NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF ELECTRICAL & ELECTRONIC ENGINEERING

IM2073 (IM201) PROJECT

ACADEMIC YEAR 2021/2022

PROJECT MANUAL

MODULE

GAME PROGRAMMING

Software Engineering Lab
(S2.2-B4-04)

NAME :_____
GROUP :_____
DATE :_____

Welcome to IM2073 Game Programming.

Contents

Introduction

Unity 3D is a cross-platform game engine with a built-in IDE (Interactive Development Environment) developed by Unity Technologies. It is used to develop video games or other digital experiences and simulations. The target hardware platforms include desktops, consoles, VR/AR headsets, and mobile devices; the devices can run any of the major operating systems, including Android, iOS, Windows, MacOS, and Linux distributions. As of 2019 the Unity engine powers over 50% of games worldwide on PCs, consoles and mobile devices. It is, as of 2021, the most popular game engine for game developers.

Please use the following account to log in the computer:
username: xxxx
password: xxxx

1. Motivation

Very few game engines on the market provide the ease of use and power available at such an affordable price point. The Unity 3D Game Engine has been used to create quite a number of successful game titles ranging from casual games all the way to Massive Multiplayer Online Games. Sample games are Kerbal Space Program and League of Legends: Wild Rift. Would-be game developers will find that learning to use the Unity 3D game engine is an invaluable stepping stone into the world of game development. For the latest software download, go to: http://unity3D.com/unity/download/

2. Objective

By the end of the course, the student should understand how the Unity 3D game engine functions. The student should also be able to create a single player game, or a simple multiplayer game using C#. Creating the game includes adding and manipulating 3D objects within the game world and writing C# scripts.

3. Scope

What this course will teach you:

- How to create and manipulate objects
- How to make the camera follow objects
- How to create and use C# scripts
- How to create and program triggers
- How to create animations
- How to use ml-agents with Unity for machine learning training

Part I: Game Programming Basics Using Unity 3D

This manual consists of two major parts. The basics of Unity 3D will be covered in Part I and a real game example will be covered in Part II.

Procedure 1: Introduction to the Unity Interface

In this part of the manual, we will explain the key sections of the interface and describe how to create basic game objects and position them in 3D space. To demonstrate the Unity interface essentials, we will create a simple scene which will allow a character to walk around on a surface.
Time to complete: 1.5 hours.

Point 1: Gettting Familiar with Unity 3D Screen Layout

Before we start, let's adjust our screen layout. Go to *Window->Layouts->Default* Your view is now split into 5 main regions:

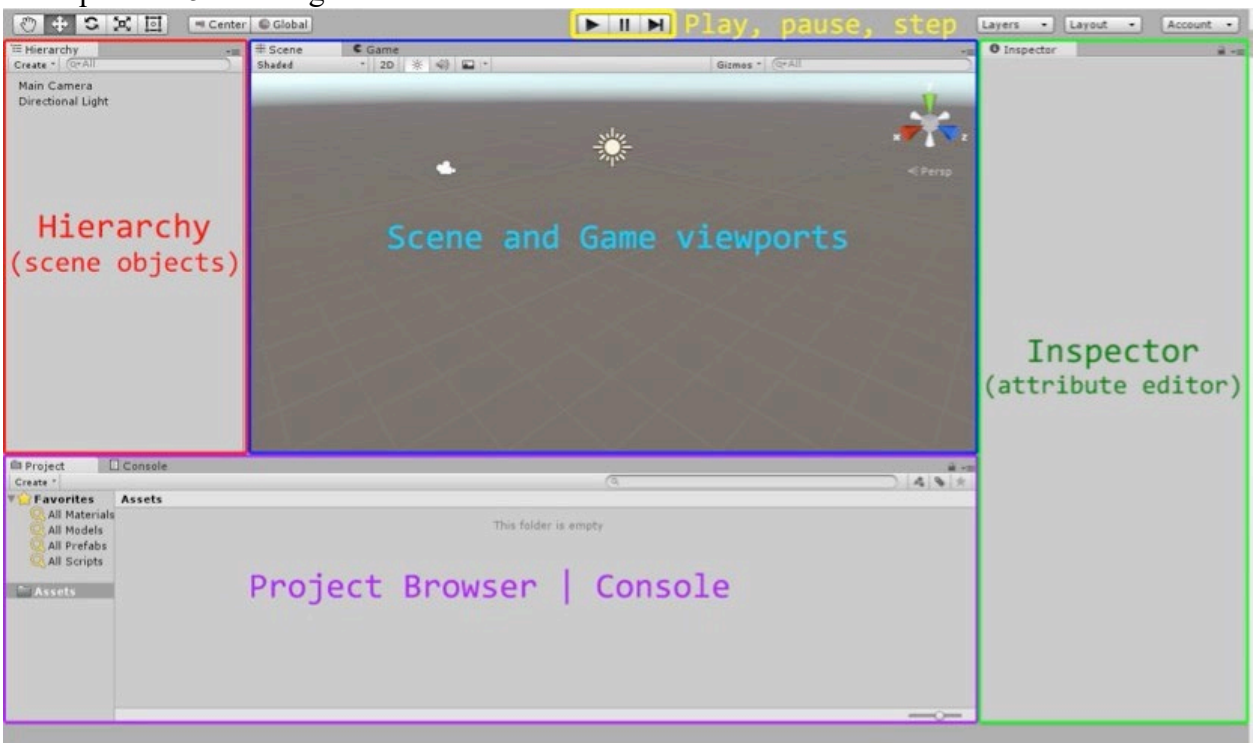
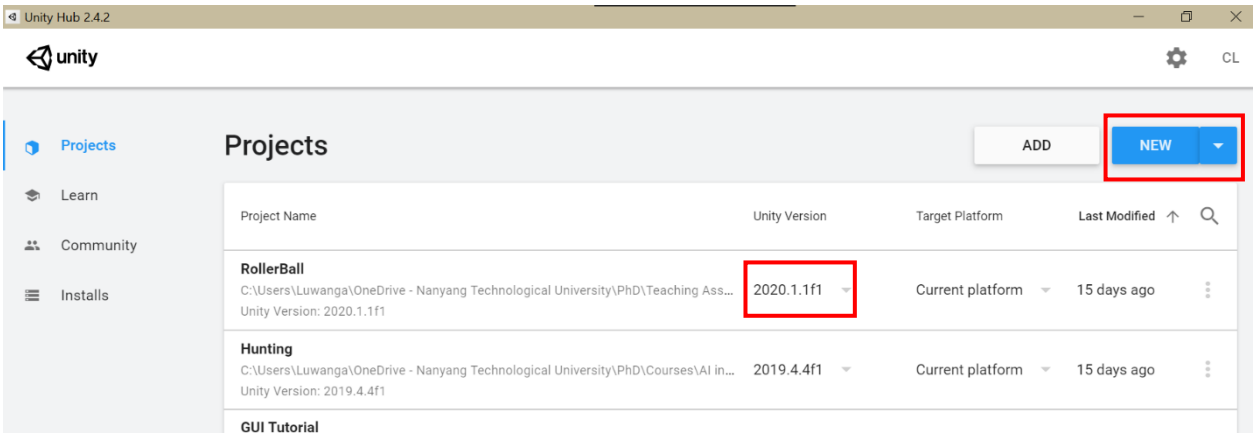


Image from Center for Virtual and Augmented Reality, NTU

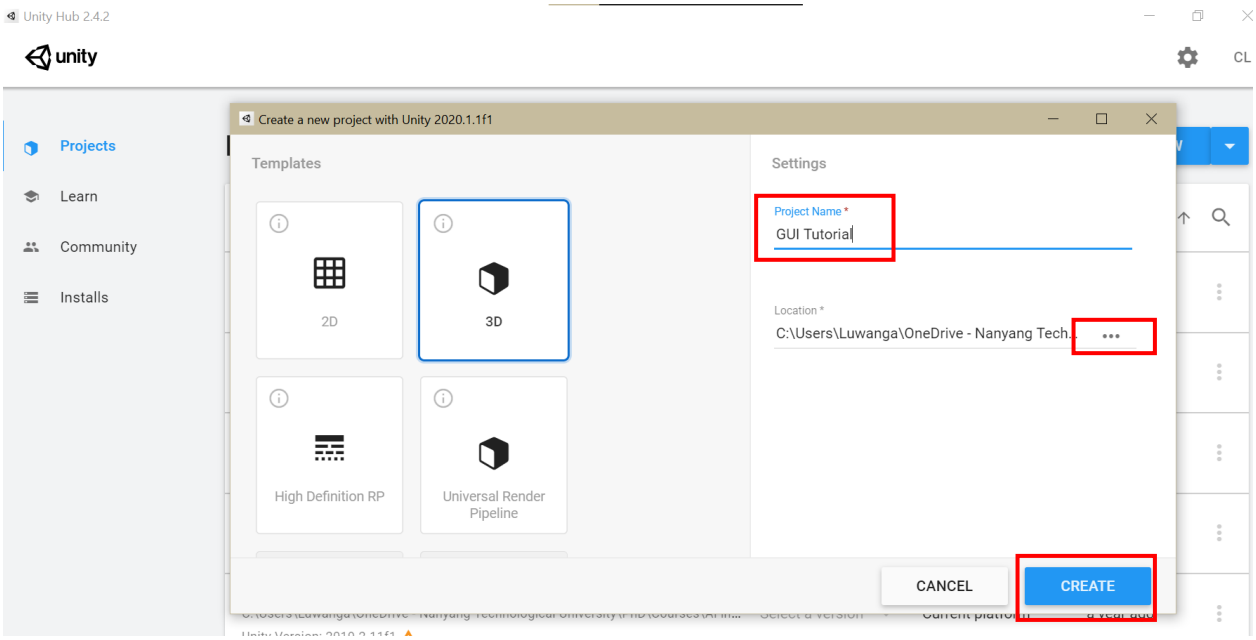
1. Scene view - for placing game objects
2. Game view - this is how the game will look when run.
3. Hierarchy view - this is a list of all game objects currently in the scene.
4. Project view - this view shows all assets that are available for use (like a palette).
5. Inspector view - this shows the properties of the selected game object.

Point 2: Finding game objects in 3D space

Unity Hub is a complementary application that is used to manage Unity projects and Unity Editor versions. S ometimes a game can only be opened with one version of Unity Editor. Let's first create a new project. Launch Unity Hub and select NEW. This will open a new project with the default Unity Editor version (2020.1.1f1). You can click the arrow next to NEW when you need to create the project with another Unity version.



Select 3D, if not already selected, and give your project a name such as "GUI Tutorial". Click the three dots next to the location field in order to choose a folder you prefer to keep your work.

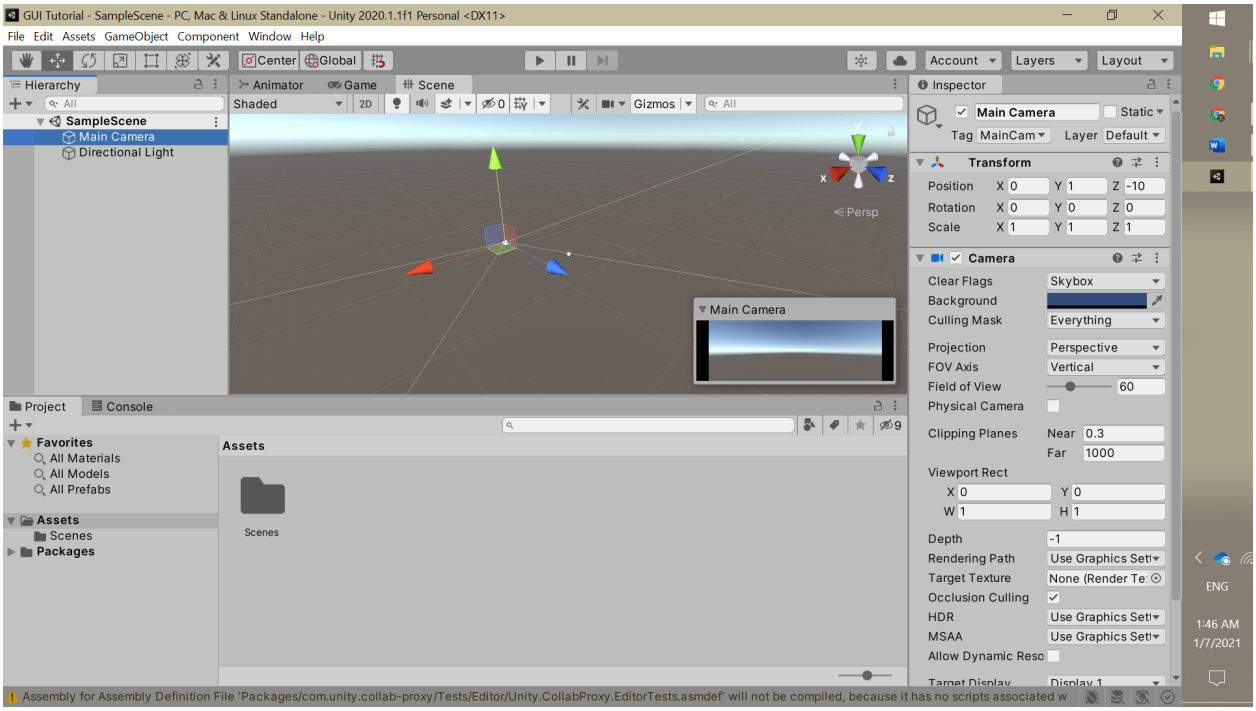


Unity will now import assets (e.g. audio, textures, 3D models) so they are available for use.

Once importing is complete you will be presented with your game world. At present the world only contains a Main Camera and a Directional Light (this is default); you can see these two items

in the Hierarchy view.

Exercise 1.2:
Step 1: Find the Camera. If you can't see the camera in your Scene view, click on the *Main Camera* in the Hierarchy view, move your cursor so that it's inside the Scene view (but don't click), then press F (Shift f = frame select) on the keyboard. The Main Camera should now be at the centre of the Scene view. This technique can be used to find any game object.

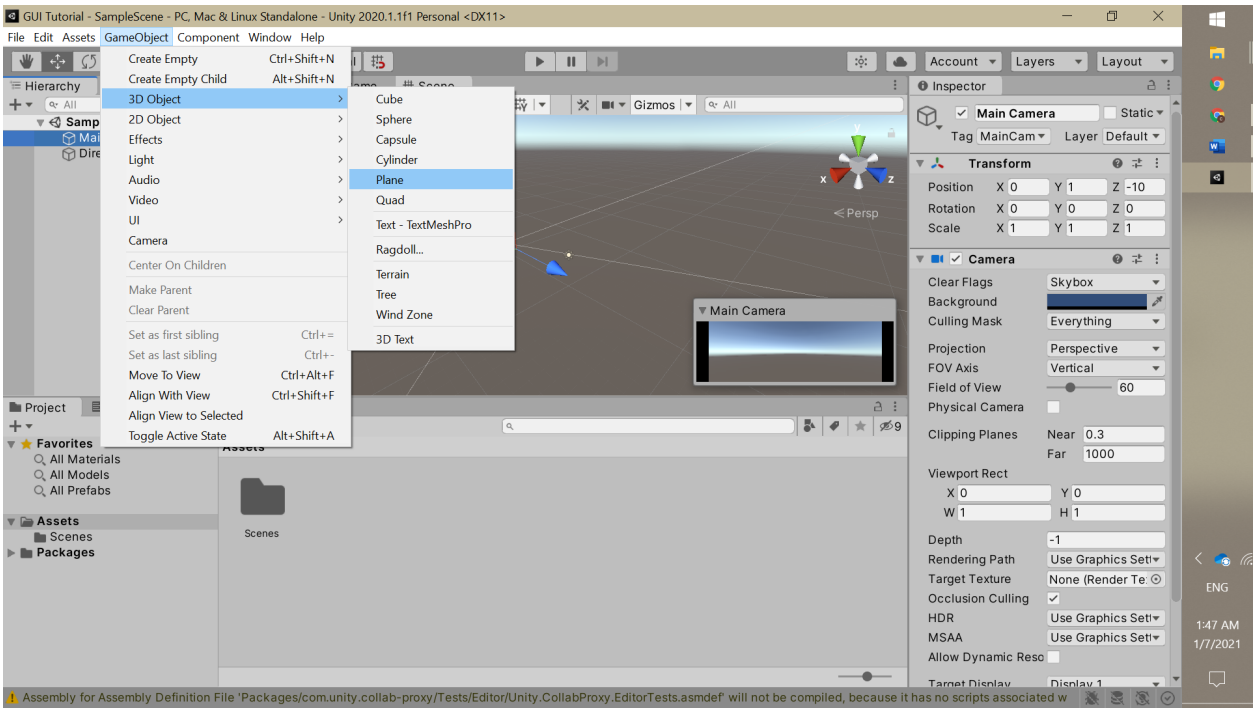


Step 2: Select Object. Any game object can be selected by either clicking on it in the Scene view or selecting it in the Hierarchy view. If you click on an object in the Scene view, you can confirm that you've selected the correct object as it will also be highlighted in the Hierarchy view.

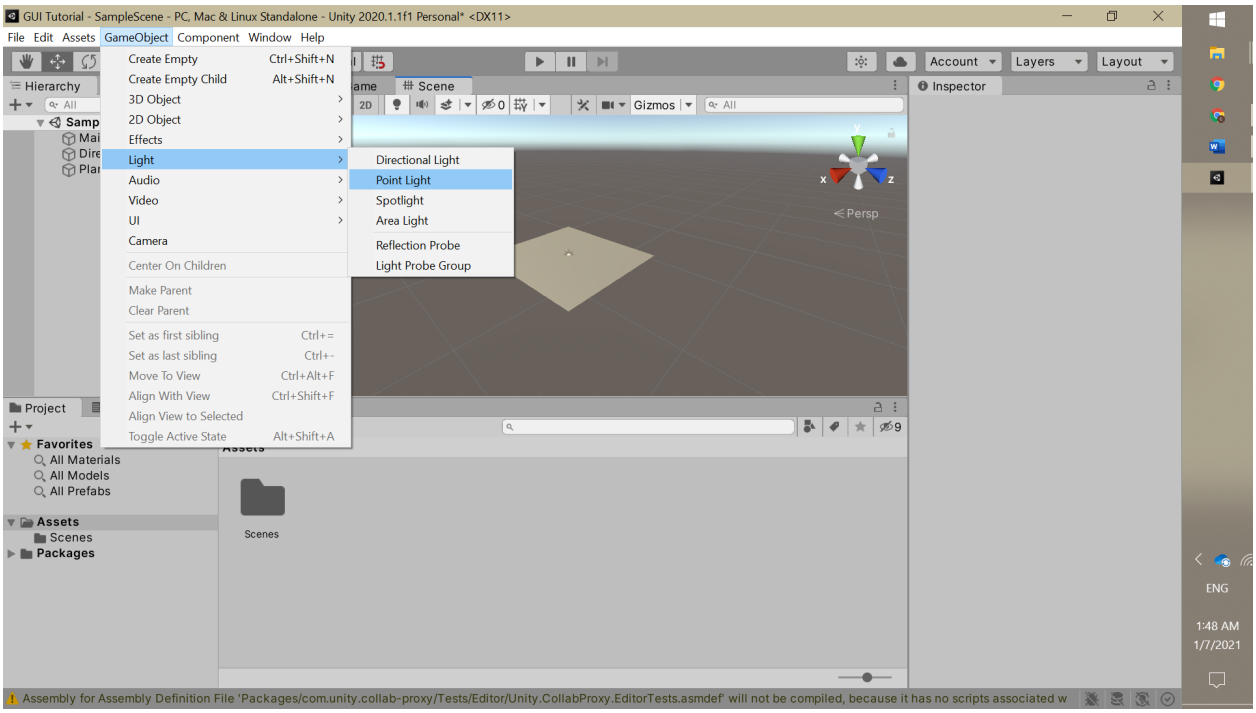
Point 3: Creating game objects

Let us add some game objects to our world!

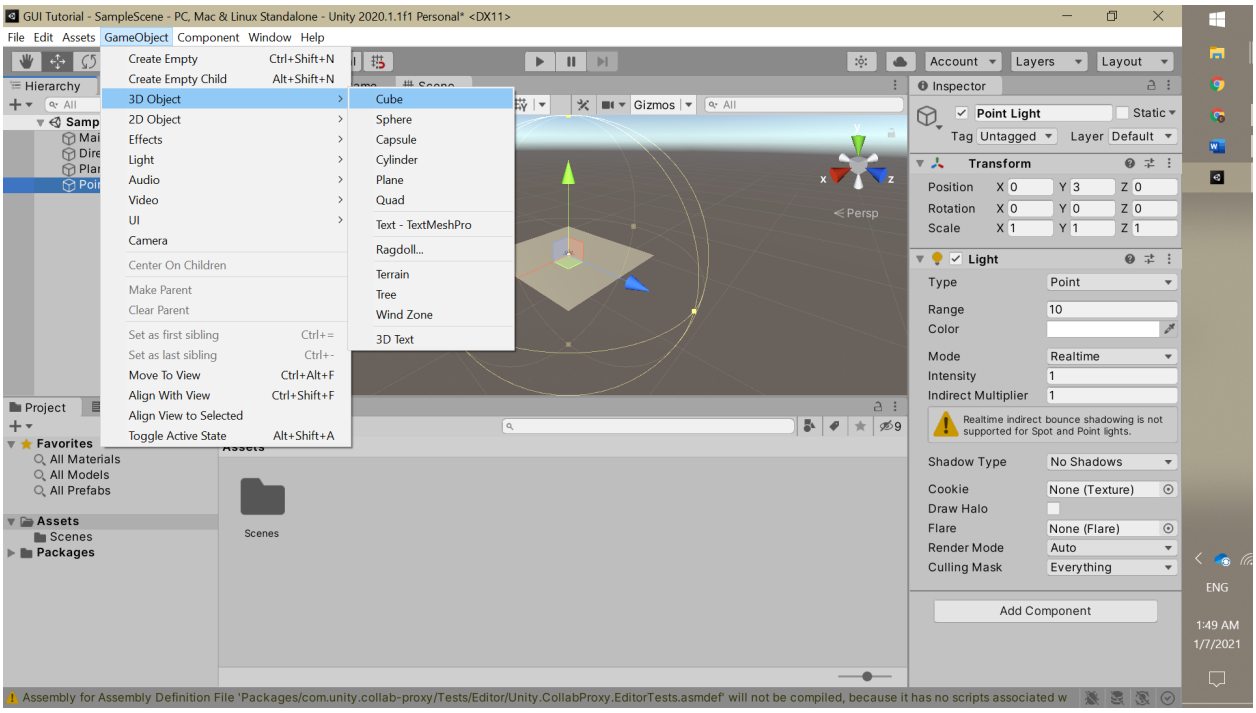
Exercise 1.3:
Step 1: Create a Plane. Select *GameObject->3D Object->Plane*, this will create a plane (a 2D surface) to allow our character to walk on.



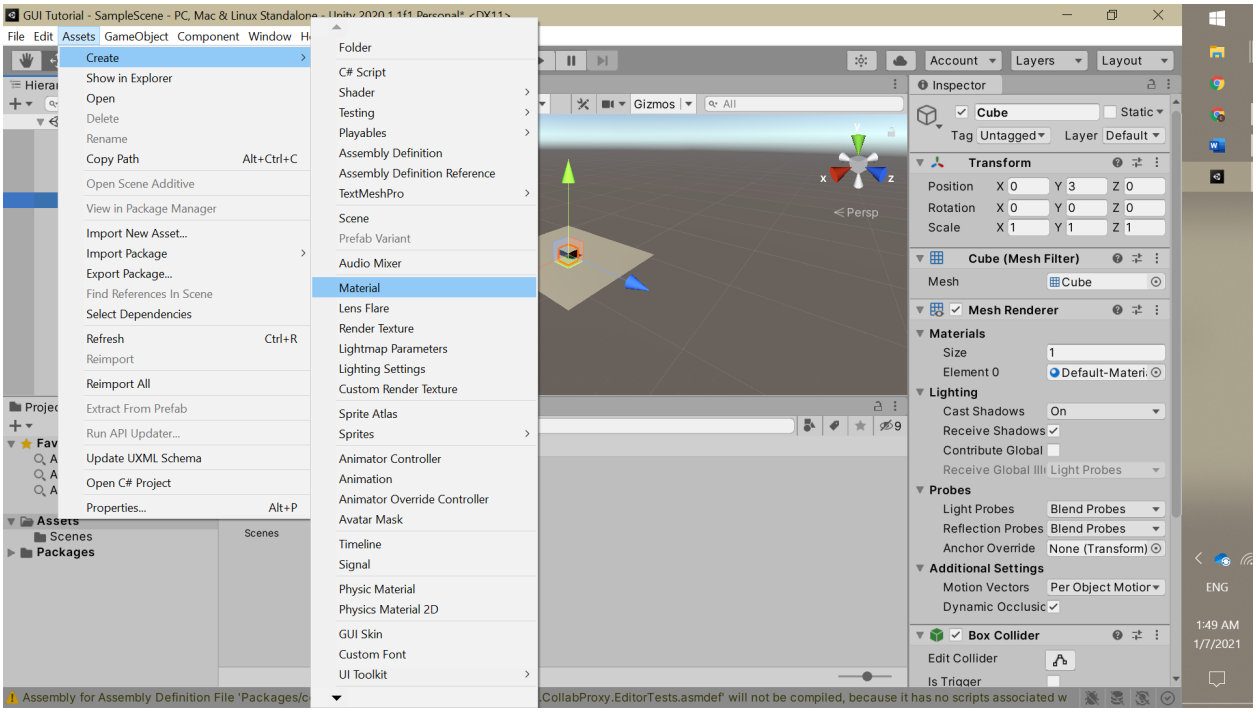
Step 2: Add a point light by selecting *GameObject->Light->Point Light*. *Point Light* needs a location and range. It sends out light in all directions, with its light intensity diminishing with distance from its location; the intensity reaching 0 at point R = range, which is a value you can set in the inspector.



Step 3: Add a cube to the game world by selecting *GameObject->3D Object>Cube*.

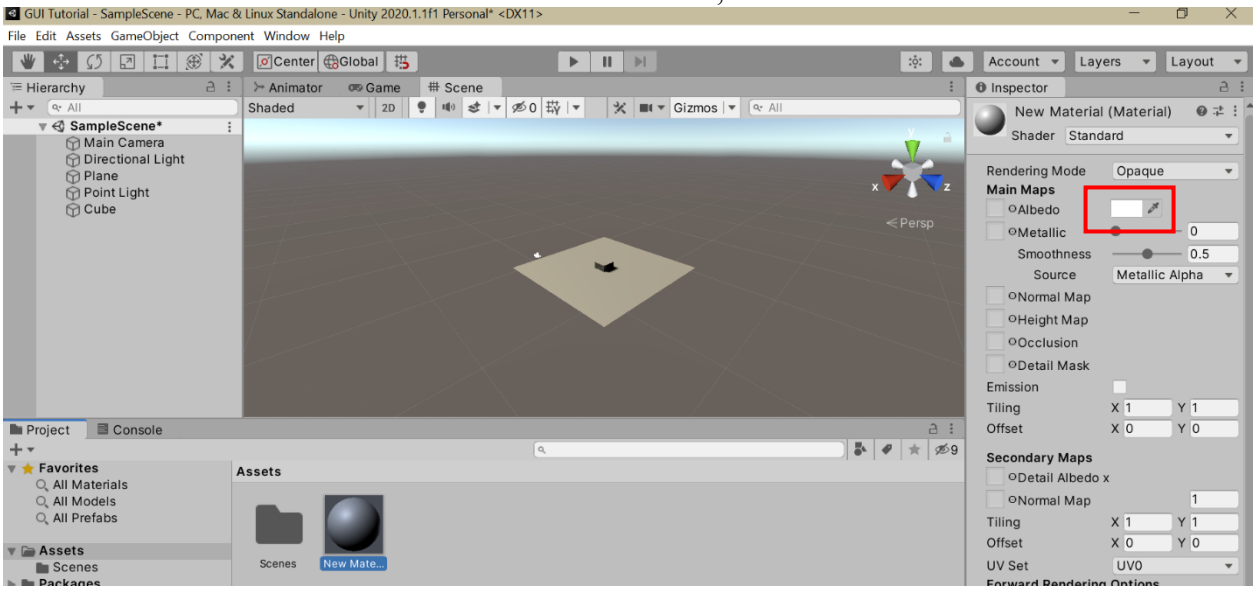


Optional :You **may add a color to the cube to** make it easily visible by doing the following:
Select *Assets>Create>Material,* this will create a material sphere.

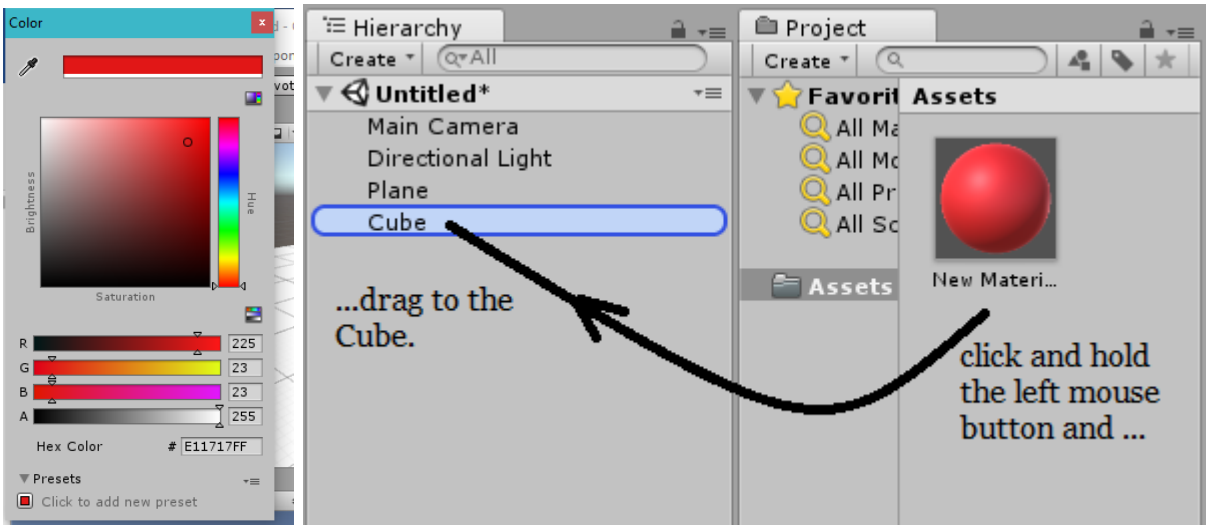


Go to the Inspector view of the material sphere, click on the white bar next to the Albedo of the Main Maps.

