time in seconds between each frame update
- event driven environment – many other events like `MouseDown`, `OnCollision` against which we can encode custom game responses

The event driven environment means that we will often be adding functions to the class(es) to respond to events that occur in certain stages of the game.

Just like `Start`, we have 2 other events `Awake` and `OnEnable` which allow us to set things up before the first frame update occurs. After this, the game is set into an event loop (each loop usually lasting a frame) - the `Update` and `LateUpdate` functions provide a way to keep a check on things while the game is playing. Finally, there are certain specific events that the program can be instructed to respond to in a certain manner (eg: `OnMouseDown` causes the camera view to shift down).

7.1.1 Referencing gameObjects in scripts

When a script is attached to game object, it can be referred to as either `this.gameObject` or simply `gameObject`. The following is a simple program that causes the transform of the object to change:

```
void Start () {
    //shift the x position of the object by 1 unit
    this.gameObject.transform.position.x += 1;
    //gameObject.transform.position.x += 1 also works
}
```

For referencing any other game object that the script is not attached to, we need to define a public variable of type gameObject that we attach to another game object/prefab in the editor. They can further be searched in the editor either using object names or tags. The following is a program to change the transform of global game objects:

```
public GameObject target1;
public GameObject target2;
public GameObject target3;

void Start () {
    //target1 is set in the editor
    target2.Find("Enemy");
    target3.FindWithTag("Player");

    //change the x position of transform by 1 unit
    target1.transform.position.x += 1;
    target2.transform.position.x += 1;
    target3.transform.position.x += 1;
}
```

7.1.2 Referencing components in scripts

Every property you see in the Unity Editor for a component can be reference and often modified through code. For example, in both of the above programs we referenced the position (`gameObject.transform.position.x`) along the X-axis and also modified it.

Quite often, there are multiple ways of implementing actions into code. Sometimes one is more efficient than the others. For example, `gameObject.transform.position.x += 1` and `gameObject.transform.Translate (1,0,0)` do exactly the same thing.

Not all components are properties of a game object like transform. To reference and/or manipulate these, we use the `gameObject.GetComponent<TYPE>()` function. The following program illustrates the above concept to the 'RigidBody' component:

```
private Rigidbody rb;

void Start () {
    //gets a reference to the RigidBody component
    rb = gameObject.GetComponent<Rigidbody>();
    //turns gravity on
    rb.useGravity = true;
}
```

NOTE: Consider the following points while programming in Unity:

1. The name of the class *must* be the same as your script name. If you do change the name of your script, you will have to change the name of the class too - Unity does *NOT* automatically do this
2. If you find yourself stuck, Unity has an excellent documentation (`https://docs.unity3d.com/`) - this can also be accessed through many IDEs like MonoDevelop and VS Code (Help > Unity API Reference)
3. Use the community forum (`https://unity.com/community`) to your advantage - if you're stumbling upon an error, it is more than likely that someone else has too
4. Finally, here is a quick reference sheet on writing basic C# code: `Reference Sheet`

7.2 Setting things up

In this project, we will dive a lot deeper into Unity by building out a custom game using our own custom scripts. Most of the time in this project, we will be modifying existing code rather than making scripts from scratch. Some of the concepts we will cover are:

- creating custom C# scripts using an IDE and basic debugging techniques
- use of classes, variables and functions to respond to events
- instantiate prefabs dynamically, obtain user input and update user interface through code

Similar to earlier projects, if you are trying this out yourself, you will have to find the assets in the GitHub repository. Once you have them downloaded, import them into the project. Since this is a single unity package file, import it by going to 'Assets' > Import Package > Custom Package... > Browse to the local file location.

