

# Introduction to Game Developement

Om Prabhu

19D170018

Undergraduate, Department of Energy Science and Engineering  
Indian Institute of Technology Bombay

Last updated July 12, 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	About myself . . . . .	2
1.2	Motivation . . . . .	2
<b>2</b>	<b>Overview of Game Development</b>	<b>3</b>
2.1	Factors to consider before starting out . . . . .	3
2.2	Hierarchy of game structure . . . . .	3
2.3	Team structure and roles . . . . .	4
<b>3</b>	<b>The Unity3D Engine</b>	<b>5</b>
3.1	Unity3D interface and configuration . . . . .	5
<b>4</b>	<b>Basic Concepts</b>	<b>7</b>
4.1	Game graphics concepts . . . . .	7
4.2	Game audio concepts . . . . .	8
<b>5</b>	<b>Solar System Simulation</b>	<b>9</b>
5.1	Importing assets into Unity . . . . .	9
5.2	Game objects and the transform component . . . . .	9
5.3	Adding behaviour to game objects . . . . .	10
5.3.1	Adding scripts . . . . .	10
5.3.2	Adjusting script parameters . . . . .	10
5.3.3	Parent-child relationships . . . . .	11
5.4	Adding materials, lighting and audio . . . . .	11

# 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.

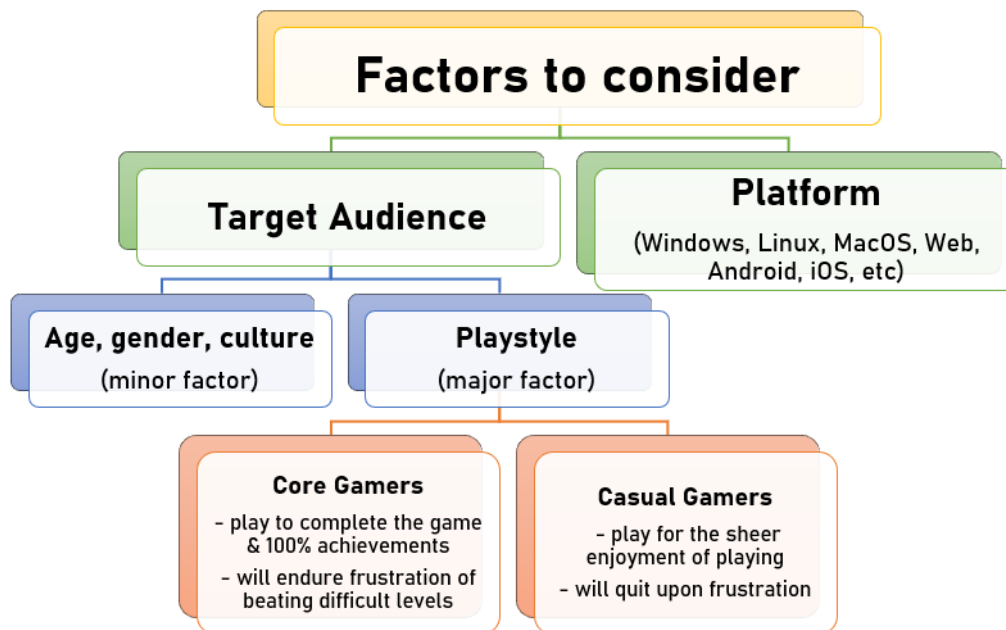
---

### 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](https://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.
-