Choose your desired color in the color chart popped up; see that the material sphere in the Assets View is colored with the color chosen, then close the color chart.



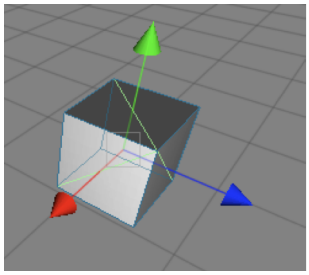
Click and hold the colored material sphere with the left mouse button and drag it to the Cube in the Hierarchy view and you will see the cube colored in the Scene view.}



Point 4: Navigating the Scene view

You can look around your 3D objects from any angle as follows.
1. **Panning and Rotating:** Move the cursor into the Scene view, press and hold the right mouse button down, and move mouse until a desired viewing angle is achieved;
2. **Zooming**: Move the cursor into the Scene view, and roll the middle mouse button, until a desired camera distance is achieved.

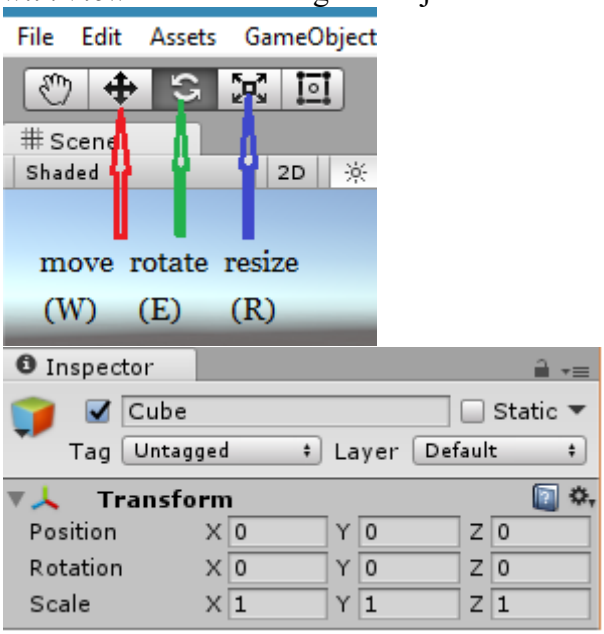
Exercise 1.4: Try all of the above actions yourself.

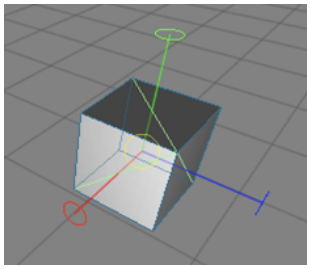


Point 5: Moving game objects

You can position any game object in 3D space by using one of the following:

1. Move tool (W on the keyboard or  on toolbar). Selecting any game object then typing W will display three arrowheads, one for each axis of 3D space. Red is the x-axis, green is the y-axis, blue is the z-axis, i.e. r,g,b, maps to x,y,z. To move an object, click the LMB on the arrowhead that represents the axis you want to move in and move the mouse as much as appropriate. As you drag the arrowheads notice the object's position values in the Inspector Panel change also. You can enter values directly into these boxes in the Inspector Ppanel also (3 boxes, 1 each for x,y,z values).
You can also re-position a game object by moving the camera to point towards the general direction of where you want the object to move to, then click on the object and select *GameObject->Align with View*. The selected game object will move to where the camera now points.





2. Rotate tool (E on the keyboard). Selecting any game object then typing E will display the rotate tool. To rotate around an axis, click the LMB on the axis of choice then move the mouse to rotate the game object. As you rotate the object notice the object's rotation values in the Inspector Panel change also. You can enter values directly into these boxes in the Inspector Panel also.

Exercise 1.5:
Step 1: Move the cube (move tool) so that it is outside of the Scene view. Use the Frame Select feature (click Cube in the Hierarchy view then press the F key) to find the cube again. Finally move the cube back so it's close to the plane.
Step 2: Move the point light so that it's above the cube and the plane.

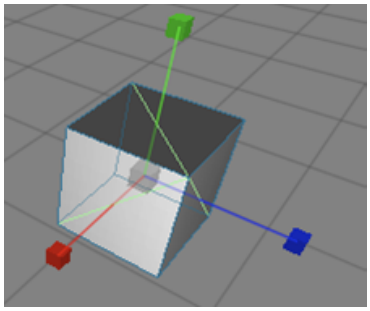
Point 6: Game view

The Game view shows how the game will look when run (complete with textures, lighting etc.). If you cannot see your game world in the Game view, it's because your camera is not looking in the

right direction.

Exercise 1.6:
Step 1: Click on the *Main Camera*, you will notice an inverted pyramid wireframe coming out from it. (If you cannot, go to *Main Camera*'s Inspector view and change the *Clipping Planes →Far* value to a smaller number, such as 137) This (pyramid) is known as the camera's viewport (what the camera can see). If the viewport is not pointing in the direction of your game objects, navigate in the scene view so that you are looking at your game objects, select the *Main Camera* then select *GameObject->Align With View*. Your Game view should now match your Scene view.
You can also manually adjust your camera's settings by either using the move and rotate tools, or changing its transform values in the Inspector view. You may also want to move your light to give better aesthetics.

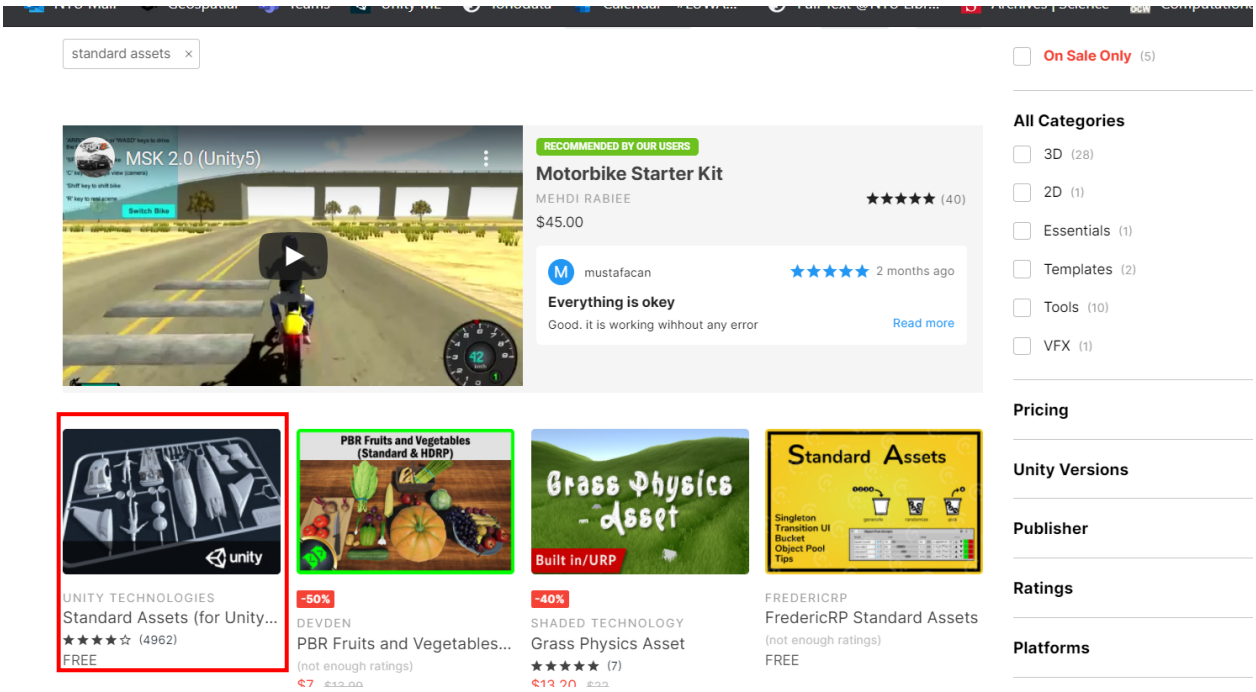
Point 7: Resizing game objects

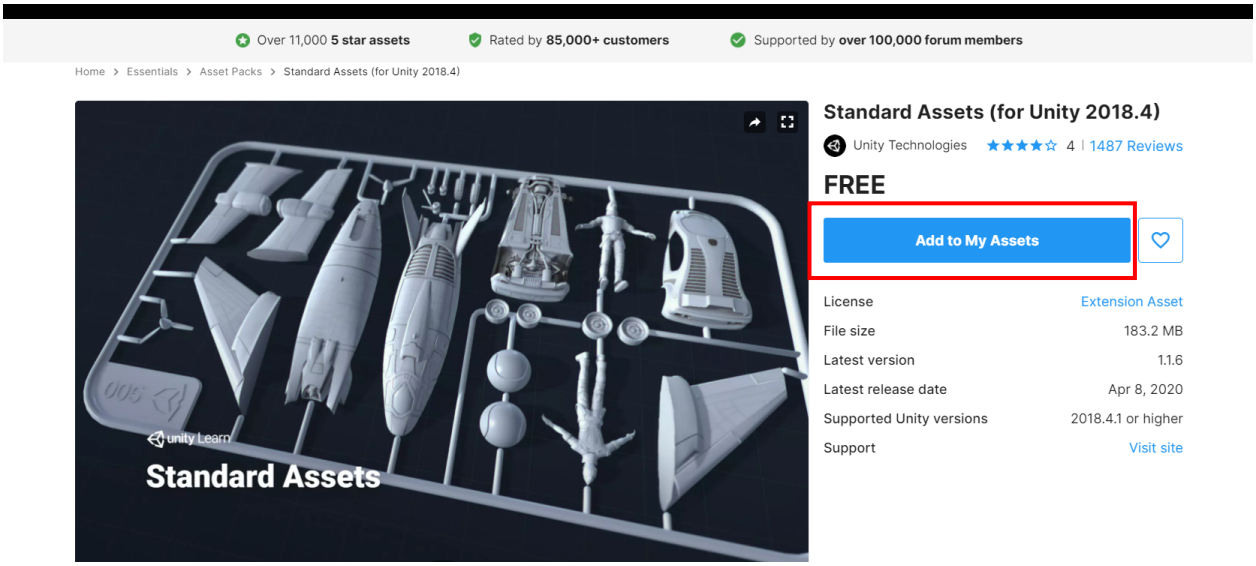


To resize an object firstly select it then press R on the keyboard (resize or scale tool). You can visually scale an object by dragging the square block at the end of the each axis, or use the Inspector view to type a precise value in.
Exercise 1.7: Resize some objects in our game world.
Step 1: Select the plane and set its x and z scale values to 10 in the Inspector view (the first and third box, as the second box is for the y value).
Step 2: Select the cube, resize it along the y-axis visually (press R and drag the green square upwards). The cube should now look like a pillar.
Step 3: Rename the cube *Pillar*. Do this by selecting it in the Hierarchy view, click on the right mouse button, choose *Rename* and enter the new name. Remember it's best to ensure game objects begin with a capital letter.

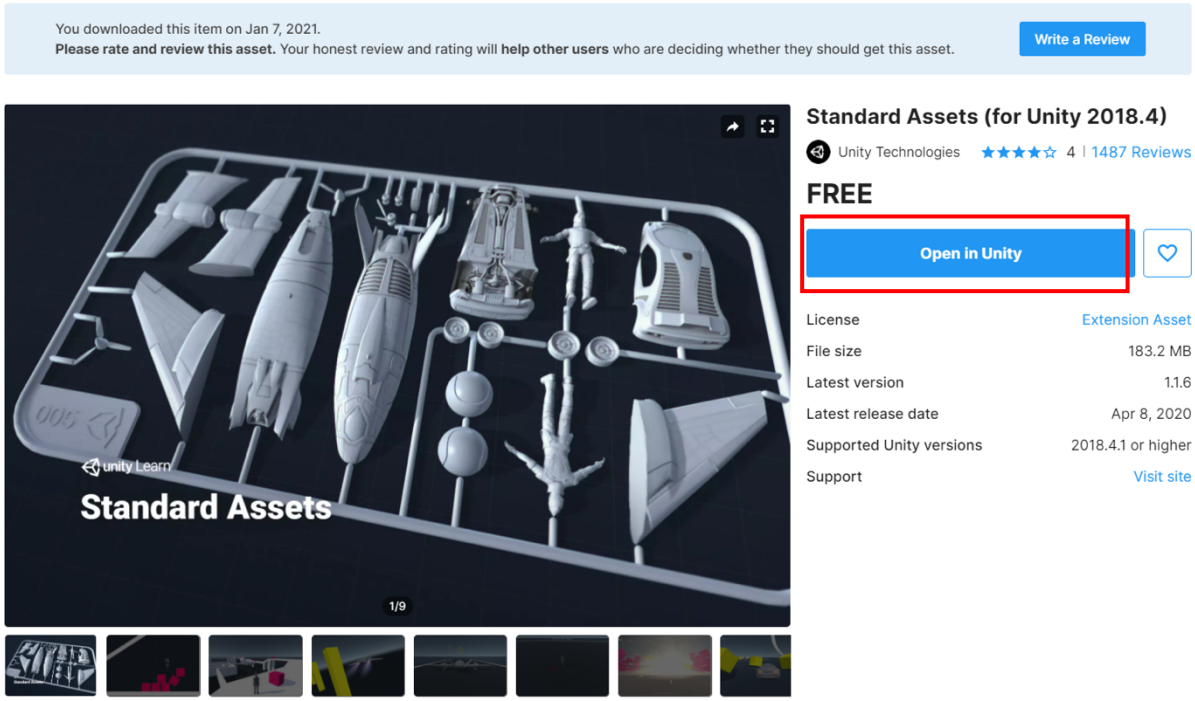
Point 8: Using Assets

Computer games are created from 3D models, textures, sound files, game code etc.; these are called assets. Unity comes with many assets shipped as *Standard Assets*. We'll now add a character to our game world from the built-in assets. Download Standard Assets (for Unity 2018.4) from the Unity Asset Store. (This is to show the process, but the use of **Standard Assets** is not heavily recommended and is the package is deprecated by Unity)
Go to *Window →Asset Store*. In the "Search for assets" search field type "Standard Assets". Choose *Standard Assets (for Unity 2018.4)*

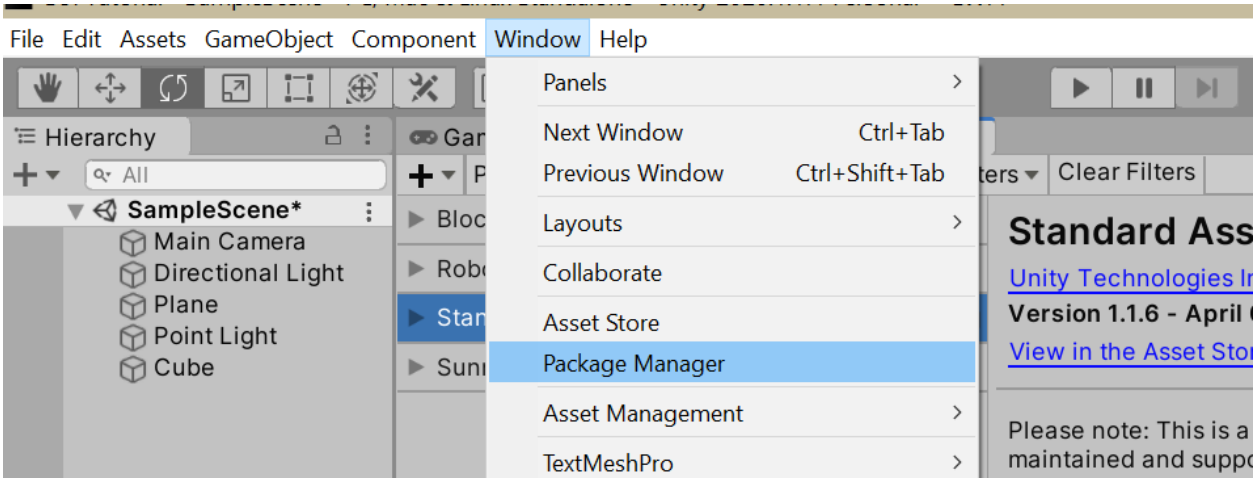


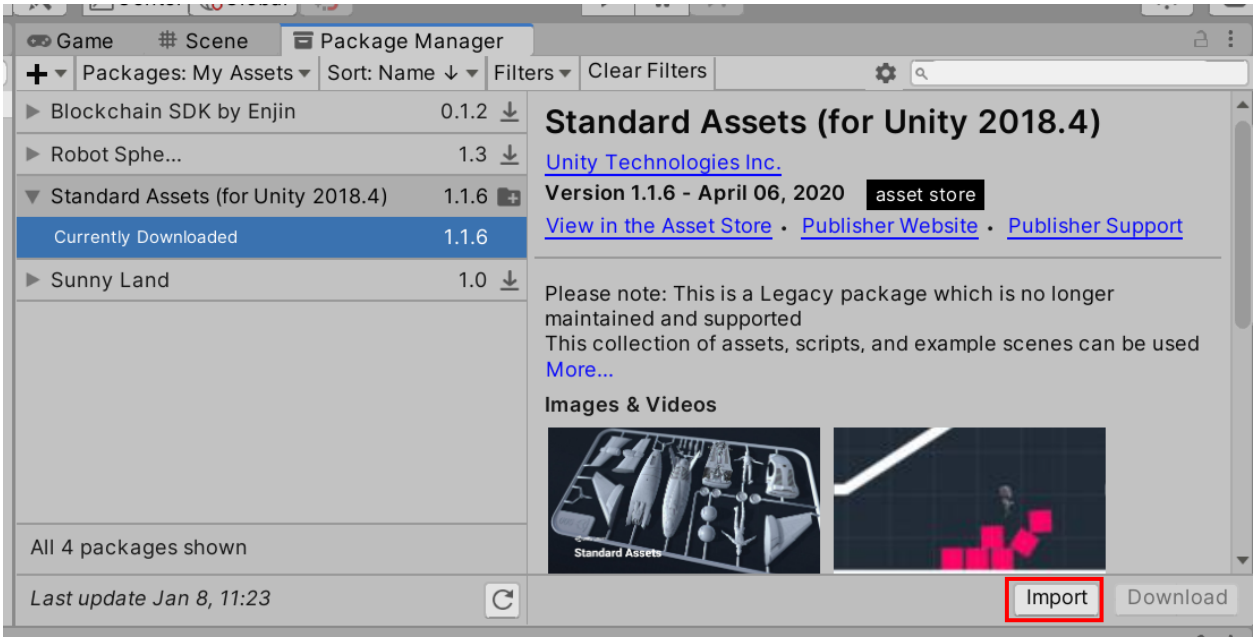


Click *Add to My Assets*. Log in in order to be able to find the assets in Unity Editor.

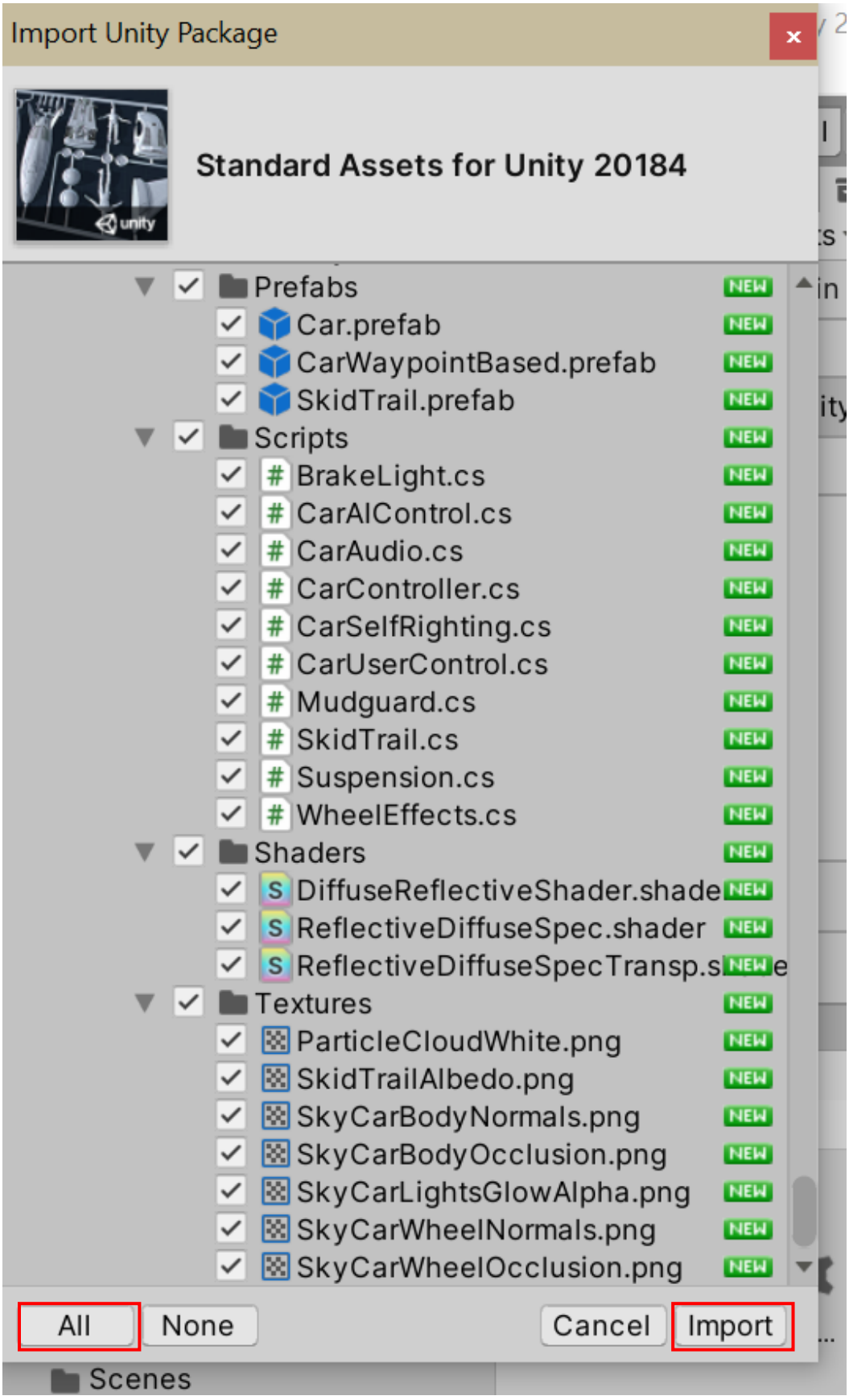


Upon clicking *Open in Unity* the *Package Manager* will be automatically opened. If this doesn't happen then go to *Window->Package Manager*



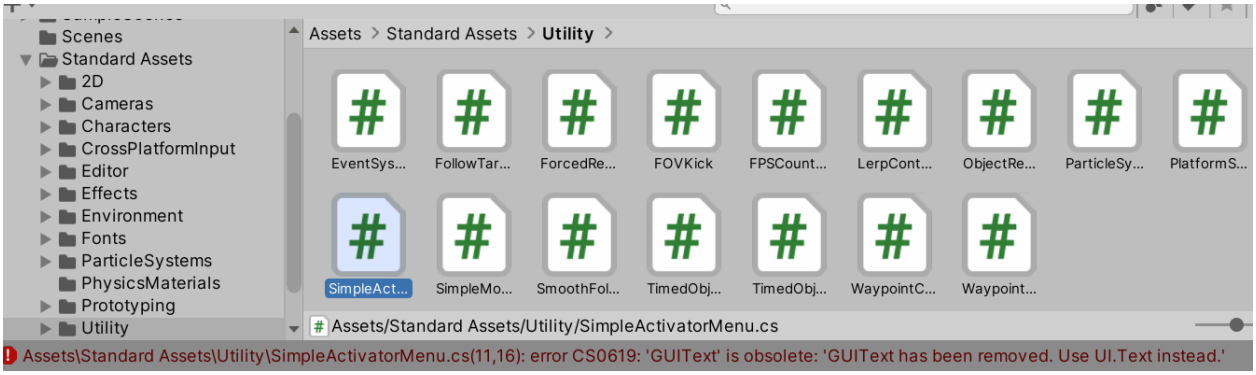


Import the package by clicking *Import.* If *Import* is not enable, click *Download* in order to download the package.


Import Unity Package
Standard Assets for Unity 20184
Prefabs
Car.prefab
CarWaypointBased.prefab
SkidTrail.prefab
Scripts
BrakeLight.cs
CarAIControl.cs
CarAudio.cs
CarController.cs
CarSelfRighting.cs
CarUserControl.cs
Mudguard.cs
SkidTrail.cs
Suspension.cs
WheelEffects.cs
Shaders
DiffuseReflectiveShader.shade
ReflectiveDiffuseSpec.shader
ReflectiveDiffuseSpecTransp.s
Textures
ParticleCloudWhite.png
SkidTrailAlbedo.png
SkyCarBodyNormals.png
SkyCarBodyOcclusion.png
SkyCarLightsGlowAlpha.png
SkyCarWheelNormals.png
SkyCarWheelOcclusion.png
NEW
All
None
Cancel
Import
Scenes

At import, click *All* in the popped out window to import all the items in the package, and then press *Import*.
After import is done, go to the Project view, click on *Standard Assets-> Characters-> FirstPersonCharacter-> Prefabs* under the *Assets* folder. You should see the *First Person Controller* (*FPS Controller)* [Note: FPS stands for First Person Shooter] inside the *Prefabs* folder. *FPS Controller* is a *Unity prefab*. **A prefab is a game object or a collection of game objects that have been pre-assembled into a logical grouping. A prefab can have defined properties, attached scripts, textures and so on.**
You may also choose to import other available packages, which will all be organized under the *Assets* folder for ease of use.

If you get an error about *GUIText* obsolescence make the correction in *Assets/Utility/SimpleActivatorMenu.cs.* Navigate to *Assets/Utility/SimpleActivatorMenu.cs* file (within Unity) and double-click to open the c# (c sharp) script in Visual Studio. Then remove *GUIText* (Search for *GUIText*) and replace with U*nityEngine.UI.Text*




```
2   using UnityEngine;
3
4   #pragma warning disable 618
5   namespace UnityStandardAssets.Utility
6   {
7       public class SimpleActivatorMenu : MonoBehaviour
8       {
9           // An incredibly simple menu which, when given references
10          // to gameobjects in the scene
11          public UnityEngine.UI.Text camSwitchButton;
12          public GameObject[] objects;
13
14
15          private int m_CurrentActiveObject;
```

Exercise 1.8:
Step 1: Drag the *FPS Controller prefab* into the Scene view. Notice how the Game view has changed, this is because the *FPS Controller prefab* comes with its own camera which takes precedent over the existing *Main Camera*. Make sure the *FPS Controller* is not caught in the plane, move it so that it is above the plane.

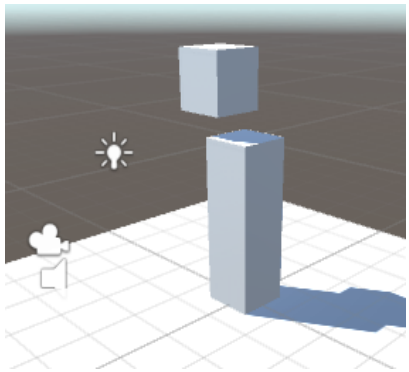
Step 2: Delete the *Main Camera* as we'll not need it now. The Game view is now what the player can see in-game.
Step 3: Start the game by clicking the *Play* button in the Unity GUI.
You can see the game in the Scene view and the Game view. Place the cursor in the Screen view, use the arrow keys to move the character around (or W,S,A,D), the mouse to look around and space to jump. [Note that you do not see any character because you, through the camera attached, are the eyes of the First Person.]
Step 4: Stop the game at any time by pressing *Esc* to show cursor and then clicking on the *Play* button again.

When the game is not running, the Scene view of the Game view may be expanded to full screen by pressing *Shift + Space* while the mouse cursor is inside any of the views (Game view, Scene view etc.). This is useful when making adjustments in the Scene view, or playing the game (Game view). Pressing *Shift + Space* inside the full screen view again returns to the previous layout mode. Note that this feature does not work if the game is currently running, either stop or pause the game first.

