Introduction to Unity
Part 1 - 2D Basics

Download 2D UFO game package here. Create a new 2D project with Unity 2019.3.X and **import** it

Adapted from :https://learn.unity.com/project/2d-ufo-tutorial

Learning Objectives for this Lab

1. Open 2D Project in Unity
2. Familiarize yourself with the editor layout
3. Arrange all your project files neatly in terms of folders: Prefabs, scripts, scenes, etc
4. Add objects to the scene
5. Getting to know how to add elements in the inspector window
6. Hooking up scripts to Objects, or other items to each element
7. RigidBody2D, Collider2D for basic Physics simulation
8. Binding Keys to simulate movements

9. Creating prefabs for reusable objects management.

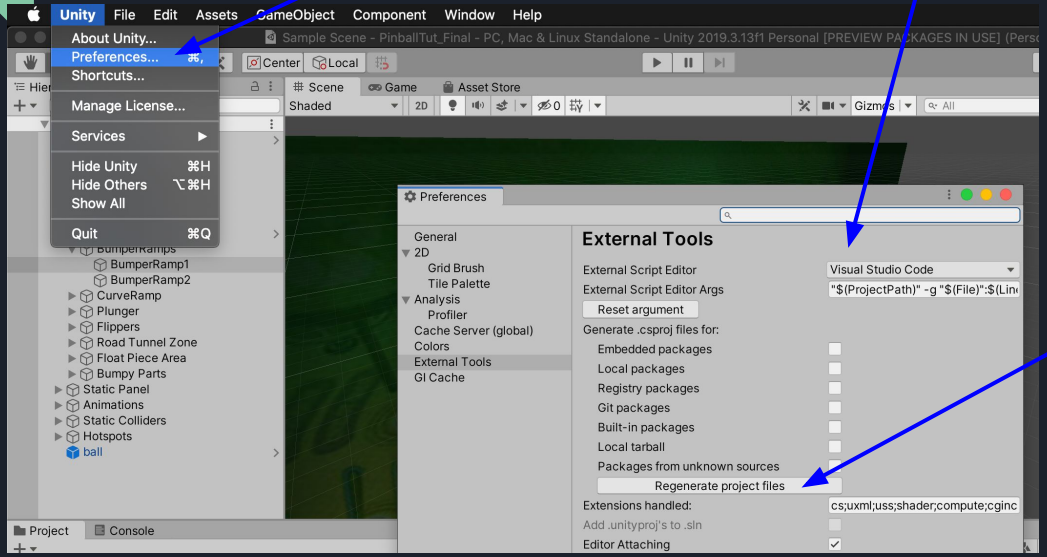
10. Adding various elements like sounds, UI elements to GameObject

11. Basic introductions to scripting and standard callback functions: update(), start(), lateUpdate(), OnTriggerEnter(), etc

12. Basic experience on *events* : OnClick() at Button

13. Creating UI Layout for players to start / pause game

Editor setting



Add your own script editor. Visual Studio Code is recommended.

See the guide : https://code.visualstudio.com/docs/other/unity

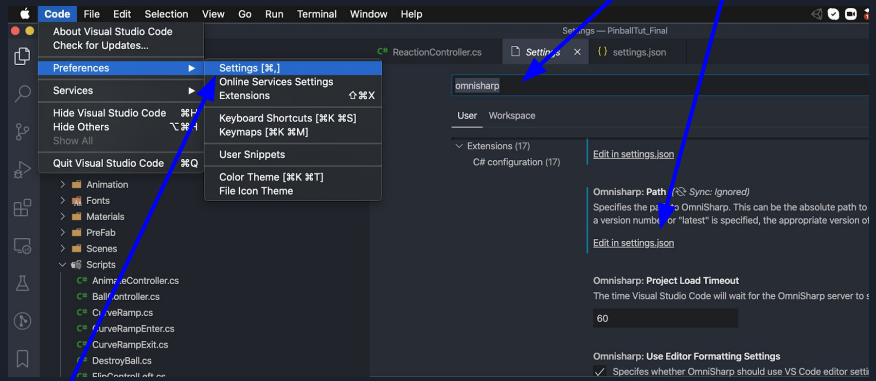
For Mac users, you need to install Mono: `brew install mono` too on top of what the guide above told you to do.

If you are NOT familiar with installing SDK/frameworks/editing .json files then just use the easy IDE Visual Studio instead (heavier, but saves you the hassle)

dotnet --info installation check (for Mac). For Windows, installation is very easy. Just follow the linked guide in the previous slides.

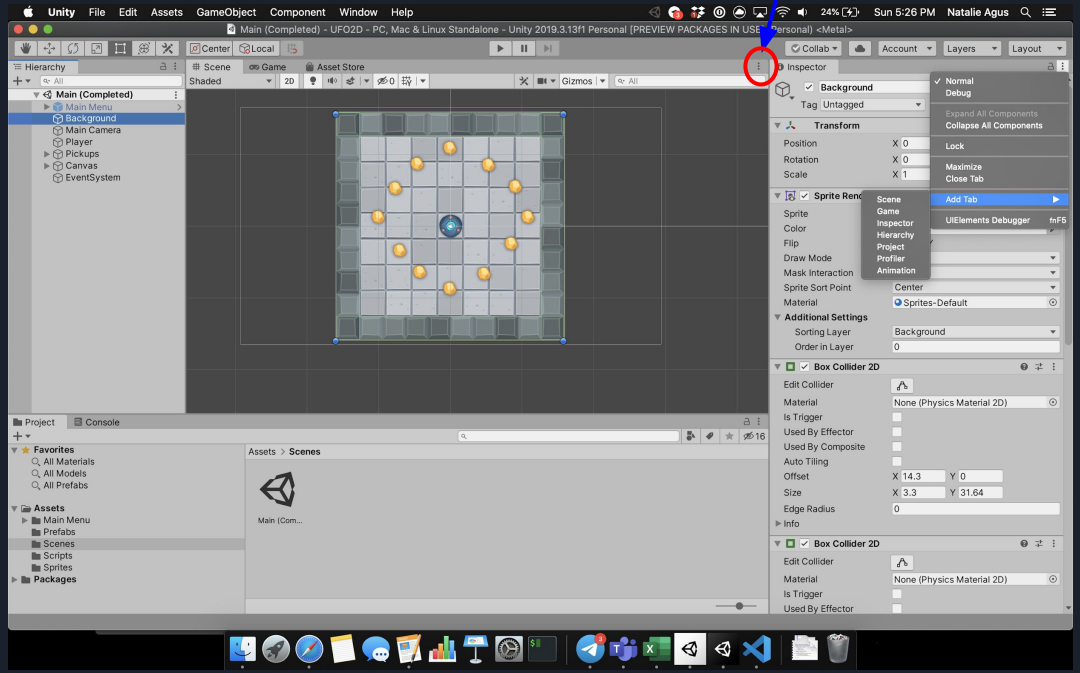
```
natalieagus@Natalies-MacBook-Pro-2 PinballTut_Final % dotnet --info
.NET Core SDK (reflecting any global.json):
 Version:   3.1.202
 Commit:    6ea70c8dca

Runtime Environment:
 OS Name:     Mac OS X
 OS Version:  10.15
 OS Platform: Darwin
 RID:         osx.10.15-x64
 Base Path:   /usr/local/share/dotnet/sdk/3.1.202/

Host (useful for support):
  Version: 3.1.4
  Commit:  0c2e69caa6

.NET Core SDKs installed:
  3.1.202 [/usr/local/share/dotnet/sdk]

.NET Core runtimes installed:
  Microsoft.AspNetCore.App 3.1.4 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 3.1.4 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]

To install additional .NET Core runtimes or SDKs:
  https://aka.ms/dotnet-download
natalieagus@Natalies-MacBook-Pro-2 PinballTut_Final %
```

Mac’s omnisharp settings.json in VSCode




```
{
    "editor.suggestSelection" :
"first",

"vsintellicode.modify.editor.sugge
stSelection" :
"automaticallyOverrodeDefaultValue
",

"latex-workshop.view.pdf.viewer" :
"tab",
    "editor.wordWrap" : "on",
    "C_Cpp.updateChannel" :
"Insiders",
    "python.jediEnabled" : false,
    "workbench.colorTheme" : "One
Monokai",
    "window.zoomLevel" : 0,

"todo-tree.tree.showScanModeButton
": false,
    "workbench.iconTheme" :
"vscode-icons",

"java.semanticHighlighting.enabled
": true,

"java.configuration.checkProjectSe
ttingsExclusions" : false,

"omnisharp.defaultLaunchSolution" :
"latest",
    "omnisharp.path" : "latest",
    "omnisharp.monoPath" : "",
    "omnisharp.useGlobalMono"  :
"always"
}
```