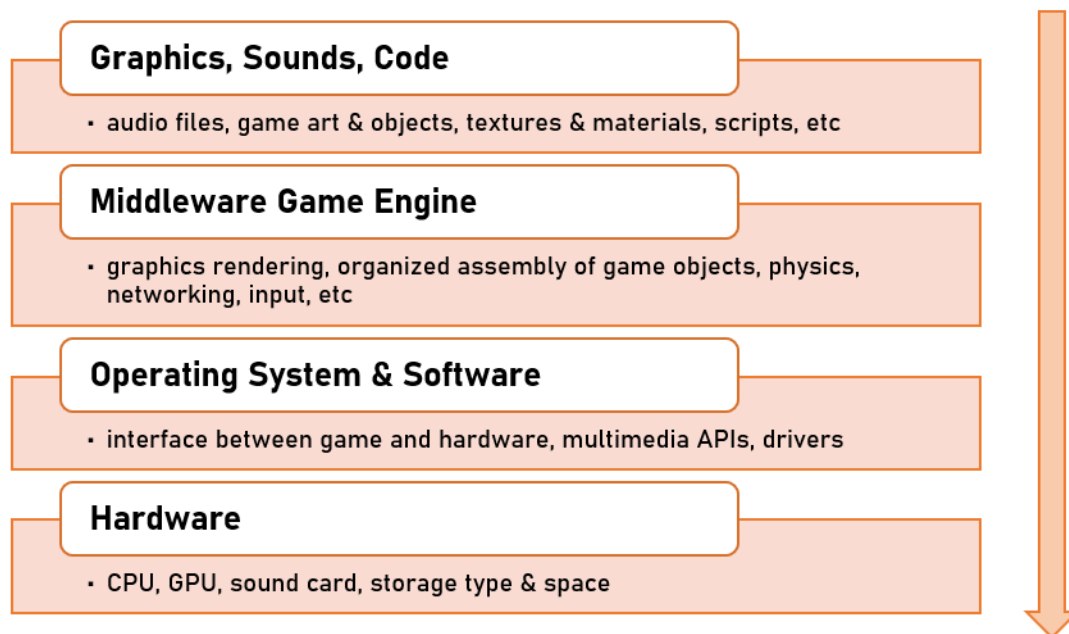
## 2 Overview of Game Development

### 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.

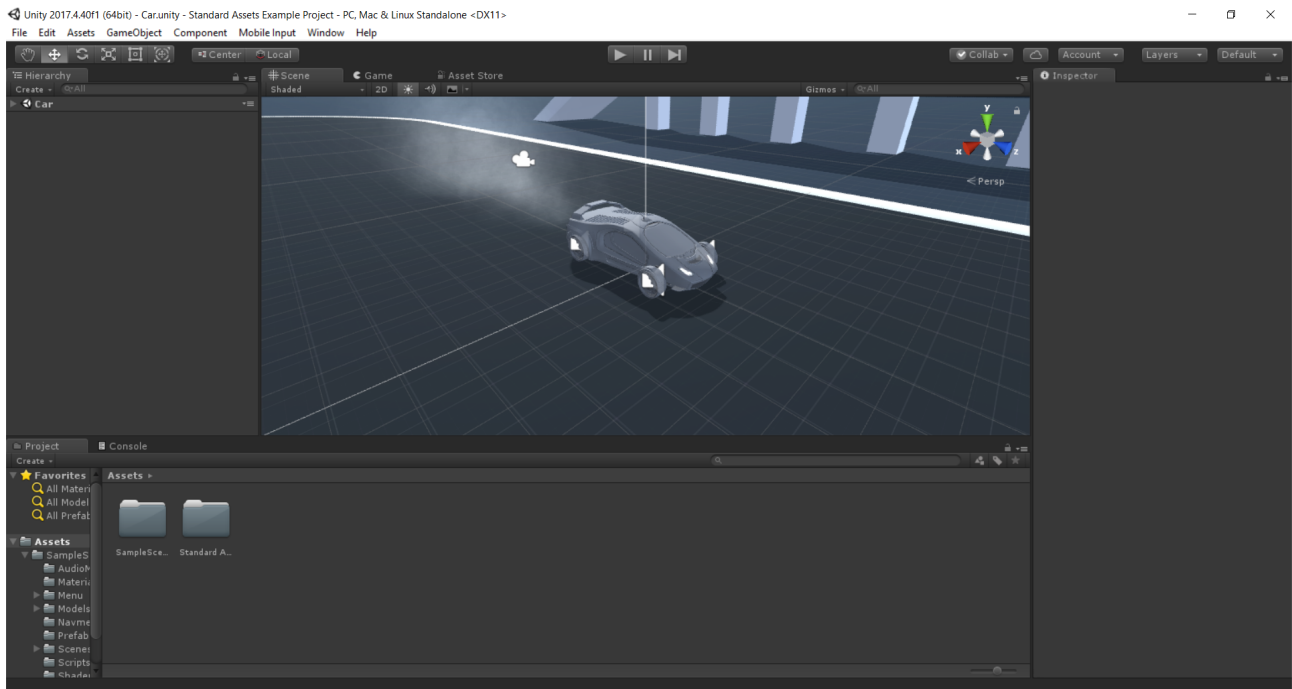
Some advantages of using the Unity3D game engine are:

- 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

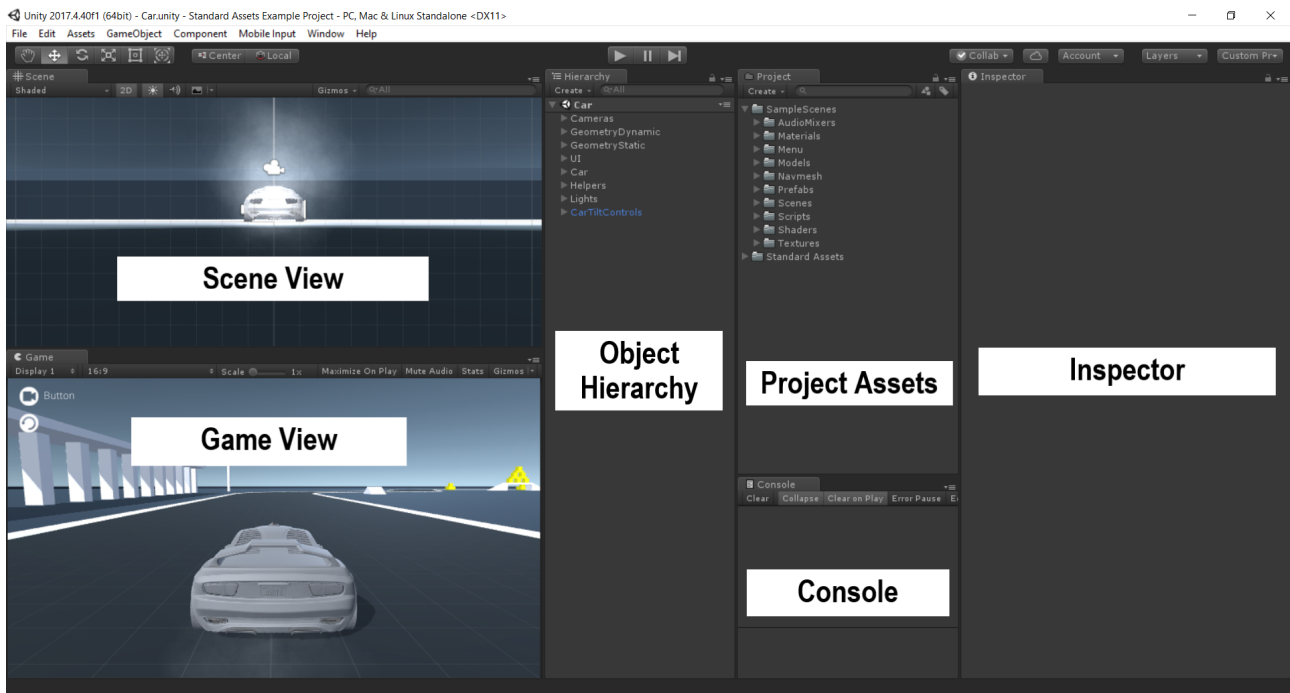
**NOTE:** If you take up this course, the instructor will mention that Unity3D offers flexibility in programming languages between C# and JavaScript. This is no longer valid after the 2019 LTS releases. However, many .NET compatible languages like C++ can be used if they are first compiled into a DLL (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

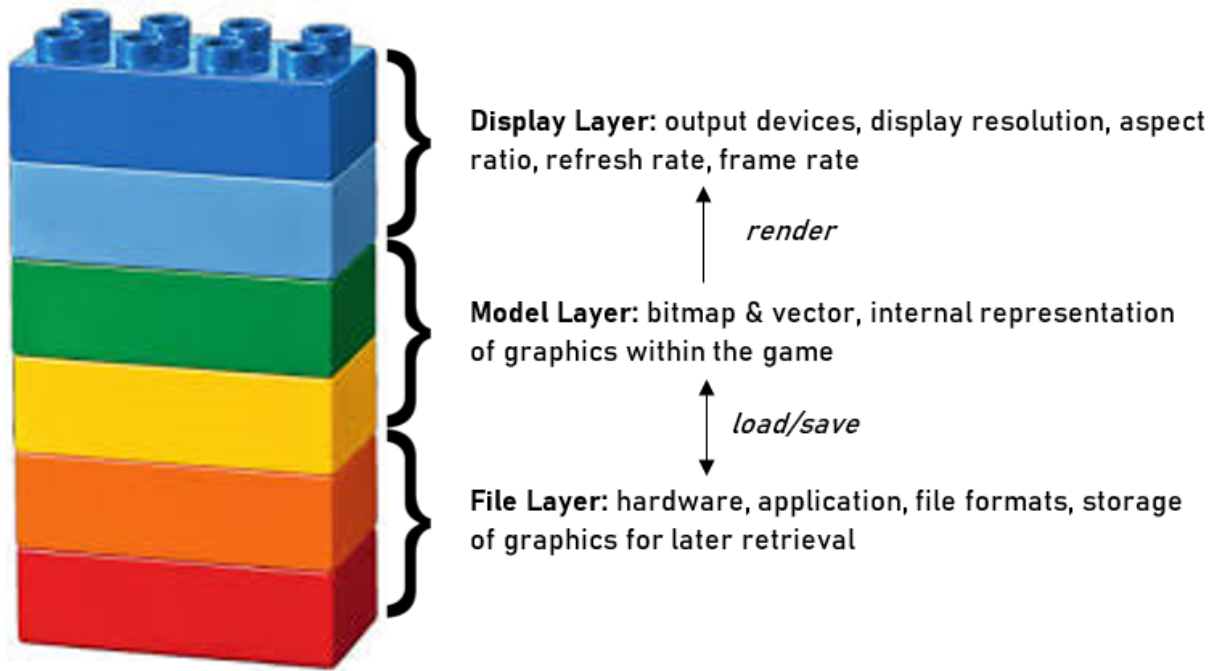
You might now want to explore the Unity editor to look at various features. There are also many quick shortcuts in Unity that you can refer to here: <https://docs.unity3d.com/Manual/UnityHotkeys.html>.