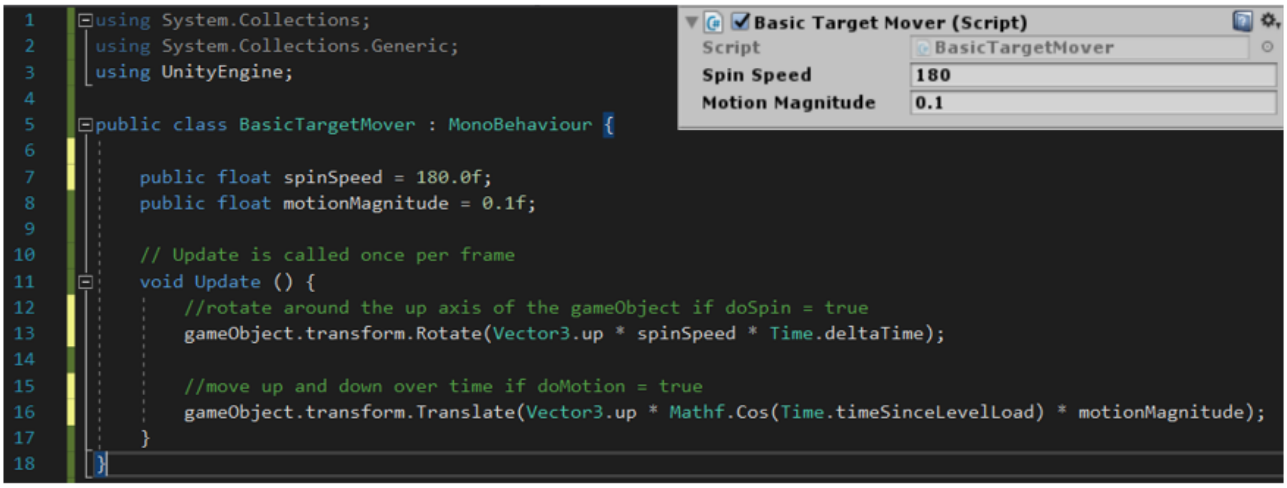
7.3 Using scripts to move objects

In this subsection, we will apply the concepts of public variables, floating point numbers, boolean values and referencing components to create a basic script for moving a target object. Firstly, create a 3D object (I have created a cube) and name is as 'Target' and apply the Target-Positive Material to it.

7.3.1 Basic object movement

A script to induce basic movement in a game object might look like this:

- `Vector3.up`: a representation of the 3D vector (0,1,0) - references the Y component of the rotate and position properties of the transform respectively
- `Time.deltaTime`: normalizes the motion for different frame rates
  - without this, the spinning would be slower on 15 FPS compared to 60 FPS
  - multiplies by the time in seconds between frames (i.e. for 60 FPS, 1/60)
  - for example at 60 FPS, Update is called 60 times per second - during every update, object rotates through (0,3,0) so total rotation in one second is (0,180,0)
  - in contrast for 15 FPS, Update is called 15 times per second - during every update, object rotates through (0,12,0) so total rotation per second is again (0,180,0)
  - as a result, consistence of spin motion is maintained but smoothness of the motion may vary
- `Mathf.Cos(Time.timeSinceLevelLoad)`: cumulative time since level load (converted to radians) - facilitates harmonic motion of the target object




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

public class BasicTargetMover : MonoBehaviour {

    public float spinSpeed = 180.0f;
    public float motionMagnitude = 0.1f;

    // Update is called once per frame
    void Update () {
        //rotate around the up axis of the gameObject if doSpin = true
        gameObject.transform.Rotate(Vector3.up * spinSpeed * Time.deltaTime);