Introduction



Set your layouts. We need at least: *inspector, game hierarchy, our game editor, project tab, and the game screen.* **You can add other things by clicking the red circle in the screen shot, then >> add Tab**

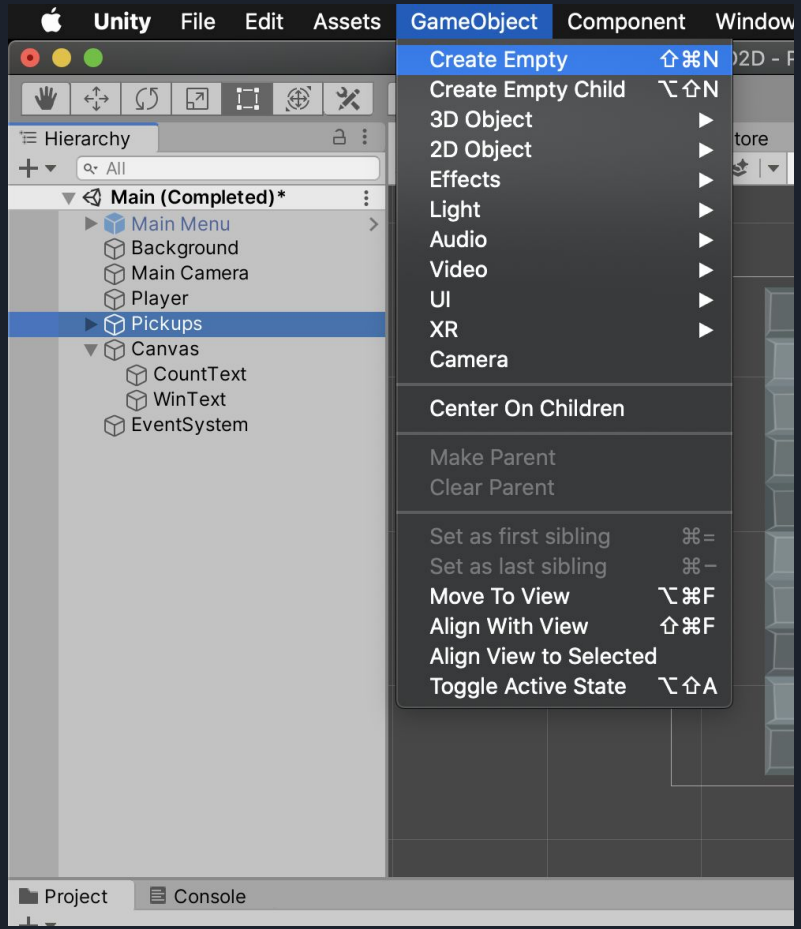
Step 1: Create hierarchical game object (for the jewel pickups)

As your first task, let's add some pickups on the scene first. The game goal is simple: steer the UFO to consume all the pickups.

To add pickup on the scene, create a "parent object" first: **Game Object >> Create Empty**

Give it a name like "Pickups". This will be the **parent** object.

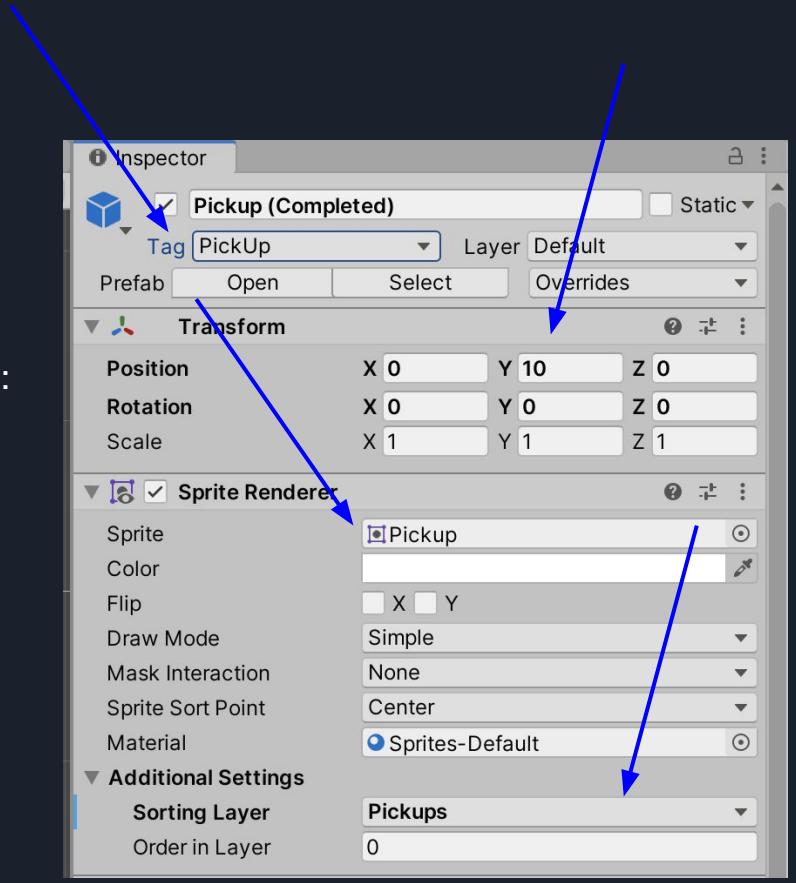
This object "cannot be seen" on the game but it acts like a "group" that will hold all the jewels.



Step 1: Create game objects (the jewel pickups)

Then, **create another game object (2D>>Sprite), set it as child under "Pickups", name it "Pickup"**:

- Place it under (0,10,0)
- Under Sprite renderer, **add the texture** named "pickup" (click the circle)
- **Add a new sorting layer,** name it "Pickups" and place it below "Background". This is so you can see a little yellow gem in your scene
- Add a **tag** to it, call it "PickUp"



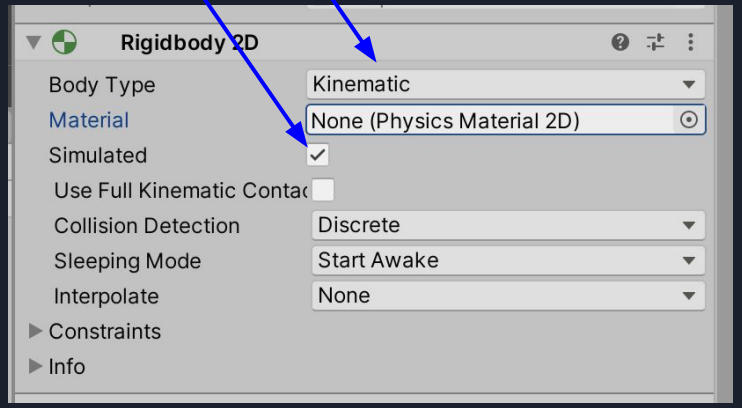
Step 2: Create RigidBody

We want to let the UFO **collides** with the pickup, so we need to set the pickup to have the RigidBody component. This way Unity can compute collision with the pickup.

Click the pickup, and at the inspector window:

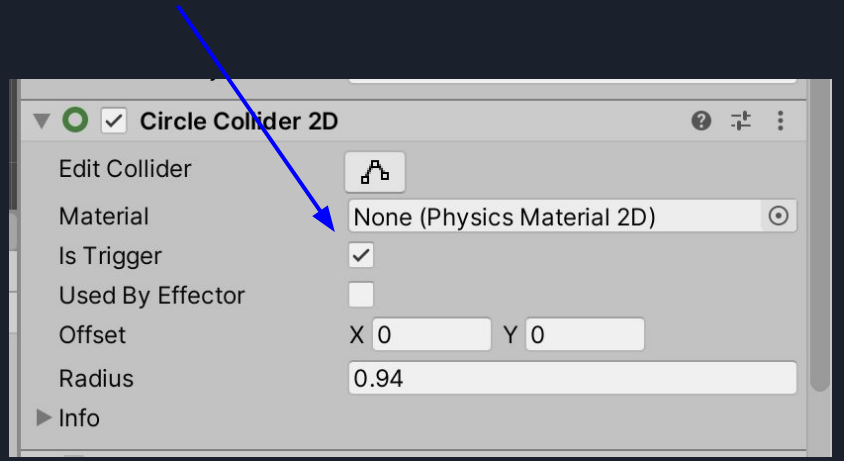
- **Add component >> RigidBody 2D, body Type: Kinematic -- allowing the body to be simulated but UNDER SPECIFIC CONTROL.** **Please read the documentation to know the other body types: dynamic and static**

- Tick the simulated box, and make sure the rest match the screenshot



Step 2: Create RigidBody + Collider

- Finally, we can **add a Circle Collider 2D** component to the sprite
- **Tick the "Is-Trigger" box,** so that we can do callback when there's collision between the player and this object, meaning that **we will handle it by script** on what will happen if some other collider collides with this object.