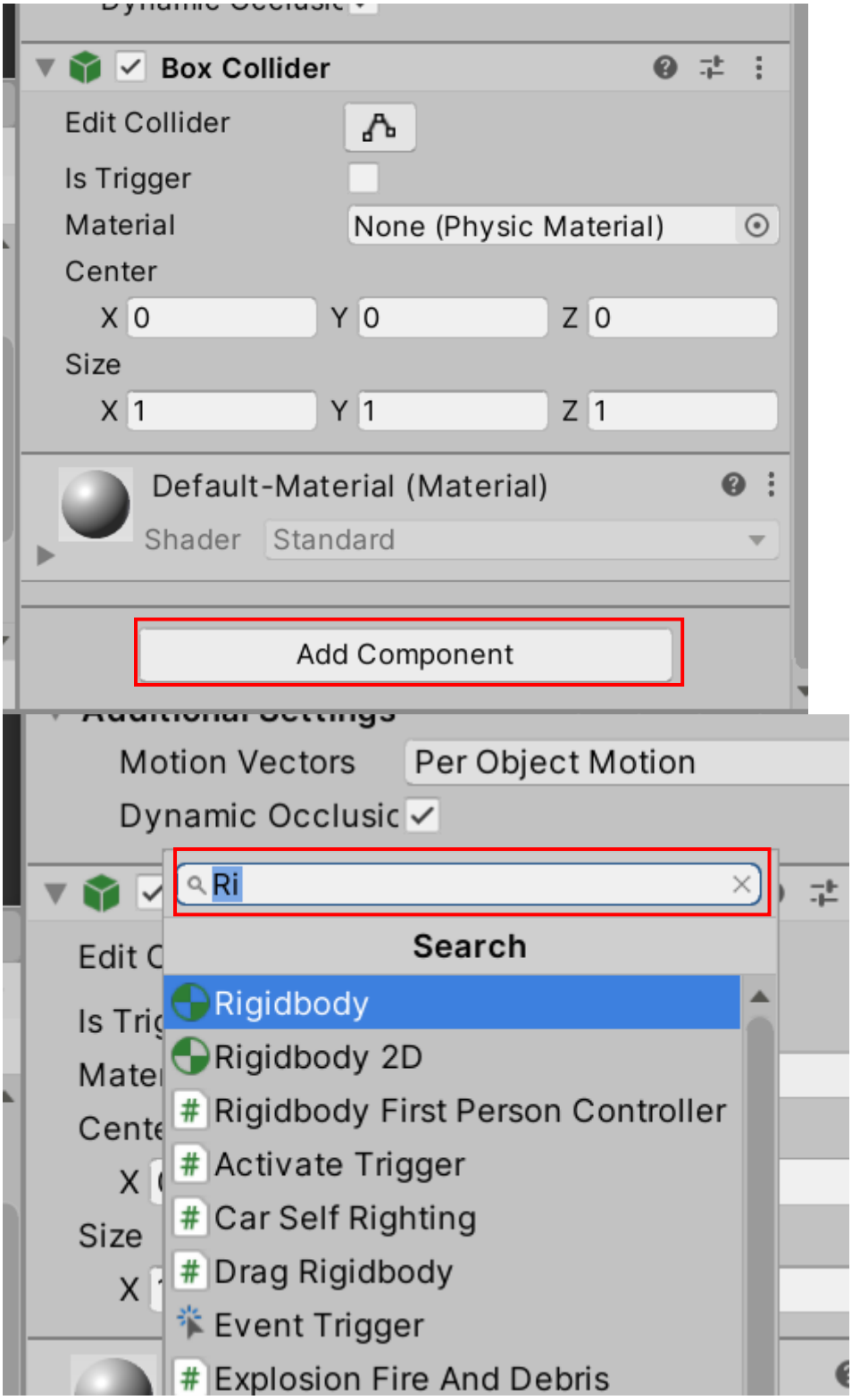
Point 9: Adding Components

Game objects can have a number of components (or behaviours) attached to them. By clicking on any game object, you can see the attached components in the Inspector view (e.g. Transform, Box Collider).
Click on *Pillar* and look at its components in the Inspector view.

Exercise 1.9: Adding components.
Step 1: Add another cube to the scene (*GameObject->3D Object->Cube*). Rename it *PhysicsCube* in the Hierarchy view.



Now we're going to add a *Rigidbody* component to this cube. **The *Rigidbody* component allows the game object to simulate real world physics behavior for example responding to the force of gravity; the object will fall downwards until it hits a surface with a collider component attached.**
Step 2: Making sure *PhysicsCube* is selected, select *Component->Physics->Rigidbody*. This adds the *Rigidbody* component to the currently selected game object. Notice that in the Inspector view the *Rigidbody* component has been added.
Alternatively in Inspector view find *Add Component* button and click it.



Step 3: Move *PhysicsCube* so that it is above *Pillar* but slightly offset (see Figure). You can also directly set the position parameters (x,y,z) in the Inspector view that belongs to *PhysicsCube* . We want to set the scene up so that when we press play, *PhysicsCube* will collide with *Pillar* and fall to the ground.

Step 4: Press Play, *PhysicsCube* should collide with *Pillar* and land on the plane.

Point 10: Duplicate

Duplicating a game object is one of the most powerful features of Unity. When duplicating an object, all features and behaviours of the object are also copied. It's a very fast way to create complex scenes. We'll now add more *PhysicsCubes* to our scene.

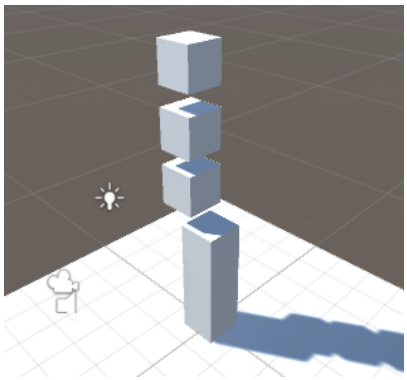
Exercise 1.10:

Step 1: Click on *PhysicsCube* and press *Ctrl+ D* (or do this from the Edit menu). Notice an extra *PhysicsCube* entry has appeared in the Hierarchy menu, however you cannot see it as it has been created at the same position as the first *PhysicsCube*.

Step 2: Move the new *PhysicsCube* by selecting the move tool (W) and move it upwards (y-axis).

Step 3: Repeat this process so there are now 3 *PhysicsCube* game objects (see Figure 2).

Step 4: Run the game and all 3 *PhysicsCube* game objects should interact and behave naturally.



Point 11: Import 3D Objects

A well-designed game requires sophisticated and polished 3D objects. Creating 3D objects is a complex and time-consuming task, however you can choose to import available 3D objects from the Unity Asset store or various CG forums. The web site (http://free3D.com/) is one of many good places to explore for free 3D models. (However, unauthorized publishing or distribution will result in a violation in Intellectual Property. You are at your own risk when downloading or using 3D model resources from the internet.)

Unity supports two main types of 3D model formats:

1. Exported 3D file formats, such as .FBX or .OBJ;
2. Proprietary 3D application files, such as .Max and .Blend file formats from 3D Studio Max or Blender respectively. Either should enable you to get your meshes into Unity.

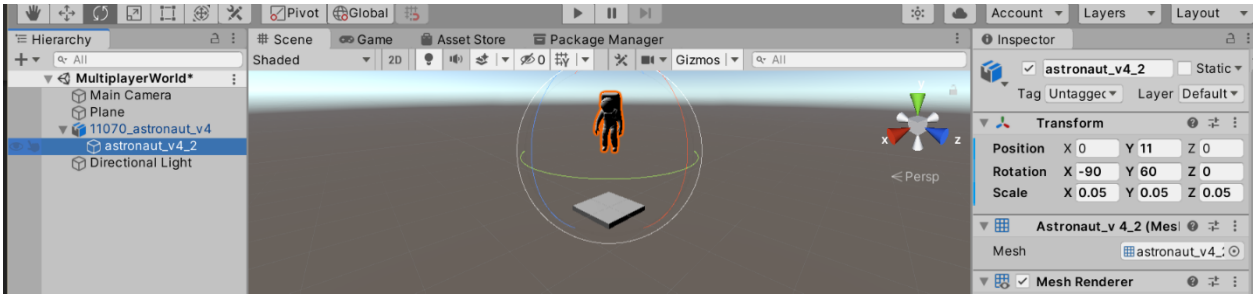
Exercise 1.11: 3D object import

We have already provided a 3D model in our course folder (in NTU Learn) under *\Resources*. In general, the models can be of different formats that are supported by Unity. We'll illustrate here how to import one of them into Unity. Start by creating a plane (10,10,1) on which the object will be.

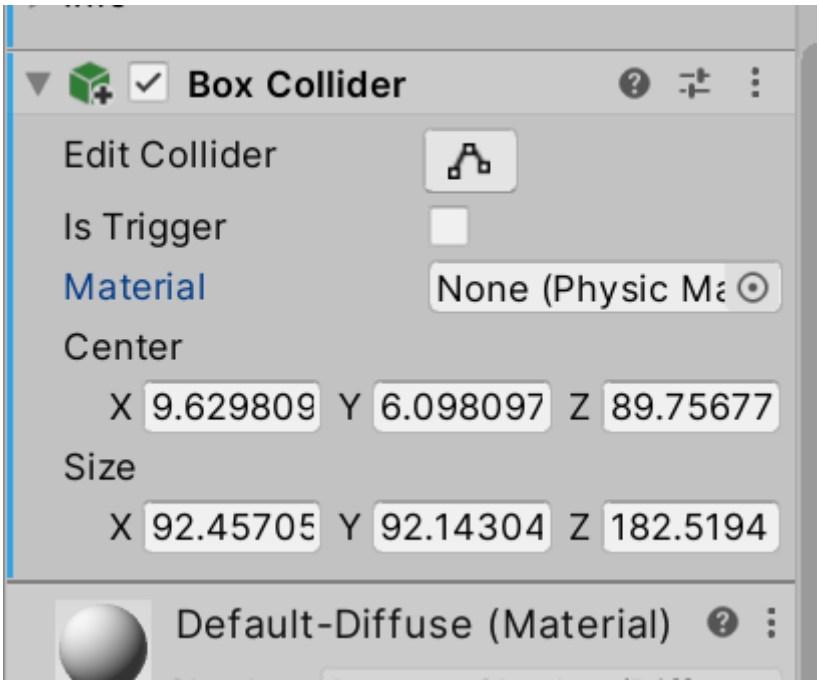
Step 1: Copy the file *astronaut.zip* to the *Assets* folder of your current project. You'll need *Winrar* to decompress the file. In this case your *astronaut.obj* file is available for use.

Step 2: You can now import the model into your scene by dragging the *astronaut* from the Project view to the Scene view.

Step 3: Set position (0,11,0), rotation to (-90,60,0), and scale to (0.05,0.05,0.05).



Step 4: Add *Rigidbody*
Step 5: Add a *Box Collider* component



Play the game

Note: If the astronaut falls off, you can play around with the position so that the astronaut does not fall off (e.g. reduce the Y value)

Procedure 2: Scripting with Unity 3D

Scripting is how the user defines a game's behaviour (or rules) in Unity. The recommended programming language for Unity is C# (C sharp). This tutorial will cover the fundamentals of scripting in Unity and also introduce key elements of the Application Programming Interface (API). You can think of the API as code that has already been written for you which lets you concentrate on your game design thus speeding up development time.

A good understanding of these basic principles is essential in order to harness the full power of Unity. This tutorial will introduce the fundamentals of scripting using C#. No prior knowledge of C# or Unity is required.
Time to complete: 2 hours.

Note: If you need help with some scripts check out the associated scripts in the course folders on NTULearn.

[Note: If Visual Studio or any editor is used to enter the script, check for error when the script is completed and save it before attaching to the GameObject.]

Point 1: Naming Conventions

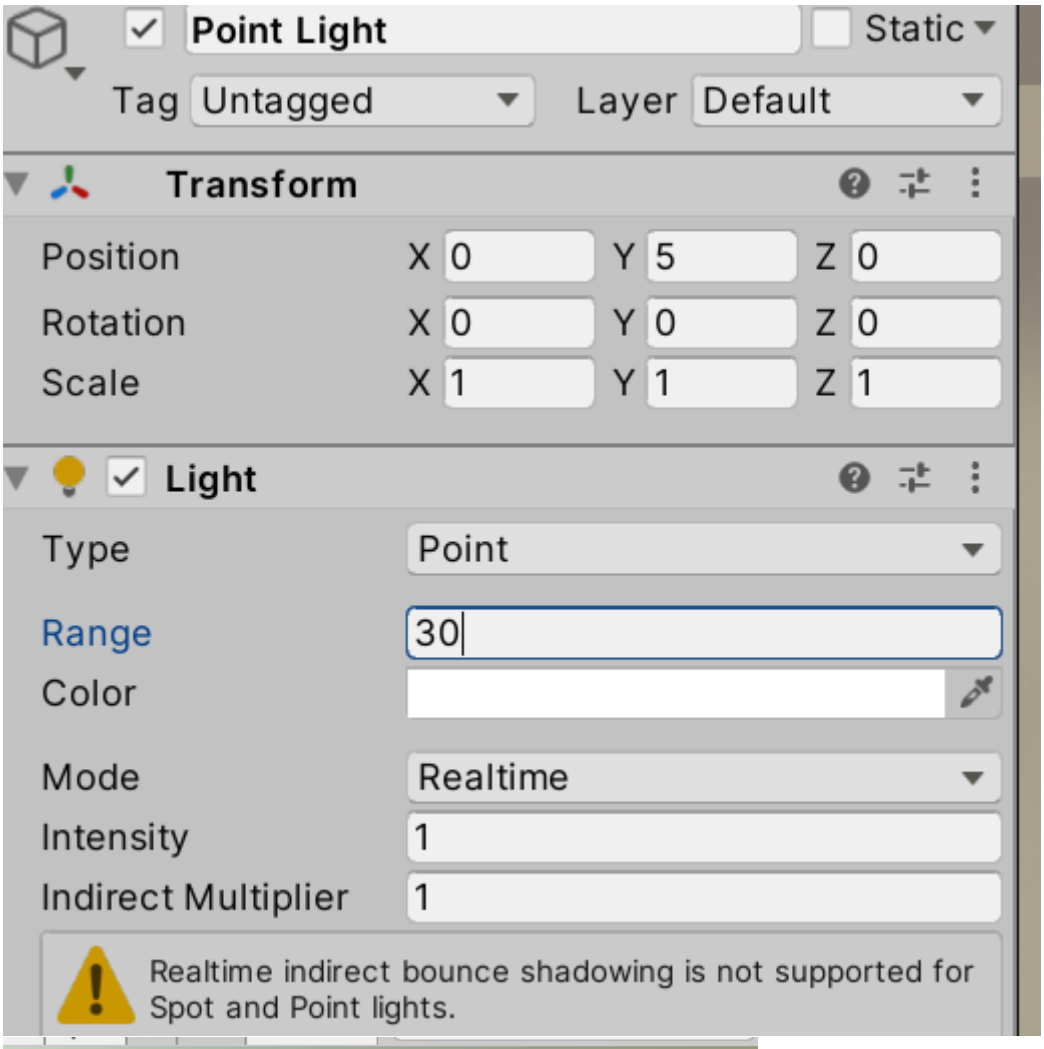
Before we begin, it is worth mentioning some conventions in Unity.
1. Variables: begin with a lowercase letter. Variables are used to store information about any aspects of a game's state.
2. Functions: begin with an uppercase letter. Functions are blocks of code which are written once and can then be reused as often as needed. Functions within a class are sometimes referred to as methods of that class.
3. Classes: begin with an uppercase letter. These can be thought of as collections of functions.
When reading example code or the Unity API, pay close attention to the first letter of words. This will help you better understand the relationship between objects.

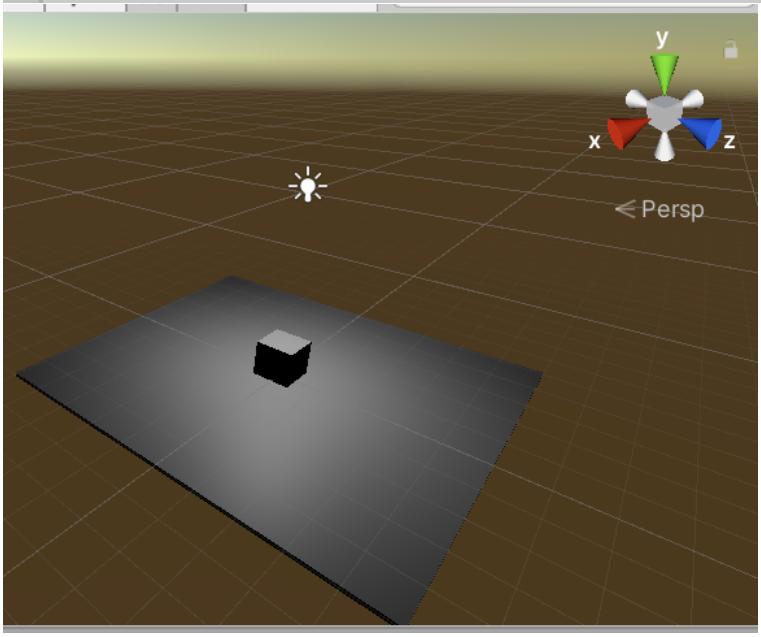
Point 2: Player Input

For our first program we're going to allow the user to move around in a simple game world.

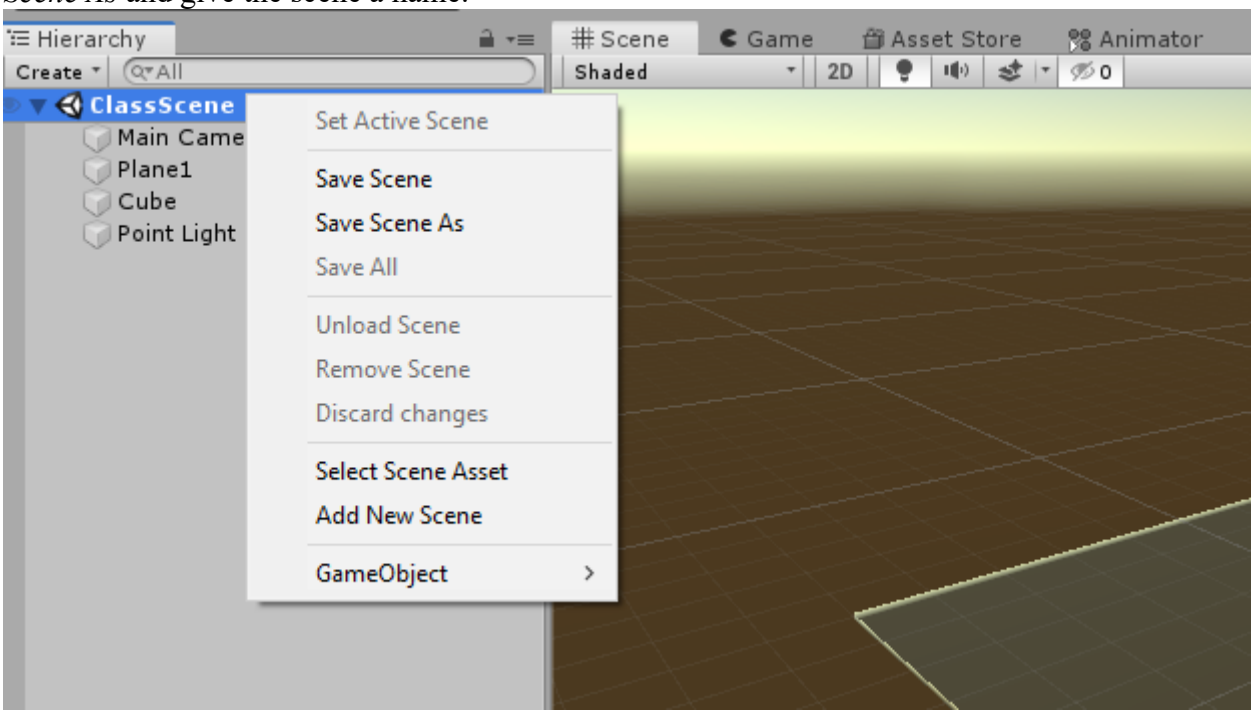
Setting the scene

Exercise 2.2a: Setting the scene
Step 1: Start Unity. Firstly, let's create a surface for the user to walk on. The surface we're going to use is a flattened cube shape.
Step 2: Create a cube with parameters: *Position (0,0,0), Rotation (0,0,0), Scale (10,0.1,10)*, it should now resemble a large flat plane. Rename this object *Plane* in the Hierarchy view.
Step 3: Create a second cube with parameters: *Position (0,0.5,0), Rotation (0,0,0), Scale (1,1,1)*. Rename the object to *Cube1*.
Step 4: Alter the *Main Camera* parameters: *Position (0,4,-20), Rotation (12,0,0), Scale (1,1,1)*.
Step 5: Delete the default Directional Light, and create a point light at *Position (0,5,0), Rotation (0,0,0), Scale (1,1,1)*; then change the range to **30**.


Point Light
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Light
Type Point
Range 30
Color
Mode Realtime
Intensity 1
Indirect Multiplier 1
Realtime indirect bounce shadowing is not supported for Spot and Point lights.


y
x
z
Persp

Step 6: Save the scene by right-clicking on the scene in hierarchy (see below) and selecting *Save Scene As* and give the scene a name.



Our first script

We're now ready to start game programming. We're going to allow the player to move around the game world by controlling the position of the main camera. To do this we're going to write a script which will read input from the keyboard, then we attach (associate) the script with the main camera (more on that in the next section). For our first program we're going to allow the user to move around in a simple game world.

Exercise 2.2b: Our first script

Step 1: Begin by creating an empty script. Select *Assets->Create->C# Script* and rename this script to *Move1* in the Project view.

Step 2: Double-click on the Move1 script and it will open with the *Update()* function already inserted (this is default behaviour), we're going to insert our code inside this function. Any code you insert inside the *Update()* function will be executed **every frame**.

In order to move a game object in Unity we need to alter the position property of its transform; the *Translate* function will let us do this. The *Translate* function takes 3 parameters for x, y and z axes. As we want to control the *Main Camera* game object with the cursor keys, we simply attach code to determine if the cursor keys are being pressed for the respective parameters:

```
void Update () {
        transform.Translate(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
}
```

The *Input.GetAxis()* function returns a value between -1 and 1, e.g. on the horizontal axis, the left cursor key maps to -1, the right cursor key maps to 1. (Note that the *transform* property refers to the Transform of the GameObject to which the script is attached; we can use it directly without declaring in C#.)

Notice the 0 parameter for the y-axis as we're not interested in moving the camera upwards. The

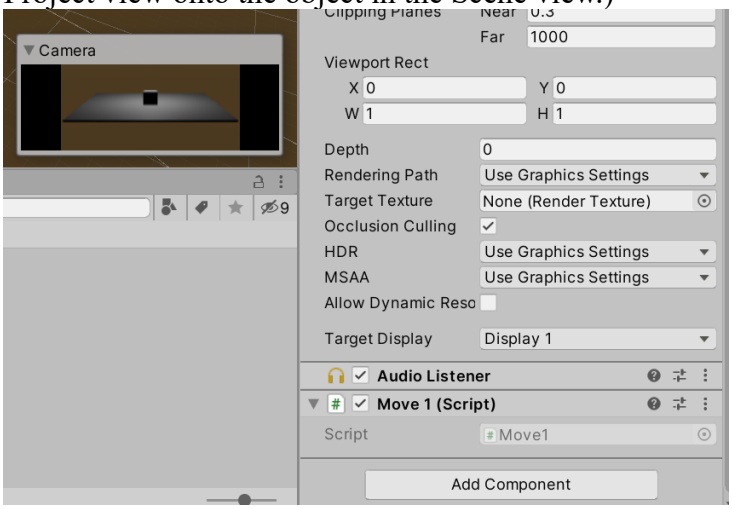
Horizontal and Vertical axis are pre-defined in the Input Settings, the names and keys mapped to them can be easily changed in *Edit->Project Settings->Input Manager*.

Step 3: Open the *Move1.cs* script and type in the code in the box above, paying close attention to wordcases.

Attaching the script

Now that our first script is written, how do we tell Unity which game object should have this behaviour? All we have to do is to attach the script to the game object which we want to exhibit this behaviour.
Note: According to the Unity Manual "Each time you attach a script component to a GameObject, it creates a **new instance** of the object defined by the blueprint". When you get an error such as "NullReferenceException: Object reference not set to an instance of an object" it may be that you're trying to access the blueprint (the class) rather than an instance of the class. Remember that a class simply tells us the properties of **potential** objects: for example, a class defining a human being could have properties such as numberOfEyes, Volume, lengthRightArm, Eat(), etc. But until we've created a human being, it does not make sense to ask "What is the lengthRightArm?"
Therefore, check that you have attached the script to a game object and properly initialized any properties that need to be.

Exercise 2.2c: Attaching the script
Step 1: To do this, first click on the game object that you wish to have the behaviour as defined in the script. In our case, this is the *Main Camera*, and you can select it from either the Hierarchy view or the Scene view.
Step 2: Next select *Component->Scripts->Move1* from the main menu. This attaches the script to the camera. You should notice that the *Move1* component now appears in the Inspector view for the main camera. (You can also assign a script to a game object by dragging the script from the Project view onto the object in the Scene view.)



Step 3: Run the game, you should be able to move the main camera with the cursor keys or W, S, A, D. Make sure you saved the script otherwise the behaviour may not be activated.

You probably noticed that the camera moved a little too fast, let's look at a better way to control the camera speed.

Delta time

As the previous code was inside the *Update()* function, the camera was moving at a velocity measured in meters per frame update. It is better however to ensure that your game objects move at the more predictable rate of meters per second. To achieve this we multiply the value returned from the *Input.GetAxis()* function by *Time.deltaTime* and also by the velocity we want to move per second.

Exercise 2.2d: Update the *Move1* script with the above code.

```
public float speed = 5f; //f is for float
void Update () {
        float x = Input.GetAxis("Horizontal") * Time.deltaTime * speed;
        float z = Input.GetAxis("Vertical") * Time.deltaTime * speed;
        transform.Translate(x, 0, z);
}
```

Notice here that the variable speed is declared outside of the function *Update()*, this is called an exposed variable, as this variable will appear in the Inspector view for whatever game object the script is attached to (the variable gets exposed to the Unity GUI). Exposing variables is useful when the value needs to be tweaked to get the desired effects; this is much easier than changing code.

Point 3: Connecting Variables

Connecting variables via the GUI is a very powerful feature of Unity. It allows variables which would normally be assigned in code to be done via drag and drop in the Unity GUI. This allows for quick and easy prototyping of ideas. As connecting variables is done via the Unity GUI, we know we always need to expose a variable in our script code so that we can assign the parameter in the Inspector view. We do this by using the "public" modifier

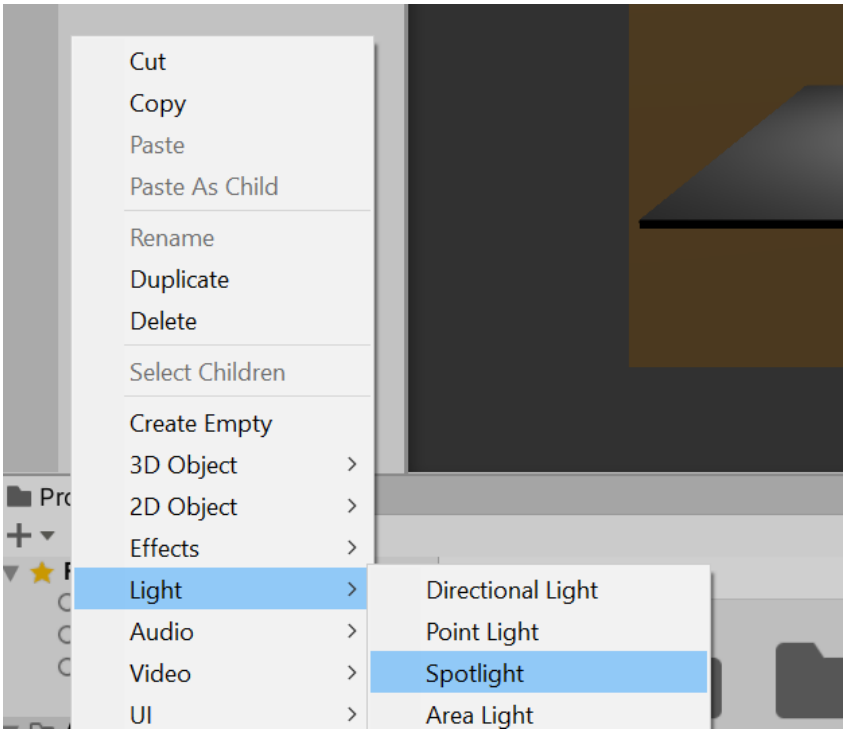
We'll demonstrate the connecting variables concept by creating a spotlight which will follow the player (Main Camera) around as they move.

We want **our new spotlight to look at wherever the main camera is**. As it happens, there's a built in function in Unity to do this, *transform.LookAt()*. If you were beginning to think "how do I do this?" and were already imagining a lot of code, then it's worth remembering to always check the Unity API for a function that already exists. We could also make a good guess at looking in the 'transform' section of the API as we're interested in altering the position or rotation of a game object.

Now we come to the connecting variables section, what do we use as a parameter for *LookAt()*? Well we could hardcode a game object, however we know we want to assign the variable via the GUI, so we'll just use an exposed variable (of type Transform).

Exercise 2.3: Connecting Variables
Step 1: Add a spotlight to the Scene with parameters: *Position (0,5,-1), Rotation (90,0,0), Scale*

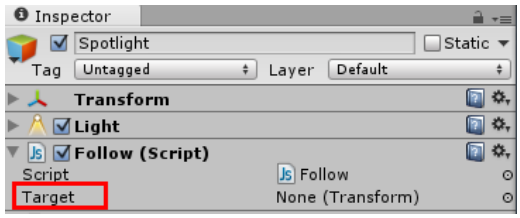
(1,1,1) and rename it to *spotlight*.



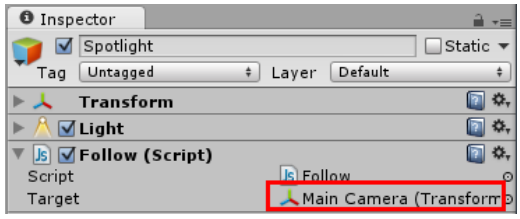
Step 2: Create a new C# script and name it *Follow.cs*. Type the code below into the script.

```
public Transform target
void Update () {
        transform.LookAt(target);
}
```

Step 3: Attach the script *Follow.cs* to the *spotlight* by first selecting *spotlight* in the Hierarchy view or the Scene view, and then dragging *Follow* from the Project view to *spotlight*'s Inspector view. Notice when the component gets added, the *target* variable is exposed.



Step 4: With the *spotlight* still selected, drag the *Main Camera* from the Hierarchy view onto the *target* variable in the Inspector view. This assigns Main Camera's position/rotation/scale (that is the *Transform* of the *Main Camera*) to the *target* variable. According to the code we composed in *Follow.cs*, the *spotlight* will now shine wherever *Main Camera* is. If we wanted the *spotlight* to follow a different game object we could just drag in a different object.



Step 5: Play the game. If you click on the Spot Light You may want to change the position of the spotlight to improve the effect.

Point 4: Accessing Components

As a game object can have multiple scripts (or other components) attached, it is often necessary to access other component's functions or variables. Unity allows this via the *GetComponent()* function.