        //move up and down over time if doMotion = true
        gameObject.transform.Translate(Vector3.up * Mathf.Cos(Time.timeSinceLevelLoad) * motionMagnitude);
    }
}
```

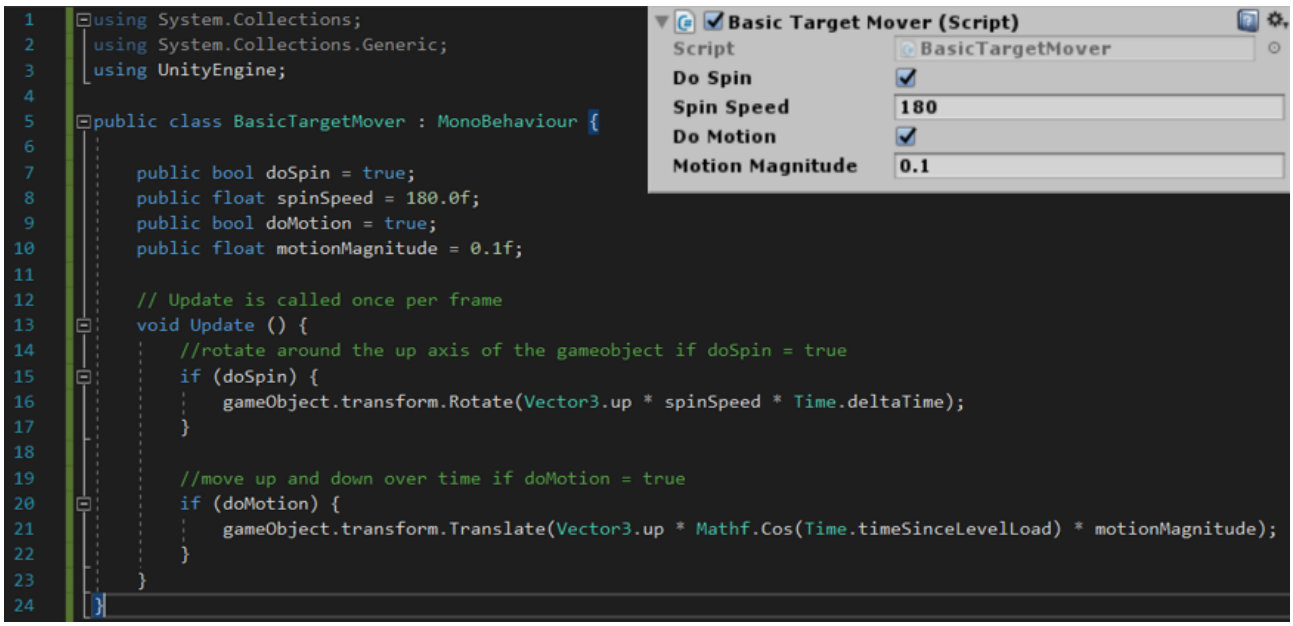
7.3.2 `bool` datatype and `if` statements

With the above code, we can make our target game object spin as well as move up and down. If we only want it to spin, we can set the Motion Magnitude value to 0 within the editor (this will not affect the value in our script), and similarly for only motion. While this is easy enough, it is not user-friendly and game designers might want simple options to switch these properties on and off. When faced with words like on and off, true and false, etc, a programmer's instinct leads them to work with the boolean datatype.

The bool datatype is a relatively simple one - it can store only values 0(false) and 1(true). We can define 2 bool variables named `doSpin` and `doMotion` and write if statements so that:

- spin motion occurs only if `doSpin` is true
- vertical motion occurs only if `doMotion` is true

And of course, we can individually switch these on/off in the editor without affecting values in the script file.

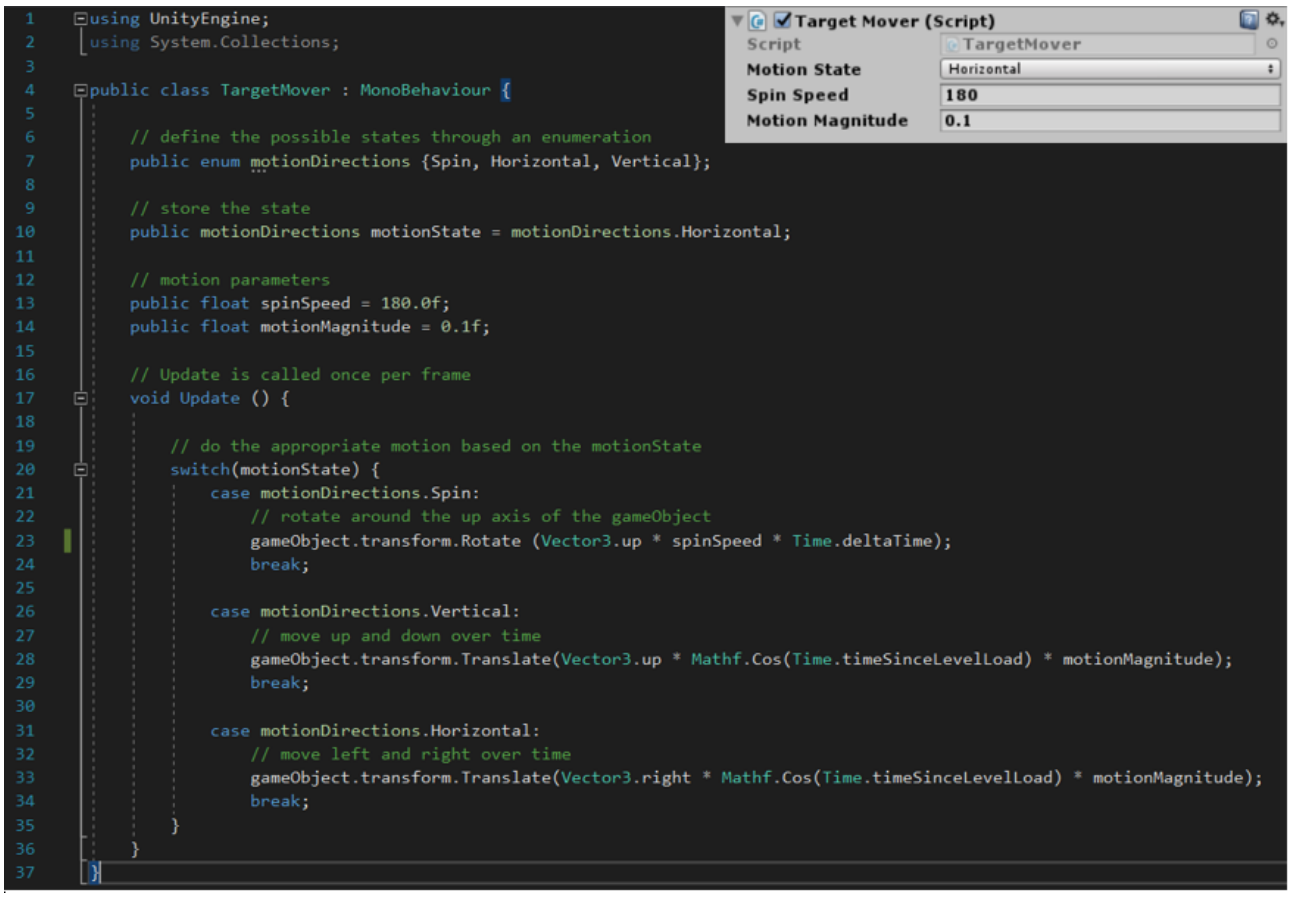



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BasicTargetMover : MonoBehaviour {

    public bool doSpin = true;
    public float spinSpeed = 180.0f;
    public bool doMotion = true;
    public float motionMagnitude = 0.1f;

    // Update is called once per frame
    void Update () {
        //rotate around the up axis of the gameobject if doSpin = true
        if (doSpin) {
            gameObject.transform.Rotate(Vector3.up * spinSpeed * Time.deltaTime);
        }

        //move up and down over time if doMotion = true
        if (doMotion) {
            gameObject.transform.Translate(Vector3.up * Mathf.Cos(Time.timeSinceLevelLoad) * motionMagnitude);
        }
    }
}
```