Step 3: Scripting to control behavior of the object

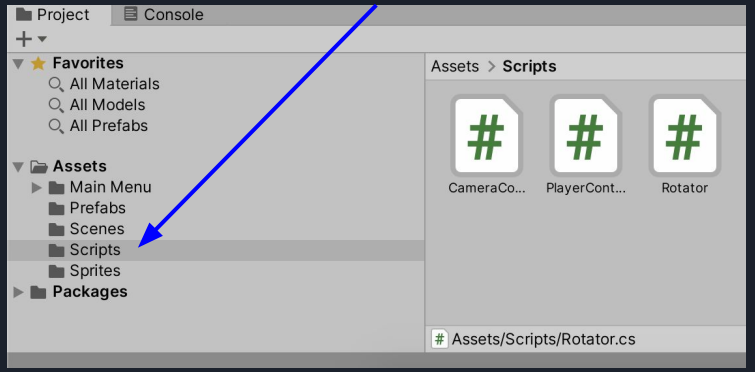
Create a new script in the script folder of the Project directory, and name it Rotator.

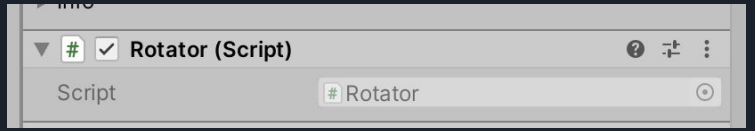
Open the script and you will see two methods there, `start()` and `update()`. To know what these are, see Unity docs on order of execution for event functions.

Under update(), **implement a line of code that will rotate the sprite by a small angle each time the update() method is called**.

Attach the script to the pickup sprite by **add component >> script**, and locate that script in your folder.

When you press play, you should see your jewels rotating.

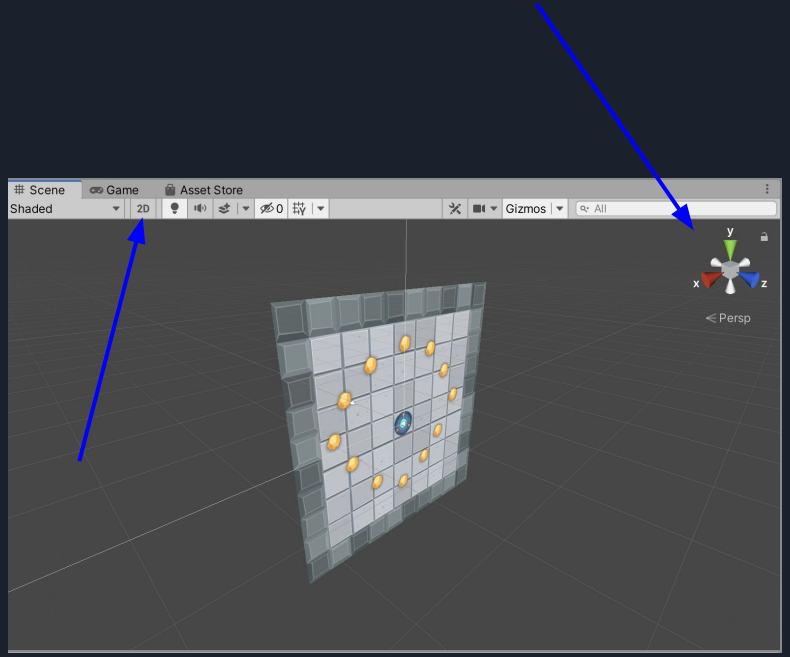




How to rotate?

- The idea is to modify “**`this`**” component that’s responsible for the orientation / position of the object : **`Transform`**
- You can use the function **`Rotate`** thats inbuilt from `Transform`
- Rotate with respect to which axis? -- In 2D games, the axis that is facing **you** (screen) is the +ve z-axis. You can toggle the “2D” button on the scene to observe the global axis.
- Therefore, you need to create a z-vector axis **`new Vector3(0,0,z)`**
- And pass it as argument to `Rotate`
- Add this code to the Update() method:

```
this.transform.Rotate(new Vector3(0,0,1.0f));
```



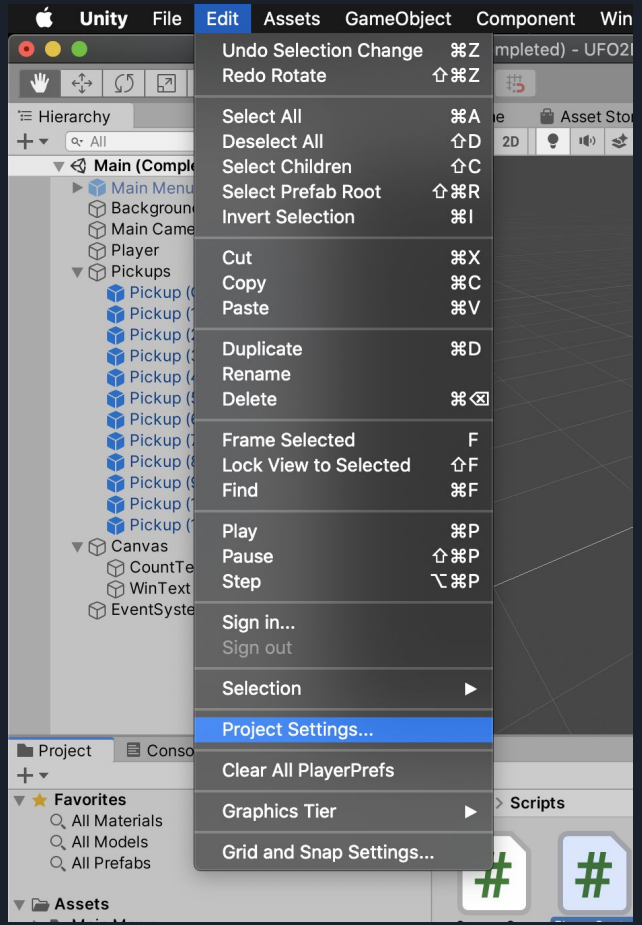
Step 4: Moving the Player

We wan’t to be able to control the UFO’s movement by ASWD buttons.

Go to Edit >> Project Settings to access Input Manager

Observe some standard axes that’s made for you: Horizontal and Vertical

Now let’s call it in the script



Step 4: Moving the Player

Go to `PlayerController.cs` script, and fill in the **`FixedUpdate`**() method to get the current horizontal and vertical input using `Input.GetAxis(AxisName)`

```
        //Store the current horizontal input in the float moveHorizontal.
        float moveHorizontal = Input.GetAxis ("Horizontal");

        //Store the current vertical input in the float moveVertical.
        float moveVertical = Input.GetAxis ("Vertical");
```

This will be the x and y forces that we will apply to the player's this RigidBody (and Unity Physics engine will take care of the rest)

```
        //Use the two store floats to create a new Vector2 variable movement.
        Vector2 movement = new Vector2 (moveHorizontal, moveVertical);
```

Step 4: Moving the Player

And then **`AddForce`** to the player's rigidbody

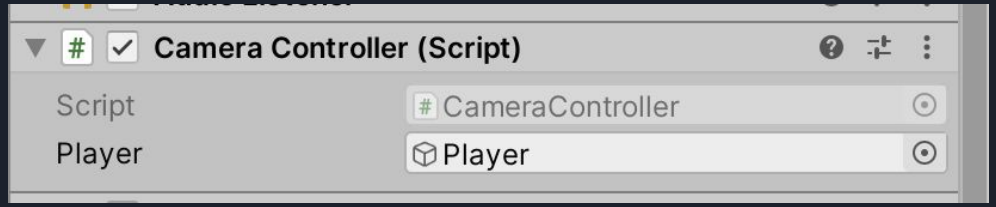
```
        //Call the AddForce function of our Rigidbody2D rb2d supplying movement multiplied by
speed to move our player.
        GetComponent<Rigidbody2D>().AddForce (movement * speed);
```

What is `speed`? It is a public variable, which means its something we can set at the Inspector of the UFO object. A convenient way to edit it.