**NOTE:** If you ever need to move/rename/delete asset files, do it in the Unity editor itself (in the Project Assets 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  $1920 \times 1080$ , dots/pixels per inch)
- aspect ratio: proportion of pixel map width vs height (eg:  $1920 \times 1080$  and  $1280 \times 720$  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 for further weeks, or head over into Unity and try out some stuff yourself.

---



## 5 Solar System Simulation

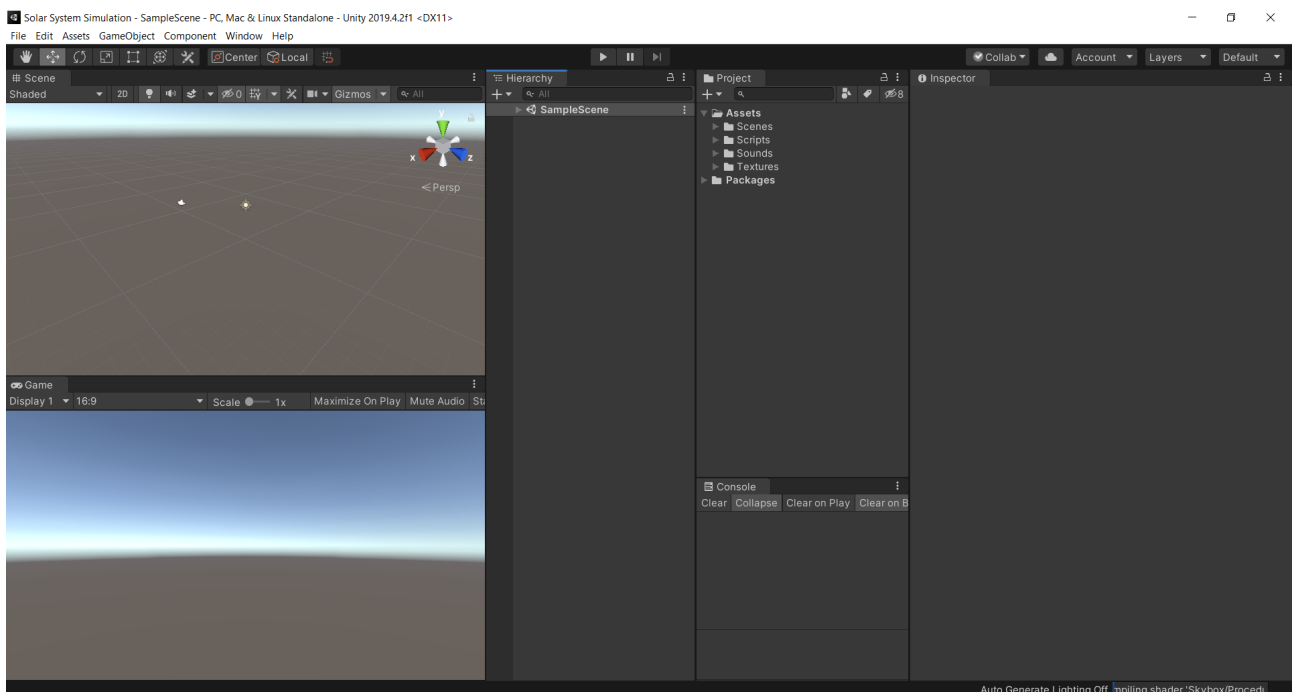
While a solar system model is not technically a game, it is useful to start small and understand the basic workflow that goes into making a game. Sadly due to copyright issues, I am not sure if I can directly provide you the link to the project assets. In case you wish to follow along as I discuss the 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**).