We're now going to add another script to our spotlight which will make it look at *Cube1* whenever the jump button (spacebar by default) is pressed. Let's think about this first. What we want to do:

1. Detect when the jump button has been pressed.

2. When jump has been pressed make the spotlight look at *Cube1*. How do we do this? Well, the *Follow.cs* script contains a variable *target* whose value determines which game object the spotlight should look at. We need to set a new value for this parameter. We could hardcode the value for the cube (see *Point 5: Doing it with code* later), however exposing the variable and assigning this via the GUI is another way of doing this.

Exercise 2.4:
Step 1: Create a new C# script and name it *Switch*. Add the code below to *Switch.cs*.

```
public Transform switchToTarget;
void Update () {
        if (Input.GetButtonDown("Jump"))
                GetComponent<Follow>().target = switchToTarget;
}
```

Step 2: Add the *Switch.cs* script to the *spotlight* and assign *Cube1* to the *switchToTarget* parameter in the Inspector view.
Step 3: Run the game. Move around and verify that the spotlight follows you as usual. Then hit the spacebar and the spotlight should focus on the *Cube1*.

Point 5: Doing It with Code

Now we are going to assign the variables via code (as opposed to the Unity GUI). Remember this is only for comparison, assigning variables via the GUI is the recommended approach.

The problem we were interested in earlier was how to tell the spotlight to look at *Cube1* when the jump button was pressed. Our solution was to expose a variable in the *Switch* script which we could then assign a value to by dropping *Cube1* onto it from the Unity GUI. There are two main ways to do this in code:

1. Use the name of the game object.

A game object's name can be seen in the Hierarchy view. To use this name with code we use it as

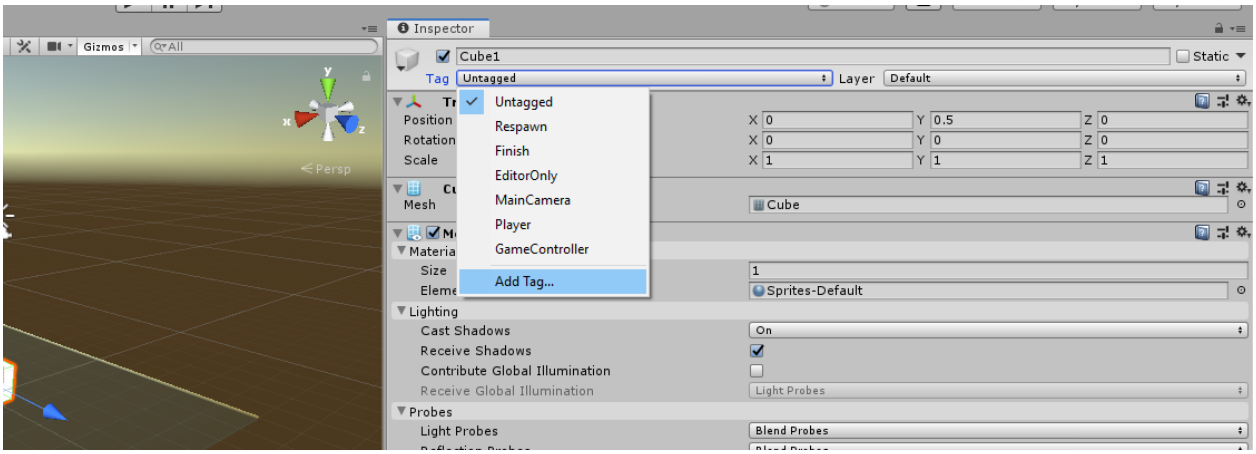
a parameter in the *GameObject.Find()* function. So if we want the jump button to switch the spotlight from *Main Camera* to *Cube1*, the code is as follows (make the changes in *Switch.cs*):

```
void Update () {
        if (Input.GetButtonDown("Jump"))
        {
                Transform newTarget = GameObject.Find("Cube1").transform;
                GetComponent<Follow>().target = newTarget;
                //the component is the Follow.cs script. We are accessing the variable “target”
                //and changing its value to “new target”
        }
}
```

Notice how no variable is exposed as we name it directly in code. Check the API for more options using *Find()*.

2. Use the tag of the game object.

A game object’s tag is a string which can be used to identify a component. To see the built-in tags click on the Tag button in the Inspector view. You can also create your own. Go to *Cube1*’s Inspector view: *Tag->Add Tag*. Directly edit the tag list as in the image. Then choose the newly added tag *Cube* to replace *Untagged*.



To edit the tag name click the *plus* symbol.

Give it the name “Cube”. You can give it any other name, but remember you’ll have to change the

input to GameObject.FindWithTag to match the name you give here.

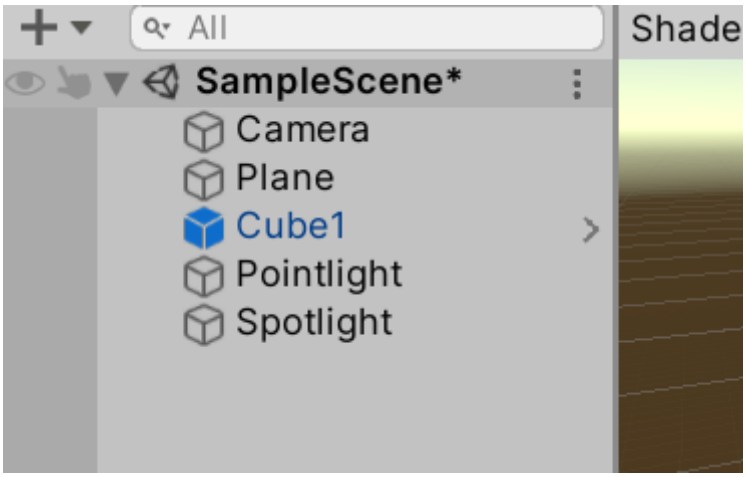
The function for finding a component with a specific tag is *GameObject.FindWithTag()* and takes a string as a parameter. Our complete code to do this is:

```
void Update () {
        if (Input.GetButtonDown("Jump"))
        {
                Transform newTarget = GameObject.FindWithTag("Cube").transform;
                GetComponent<Follow>().target = newTarget;
        }
}
```

Point 6: Instantiate

It is often desirable to create objects during run-time (as the game is being played). To do this, we use the Instantiate function.
Let's show how this works by instantiating (spawning) a new game object every time the user presses the fire button (either the left mouse button or left Ctrl on the keyboard by default).
So what do we want to do? We want the user to move around as usual, and when they hit the fire button, instantiate a new object. A few things to think about:
1. Which object do we instantiate?
2. Where do we instantiate it?
Regarding which object to instantiate, the best way of solving this is to expose a variable. This means we can state which object to instantiate by using drag and drop to assign a game object to this variable. As for where to instantiate it, for now we'll just create the new game object at the previous game object's position.
The Instantiate function takes three parameters: (1) the object we want to create, (2) the 3D position of the object and (3) the orientation of the object.

Exercise 2.6: Create prefab and instantiate it
When an object is instantiated, it is usual for that object to be a prefab. We'll now turn the *Cube1* game object into a prefab.
Step 1: First of all, let's add a *Rigidbody* component to *Cube1* to resemble a cube in a physical world. In *Cube1*'s Inspector view, *Add Component->Physics->Rigidbody*.
Step 2: drag the *Cube1* game object from the Hierarchy view into the Project browser and rename it to *Cube*. If you have a *Prefabs* folder, drag directly into that folder. Notice the icon for *Cube1* (an instance of the prefab *Cube)* in Hierarchy view changes.

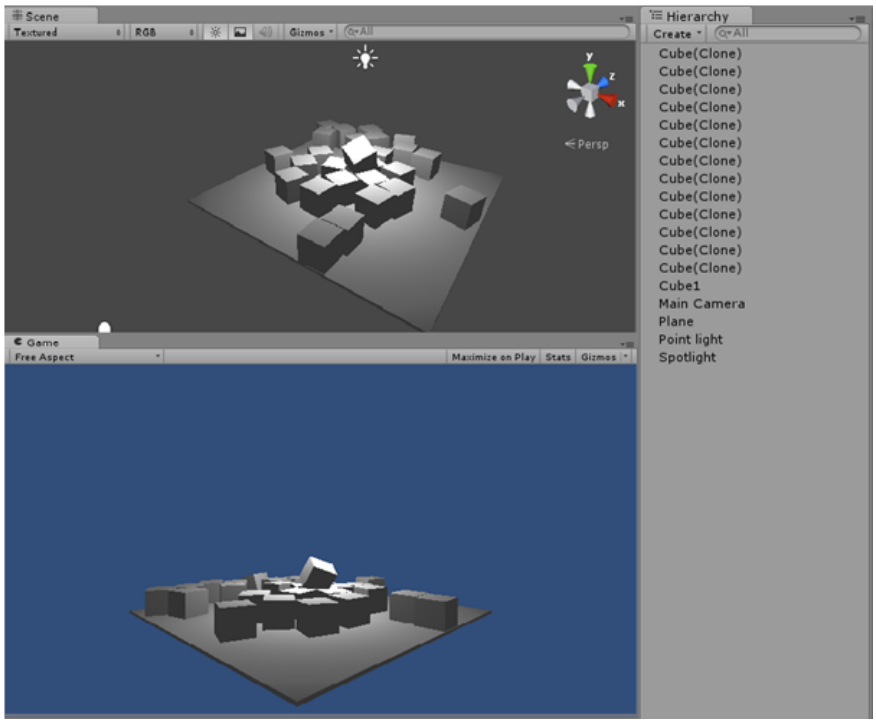


Step 4: Create a new C# script and name it *Create*. Insert the code below.

```
public GameObject newObject
public Transform refPos
void Update () {
    if (Input.GetButtonDown("Fire1")) {
        Instantiate(newObject, refPos.position, refPos.rotation);
    }
}
```

refPos.position and *refPos.rotation* are the position and rotation of the spawned object that we are going to assign a value to via GUI.

Step 4: Attach this script to the *Main Camera*. In *Main Camera*'s Inspector view, assign the *Cube* prefab to the *newObject* variable, and assign the *Cube1* object to the *refPos* variable.
Step 5: Play the game and move around as usual. Each time the fire button is clicked (LMB or left Ctrl), you should notice a new *Cube1* clone (the prefab assigned to *newObject*) appearing at the position of *Cube1* (the object assigned to *refPos*).



Point 7: Common script types

Whenever a new C# script is created, by default it contains an *Update()* function. This section will discuss other common options available, simply replace the name of the *Update()* function with one from the list below.

FixedUpdate(): Code placed inside this function is executed at regular intervals (a fixed framerate). It is common to use this function type when applying forces to a Rigidbody.

Awake(): Code inside here is called when the script is initialized.

Start(): This is called before any *Update()* function, but after *Awake()*. The difference between the *Start ()* and *Awake()* functions is that the *Start()* function is only called if the script is enabled (if its checkbox is enabled in the Inspector view).

OnCollisionEnter(): Code inside here is executed when the game object the script belongs to collides with another game object.

OnMouseDown(): Code inside here is executed when the user moves the mouse over a game object which contains a GUIElement or a Collider and performs a click.

```
// Loads the level named "SomeScene" as a response
// to the user clicking on the object
Using UnityEngine.SceneManagement
void OnMouseDown () {
        SceneManager.LoadScene ("SomeScene");
}
```

OnMouseOver(): Code inside here is executed when the mouse hovers over a game object which contains a GUIElement or a Collider.

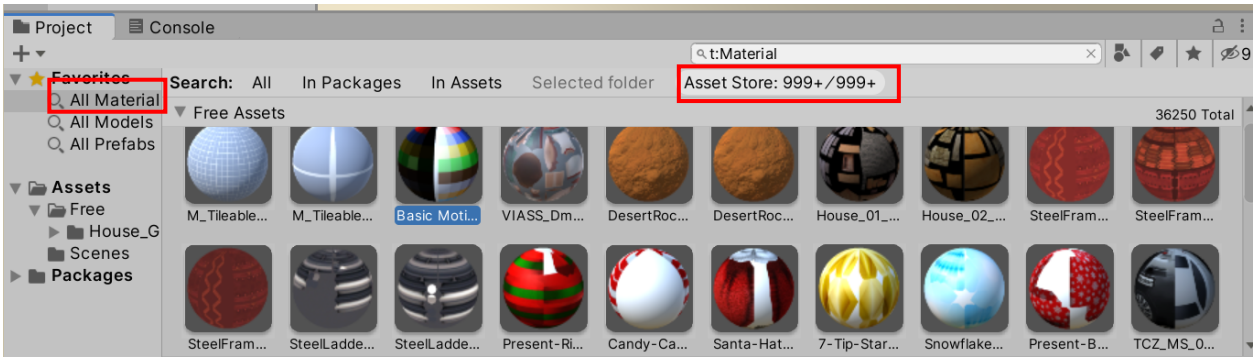
```
// Change color of the material to red (or other colorVal)
// while the mouse is over the mesh
void OnMouseOver () {
        GetComponent<Renderer>().material.SetColor("_Color", new Color(colorVal,0,0);
}
```

Procedure 3: Multiplayer Local

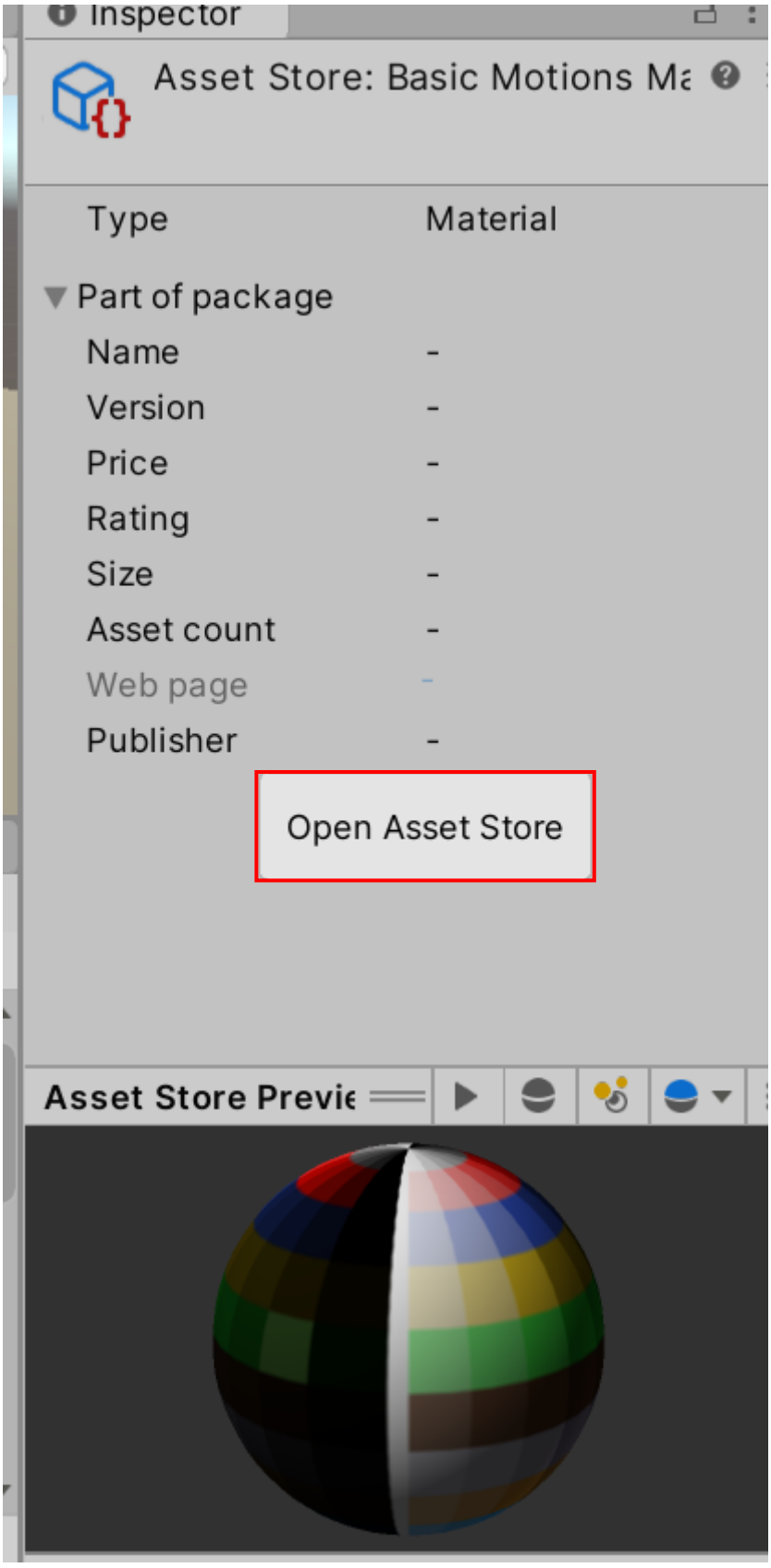
You can have two or more players play the game on the same computer. This does not require networking between the players. Rather, player A is assigned one set of keys while player B is assigned another set. In this tutorial you will allow two players to play at once.
Time to complete: 3 hours.

Point 1: Adding material to objects

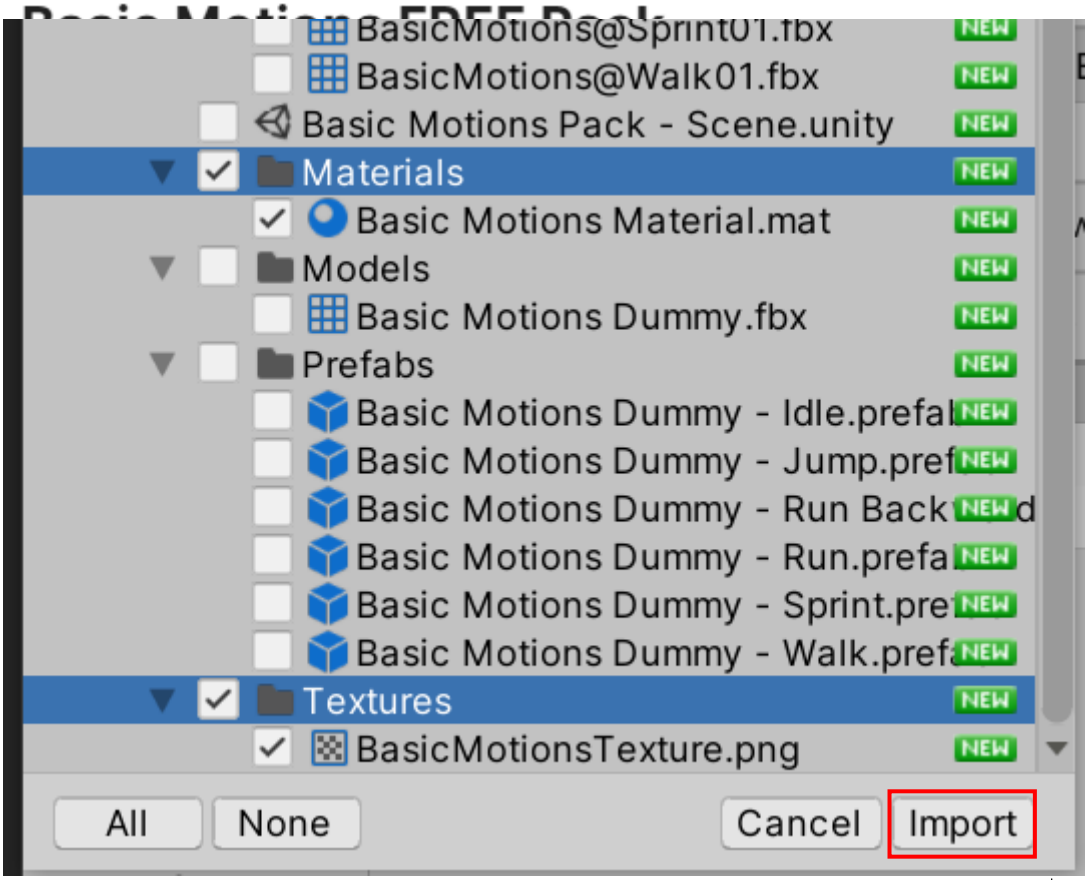
When we have many players we need a way to tell them apart. One way to do this is to assign their avatar (their representation in the game) a different sign. For our case the ball is the representation of the player. So whenever we have more than one ball we want each ball to be different.
Exercise 3.1: Creating a new scene with game objects of different materials.
Step 1: Create a new scene and call it *MultiplayerWorld.* In the hierarchy, right-click on current scene. Select *Add New Scene.* Right-click on the newly-created scene (look for Untitled as the name) and select *Save Scene As.* Open the Scenes folder by double-clicking. Type "*MultiplayerWorld*" as the file name and press Enter key or **Save** button
Step 2: Create a cube with parameters: *Position (0,0,0), Rotation (0,0,0), Scale (10,0.1,10)*, it should now resemble a large flat plane. Rename this object *Plane* in the Hierarchy view.
Step 3: Create a new sphere with parameters: *Position (0,0.5,0), Rotation (0,0,0), Scale (1,1,1)*. Rename the object to *Ball*.
Step 4: Download new materials for the ball. Under Favorites go *All Materials-> Asset Store.* Access the Free Assets. (You will need to log into your account to import packages). Choose something with stripes so you can see the rotation/spin. We give the ball this rotation behaviour in the script.



Then click click in *Open Asset Store*

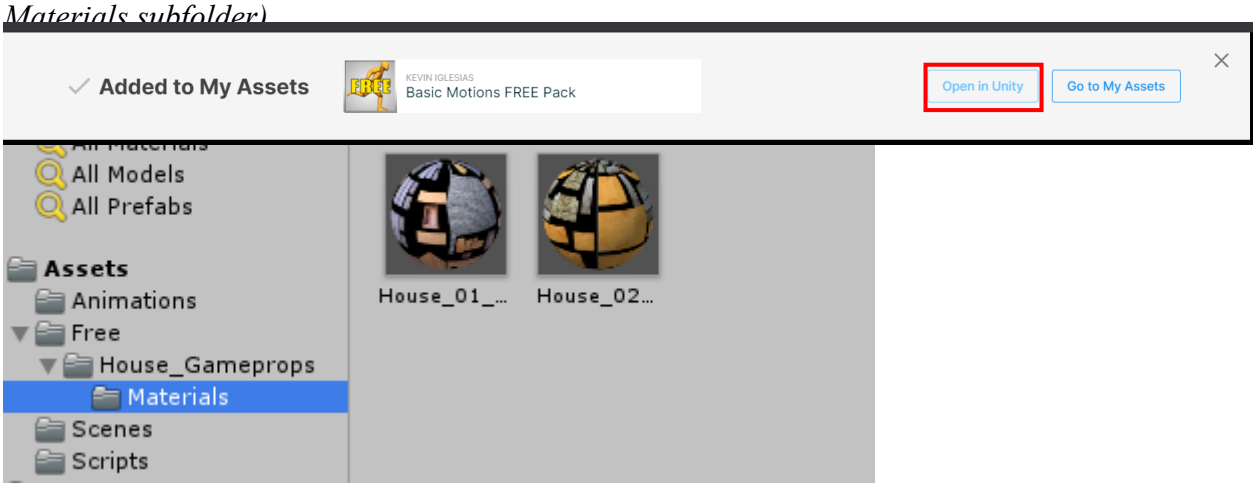

Inspector
Asset Store: Basic Motions Ma
Type
Material
Part of package
Name
-
Version
-
Price
-
Rating
-
Size
-
Asset count
-
Web page
-
Publisher
-
Open Asset Store
Asset Store Previe

Then *Add to My Assets*



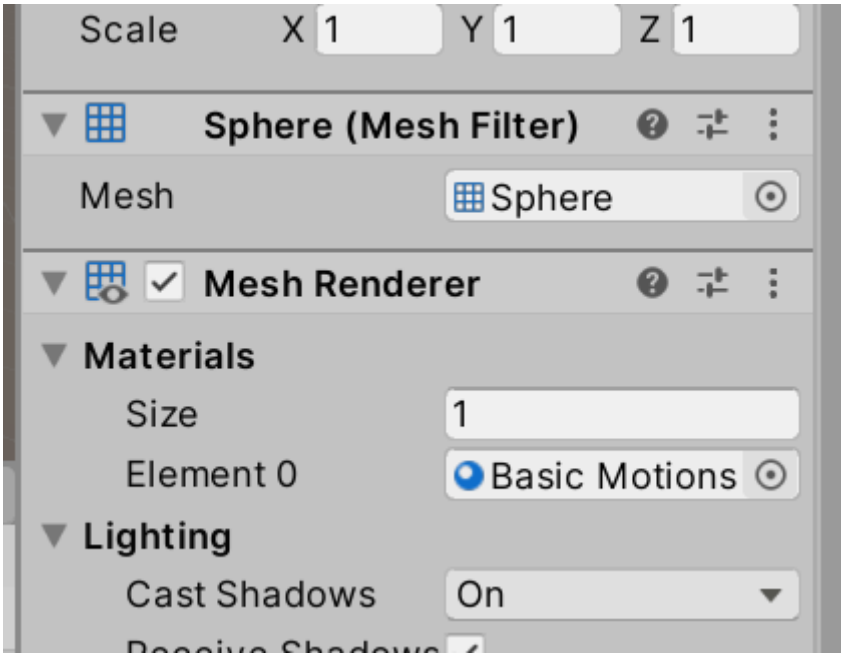
extures options,

The material will now appear in your subfolders for example in Assets->Free->(Find the *Materials subfolder)*

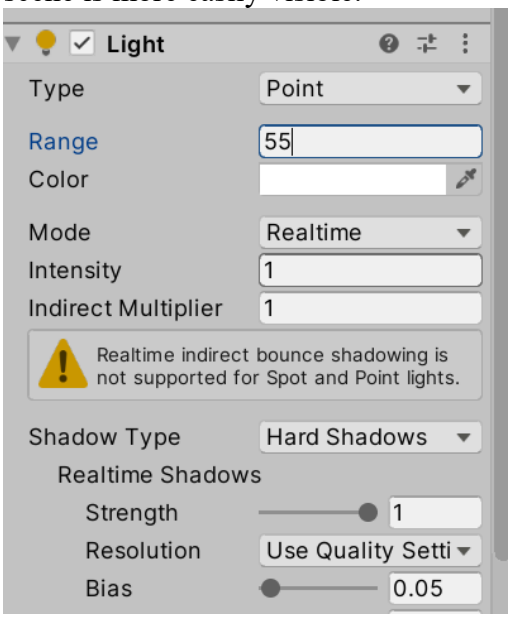


Download ***two*** material assets of your liking.

Finally, you can drag the new material onto the *Materials->Element 0* field of the Mesh Renderer component attached to the ball

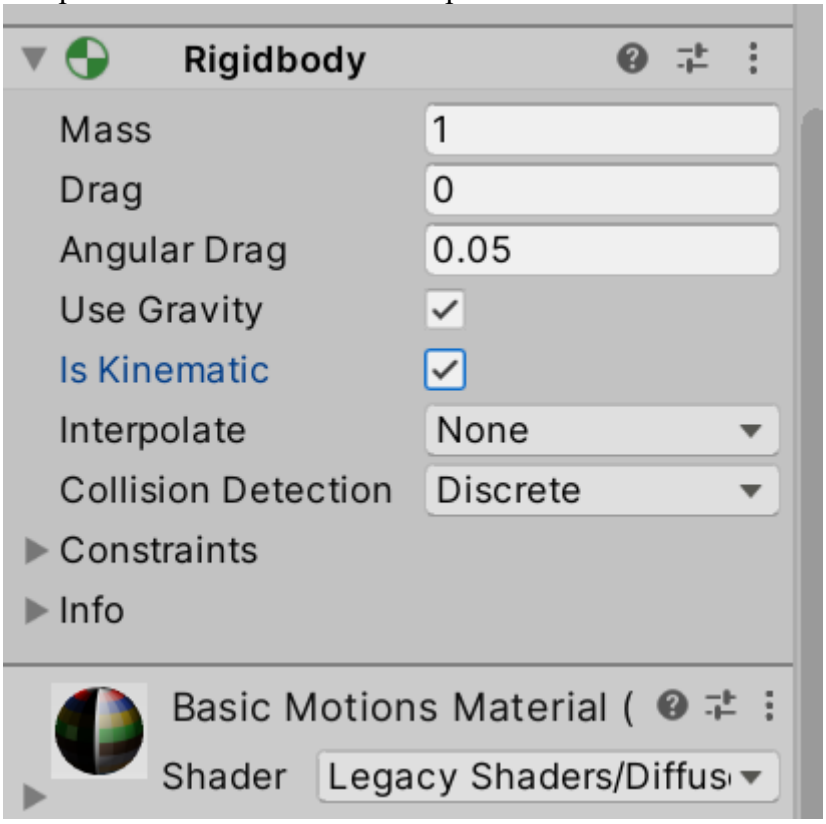


Step 5: Alter the *Main Camera* parameters: *Position (-8,3,-8), Rotation (0,45,0), Scale (1,1,1).*
Step 6: Delete the default Directional Light, and create a point light at *Position (0,5,0), Rotation (0,0,0), Scale (1,1,1).* Change the *Range* to 55 , and Shadow Type to "Hard Shadows" so that the scene is more easily visible.



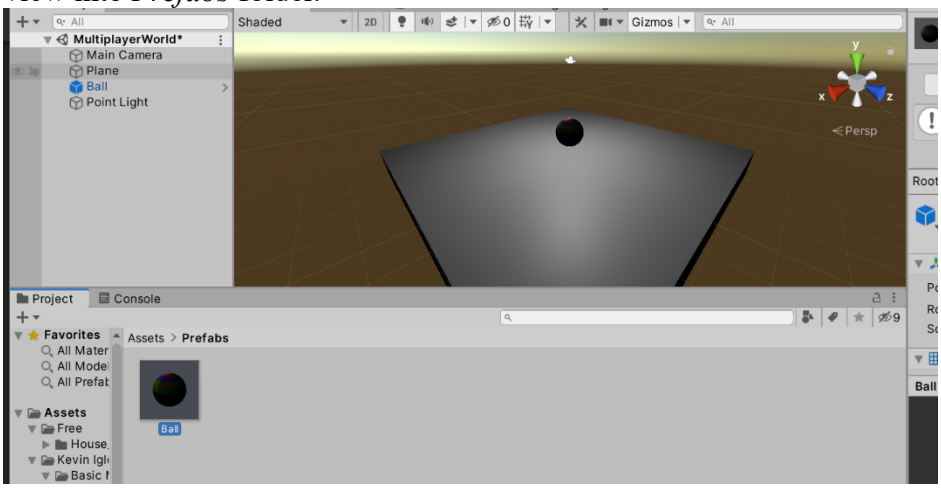
Step 7: Add a *Rigidbody* component to *Ball*. Go to the Inspector view of *Ball*, *Add Component* and then type start typing *"Rigidbody"*. Select *Rigidbody.* Be sure ***not*** to select *Rigidbody 2D*.