Step 4: Moving the Player and Camera

Notice how the camera doesn't move with the player. We can move it as the player moves. Create/ Open `CameraController.cs`, and add this method below. This script should be added as component to your Main Camera, and the public variable Player should be hooked to the player gameobject (your UFO). This way we can have "access" to its component from another gameObject (this camera). Set "offset" to be the initial vector difference between player's transform and camera's transform in Start(). Then do:

```
    // LateUpdate is called after Update each frame
    void LateUpdate ()
    {
        // Set the position of the camera's transform to be the same as the player's, but offset by the
calculated offset distance.
        transform.position = player.transform.position + offset;
    }
```



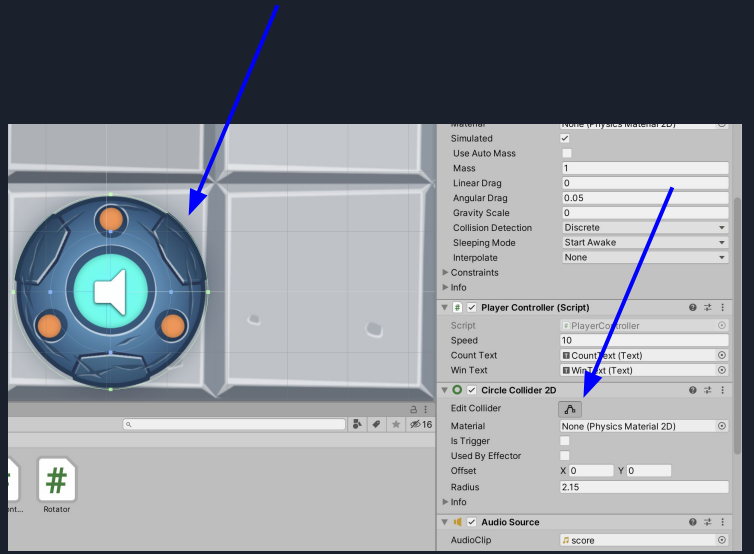
Step 5: Implement Collision trigger

Now we can move the UFO around, how do we *do something* when it collides with the Jewel?

Remember we added circle collider to the PickUp? And set IsTrigger? We can use it here.

So notice in the UFO inspector, that it also has a circle collider component. Unity Physics engine will automatically detect colliders ….colliding.

You can *edit* the bounding volume of a collider to match the *sprite* texture, so that it “fits”



Step 5: Implement Collision output

Fill up the OnTriggerEnter2D: `void OnTriggerEnter2D(Collider2D other)` method in the PlayerController. This method is an automatically triggered when another object enters a trigger collider attached to this object.

Check that if it collides with another gameObject called “PickUp” as the *tag* (that’s why we added tag to our jewel earlier), then make the jewel inactive (so it disappears from screen) -- or you can just destroy it.

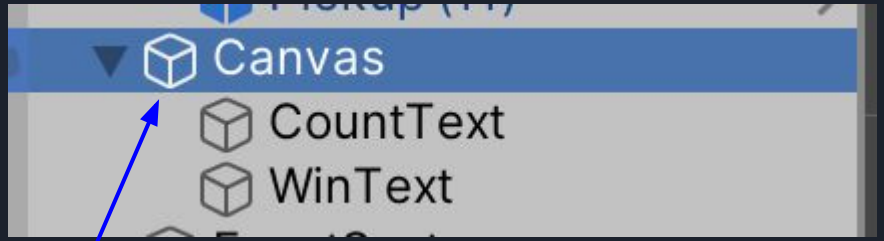
```
        //Check the provided Collider2D parameter other to see if it is tagged "PickUp", if it is...
        if (other.gameObject.CompareTag ("PickUp"))
        {
            //... then set the other object we just collided with to inactive.
            other.gameObject.SetActive(false);
            // Or destroy it… your choice
            // Destroy(other.gameObject);
        }
```

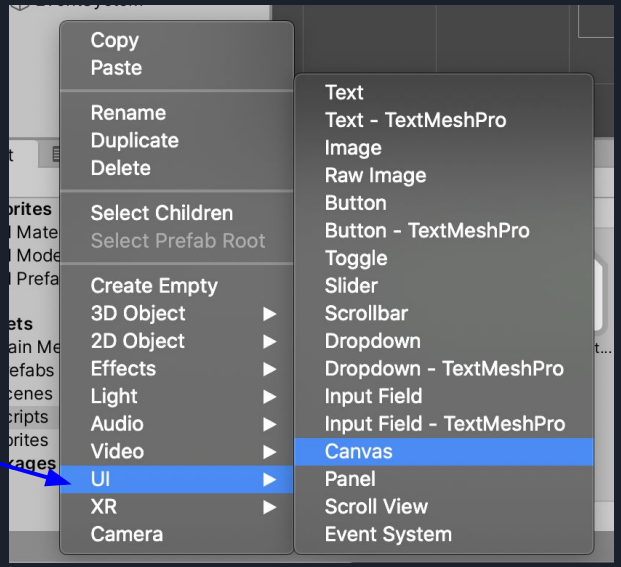
If you didn’t set the Jewel’s Circle collider to IsTrigger, then the UFO will just bump into it and….nothing happens

Step 6: Add scoring system

A game is not very fun without any scoring system. We need to display the current score as a *canvas* overlay

Create three game objects, Canvas (type UI >> Canvas) and two children called CountText (Type UI >> Text) and WinText (Type UI >> Text).

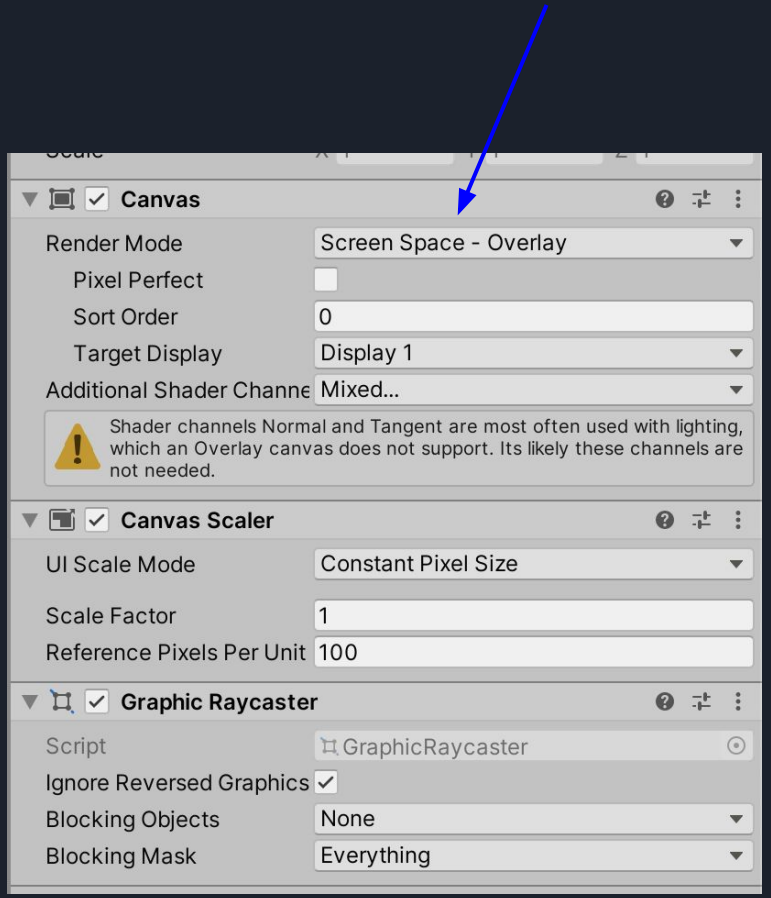




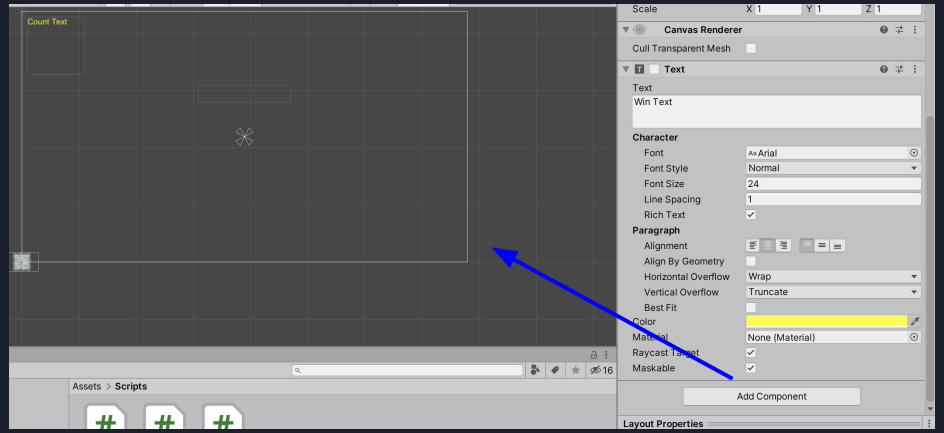
Step 6: Add scoring system

Notice in the canvas inspector we have Render Mode as Screen Space -- Overlay.