7.3.3 `enum` datatype and `switch` statements

Up till now we can effectively switch on and off the individual motion components. But what if we have more than two types of motion? We can go on creating bool variables and if statements but that would make the component interface in Unity too complicated and cluttered. A better solution is to use the enumeration datatype.

Let us check out the `TargetMover.cs` script we have in our assets directory. One disadvantage is that we can only choose one type of motion (i.e. spin and vertical motion cannot be switched on simultaneously).




```
using UnityEngine;
using System.Collections;

public class TargetMover : MonoBehaviour {

    // define the possible states through an enumeration
    public enum motionDirections {Spin, Horizontal, Vertical};

    // store the state
    public motionDirections motionState = motionDirections.Horizontal;

    // motion parameters
    public float spinSpeed = 180.0f;
    public float motionMagnitude = 0.1f;

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

        // do the appropriate motion based on the motionState
        switch(motionState) {
            case motionDirections.Spin:
                // rotate around the up axis of the gameObject
                gameObject.transform.Rotate (Vector3.up * spinSpeed * Time.deltaTime);
                break;

            case motionDirections.Vertical:
                // move up and down over time
                gameObject.transform.Translate(Vector3.up * Mathf.Cos(Time.timeSinceLevelLoad) * motionMagnitude);
                break;

            case motionDirections.Horizontal:
                // move left and right over time
                gameObject.transform.Translate(Vector3.right * Mathf.Cos(Time.timeSinceLevelLoad) * motionMagnitude);
                break;
        }
    }
}
```

Some points to note about this are:

- `public enum motionDirections Spin,Horizontal,Vertical`: basically an array where we define a list of all possible motion states
- `public motionDirections motionState = motionDirections.Horizontal`: initializes the motion state to horizontal (can be changed in the editor)
- the `switch` statement:
  - `case motionDirections.Spin`: initializes a case which instructs the program to check whether or not `motionDirections.Spin` is true
  - every case must contain a `break` statement (so that the switch case breaks on the first occurrence of a satisfied case and ignores all further cases)
  - there can be an optional `default` case which runs if none of the cases are satisfied

7.4 Player, camera, projectiles and shooting

(This section is currently in the works. Expect it to be up within a week's time.)