Check (activate) the *Is Kinematic* box: this means that the gameObject to which this Rigidbody component is attached will **not** respond to forces.



Step 8: As per Exercise 2.6, create **and attach a new script called "*Ball.cs*"** which will define the behaviour of the ball. Copy and paste the code from the Resources folder of the class.

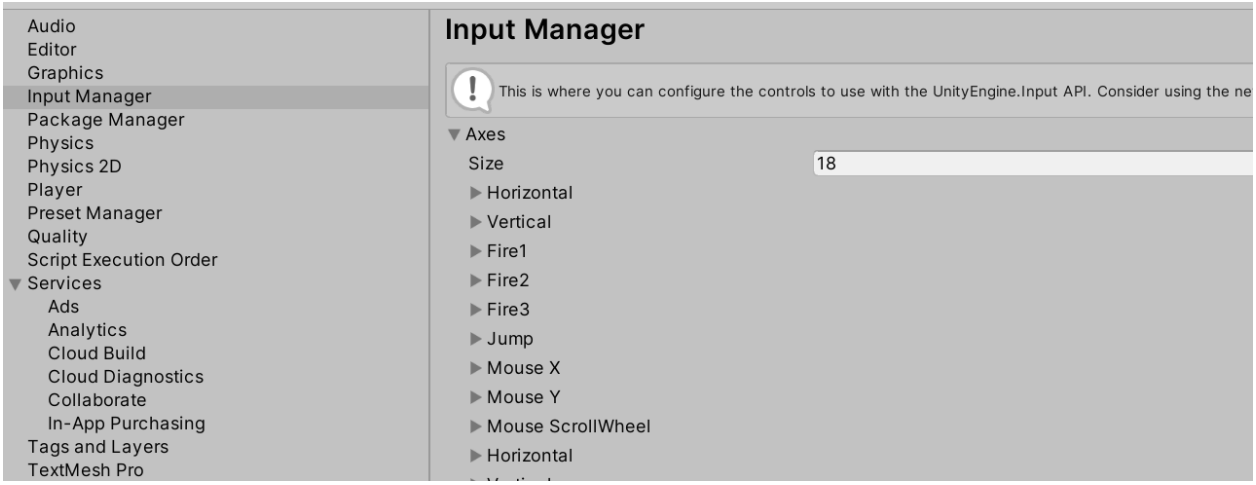
Step 9: Create a folder under Assets. Call this new folder *Prefabs*. Drag *Ball* from the hierarchy view into *Prefabs* folder.



Point 2: Adding new player Input

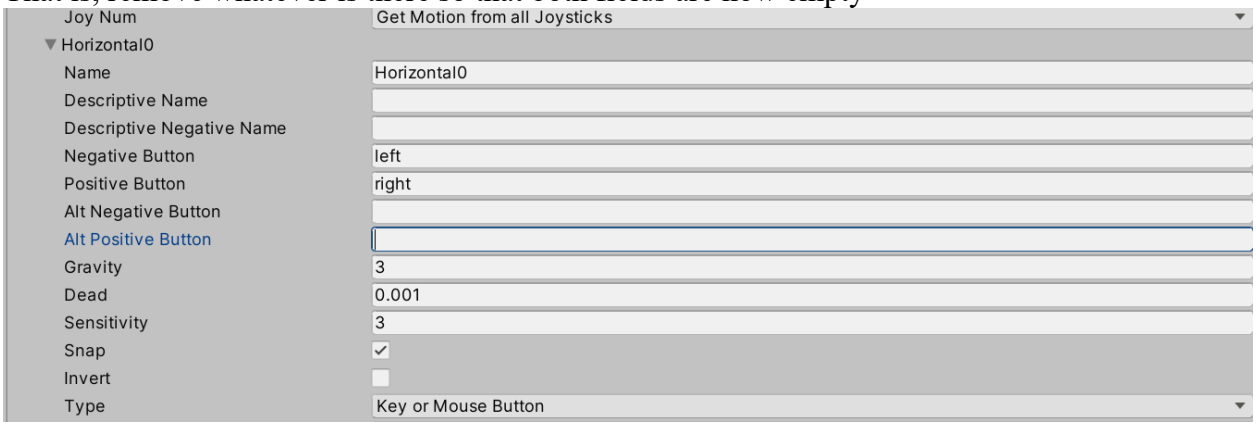
Whenever a new player joins we want to assign this new player their own set of keys to express their behaviour in the game world.
Exercise 3.2: Creating new input settings

Go to *Edit->Project Settings->Input Manager->Axes.* Click *Axes* if the fields are not shown.



Step 1a: Click on *Horizontal* and in the name field type "*Horizontal0*". The last character is the number zero.

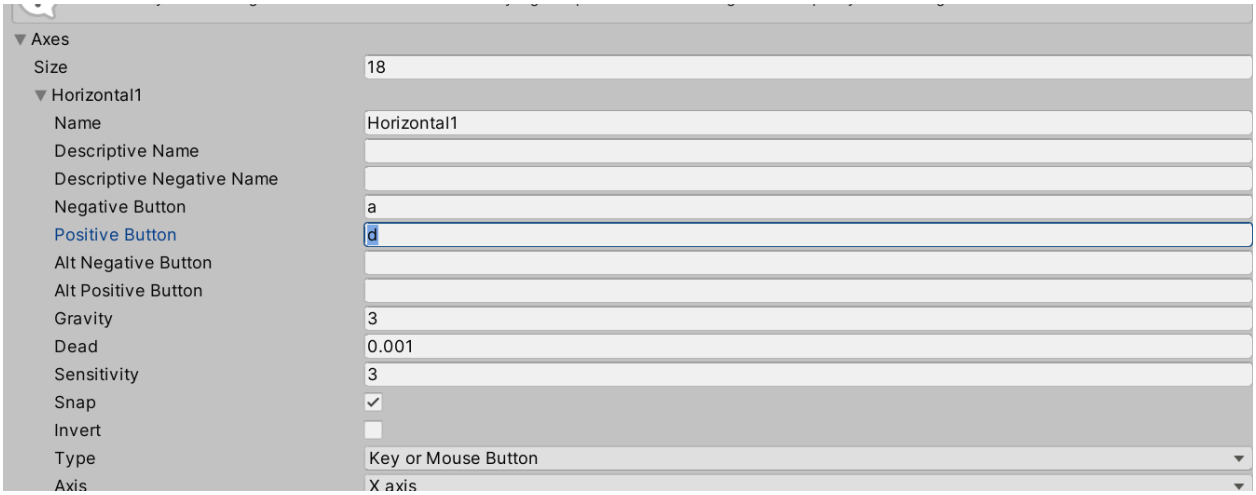
Step 2a: Click on *Horizontal0* and clear the *Alt Negative Button* and *Alt Positive Button* fields. That is, remove whatever is there so that both fields are now empty



Step 3a: Righ-click on *Horizontal0* and select *Duplicate Array Element*

Step 4a: Click on the new *Horizontal0* and in the name field type "*Horizontal1*". The last character is the number one.

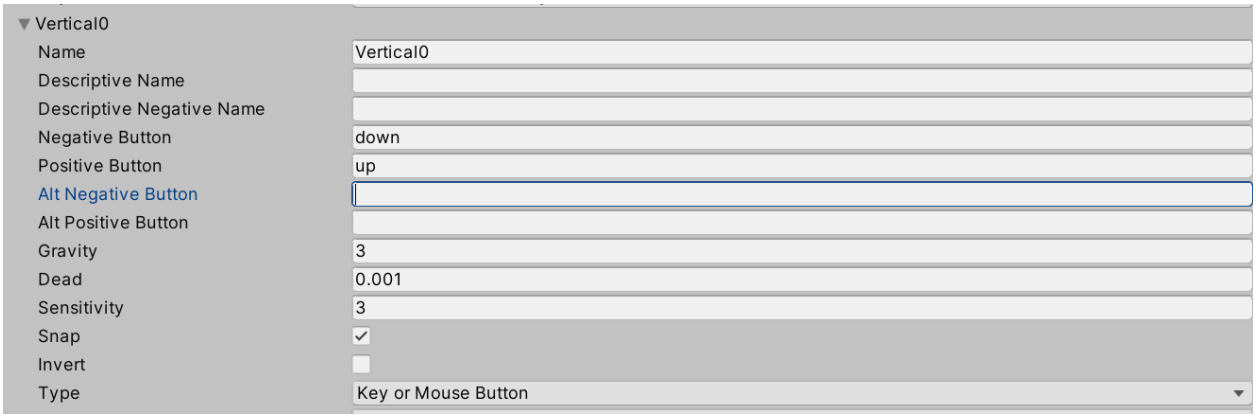
Step 5a: In the *Negative Button* field type the letter *a* and in the *Positive Button* field type the letter *d*



We now do the same for the vertical axis

Step 1b: Click on *Vertical* and in the name field type "*Vertical0*". The last character is the number zero.

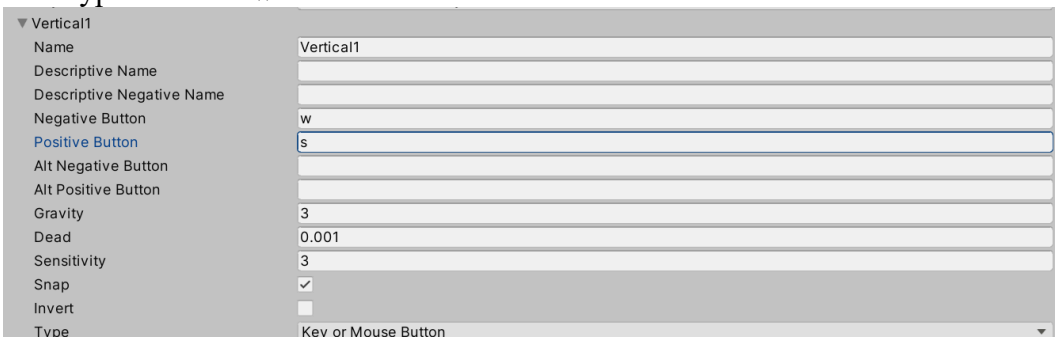
Step 2b: Click on *Vertical*0 and clear the *Alt Negative Button* and *Alt Positive Button* fields. That is, remove whatever is there so that both fields are now empty



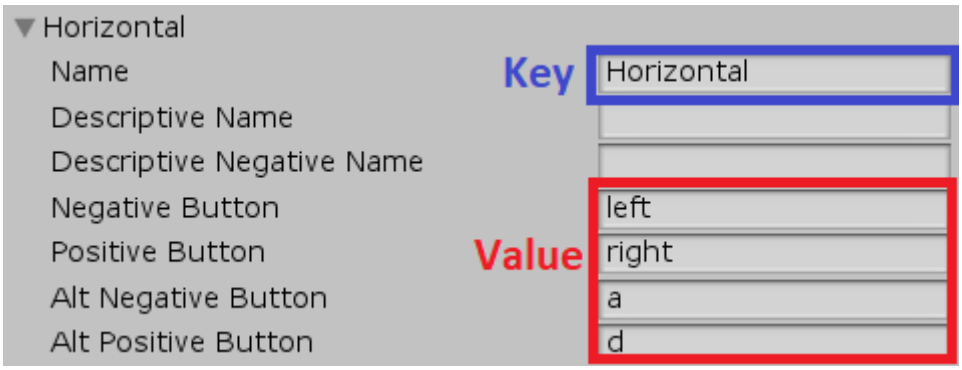
Step 3b: Righ-click on *Vertical0* and select *Duplicate Array Element*

Step 4b: Click on the new *Vertical0* and in the name field type "*Vertical1*". The last character is the number one.

Step 5b: Click on *Vertical1*. In the *Negative Button* field type the letter *s* and in the *Positive Button* field type the letter *w*



Note that the input is stored as dictionary with Name (e.g. Horizontal) as the Key and Value is a float value between -1 and 1 depending on input



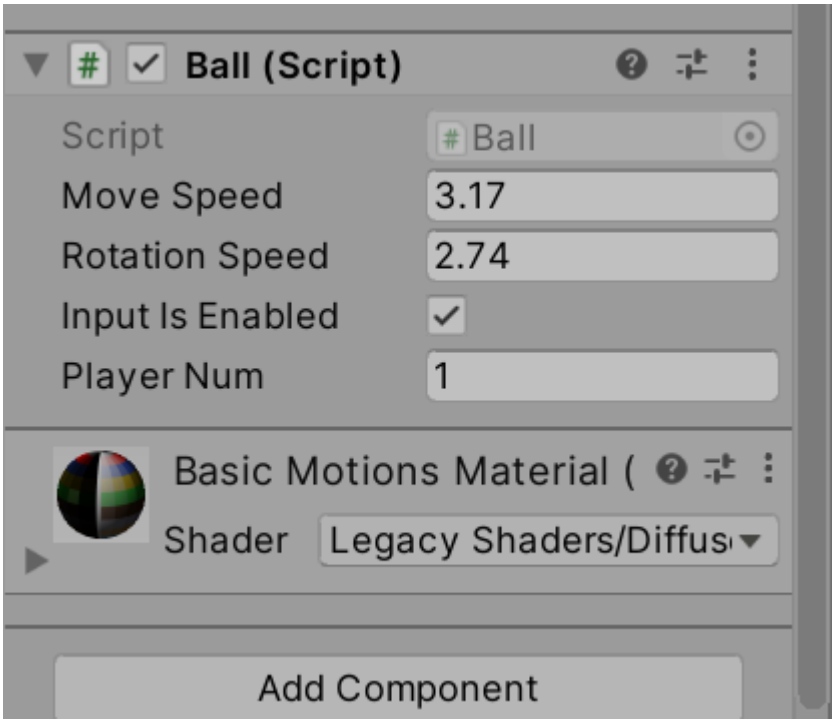
Exercise 3.3: Modifying the *Ball.cs* script to account for multiple balls in the scene. Make these changes:

a. Initialize the _moveSpeed to 5.0f
b. Initialize the _rotationSpeed to 30.0f
c. Add a new variable public int _playerNum. We want each player to be assigned a unique number so that we assign that player unique set of keys (in this case, one of the two sets)
d. Change the MovementInput method:

```
public int _playerNum;

protected void MovementInput()
{
    mVerticalInputValue = Input.GetAxis(mVerticalAxisInputName + _playerNum );
    mHorizontalInputValue = Input.GetAxis(mHorizontalAxisInputName + _playerNum);
}
```

To test if the logic works, play the game. Then whilst in Game play mode, go to Ball in the hierarchy and in its inspector view find “Player Num” field under the “Ball” script component. Check what value is there. If it is zero then moving using the arrow keys on the keyboard should work. Change Player Num to 1 (the number one). Then you should be able to get the same behaviour as the arrow keys but now using the keys a,d (horizontal) and w,s (vertical). Note that any other number apart from zero and one will not move the ball because we did not set any such inputs in Input settings. You may add more players, but for each player assign a new set of keys in the manner explained.

Note: If the ball is not moving or rotating increase the _moveSpeed and _rotationSpeed.



Point 2: Managers

We may at times need to distinguish between game objects when there can be many of them. The game objects can be distinguished by how they behave as well as how they look in the game world. Specially written scripts can be defined to handle these and other aspects of the game world that may affect more than one object at once.

GameManager.cs
The game manager manages many aspects of the game during play. Its tasks may include keeping track of the score for each player, killing objects (that have lost for example), tracking game states, and so on. A sample script GameManager.cs has been provided.
BallManager.cs
One could also script a manager that manages just specific objects in the game world. A simple sample script has been provided (BallManager.cs) that manages the balls in the game world.

Currently we have provided these sample Managers. You may build upon these managers to allow for more complex logic.

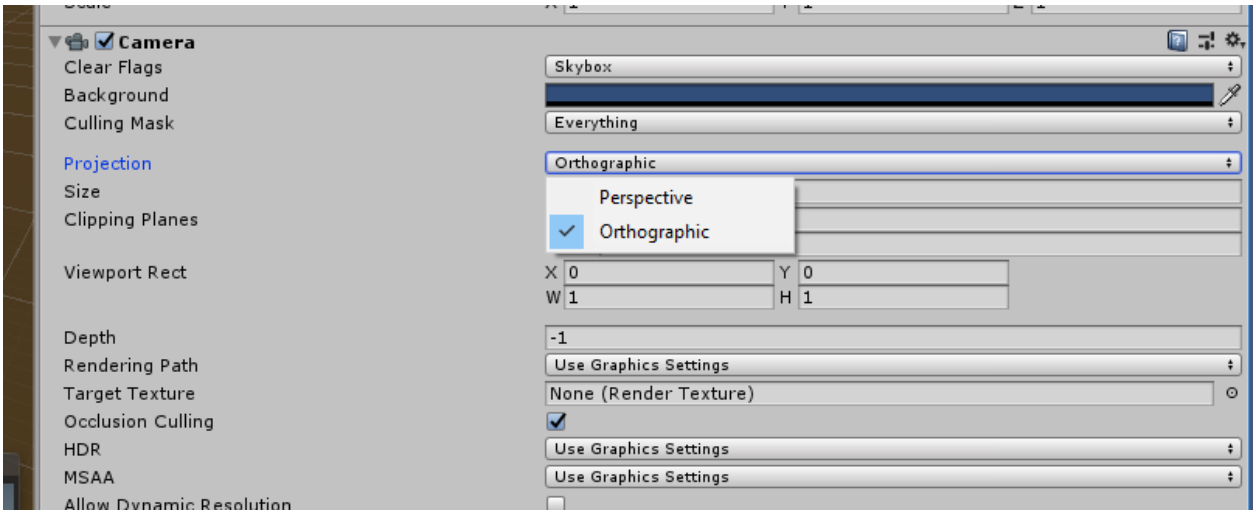
Procedure 4: Follow Cam

We now want to follow a specific object in the game world. In fact, when there are more than one object of interest that we want to see at the same time, we want to set up the camera to show us both. In this section you'll set up the camera to always follow the balls. In the case of two or more balls the camera will with resize to show all the balls of interest.

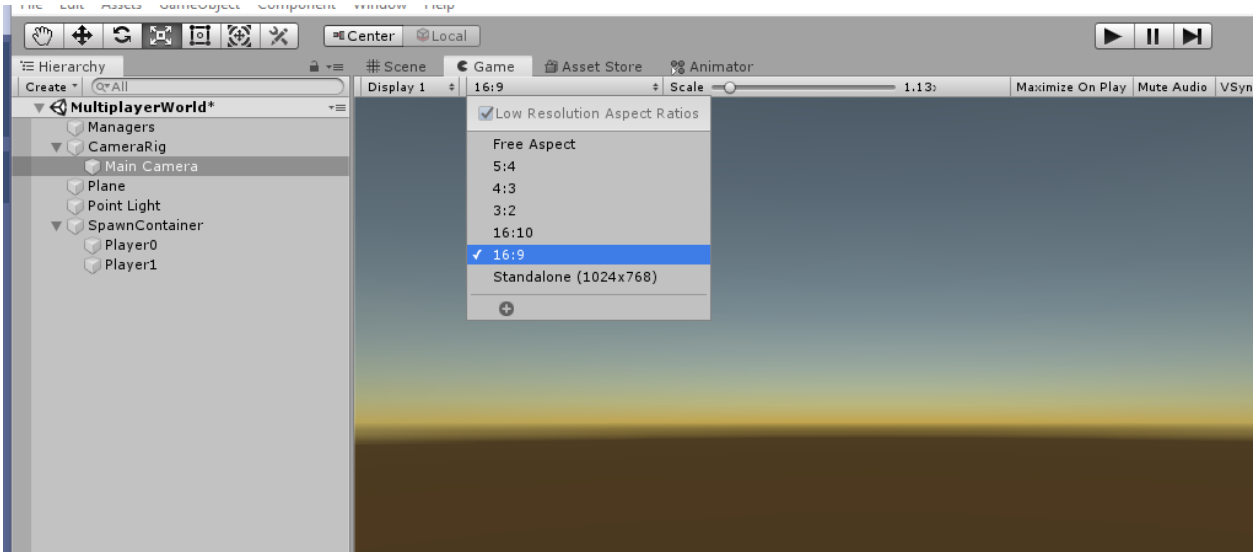
To reduce the amount of text, we're now going to use shortcut instruction,assuming that you are familiar with where to do the following tasks in Unity.

Exercise 4.1: Make the Camera Follow an Object

In the Inspector View, change the Main Camera's Camera component's Projection to **Orthographic**



Change **aspect ratio to 16/9**



1. Create a new Empty object by right-clicking in the hierarchy and selecting appropriately
2. Rename the object to CameraRig
3. Drop the Main Camera into CameraRig
4. Set rotation of CameraRig to (40,60,0)
5. Set positon of camera to (0,0,-65)
6. Create a new script called *CameraRig.cs* and attach this script to CameraRig GameObject
7. Add the following properties/variables to the script
    a. Camera mCamera which is a reference to Main Camera
    b. *float _dampTime*—approximate time for camera to focus smoothly (0.2f)
    c. *float _screenEdgeBuffer*—Space between the target at the edge of the screen (4f)
    d. *float _minSize* –Minimum zoom size (6.5f)
    e. *float mZoomSpeed* – Reference to speed for the smooth damping of the orthographic size
    f. *Vector3 mMoveVelocity* – Reference velcoty for the smooth damping of the position
    g. *Vector3 mAveragePos* – The average position between all balls
    h. *Transform[] _ballsTrans* – List of transforms of all the balls
8. Find the average position: Total position = the position of all balls and average position = total position/number of tanks.

Smooth transition from the current position to average position using the speed of _moveVelocity in _dampTime. We don't want to suddenly jump from currentPosition to averagePosition.

The Zoom() method of *CameraRig.cs* has been provided, and does the following:

-Zoom in and out to make sure all balls will be visible

So, the Zoom() method finds the *required size* (to make sure all balls are visible) and then it does a smooth transition between *current size* to *required size* using speed of *_zoomSpeed* in *_dampTime*

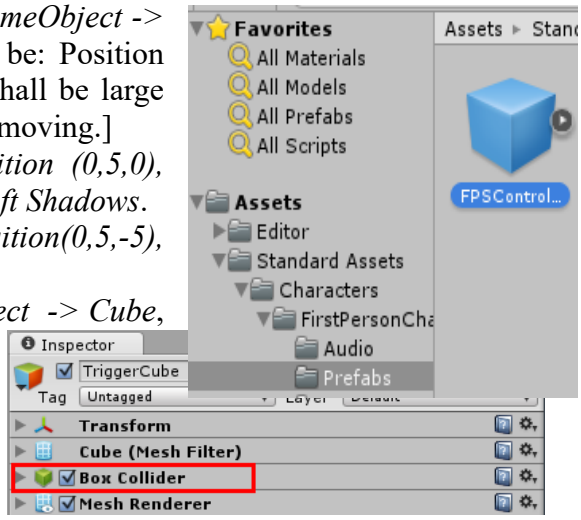
Procedure 5: Programming Trigger

Content of this section is with reference to "Making Trigger Zones in Unity" by Christian Barrett.
Time to complete: 30 min.

Triggers are a powerful tool in developing Unity games that every developer should know how to use. Our goal here is to show how to set up an object to be a trigger and detect if it's been triggered from script.

We're going to start by making a cube for a cube-shaped trigger zone. In actuality, anything that has a collider or can get a collider can be a trigger zone, including custom meshes with custom mesh colliders.

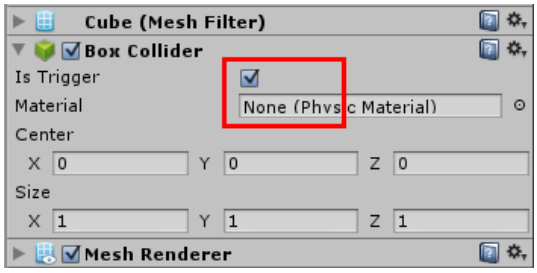
Exercise 5.1a: Prepare scene for the trigger
Step 1: Create a new scene and save it as *TestTrigger*. Import standard asset *Characters*: *Assets->Import Package->Characters*. Tick all the boxes in the popped out window to import all the items in the package, then press *Import*.
Step 2: Create a new Plane and name it *Ground*: *GameObject -> 3D Object ->Plane*. Your Plane parameters should be: Position (0,0,0), Rotation (0,0,0), Scale (6,1,6). [The plane shall be large enough to prevent the capsule fall off the edge while moving.]
Step 3: Adjust Directional Light parameters: *Position (0,5,0), Rotation (10,0,0), Scale (1,1,1). Shadows->Type->Soft Shadows*.
Step 4: Adjust the Main Camera parameters to: *Position(0,5,-5), Rotation(30,0,0), Scale(1,1,1).*
Step 5: Create a new Cube: *GameObject -> 3D Object -> Cube*, and rename it *TriggerCube*, with parameters: *Position (0,2,0), Rotation (0,0,0), Scale (4,4,4).*
Step 6: Add a collider to *TriggerCube* if it is not already a component of cube in the Inspector view: *Add Component->physics->Box Collider*.



Step 7: Create a FPSController. Go to Project view, Select *Assets->Standard Assets-> Characters -> FirstPersonCharacter->Prefabs->FPSController*, drag it into the scene, and rename it *CapsuleFPC*, with parameters: *Position (5,1,0), Rotation (0,0,0), Scale (1,1,1).* [This will put the capsule outside the TriggerCube.]

So we have our *CapsuleFPC* to move around, and *TriggerCube* to be set as trigger when *CapsuleFPC* enters its volume. Now we want our *TriggerCube* to detect such entrance.

Exercise 5.1b: Set up trigger

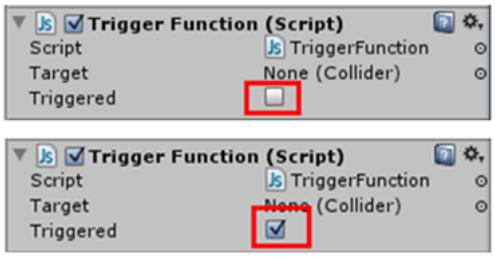


Step 1: Select *TriggerCube*, Expand the Box Collider component in the Inspector view, and check the *Is Trigger* box.

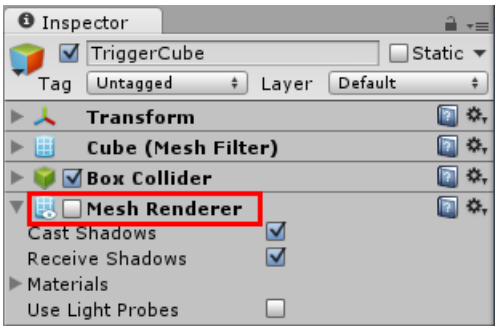
Step 2: Create a new script: *Assets->Create->C#*, and rename it *TriggerFuncion*. Attach the new script to *TriggerCube*. Double click open the script, and key in the follwing code.

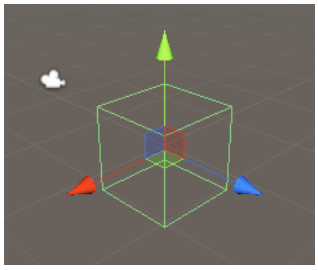
```
using UnityEngine;

public class TriggerFunction : MonoBehaviour
{
    public bool triggered = false;
    public void OnTriggerEnter(Collider other)
    {
        // Change the cube color to green.
        MeshRenderer meshRend = GetComponent<MeshRenderer>();
        meshRend.material.color = Color.green;
        Debug.Log(other.name);
        triggered = true;
    }
}
```

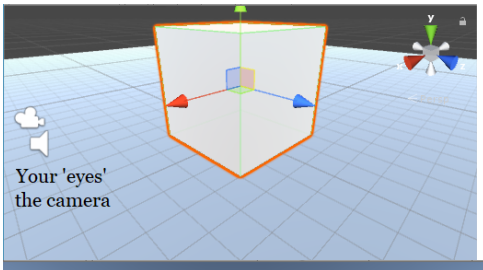
Now run the game. With all the components attached to the prefab *CapsuleFPS*, you will be able to use your mouse to change the camera direction, and use your arrow key to move the capsule about in the scene [key w or △ to move forward, s or ▽ to move backward, a or ◁ to move left and d or ▷ to move right] [You may also click and hold the right mouse button and move around to have a good view of the TriggerCube in the Game view. As you are the First Person Shooter, you cannot see yourself, only what is in front of you.]. Now observe the Inspector view for *TriggerCube*. In the Trigger Function component, the variable *Triggered* is ticked off when the game runs. However, when you move the *CapsuleFPS* into *TriggerCube* [you cannot see the cube in the Game view once the capsule is inside], it will be ticked, which signals that the function *OnTriggerEnter* has been called.

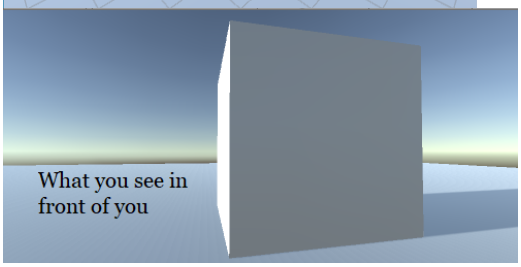


There's one last optional thing you can do, which is to create an invisible trigger zone. Only do this if you want to detect where the player is without the trigger zone being visible during gameplay (so this is probably not good for collectibles in the game). To remove the graphics of your trigger object, look at *TriggerCube* in the inspector and remove the Mesh Renderer component. Now your object is invisible, but still detects when something triggers it.