This means that the canvas “sits” on top of our view nand we don’t need to care about its location.



Step 6: Add scoring system



Click on the CountText and modify the color, font, location, etc so that it is placed at your desired location on the “canvas”. This will contain our score value.

Since CountText is the *child* object of Canvas, notice that the position is relative to the parent object, and not absolute to the world. This makes things easier to position a text. The position of the same CountT

Step 6: Add scoring system

Now we need to “hook” this UI Text to our game. Go to PlayerController.cs, and add the public variable:

```
    public Text countText; //Store a reference to the UI Text component which will display the number of pickups collected.
    private int count;              //Integer to store the number of pickups collected so far.
```

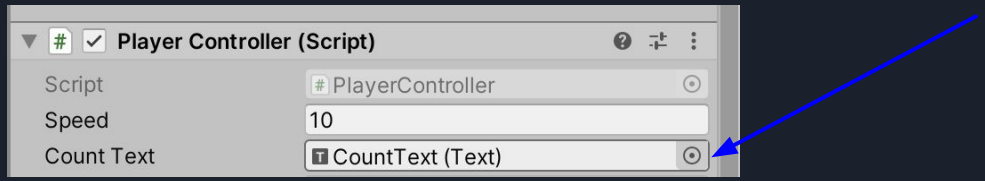
Create a method that sets this text depending on a current score

```
    //This function updates the text displaying the number of objects we've collected and displays our victory message if we've collected all of them.
    void SetCountText()
    {
        countText.text = "Count: " + count.ToString ();
    }
```

Step 6: Add scoring system

Dont forget to initialize `count = 0` in the `Start()` function.

Hook up the text into the public variable by clicking the circle button in the script section in the Player's inspector:



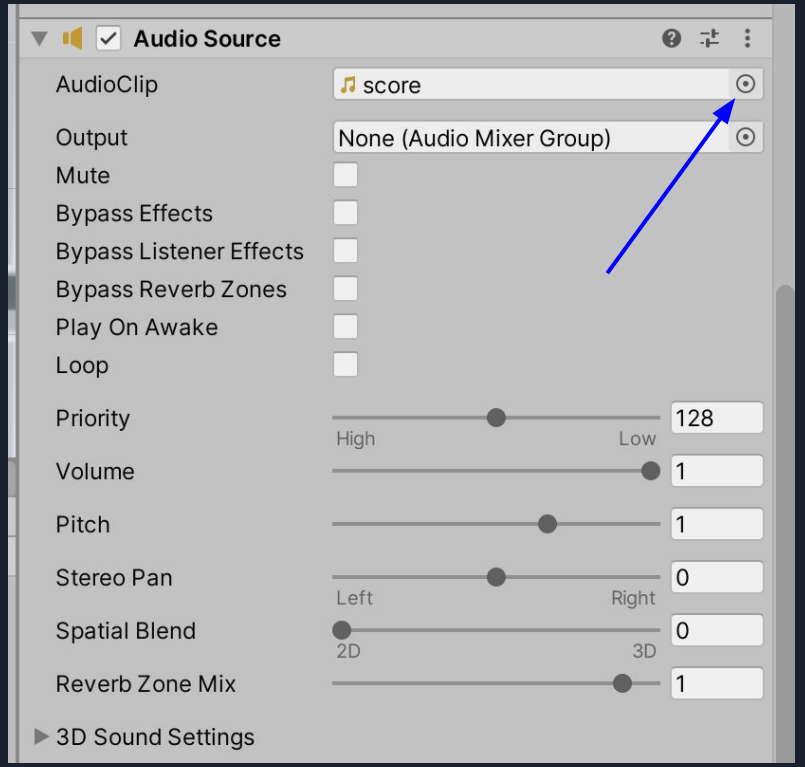
Finally, update the count variable and call the function SetCountText whenever the player collides with the Jewel inside our `OnTriggerEnter2D` function:

```
            //Add one to the current value of our count variable.
            count = count + 1;
            //Update the currently displayed count by calling the SetCountText function.
            SetCountText ();
```

Step 7: Add sound effect

We need to create an audio source component to play any sound. Create an Audio Source component on the Player's object, select the score audio clip (click the circle button again).

Who *listened* to this audio source? The answer is **our camera. Click on the Main Camera and you will find Audio Listener attached to it.**

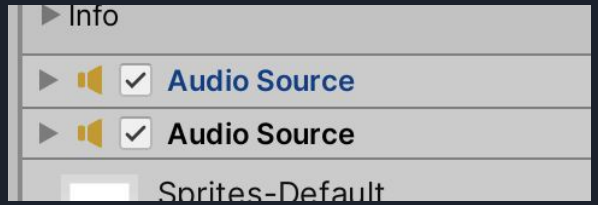


Step 7: Add sound effect

Now we want this “score” sound to play each time the player collides with the jewel. Add this code inside the `OnTriggerEnter2D` method:

```
            GetComponent<AudioSource>().Play();
```

What if we have >1 components of the same type in the Object? We can use this method instead and address the sources by *index*. The index will match the order of appearance of the components in the inspector

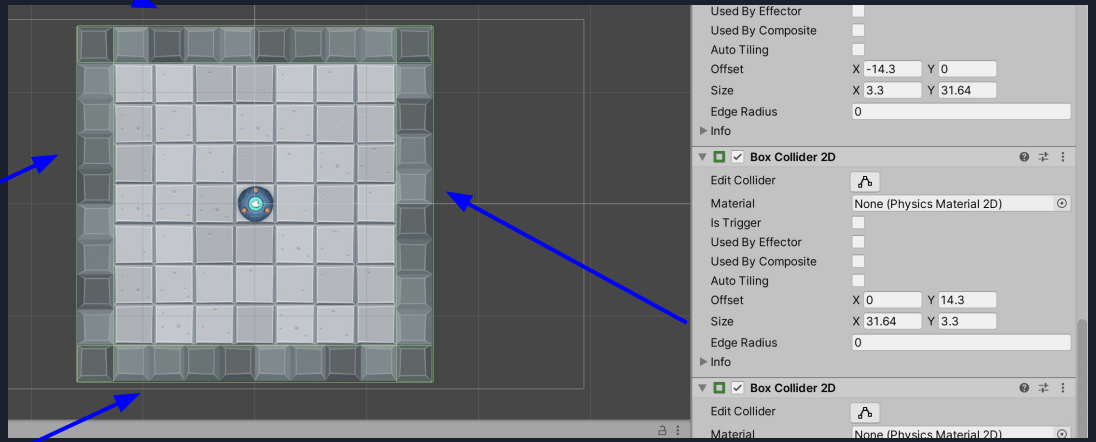



```
        AudioSource source =
GetComponents<AudioSource>()[1]; // this refers to the
bottom (second) one
```

Step 8: Bound your player

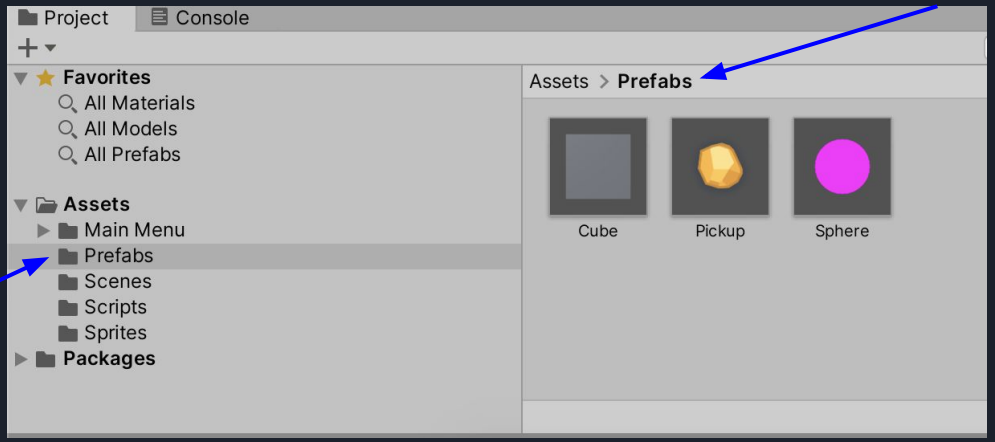
We wouldn’t want our UFO to fall out of the screen into oblivion. Click the Background Object and add **four BoxCollider2D** component.