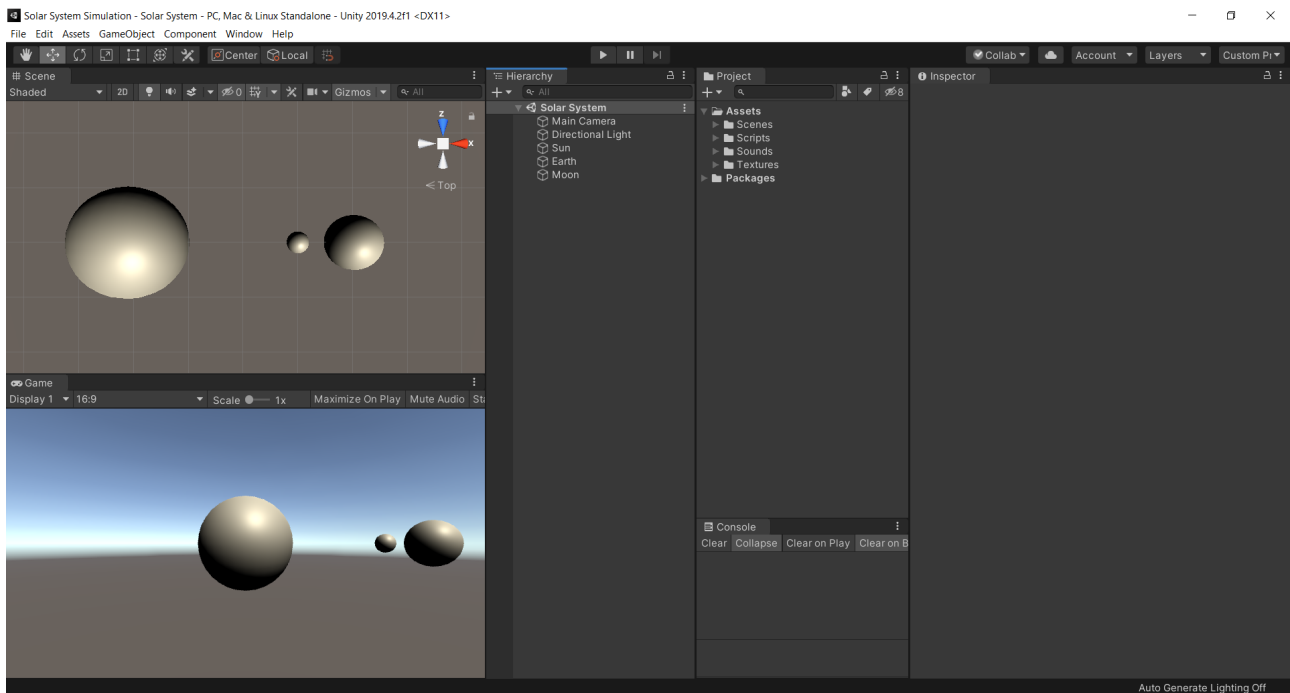
### 5.2 Game objects and the transform component

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

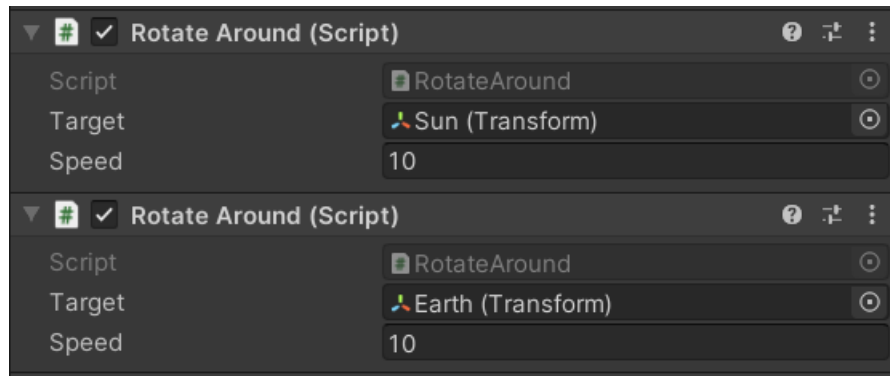
You will notice that a new ‘Rotate Around’ component has been added into the inspector panel for the particular game object. It also contains two parameters - target (the object around which the selected game object will rotate) and speed (angular speed of rotation in degrees per second).

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

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

---