Your 'eyes'
the camera


What you see in
front of you

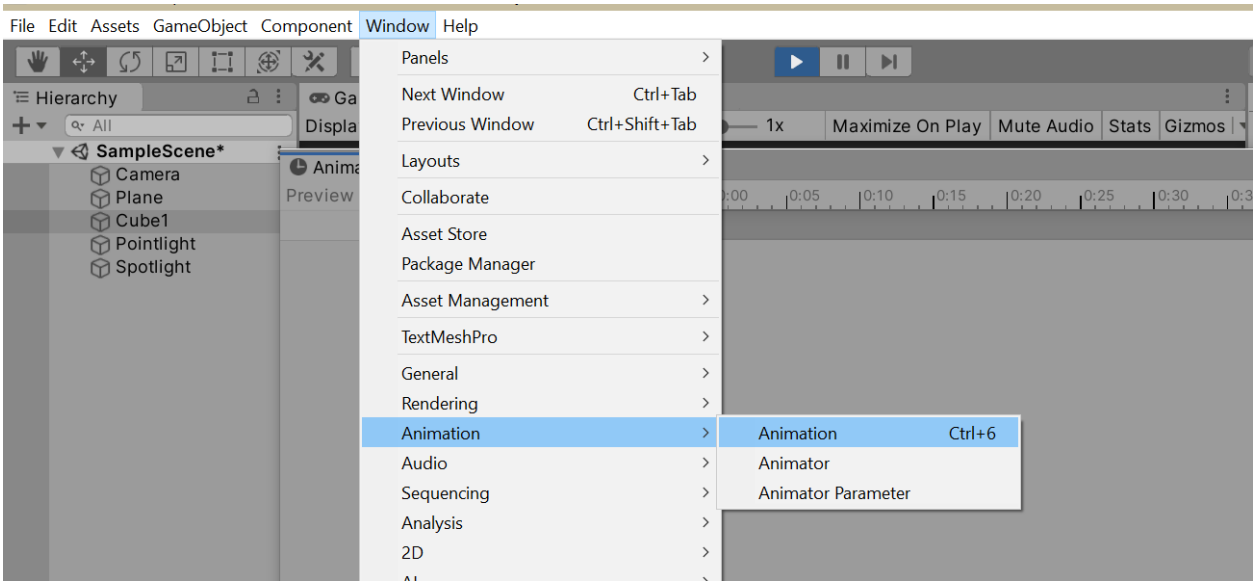
Procedure 6: Creating Animations

Animating an object can be easily done in Unity. We assign an Animator component to the object that needs to be animated. The Animator component itself has subcomponents such as the animationg controller; the controller defines the transition between different animations that the object may have to undergo. In this we part we 1) create animations 2) update the state machine logic of the animation controller 3) assign the animator to a GameObject in our scene

Time to complete: 3 hours.

Point 1: Creating an Animation

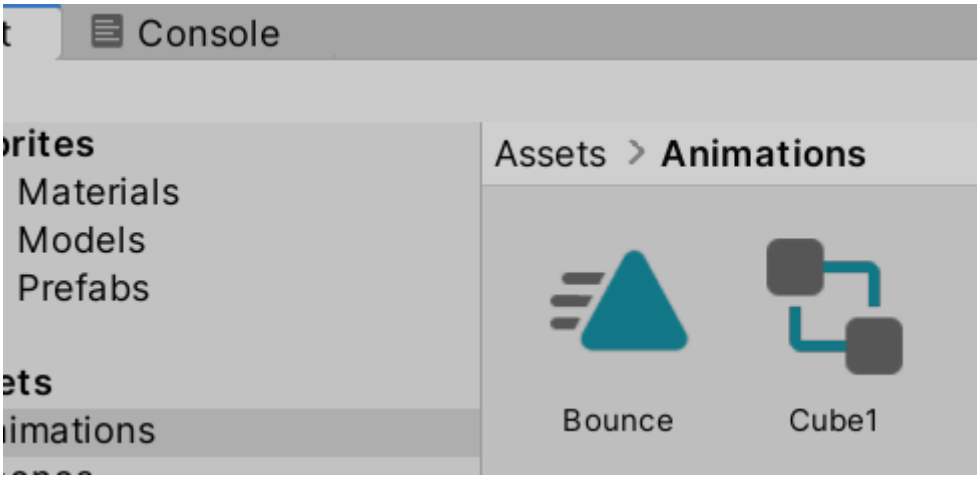
We can 1) creat an animation from scratch 2) import an animation and, 3) modify an imported animation.
Exercise 6.1: Create an aimation
Step 1: Create a cube at (1,1,0)
Step 2: Select the newly created "Cube" in the hierarchy. Then go to Window→Animation→Animation.



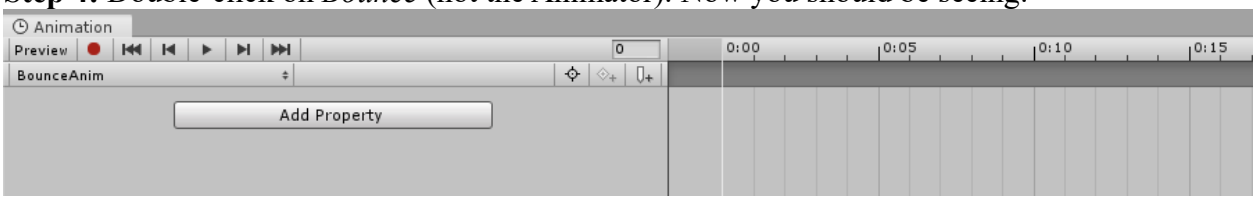
A window like this opens:

Step 3: Click Create and name it *Bounce* Save in Assets→Animations. Create an Animations folder under Assets if you do not have one already. The extension will be .anim. You may see a window like this:

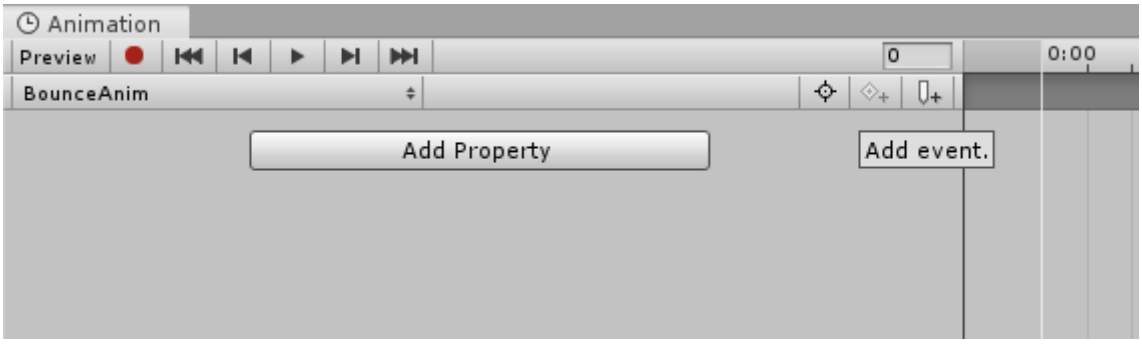
Alternatively, two Assets will have appeared under Assets folder, or under Animations folder if you created and saved your animation in this folder. One new asset is *Bounce* animation and another is an animation controller which may have been named *Cube1* or some other name.



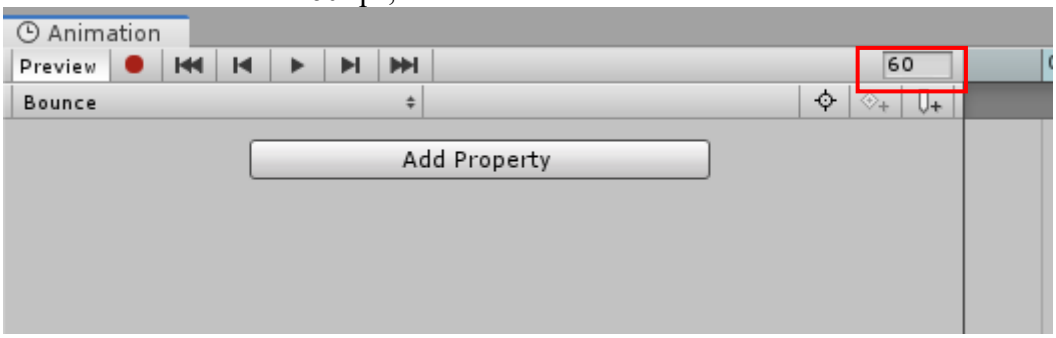
Step 4: Double-click on *Bounce* (not the Animator). Now you should be seeing:



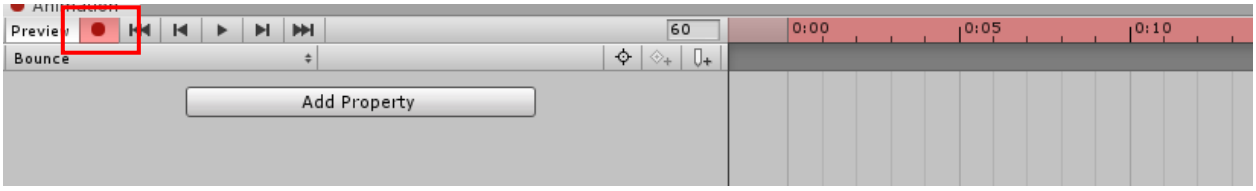
Step 5: Many properties of a GameObject can be animated; these include its transform (position, rotation, and scale), its material properties, or even some of the properties that we assigned in script such as the _moveSpeed of the balls in our earlier examples!



In the input field that now has 0 (zero), put in **60**. With this you are telling Unity to divide what you'll soon record across 60 frames. So if the system (the application) displays 60 frames per second, then the recorded changes will take place in ONE second. However, if the application frame rate is other than 60 fps, then the time will be different.



Step 7: Click the red record button. Note the change in how the time line looks; it is now highlighted

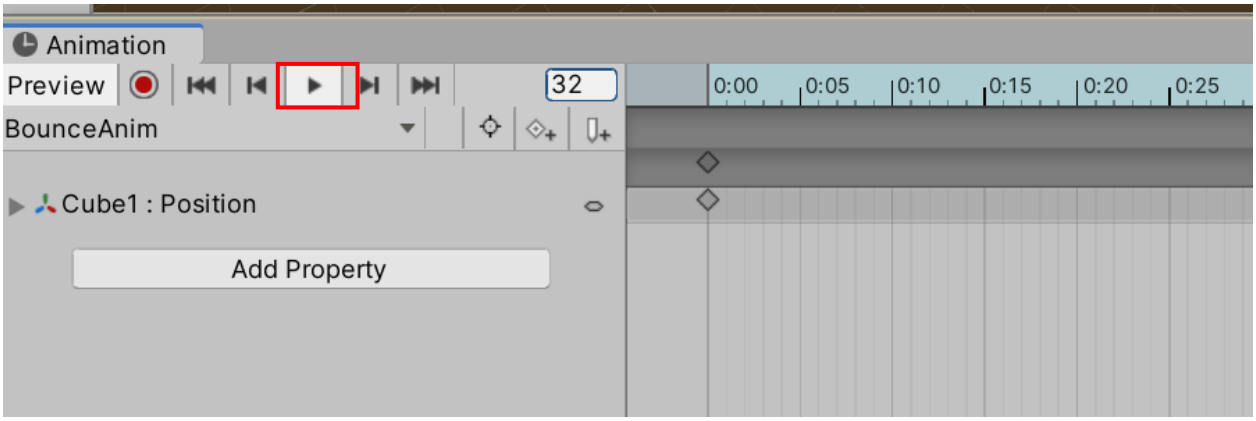


If you did not get the change in highlighting color, it is likely that you do not have an object

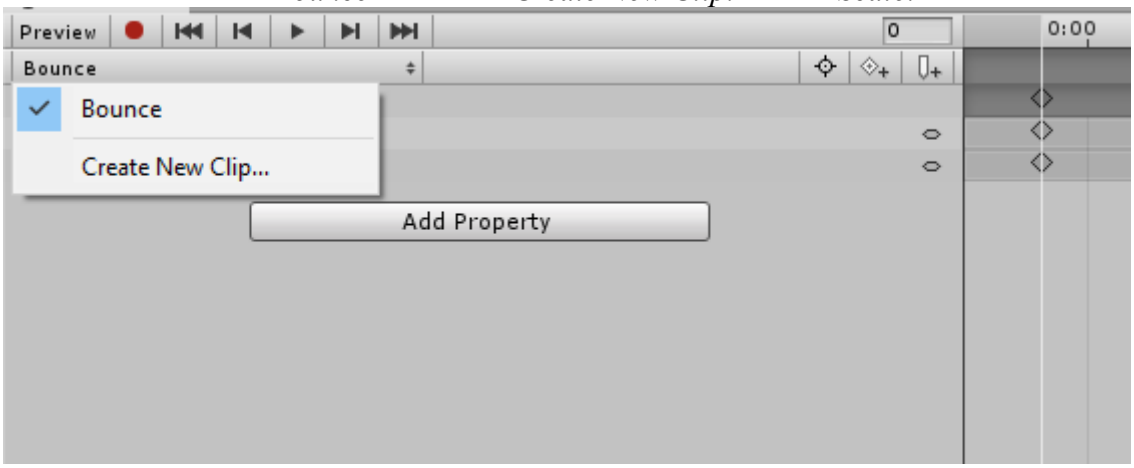
selected. Go to the hierarchy and select the cube (the one created in step 1). Go back to the Animation window and click record button.

Step 8: While still in recording mode, return to the Unity Scene view and move the cube away from its current position to another point. Also, rotate it however you want.

When satisfied, press the record button again to stop recording. Then press the play button to see your animation! We named this animation "Bounce", so can you make it actually bounce? Which property would you change?



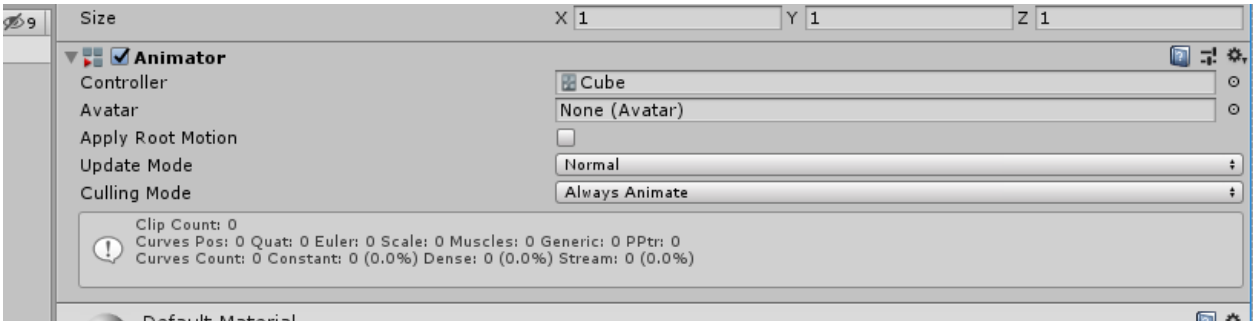
Step 9: Create another animation that scales (changes its size) the Cube and call this animation *Scale*. A quick way to do this is to go to the Animations folder, and double-click on *Bounce* animation. Then select the Cube GameObject in the hierarchy. Back in the animation click the up-down arrows next to *Bounce* and select *Create New Clip.* Call it *Scale.*



Using the steps outline above, create an animation that **scales** the object.
We now have **two animations.**

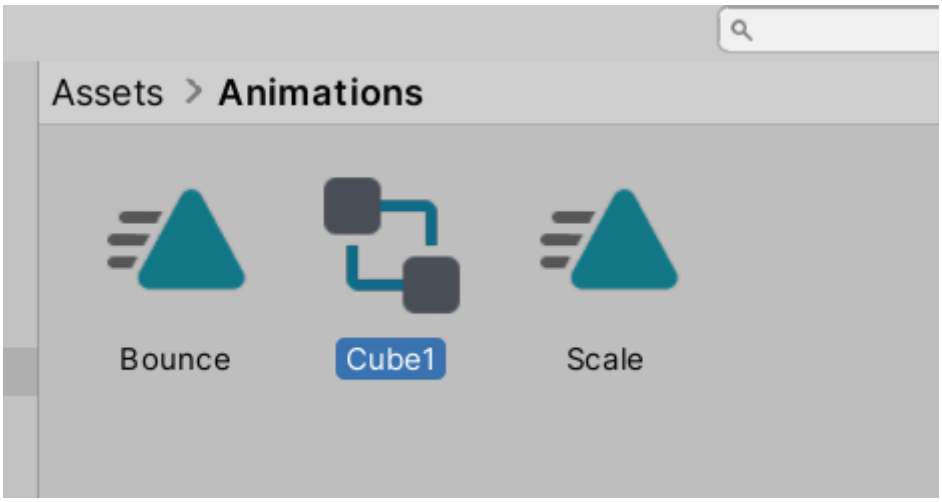
Point 2: Setting up the controller

You may note that in the Inspector for Cube GameObject, there is a new component called Animator:

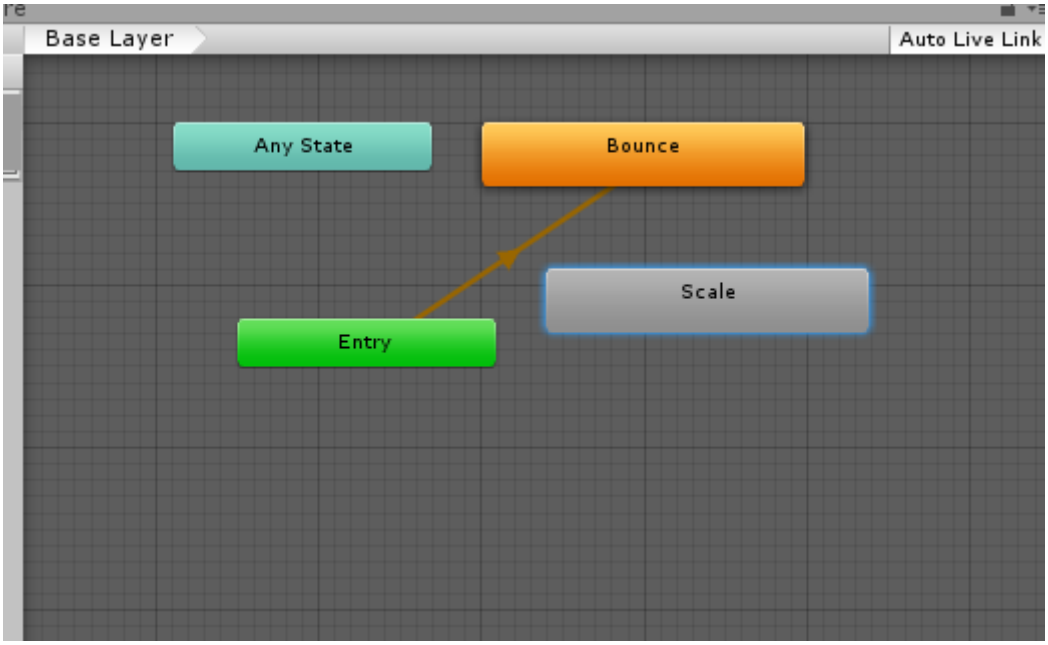


The Animator has a controller to control transitions between the different animations—in our case only two. And of course it also needs to have the animations, which we have created already. Wonderful!

Exercise 6.2: Transitions between animations
Step 1: Double-click on the Animation Controller (highlighted in screen shot below)

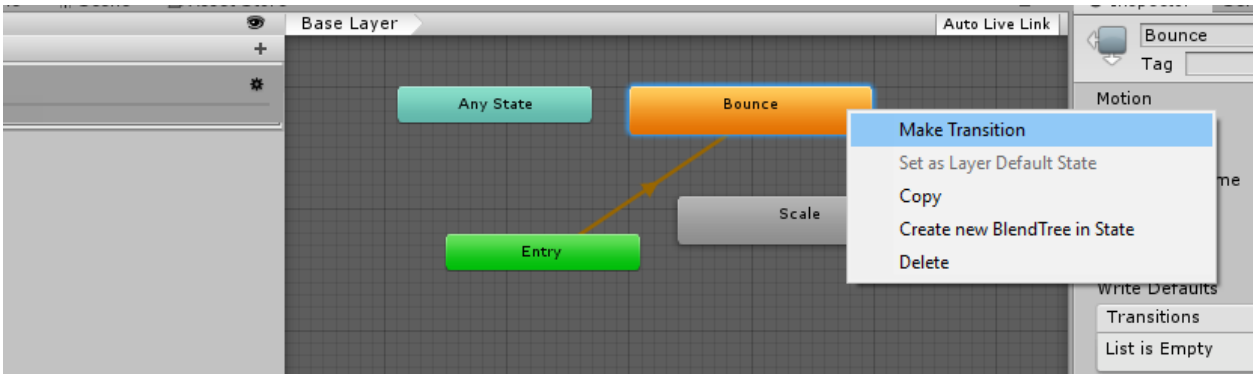


You should now see:

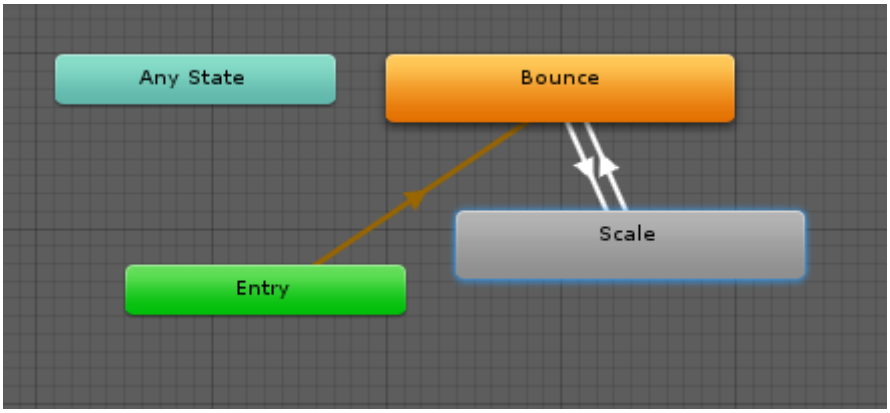


This shows that when you enter Game mode, your default animation is *Bounce.* You can change this. One way to do this is to right-click on *Scale* (or whichever animation you want to make default) and select *Set as Layer Default State*

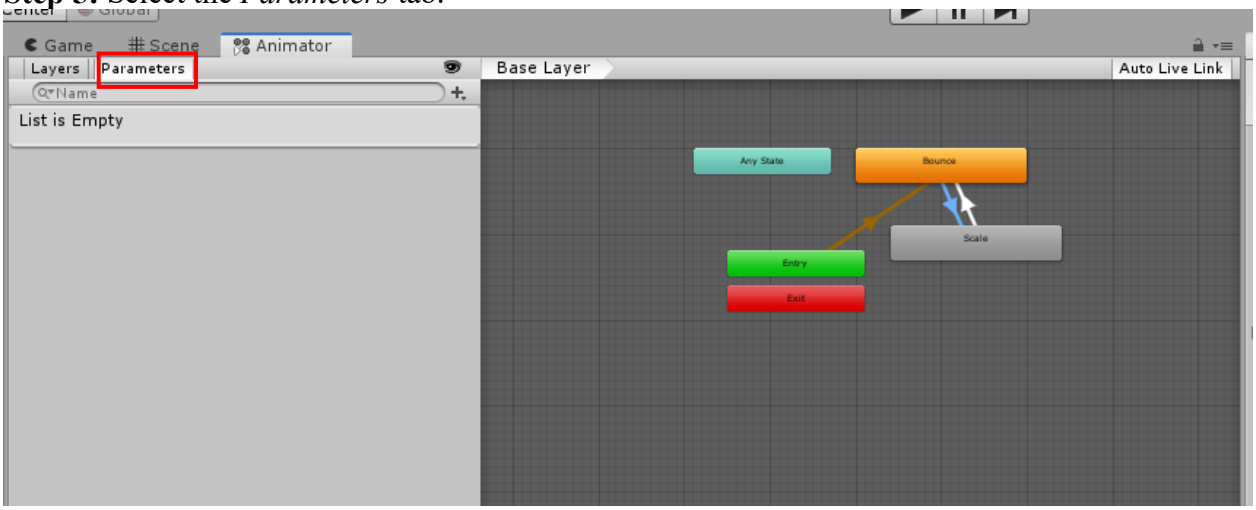
Step 2: Right-click on *Bounce* and select *Make Transition*



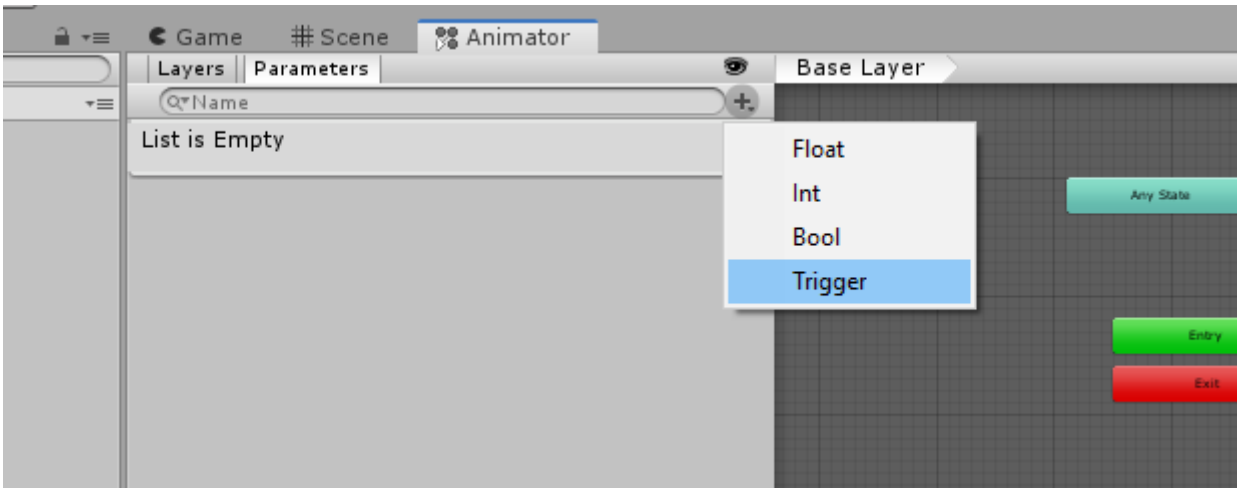
Step 3: An arrow is instantiated. Point it at *Scale* and left-click to connect.
Step 4: Do the likewise to connect *Scale* to *Bounce*



Step 5: Select the *Parameters* tab.

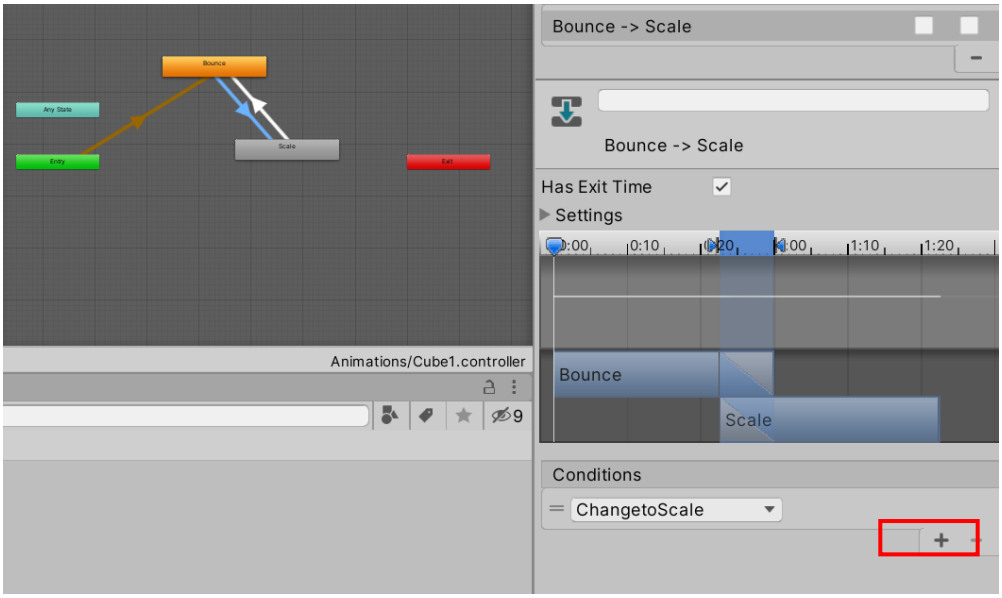


Step 6: Click the + icon next to the search field and select *Trigger*

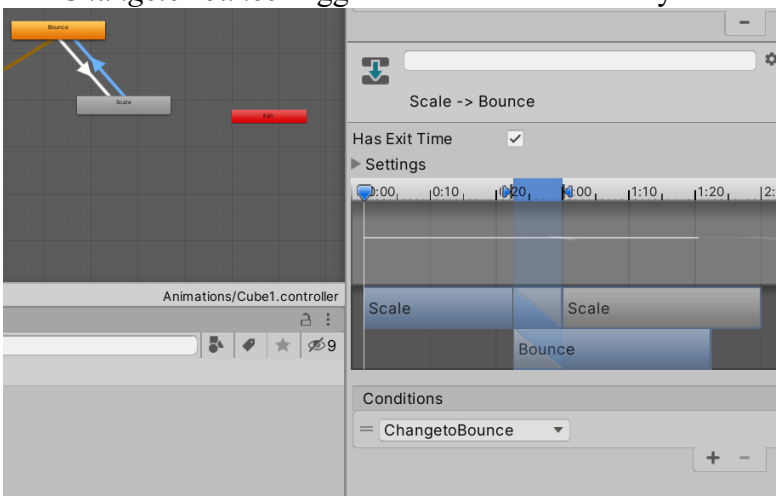


Step 7: Name this Trigger as *ChangetoBounce*.
Step 8: Create a new trigger and name this Trigger as *ChangetoScale*.