Edit it such that the border matches the tile edges (see the faint green line around the darker tiles). Don’t make it IsTrigger. We just want the Unity PhysicsEngine to handle this and prevent the player to cross the dark edges of the background.



Step 9: Add more pickups using *prefabs*

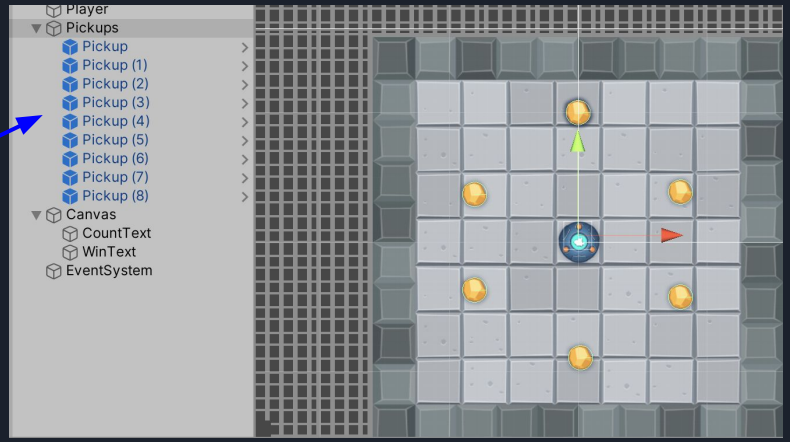
Drag the Pickup GameObject into the “Prefabs” Folder in your project. You should see a prefab automatically created like this. Prefab is basically a reusable game object. You can change the prefab master and all of your copies will reflect the change. All you need to do is create a “Prefab” Folder under Assets and drag your objects there.

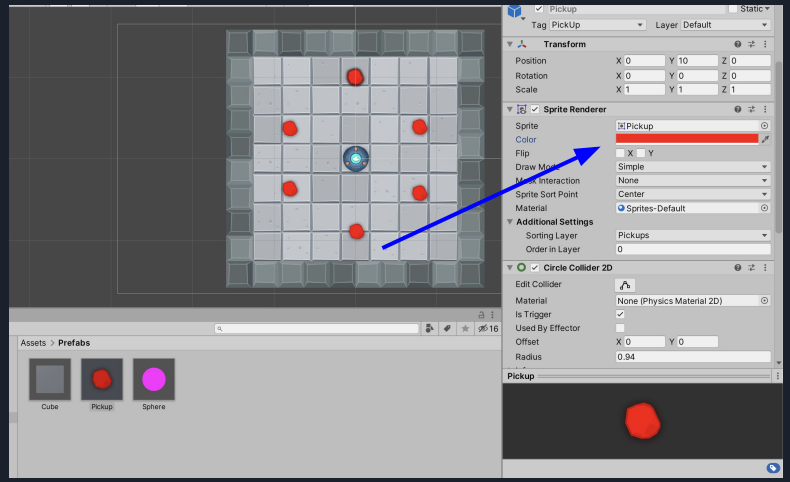


Step 9: Add more pickups using *prefabs*

A prefab is indicated by the blue cube symbol. Copy paste a few other jewels in your screen.

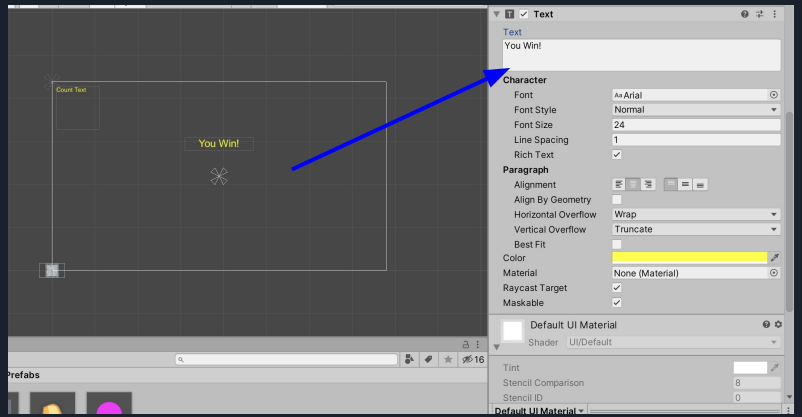
You can change the master prefab's color, and the rest will follow too, as shown on the right screenshot.





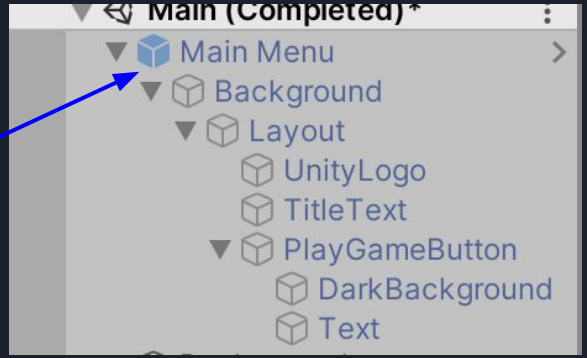
Step 10: Set the endgame condition

Some game has no end, but let's say we ate all the jewels, and we want to give some kind of indication that the game ends. Go to WinText under canvas, position it and create some kind of win text. Hook it up as a public variable in PlayerController.cs. In the start() method, set its enabled property as false (so we cant see it). Set its enabled property as true when counts of jewels reached a certain number.

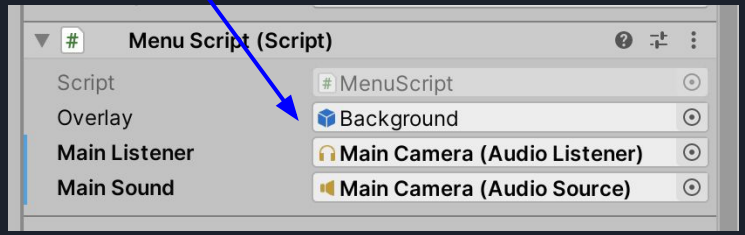


Step 11: Adding UI “Menu”

Common game has a proper “start game” menu. We can use canvas too for this. Click on the Main Menu, and tick the enabled box at the inspector so you can now see it on the scene.



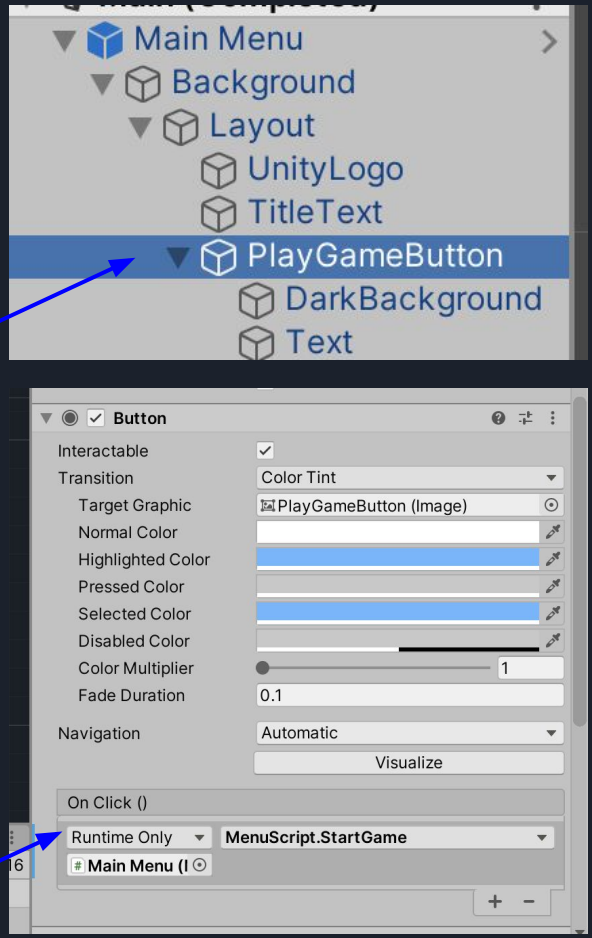
Click on the Main Menu Object and add the script: `MenuScript.cs`. Hook up the necessary variables (add AudioSource for camera with “happy” as background song)



Step 11: Adding UI “Menu”

Click on the PlayGameButton object. Under its Button component in the inspector, hook up the MenuScript.StartGame method to the OnClick() event. You need to select the script thats under the SCENE tab and not the assets tab.

This indicates that when the button is clicked, the game starts.



Step 11: Adding UI “Menu”

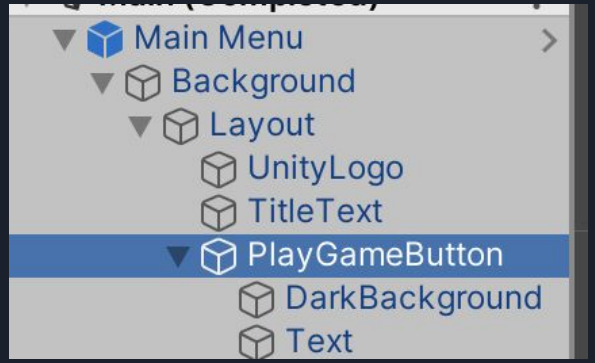
Open MenuScript.cs and look at the `Awake()` method that calls `ShowLaunchScreen()` custom method. Note the line:

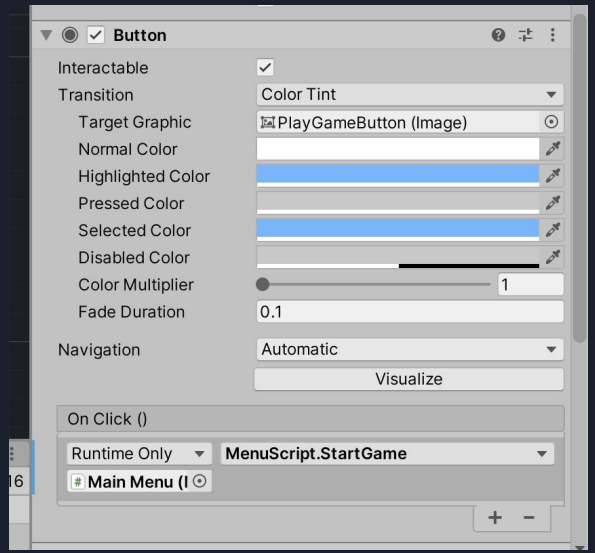
```
Time.timeScale = 0f;
```

The line above is handy to *pause a game*.

Notice you cannot move the player when the overlay is on at the beginning.

After the button is clicked, the StartGame() method is triggered, enabling camera listener, the background music, as well as setting the timeScale into 1 >> normal speed simulation.





Step 12: Adding “Pause” game

To enable some kind of “pausing” the game, add this method to `MenuScript.cs`.

```
    void Update(){

        bool escapeEntered = Input.GetKeyDown("escape");

        if (escapeEntered) ShowLaunchScreen(); // pause game

    }
```

Summary of what we have learned

1. How to create objects and place them in the scene, keeping them in proper **hierarchies**
2. **Adding elements** to the objects, i.e: RigidBody to allow for collision
3. Controlling movement of a GameObject using a script and **binding** them to keys
4. **Scripting,** i.e: detecting collision and deciding what do we do about it with **IsTrigger**, define how the gameobjects should *behave* in the game using **update()** function
5. Adding *duplicates* of game objects using ***prefabs***
6. Adding **sound effects**
7. Keeping up **scores**,
8. Setting **end-game** condition
9. Presenting **“menu”** (when game starts, when game ends)
10. Very brief view on *events* in Unity -- the onClick() event

Checkoff

Modify the UFO game to add **ONE INTERESTING FEATURE that correlates with** one of the *highlighted (informal)* rules of game design that you have learned in Week 1.

Record your screen to create a 1 minute video of you demonstrating the feature quickly and stating the rule you tried to follow.

Upload your video to Youtube (or any other online platform), and SUBMIT THE LINK. Please DO NOT submit the video directly!

See submission link for due dates.

Introduction to Unity

Part 2 - Inheritance, Properties, Events, Delegates, and Polymorphism

Material: Use your 2D UFO game (finished version) from Lab 1

Motivation

Imagine now we have like 4 different types of pickups:

1. COIN
2. GREENJEWEL
3. YELLOWJEWEL
4. GARBAGEJEWEL

And that we want each to increase or decrease the player's score. For example, player is supposed to avoid "garbageJewel" because it will penalise the current score.

How can we do this? Create prefab for each? And script for each? Place them on screen manually? Nah! Lets try to create them using scripts.

Step 1: Inheritance for common objects, enumerate types

Since each “item” has three common attributes: type, damage, and health, we can create a base class for this, and lets call it Consumable. Create a new C# script, without inheriting anything:

```
using System.Collections;

using System.Collections.Generic;
using UnityEngine;

[System.Serializable] // System.Serializable is so that the editor field appears at
// Unity during runtime and you can change the values conveniently for debugging or testing
public class Consumable
{
    public float damage;
    public float health;
}
```

To make it a little fancier, for “type”, instead of naming them 0, 1, 2, 3 for each item, we can enumerate them instead like this:

```
public enum items{
    GREENJEWEL = 0,
    YELLOWJEWEL = 1,
    GARBAGEJEWEL = 2,
    COIN = 3
}
```

Now we can declare a variable of type “items” inside Consumable.cs too:

```
public items type;
```

Create the constructor:

```
public Consumable(items type, float damage, float health){
    this.type = type;
    this.damage = damage;
    this.health = health;
}
```

Then, create two more scripts: `Jewel.cs, Coin.cs` that inherits Consumable, for example:

```
using UnityEngine;
using System.Collections;
[System.Serializable]
public class Coin : Consumable
{
}
```

And,

```
using UnityEngine;
using System.Collections;
[System.Serializable]
public class Jewels : Consumable
{
}
```

For Jewel, it is a class that should also have a list which contains all Jewels in the scene. They are going to be “spawned” randomly at runtime. Add these variables to Jewel.cs:

```
    public int amount;
    public GameObject[] jewels;
    private int index = 0;
```

The way “protected”, “private”, or “public” works is the same as Java. Note that there’s “amount” declared at the beginning that will dictate how much the size of jewels array should be later on.

And then fill in the Constructor:

```
    public Jewel(items type,
                 int amount,
                 float damage = 0,
                 float health = 0) : base (type, damage, health)
    {
        this.amount = amount;
        jewels = new GameObject[amount];
    }
```

Remember Jewel inherits Consumable so it also has “damage” and “health”. It also must invoke the Consumable’s constructor, and this can be done using `base` keyword.

Look at how `optional arguments` are added, similar to other languages.

The same steps should be done for Coin.cs. To make things simple, let's say we just want to have one Coin, and coin has a new variable called `cost`. We will use it later in the game.

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

[System.Serializable]
public class Coin : Consumable
{
    public int amount = 1;
    public int cost;
    protected GameObject coin;

    public Coin(items type, int cost, float health = 0) : base(type, 0, health)
    {
        this.cost = cost;
    }
}
```

Step 2: Using Properties

Notice how we have to make some variables public if we want to access it from other classes? We can improve our code by using properties instead -- that is other classes can “read” its value, but only itself can “set” it. You know this certainly, it’s called *getter* and *setter*.

We can replace it as such -- with public get and private set for the cost variable inside coin.

__cost is the private variable, and cost is a property.

```
// property getter & setter
private int __cost;
public int cost{
    get{
        return __cost;
    }
    private set{
        if (value > 0){
            __cost = value;
        }
    }
}
```

Step 2: Using Properties

You can also use *autoproperties*. The only caveat with this is that unlike normal properties in the previous slide, you cannot initialize it when you declared it. You need to initialize it later in the constructor instead. Change Jewel.cs amount variable to property like this too.

```
public int amount{get ; private set;}
```

For variables in Consumable.cs:

```
public items type{get; private set;}
public float damage{ get; private set;}
public float health{ get; private set;}
```

Is it necessary to do this? Well, your game will *work just fine* without these and setting all variables into public, but this is just ***good coding habit.***

Step 3: Spawning from script, creating a ‘Manager’

It is good to have a script that “manages” all consumables in your screen, so that you don’t have to manually place them all on the screen. It also makes your code more flexible, and neat.

Create a new script called ItemManager.cs inheriting the normal MonoBehavior class. We are going to attach this script to one of the objects in the scene.

Then create all these variables.