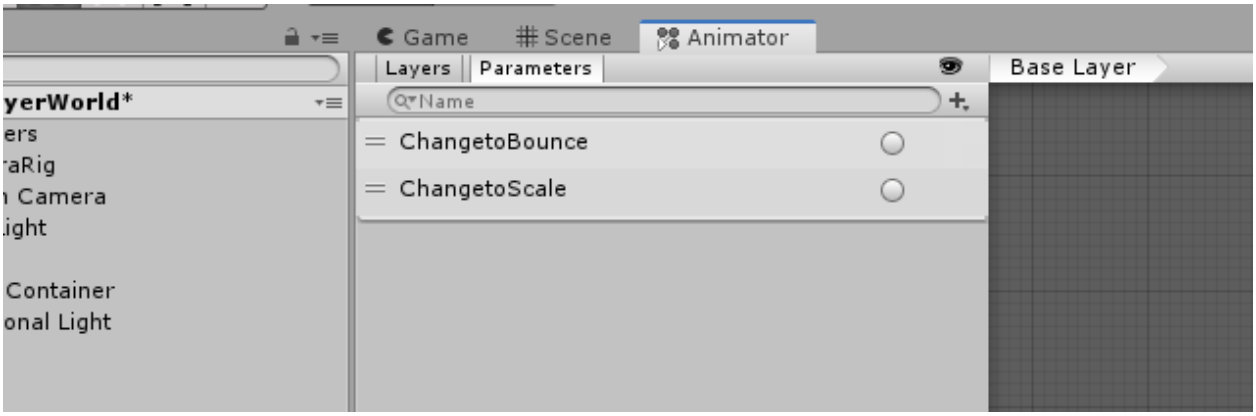
Step 9: Select the *Bounce* to *Scale* arrow. Under *Conditions,* click the + button and select *ChangetoScale*



Step 10: Select the *Scale* to *Bounce* arrow. Under *Conditions,* click the + button and select *ChangetoBounce* trigger if it isn't automatically selected.



What we have done is to define when to transition between the two animations. As you can see the triggers **are not activated.**



We do this activating in code.

Point 3: Scripting for animation control

The final part for animation!

Exercise 6.2: Player manipulating the transition

Step 1: In your Scripts folder (create one if you don't have it) create a new C# script called *CubeAnim.cs.* **Attach *CubeAnim* script to the Cube that we have been working with**
Step 2: *Type* the following code into the script

```
using UnityEngine;

0 references
public class CubeAnim : MonoBehaviour
{
    Animator anim;
    // Start is called before the first frame update
    0 references
    void Start()
    {
        anim = GetComponent<Animator>();
    }

    // Update is called once per frame
    0 references
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.B))
        {
            anim.SetTrigger("ChangetoBounce");
        }
        if (Input.GetKeyDown(KeyCode.G))
            anim.SetTrigger("ChangetoScale");

    }
}
```

Basically this code checks if the user has pressed B and if so, changes to the *Bounce* animation (By setting the *ChangetoBounce* trigger ON). If the user presses G, the (game) state machine transitions to the *Scale* animation.
From here one can create very complex State Machines (the Animation Controller is more or less a state machine with the animations being different states. In general however states need not be synonymous with animation; a state can include animations and many other variable states)

And now you're ready to create your own amazing animations!!

Procedure 7: Using ML-Agents toolkit to train AI agents in Unity (Optional)

Applications such as autonomous cars require simulating the behaviour of the "agent" in an environment that is as close as possible to the real environment. An agent is a representation in the game (simulation) world. If you want to train a car to drive itself through Singapore, you would need to create an environment that is like Singapore as much as possible. In actuality you may not need to include extraneous details such as insides of buildings and parks. For this simulation, how accurately the roads are represented in the simulation may be the most important.
Once you have the environment set up, you need to create an agent that would operate in that environment. For the autonomous car example, that would be the car. While not all the exact details of how the car works may be needed, it helps to include a lot of the expected behaviour. For example, a truck will likely need to be assigned a much higher aerodynamic coefficient (a factor that affects how much drag the moving vehicle experiences) than a well-designed streamlined sports car.
Once you have the environment and the agent, it is time to cognify the agent! You can think of this as assigning a brain to the car—but don't read too much into this metaphor, as it is useful only as metaphor. There are many ways to actually train the brain, and that is where **ML-Agents** comes in.
ML-Agents is a Python toolkit with which one can train AI agents in Unity. It allows Unity developers to access the widely available machine learning libraries written in Python. It provides an interface between Python and Unity.

Setting up:

1. Install Anaconda. Install the Python 3.6 version.

   Anaconda allows us to run different distributions of Python in separate environments. One environment may have packages that another does does not. To see the packages associated with the current environment issue *conda list* command at the Anaconda Prompt Terminal There will be a star next to the environment that is currently active.

2. Once installed, open Anaconda Prompt application by searching for it. You may see a screen like so:

3. Create a new environment by typing the following and press ENTER:

   conda create --name ml-agents python=3.6

```
(base) C:\Users\luwa0001>
(base) C:\Users\luwa0001>conda create --name ml-agents python=3.6
Collecting package metadata (repodata.json): done
Solving environment: done
```

We created an environment that is using Python 3.6. We can specify other packages and versions, but this will do for now.

4. To confirm that the new environment has been created
   conda env list

```
(base) C:\Users\luwa0001>conda env list
# conda environments:
#
base                  *  C:\Users\luwa0001\AppData\Local\Continuum\anaconda3
PythonCPU                C:\Users\luwa0001\AppData\Local\Continuum\anaconda3\envs\PythonCPU
ml-agents                C:\Users\luwa0001\AppData\Local\Continuum\anaconda3\envs\ml-agents
pythonCPU                C:\Users\luwa0001\AppData\Local\Continuum\anaconda3\envs\pythonCPU
pythonGPU                C:\Users\luwa0001\AppData\Local\Continuum\anaconda3\envs\pythonGPU
```

Make sure you see **ml-agents** there. The starred environment is the active one. Note that Anaconda (currently) seems to save environment names without regard to case of letters. So in the above, PythonCPU and pythonCPU are actually one environment even though they are shown as two separate ones in the output.

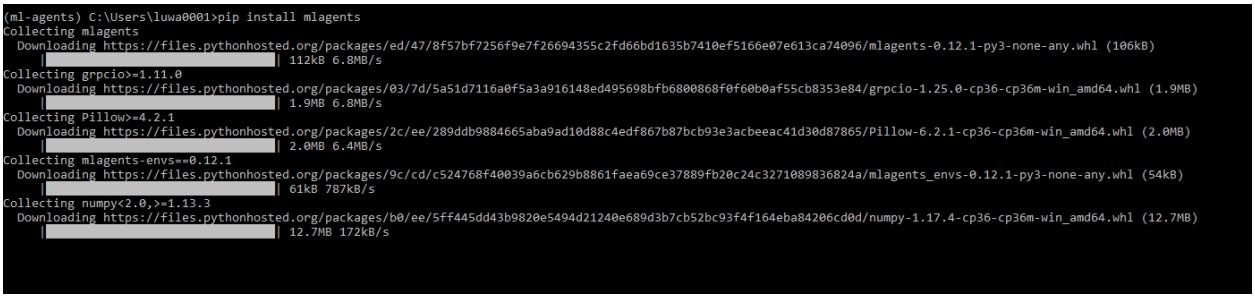
5. Activate the newly created environment:
   conda activate ml-agents

```
(base) C:\Users\luwa0001>conda activate ml-agents

(ml-agents) C:\Users\luwa0001>
```

Note: The name of the active environment is shown in bracket. For example in the above screenshots the active environment changed from *base* to *ml-agents* after we issued the request for activating ml-agents.

6. Install git by *conda install -c anaconda git*

7. **Install mlagents via *pip install mlagents***



8. To test that the ml-agents package has been correctly installed issue this command
mlagents-learn --help
Note that this is two dashes before help. Your success will look like this:

```
(ml-agents) C:\Users\Luwanga\OneDrive - Nanyang Technological University\PhD\Te
aching Assistantship\IM2073-Game-Programming>mlagents-learn --help
usage: mlagents-learn.exe [-h] [--env ENV_PATH] [--resume] [--force]
                          [--run-id RUN_ID] [--initialize-from RUN_ID]
                          [--seed SEED] [--inference] [--base-port BASE_PORT]
                          [--num-envs NUM_ENVS] [--debug] [--env-args ...]
                          [--cpu] [--torch] [--tensorflow] [--width WIDTH]
                          [--height HEIGHT] [--quality-level QUALITY_LEVEL]
                          [--time-scale TIME_SCALE]
                          [--target-frame-rate TARGET_FRAME_RATE]
                          [--capture-frame-rate CAPTURE_FRAME_RATE]
                          [--no-graphics]
                          [trainer_config_path]
```

9. Now we will clone a Unity Project developed by Unity Technologies. This Project has examples to get us started with using mlagents. *cd* to a directory where you will create your Unity Project. Take note of the location. As an example, I want to save my clone in the Machine Learning folder like so:
cd "OneDrive - Nanyang Technological University\Machine Learning"

   When you're in the folder you want, issue this request in Anaconda Prompt:

```
git clone --branch latest_release https://github.com/Unity-Technologies/ml-agents
```

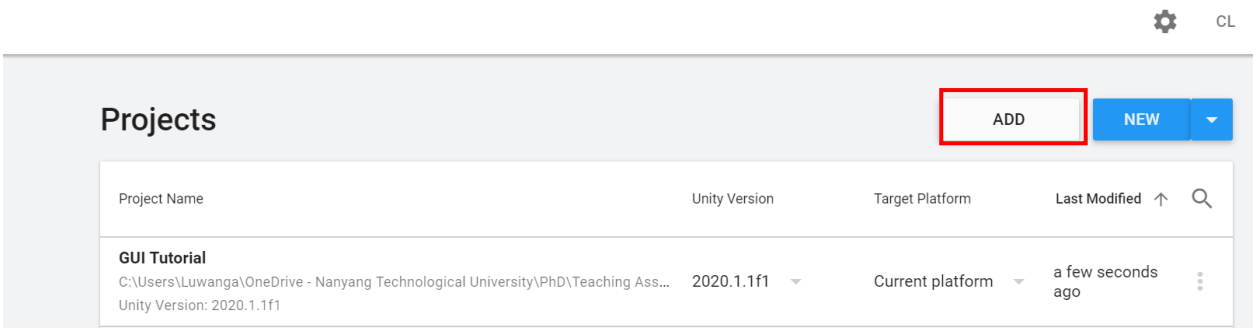
   This will create a folder called "ml-agents". Inside this folder is **Project** subdirectory. This subdirectory contains several example environments (Assets/ML-Agents/Examples) and we'll now go through one of the examples to understand how the process of setting up agents and environments works.

   Note: "ml-agents" the folder is different from "ml-agents" the Python virtual envrironment

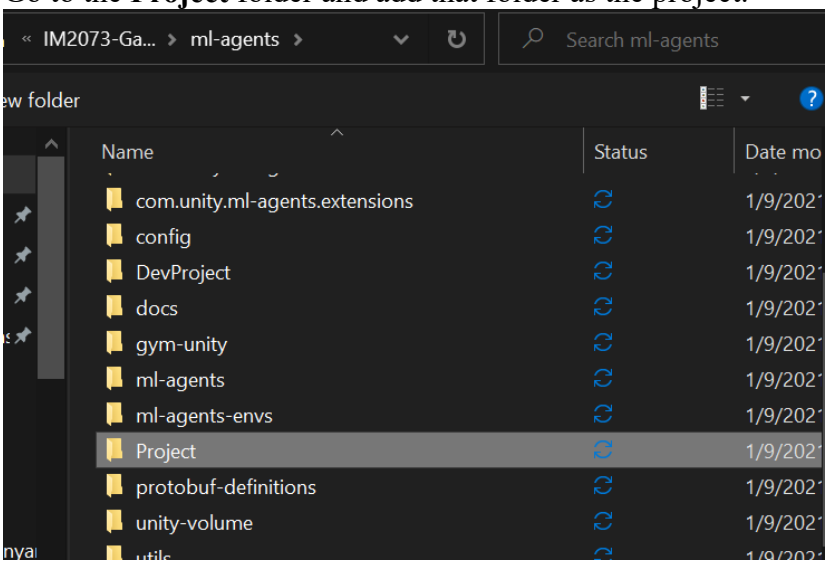
For more fun with conda commands go here

Point 1: Training an Agent

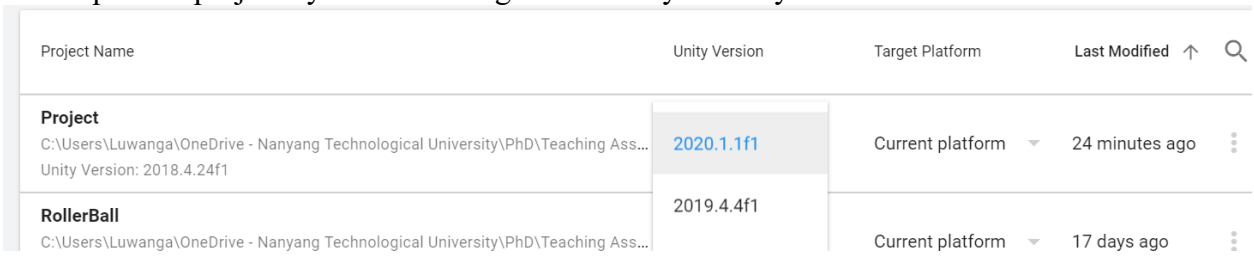
1. Exit Unity Editor and reopen Unity Hub. Then click *Add*



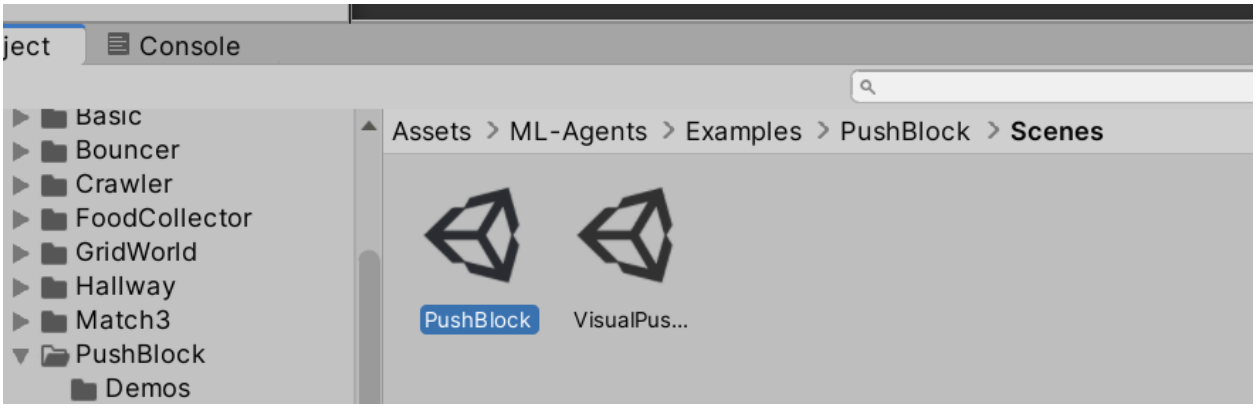
2. Go to the **Project** folder and add that folder as the project.



Then open the project by after selecting which Unity Editor you want to use:



3. Go to Pushblock scene (ML-Agents/Examples/Pushblock/Scenes/PushBlock). Double-click the PushBlock scene to open it.



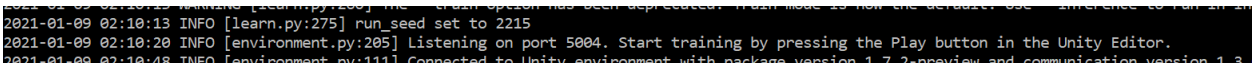
Here are the details about this example. Go to the Pushblock section to understand the goal, environment set up and other details.
Essentially we have a robot (it is represented in the scene by the the blue cube) and a block (white block). The goal is for the robot to push the block to the green area. We want to train it to be able to do this all by itself!

4. Go to Anaconda and INSIDE the folder *ml-agents* (that is, in Anaconda *cd* into this folder if you are not there currently) issue this request:

   mlagents-learn config\ppo\Pushblock.yaml --run-id="myFirstRun"

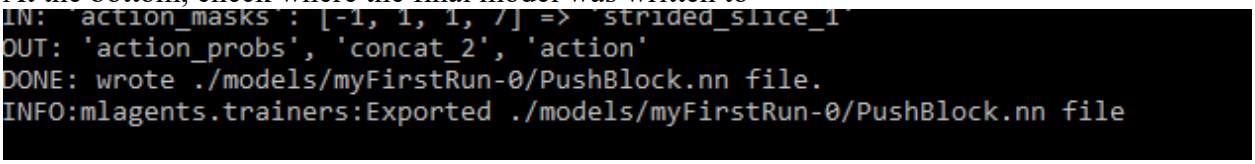
   Note the double dashes before run-id.
   mlagents-learn takes the *positional* argument config/ppo/Pushblock.yaml. This specifies details about the trainer module, and must come first when submitting the arguments. Other parameters can be specified used the *param=value* syntax. For example, run-id is a label to help us identify runs. You can call it whatever you want to.
5. Once you have pressed ENTER, you'll see this

   You will see a message that informs you to PRESS PLAY button in Unity.



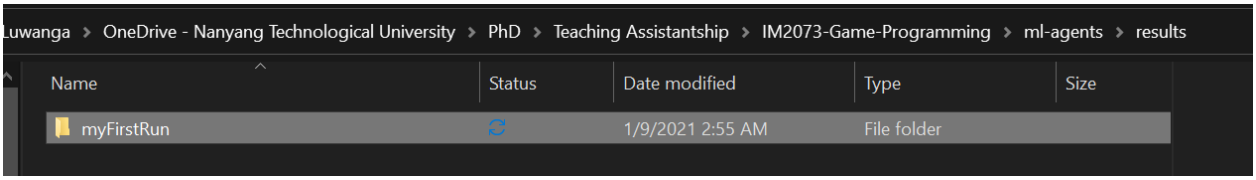
Go to Unity and press the PLAY button. The training will start and you can watch the scene to see robots learning.
Training will take several minutes or at times hours.

6. At the bottom, check where the final model was written to

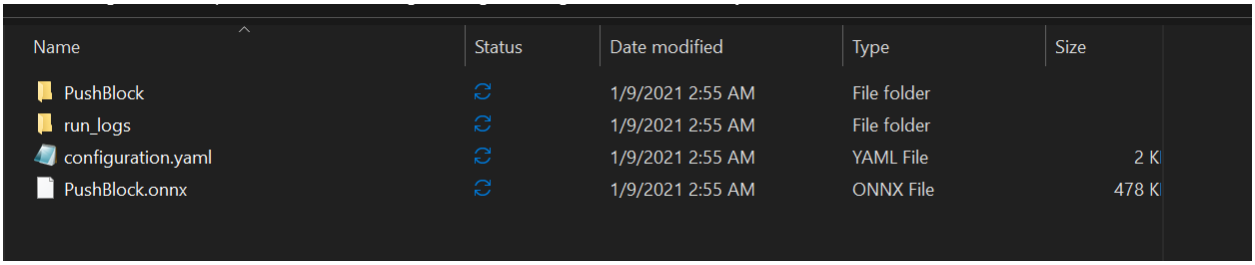


7. Go to the ml-agents folder that you cloned from GitHub. Go to *results* subdirectory

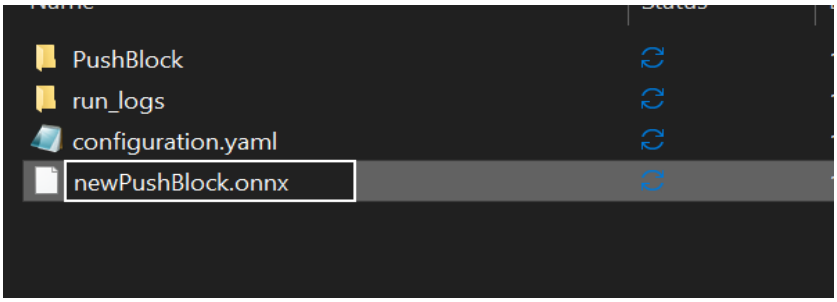
Inside models you'll find a folder that has the same name as the run-id name that you gave the trainer. *myFirstRun*, where my run-id identifier name is *myFirstRun*.



Open this folder to find *PushBlock.onnx*. onnx stands for Open Neural Network Exchange, a file format for seamless exchange of ML models

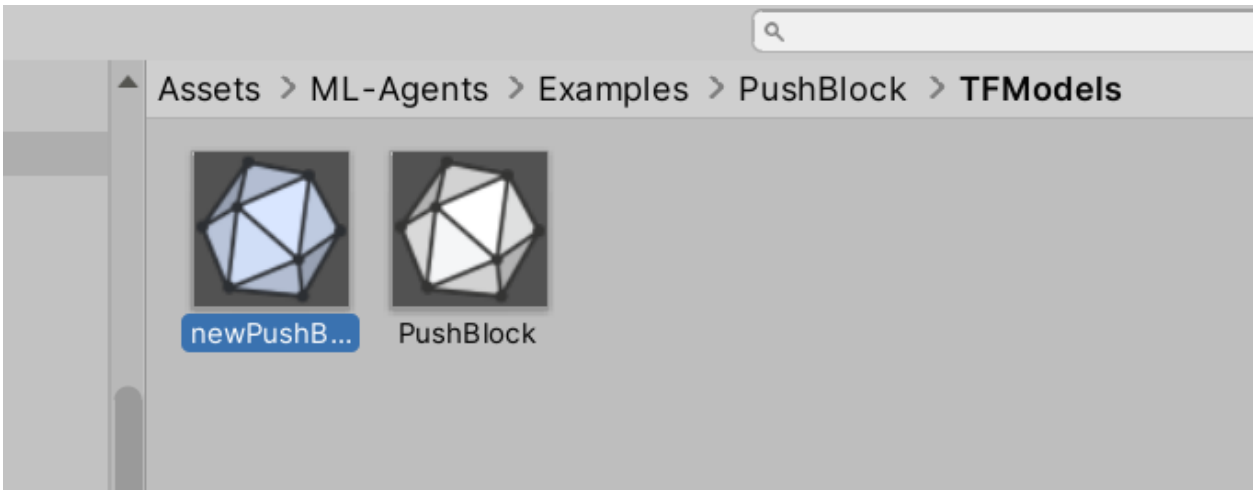


8. Rename *PushBlock.onnx* to *newPushBlock.onnx*

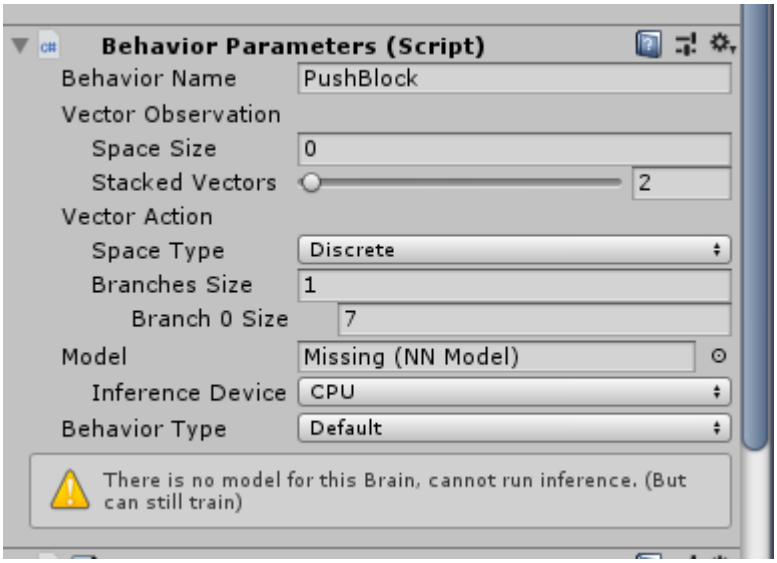


9. In Unity, go to TFModels folder of the PushBlock example (Assets/ML-Agents/Examples/PushBlock/TFModels)

10. Drag the *newPushBlock.onnx* file that you got in (8) into this TFModels folder.

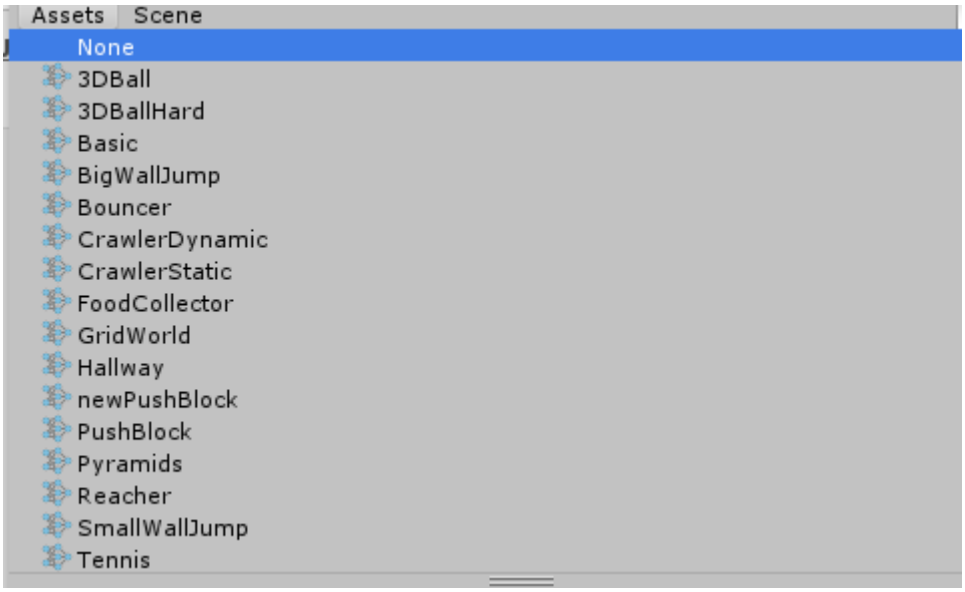


11. Go to Unity and expand the *Area* GameObject. Find the *Agent* GameObject and in its Inspector view find the *Behavior Parameters* script component.

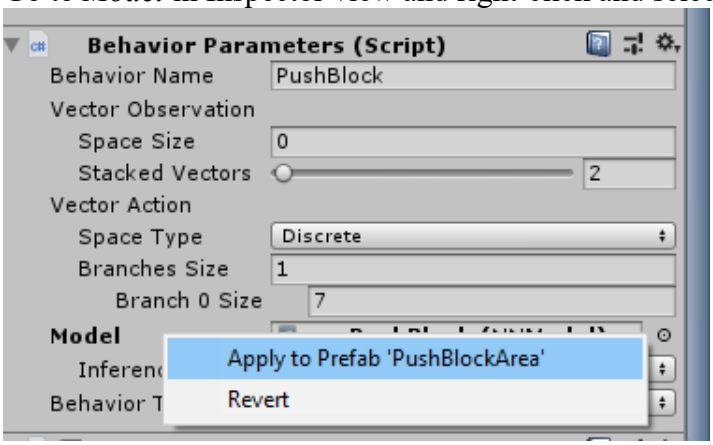


We want our recently *trained model* to be the value for the exposed *Model* field of the *Behavior Parameters* script. Now it is showing "Missing (NN Model)"

12. Click the little circular button at the end of the *Model* field. A new window opens like so:



13. Select *newPushBlock*.
    Alternatively you can drag and drop.
14. Go to *Model* in Inspector view and right-click and select "Apply to prefab 'PushBlockArea'".



15. Press *Play*. Your trained neural network model is now in business. Congratulations!

Point 2: Understanding Reinforcement Learning and Unity

We have just trained our AI agent to push a block to a goal. How did that work?
We used what is called Reinforcement Learning. This is a form of Machine Learning that can be implemented based on positive and negative reward for an agent, depending on whether the agent (learner) is taking *actions* that are towards achieving the goal or not. For purposes of understanding how this works, let us look at four key concepts: *Academy, Agent, Brain,* and

Environment

1. *Academy*. You may think of Academy as the "place" where the training of agents takes place. Every AI simulation or game must have a class that inherits from the *Academy* superclass. We however are not required to override the parent methods.

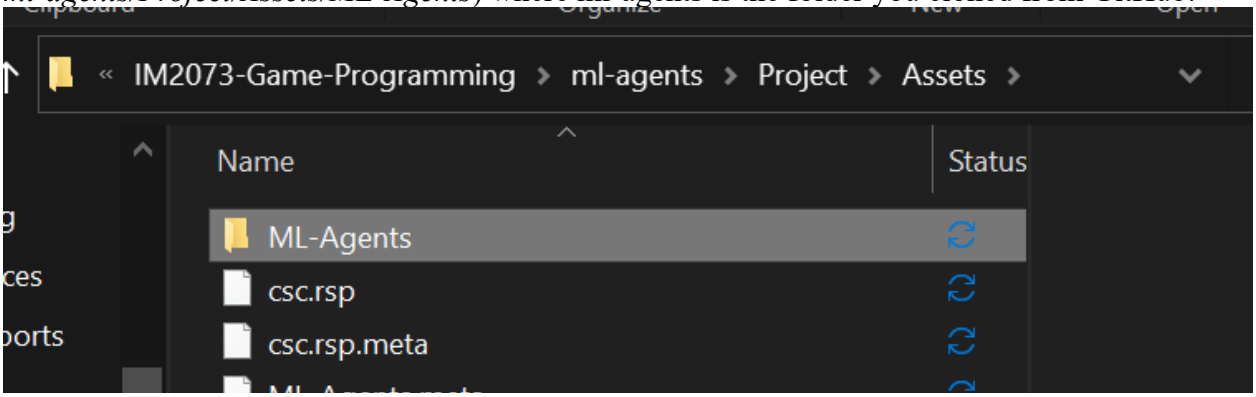
2. *Agent* is the intelligent object that we want to train. Its visual representation in the scene can be anything we desire—it could a box, a sphere, a human-like thing. But we'll later attach a script that inherits from the *Agent* superclass, and, among other things, configure what *observations* the *Agent* will make and what *actions* to take.

3. The *Brain* acts somewhat like the brain that you possess, but this is a mathematical abstraction of it. (Take this merely as a useful metaphor, because the *Brain* in Unity/Python is much unlike the animal brain). This *Brain* is a neural network model that takes as input the *observations* sent by the *Agent.* The *Agent* gets the *observations* using its sensors (which may be implied rather than explicit). Once the *Brain* receives *observations* it does computations and sends to the *Agent* an *action* vector. The *action* vector is a set of numbers which are the output of the neural network model. Note that the brain does not know what these numbers will do, or what they mean for the *Agent.* For example an *action* vector could be two dimensional $v_{action} = [a\ b]$. The *Agent* may know that a corresponds to movement in the x direction, and b movement in the z direction. But the *Brain* is not aware of this. The *Brain* makes decisions based on the *reward* that the *Agent* gets following the *action* vector that the *Brain* had produced.

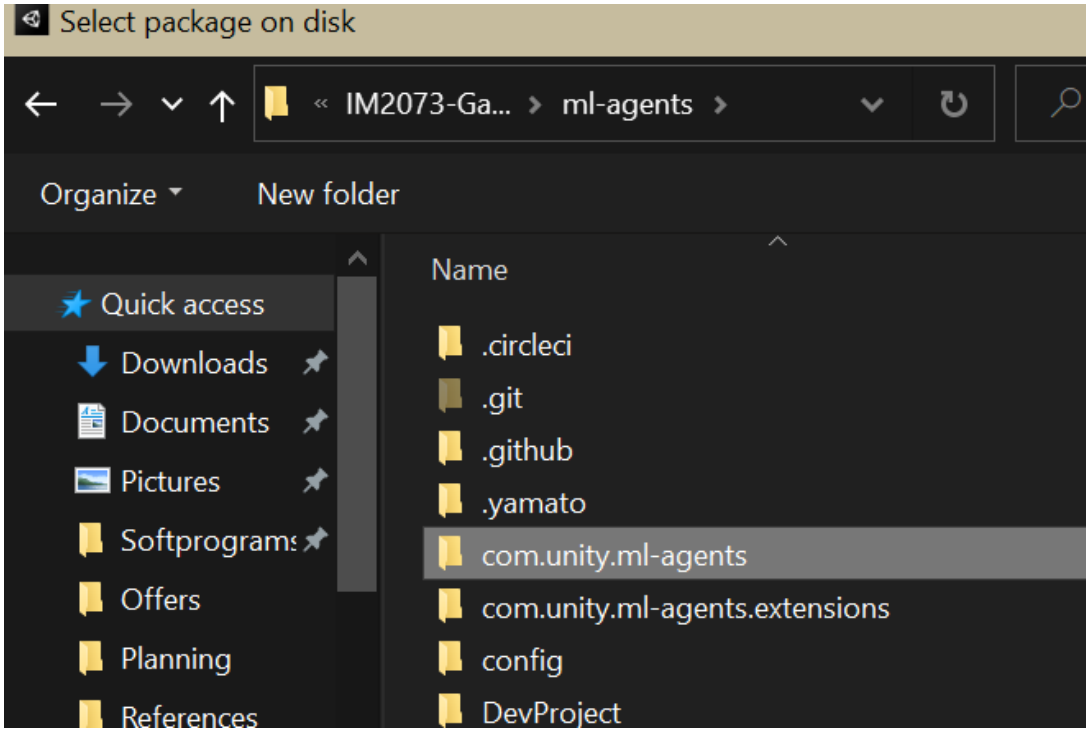
4. *Environment* is where the training will take place. It includes the objects that the *Agent* encounters (or, if it is a simulation of some real world scenario rather than a game, the environment would include the objects that the *Agent* can be expected to encounter in the real (physical) world).

Part II: Creating Your Own AI Game

Create a new project and call it "RollerBall".
Import the ML-Agents package into newly created project (to find your ML-Agents folder go to: *ml-agents/Project/Assets/ML-Agents*) where ml-agents is the folder you cloned from GitHub.

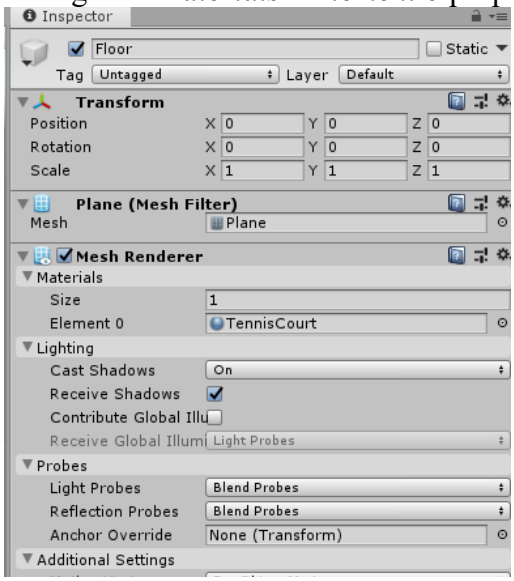


1. Go to *Window->Package Manager*



A. Create the Unity Environment

1. Create a plane at the origin with rotation = (0,0,0) and scale of =(1,1,1). Name it *Floor*
2. Change the *Materials* →*Element 0* property in Mesh Renderer to *TennisCourt*

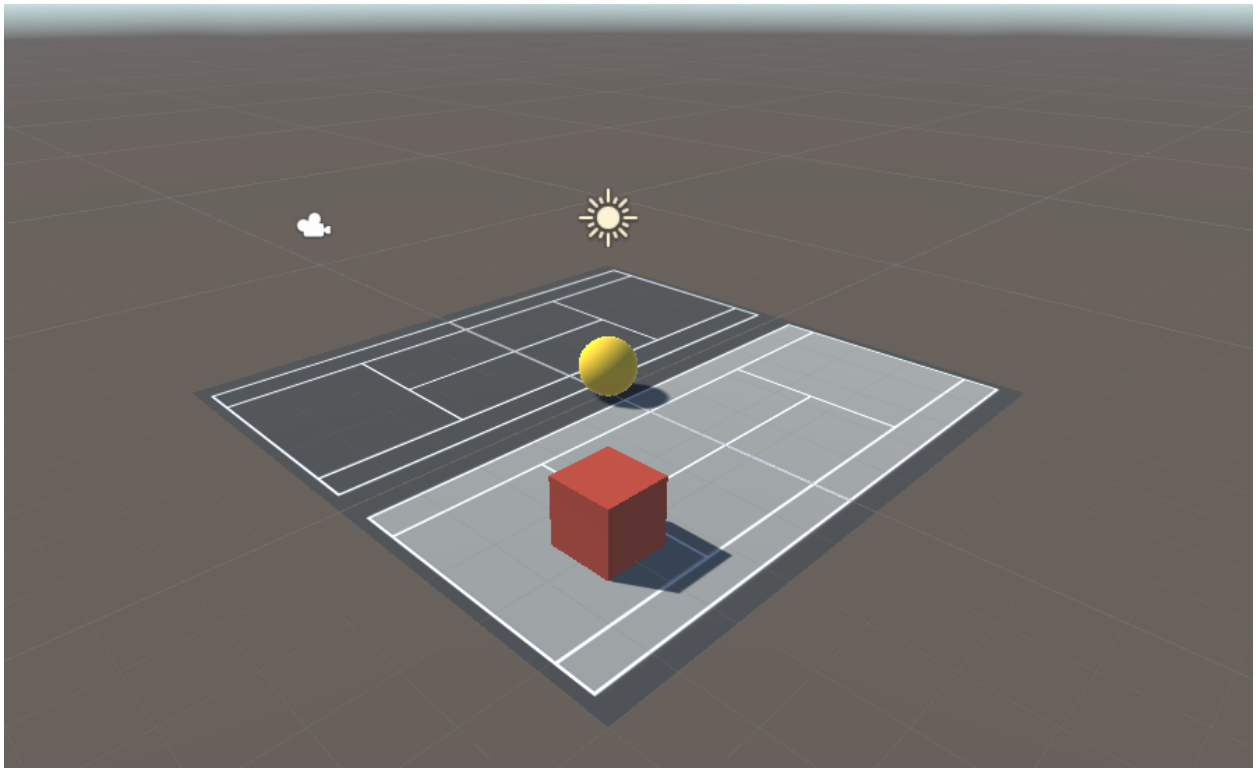


3. Add a cube with Transform Position = (3, 0.5, 3), Rotation = (0, 0, 0), Scale = (1, 1, 1) and name it *Target*

4. Change the material property to *Red*

5. Add a sphere with Transform Position = (0, 0.5, 0), Rotation = (0, 0, 0), Scale = (1, 1, 1). Name the sphere *RollerAgent*
6. Change the default material for *RollerAgent* to be *Yellow*
7. Add *RigidBody* component to *RollerAgent*

Now the environment should look somewhat like (may depend on your choice of materials):



B. Implement an Agent

1. Select *RollerAgent* GameObject and add a new C# script component to it. Name the script *RollerAgent.cs*
Note: The script name must match the class name. Verify this for every script that you create.

2. Double-click the newly-created script to edit it.

3. Once in the editor make the following changes

a. Add statement *using Unity.MLAgents;*

We have used the statement using MLAgents to inform the system to use the MLAgents namespace. The relevant resources are found under Assets/ML-Agents. MLAgents namespace contains the classes we want to use, such as Agent and Academy. For example our RollerAgent class now inherits from the Academy superclass. This implies that RollerAgent has methods and variables from Academy plus whichever others we may want to give it.

b. Change the superclass from *MonoBehaviour* to *Agent*
This is because we are now inheriting from the *Agent* class

c. Declare a new variable called *rBody*

d. In the *Start* method enter *rBody = GetComponent<RigidBody>()*
We are getting the RigidBody component that is attached to the same object that this script is attached to.

e. After the start of each round of training (that is, after the agent reaches the target or somehow forced to restart—in this case if it falls off the floor) the method *OnEpisodeBegin()* is called. Let us override this method to do: 1) send the *RollerAgent* back to starting point, resetting both its position and momentum to zero and 2) moving the target to some random position**.** **Complete** the code to arrive at the following:

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

public class RollerAgent : Agent
{
    Rigidbody rBody;
    //this Start() function is called once
    void Start()
    {
        rBody = GetComponent<Rigidbody>(); //get the RigidBody component
//attached to the same object that this script is attached to.
    }

    public Transform Target;
    //this function is at the end of each training round.
    public override void OnEpisodeBegin()
    {
        if (this.transform.position.y < 0) //if the ball falls off the
//floor its y position will be negative
        {
            // If the Agent fell, zero its momentum
            this.rBody.angularVelocity = Vector3.zero;
            this.rBody.velocity = Vector3.zero;
            this.transform.position = new Vector3(0, 0.5f, 0);
        }

        // Move the target to a new random spot
        Target.position = new Vector3(Random.value * 8 - 4,
                                      0.5f,
                                      Random.value * 8 - 4);
    }
}
```

f. The *RollerAgent* collects observations that it then sends to the *Brain* for decision making. These observations can be pictures, sounds, distances to chosen objects, etc. In our case we want the *RollerAgent* to collect **eight observations** namely, its own velocity (2 values: x, z components), its own position (3 values: x, y, z components), *Target*'s position (3 values: x,y,z components).

Let's add these values as arguments to method *AddObservation*. To add the Target's position for example, *sensor.AddObservation (Target.position).* Note that we can do this because we declared a Transform variable called *Target.* Create a new method called *public override void CollectObservations()* and add the other observations to arrive at:

```
public override void CollectObservations(VectorSensor sensor)
{
    // Target and Agent positions
    sensor.AddObservation(Target.position);
    sensor.AddObservation(this.transform.position);

    // Agent velocity
    sensor.AddObservation(rBody.velocity.x);
    sensor.AddObservation (rBody.velocity.z);
}
```

g. The observations are sent to the *Brain*. The brain does computations and sends back a response that is called (this is just a label) an *action vector.* What the action vector values actually mean is up to us. These are numbers that are the output of a neural network (or equivalent) after getting the input which includes the observations.
The action vector is the argument to the method *onActionReceived.* But how many elements will be in this action vector? We set this using *Vector Action Space Size* variable of the *Brain* which we will do soon in Unity Editor. For now, receive the *vectorAction* from the Brain and do something with it:

```
public override void OnActionReceived(ActionBuffers actionBuffers)
  {
    Vector3 controlSignal = Vector3.zero;
    controlSignal.x = actionBuffers.ContinuousActions[0]
    controlSignal.z = actionBuffers.ContinuousActions[1]
    rBody.AddForce(controlSignal * speed);
```

h. Add *Decision Requester* component (a script) to RollerAgent

i. Next, we will provide *Rewards*. In reinforcement learning the idea is to offer a reward each time an agent achieves the expected goal. Designing a proper reward mechanism results in more realistic or interesting behaviour. In this case we will provide a *Reward* of 1.0f (f for float) each time the *RollerAgent* reaches the target. Reaching a target is defined here as being a distance of less than *1.42f* from the Target. To set the reward, supply the reward as an argument to *SetReward* method.

```
    float distanceToTarget = Vector3.Distance(this.transform.position,
    Target.position);
    if (distanceToTarget < 1.42f)
    {
        SetReward(1.0f);
        EndEpisode();
```

```
    }
```

If the agent falls off the floor, we want to reset the Agent. This is what `EndEpisode()` is for.

```
    // Fell off platform
    if (this.transform.position.y < 0)
    {
        EndEpisode();
    }


    ==========================================
    //The RollerAgent.cs code becomes the following:

    using System.Collections.Generic;
    using UnityEngine;
    using Unity.MLAgents;
    using Unity.MLAgents.Sensors;
    using Unity.MLAgents.Actuators;

    public class RollerAgent : Agent
    {
        public Transform Target;
        Rigidbody rBody;

        public float forceMultiplier;
        //this Start() function is called once
        void Start()
        {
            rBody = GetComponent<Rigidbody>(); //get the RigidBody
component attached to the same object that this script is attached
to.
        }

        //this function is called at the start of each episode of
training.
        public override void OnEpisodeBegin()
        {
            if (this.transform.position.y < 0) //if the ball falls
off the floor its y position will be negative
            {
                // If the Agent fell, zero its momentum
                this.rBody.angularVelocity = Vector3.zero;
                this.rBody.velocity = Vector3.zero;
                this.transform.position = new Vector3(0, 0.5f, 0);
            }

            // Move the target to a new random spot
            Target.position = new Vector3(Random.value * 8 - 4,
                                          0.5f,
                                          Random.value * 8 - 4);
        }

        public override void CollectObservations(VectorSensor
sensor)
```

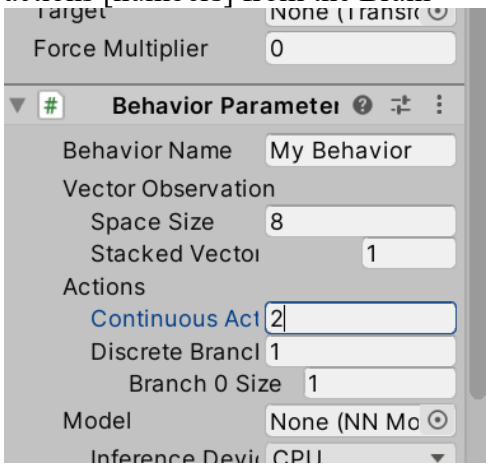
```
        {
            // Target and Agent positions
            sensor.AddObservation(Target.position); //three numbers
encoding position of Target
            sensor.AddObservation(this.transform.position); //three
numbers for position of RollerAgent

            // Agent velocity
            sensor.AddObservation(rBody.velocity.x); //one number
            sensor.AddObservation(rBody.velocity.z); //one number
        }

        public override void OnActionReceived(ActionBuffers
actionBuffers)
        {
            // Actions, size = 2
            Vector3 controlSignal = Vector3.zero;
            controlSignal.x = actionBuffers.ContinuousActions[0];
            controlSignal.z = actionBuffers.ContinuousActions[1];
            rBody.AddForce(controlSignal * forceMultiplier);

            // Rewards
            float distanceToTarget =
Vector3.Distance(this.transform.localPosition,
Target.localPosition);

            // Reached target
            if (distanceToTarget < 1.42f)
            {
                SetReward(1.0f);
                EndEpisode();
            }

            // Fell off platform
            else if (this.transform.localPosition.y < 0)
            {
                EndEpisode();
            }
        }



        public override void Heuristic(in ActionBuffers actionsOut)
        {
            var continuousActionsOut = actionsOut.ContinuousActions;
            continuousActionsOut[0] = Input.GetAxis("Horizontal");
            continuousActionsOut[1] = Input.GetAxis("Vertical");
        }


    }
```

C. Unity Setup

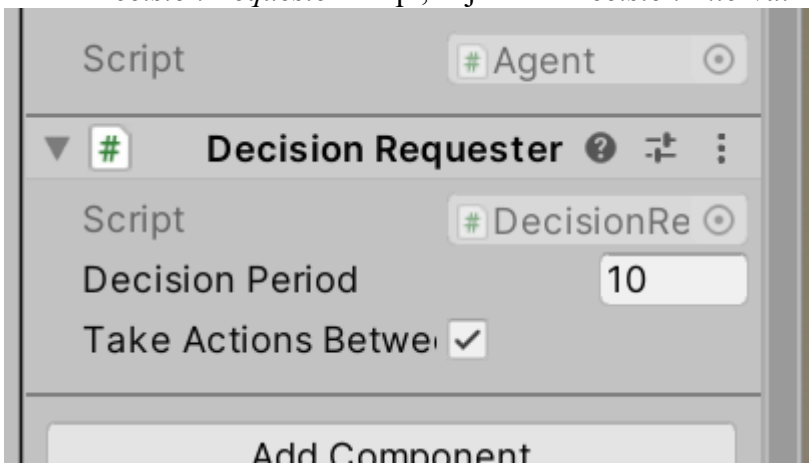
Back in Unity, attach the *RollerAgent* script to the *RollerAgent* GameObject if you haven't already. Do this by dragging the script onto the GameObject called *RollerAgent*

In the Inspector view,

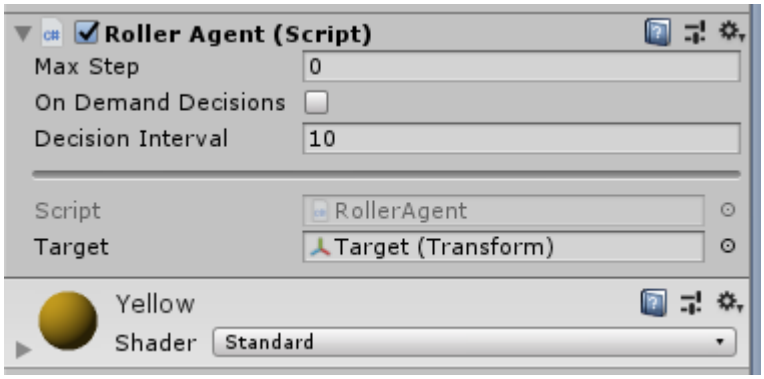
a. Add the *Behavior Parameters* script. To do so, select *RollerAgent* GameObject and go to Add Component→Search for *Behavior Parameters*
b. Set *Behavior Name* to *RollerBallBrain*
c. *Vector Observation* → *Space Size* to 8 //this is because we are sending 8 observations to the Brain
d. *Vector Action* → *Continuous Action* to 2. This is because we are receiving to actions [numbers] from the Brain



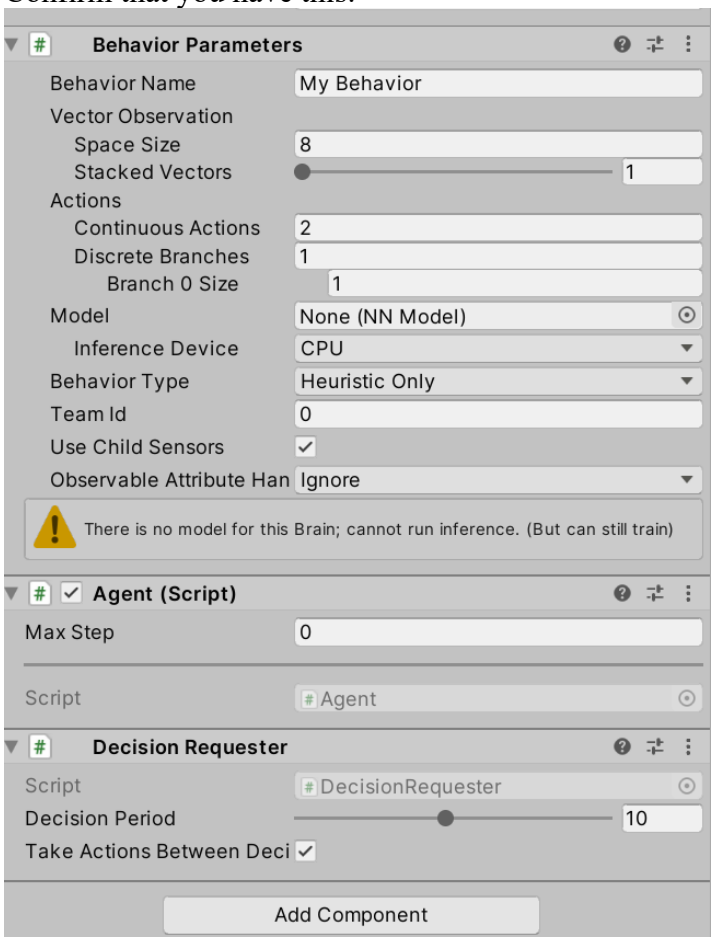
e. Add the *Decision Requester*
f. Under *Decision Requester* script, adjust the *Decision Interval* to 10



g. Still under *RollerAgent* drag the GameObject called *Target* and drop it into the Target field of the *RollerAgent* script



h. Confirm that you have this:



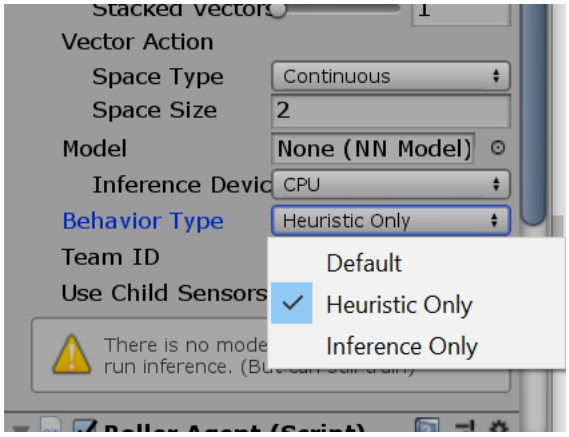
D. Testing Environment

To ensure that the game is working as expected we first try to play it manually.
To allow you to provide input to the Agent (you, instead of the brain are the one who then provides the elements of the action vector) you implement the *Heuristic* method in *RollerAgent.cs* like so:

```
    public override void Heuristic(in ActionBuffers actionOut)
    {
        var continuousActionsOut = actionsOut.ContinuousActions;
```

```
        continuousActionsOut[0] = Input.GetAxis("Horizontal");
        continuousActionsOut[1] = Input.GetAxis("Vertical");
    }
```

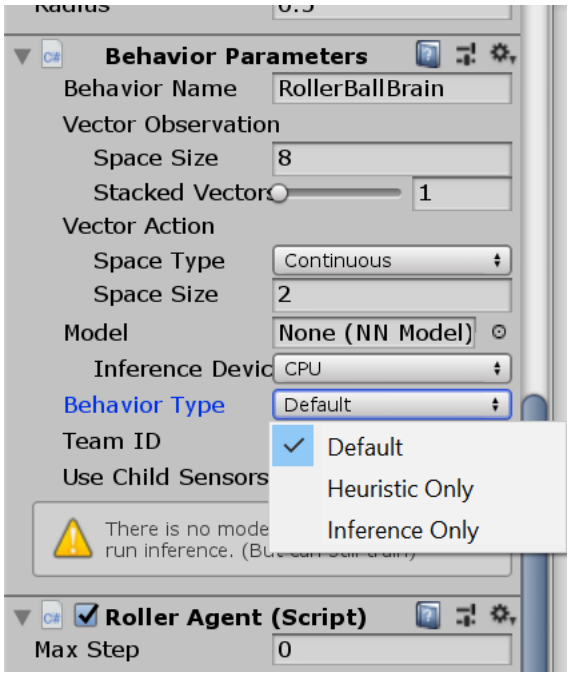
Under *RollerAgent*'s *Behavior Parameters* ensure the Behavior Type is *Heuristic Only*. To test, press Play and use arrow keys to move the *RollerAgent* around. Move towards the *Target* and see if the *RollerAgent* resets



E. Training in the Environment

Let us now train our agent in the environment!

For *RollerAgent* change the *Behavior Parameters*'s Behavior Type to *Default*



In Anaconda be sure you're in the directory that contains *ml*-agents.
In the *ml-agents* folder (that you cloned from GitHub), find the

config/ppo/Pushblock.yaml file in the config folder and copy the contents. Create another text file, paste the contents from *Pushblock.yaml* and save this new file as *config.yaml*. In the config file change *Pushblock* to *RollerBallBrain.*
Let's change some hyperparameters in our *config.yaml* file to speed up the training of this simple environment. Find the variables *batch size* and *buffer size* and change to:

```
batch_size: 10
buffer_size: 100
```

Note that this changes the training command in Anaconda. Initially we had:

mlagents-learn config\trainer_config.yaml --run-id="myFirstRun" –train

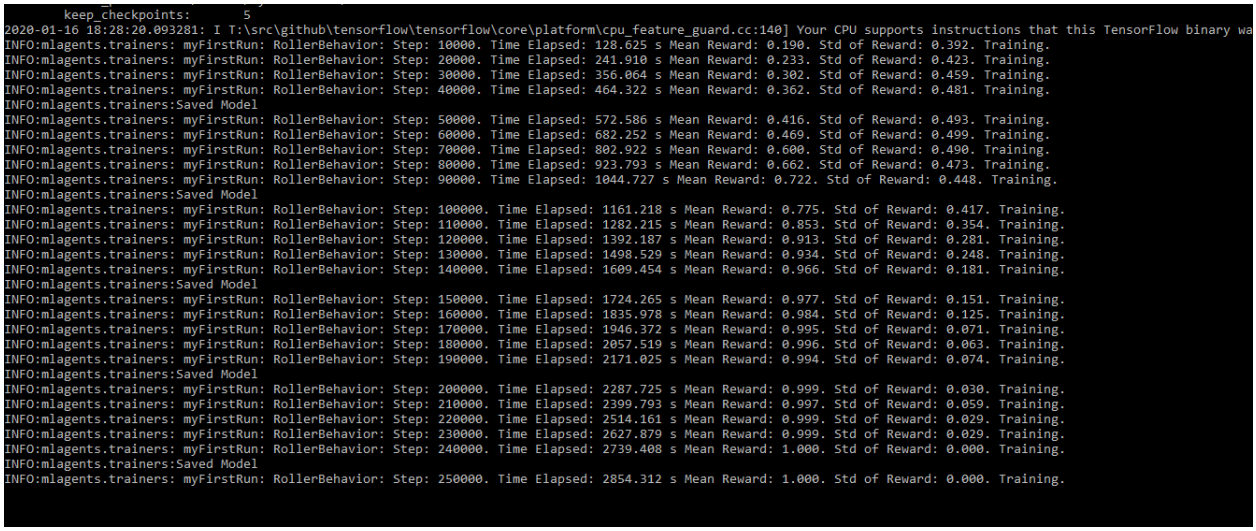
Note: There is an underscore between trainer and config

Can you guess what needs to change?
…
Replace *trainer_config.yaml* with *config.yaml*

Use your recently acquired skills to do the rest of the training. The model that you have trained will be in *ml-agents/models*.
Remember:

a) to issue the training command only when you are inside the *ml-agents* folder. Otherwise you will need to supply the absolute path to the *config.yaml* file.
b) Before training, issue *conda activate ml-agents* to activate the Python environment that contains the packages we need. Note: Python environments **are not the same** as the Unity Environment that we do the training in.

If the training is going well, your success may look as follows (on a slow pc…):

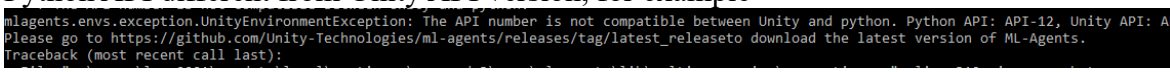


Note: The training in the screenshot above was manually stopped by pressing Ctrl + C in Anaconda

Prompt. The reason for manually stopping was that the Mean Reward had reached maximum (1) with zero Standard Deviation, which is ideally what we would want to have. In more complex environments things may never get this good. The above run was with the original *trainer_config.yaml* file (i.e. batch_size: 1024 and buffer_size: 10240)

Troubleshooting:

a. Python API different from Unity API version, for example



   Solution:
   i. uninstall and install mlagents agent by requesting Anaconda to:
      pip uninstall mlagents
      pip install mlagents
   ii. Make sure you're using the latest Unity Editor version. Sometimes it may be worth trying an older version, however.

b. Takes forever to train:
   Solution: change the hyperparameters in the *config.yaml* for example reduce the number of *hidden_units*

   This whole game is available on NTU Learn

You are now ready to take on the world with your game dev skills. All the best!

Part III: Design Your Game

Objective

The objective of this project is to assess your understanding of the course material and the ability of game programming as well as game design.

Requirement

Design a story/scenario where an agent learns to perform desired actions in an environment. You must create your own environment, and populate it with characters and GameObjects so that you are telling a fun story.
An examiner may check the following aspects of your game.

1. Inclusion and use of at least one 3D model from third parties.
2. At least one trigger in the game.
3. Use of machine learning

Prepare your project report and highlight individual's contribution of code by specifying "*//IM2073 Project*" at beginning of your code blocks and "*//End code*" at the end. The examiners will assess your code by using the search functions in Unity to find your own code. Each group has two students.
Here are some other suggestions of what you can do based on what we have learnt so far*:*

1. Create animations
2. Create multiple agents in the game who may learnt *different* behaviors
3. Create multiple training areas for the same scene

Evaluation Criteria

You can learn a lot by incorporating the following ideas in your game

1. Create a novel and interesting game story;
2. Use beautiful 3D models that make sense for the story you're telling or designing;
3. Include some new built-in functions or component which have not been taught in this project.

Part V: Game Scripts

Game scripts are available on NTU Learn. These are useful for reference, but read carefully to understand what is going on. If anything is not clear, please contact the class instructors and TAs; they would be glad to help!