```
    //items variables, remember all of
them are 'Consumable'
    private Jewel greenJewel;
    private Jewel yellowJewel;
    private Jewel garbageJewel;
    private Coin coin;

    //prefabs variables
    public GameObject Background;
    public GameObject CoinPrefab;
    public GameObject GreenJewelPrefab;
    public GameObject YellowJewelPrefab;
    public GameObject GarbageJewelPrefab;
    private float rangeX, rangeY;
```

Step 3: Spawning from script, creating a ‘Manager’

Under Start(), create all the Jewels and Coin:

```
        //create instances for jour jewels and coins
        greenJewel = new Jewel(Consumable.items.GREENJEWEL, Random.Range(1, 10), 0 ,5);
        yellowJewel = new Jewel(Consumable.items.YELLOWJEWEL, Random.Range(1, 5), 5, 10);
        garbageJewel = new Jewel(Consumable.items.GARBAGEJEWEL, Random.Range(1, 3), 10, 0);
        coin = new Coin(Consumable.items.COIN, 100, 20); //cost is 100, health is 20

        //get the background
        rangeX = Background.GetComponent<SpriteRenderer>().sprite.bounds.extents.x*0.8f;
        rangeY = Background.GetComponent<SpriteRenderer>().sprite.bounds.extents.y*0.8f;
```

At this point, we are simply creating the classes to handle three types of Jewels and one Coin.
We need to write another method to actually **Instantiate** the prefab.

Step 3: Spawning from script, creating a ‘Manager’

Add this method inside ItemManager.cs:

```
   private void instantiate(){
       //instantiate
       for (int i = 0; i<greenJewel.amount; i++){
           greenJewel.add(Instantiate(GreenJewelPrefab, new Vector2(Random.Range(-rangeX, rangeX),
Random.Range(-rangeY, rangeY)), Quaternion.identity));
       }
       //instantiate
       for (int i = 0; i<yellowJewel.amount; i++){
           yellowJewel.add(Instantiate(YellowJewelPrefab, new Vector2(Random.Range(-rangeX, rangeX),
Random.Range(-rangeY, rangeY)), Quaternion.identity));
       }
       //instantiate
       for (int i = 0; i<garbageJewel.amount; i++){
           garbageJewel.add(Instantiate(GarbageJewelPrefab, new Vector2(Random.Range(-rangeX,
rangeX), Random.Range(-rangeY, rangeY)), Quaternion.identity));
       }

       coin.create(Instantiate(CoinPrefab, new Vector2(Random.Range(-rangeX, rangeX),
Random.Range(-rangeY, rangeY)), Quaternion.identity));

   }
```

Step 3: Spawning from script, creating a ‘Manager’

Add this method inside Coin.cs:

```
    public void create(GameObject coin){
        coin.GetComponent<BasicObjectScript>().type = type;
        coin.GetComponent<BasicObjectScript>().index = 0;
        this.coin = coin;
    }
```

Add this method inside Jewel.cs:

```
    public void add(GameObject jewel){
        if (index < amount){
            //set its index
            jewel.GetComponent<BasicObjectScript>().index = index;
            jewel.GetComponent<BasicObjectScript>().type = type;
            jewels[index] = jewel;
            index ++;
        }

    }
```

Note: you might need to comment out lines with BasicObjectScript first to run at this point

Step 3: Spawning from script, creating a ‘Manager’

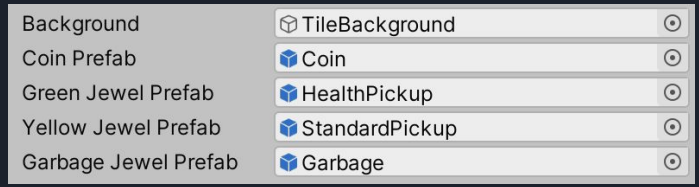
Then call instantiate() in Start() inside ItemManager.cs:

```
        //create instances for jour jewels and coins
        greenJewel = new Jewel(Consumable.items.GREENJEWEL, Random.Range(1, 10), 0 ,5);
        yellowJewel = new Jewel(Consumable.items.YELLOWJEWEL, Random.Range(1, 5), 5, 10);
        garbageJewel = new Jewel(Consumable.items.GARBAGEJEWEL, Random.Range(1, 3), 10, 0);
        coin = new Coin(Consumable.items.COIN, 30, 20); //cost is 30, health is 20

        //get the background
        rangeX = Background.GetComponent<SpriteRenderer>().sprite.bounds.extents.x*0.8f;
        rangeY = Background.GetComponent<SpriteRenderer>().sprite.bounds.extents.y*0.8f;

        //initialise all objects
        instantiate();
```

Step 3: Spawning from script, creating a ‘Manager’

Test:

1. Create an empty game object, name it PickupManager and attach the ItemManager.cs script to it.
2. Create the new Jewel prefabs however way you want it. Just copy the existing Pickup prefab and modify the colors to signify different Jewels, change the texture to make a Coin. So each pickup must have: Rotator.cs, RigidBody2D (simulated ON, Kinematic), and CircleCollider2D (isTrigger ON), tag set to PickUp
3. Hook up the items from the prefab to the script. Run the game, you should see your score +1 each time it gets any pickup type



Step 4: Triggering Events using Delegates

Now if we want to update the scores with a slightly more complicated rule:

1. Garbage Jewel will -20 of the score * speed multiplier
2. Yellow Jewel will +10 of the score if the UFO is almost at rest when hitting it, otherwise it the score will be updated as = 10 - 5 * speed multipler
3. Green Jewel will +5 of the score
4. Coin will +20 of the score IF the current score is > __cost

We can of course naively implement this inside the OnTriggerEnter2D inside the PlayerController.cs. But we need to have **different tags** for different prefab, and lots of if-elses to implement them all. Not to mention we need to set them to be inactive each time its triggered with the player. We can do this in a neater and more organised way using events and delegates.

What is a Delegate?

Delegate is just a variable that can be assigned to a method. It is useful for something called multicast. We can declare a delegate like a method, then assign a variable to it. Add this inside ItemManager.cs:

```
    //Create delegate. Any method that has the same signature as this can be assigned to this
delegate.
    //Delegates are useful for multicast
    public delegate void ActionHit(Consumable.items type, int index);
    //Common usage of delegate, by assigning a variable
    public ActionHit actionHit;
```

So ActionHit is the delegate, and actionHit is the *variable*. We can assign any method to actionHit as long as the method has the **SIGNATURE** of the delegate: returns `void`, accepts 2 arguments, first one is of type `Consumable.items`, and the other is `int`.

Events and Delegate

Given the following delegate:

```
public delegate void ActionHit(int index, float damage);
public ActionHit action_hit;
```

Which of the following methods can be assigned to this delegate?

YES

```
public void method1(int index, float damage, int cost = 0){
    // some code here
}
```

```
public int method2(int index, float damage){
    // some code here
}
```

YES

```
public void method3(int index, float damage){
    // some code here
}
```

```
public void method4(void){
    // some code here
}
```

What is a Delegate?

Lets create dummy methods inside ItemManager.cs which signature matches the delegate:

```
public void hit1(Consumable.items type, int index){
    Debug.Log("hit1 called");
}

public void hit2(Consumable.items type, int index){
    Debug.Log("hit2 called");
}
```

Then, under Start() add these lines:

```
// quick delegate example
// assigning methods with the same signature to the delegate variable
actionHit = hit1;
actionHit += hit2;

// cast!
actionHit(Consumable.items.COIN, 2);
```

Basically, we add hit1 and hit2 into actionHit variable, and we can call actionHit like a normal method. This will cast hit1 and hit2 both! -- hence *multicast*. Very convenient.

What is an Event?

An event is a *specialised type* of delegate variable. So instead of creating a variable actionHit to cast the methods, we create an event instead.

```
public static event ActionHit onHit;
```

1. We make the event *static* so that other classes can access it using the classname: ItemManager.
2. Classes can *subscribe or unsubscribe* to this event
3. However, other classes CANNOT invoke it, e.g calling onHit(Consumable.items.COINS, 2)
4. Only this class (the class where the event is declared, i.e: ItemManager) can invoke it

This provides *protection* as opposed to vanilla delegate variable actionHit in the previous slides, as anybody with the access to the delegate variable can invoke it.

Subscribe to onHit event

Inside Jewel.cs constructor and Coin.cs constructor, subscribe to the onHit event:

```
        ItemManager.onHit += hit;
```

Inside the Coin.cs, of course write the method hit that fits the ActionHit **signature**:

```
    //the method to be assigned to the
delegate
    public  void hit(items type, int index){
        if (type == this.type){
            coin.SetActive(false);
            amount = 0;
        }
    }
```

Do the same to Jewel.cs, add this code:

```
    public void hit(items type, int index){
        if (type == this.type){
            jewels[index].SetActive(false);
//make the jewel inactive
            amount --;
        }
    }
```

Write a method that invoke onHit event inside ItemManager.cs

It makes sense for the ItemManager to be the one keeping the scores, as it knows how many items are alive on your screen. So lets do some refactoring, add the following variables in ItemManager.cs:

```
    public int score{get; private set;} //properties that can be read by others but not set

    public Text countText;          //Store a reference to the UI Text component which will display the number of pickups collected.
    public Text winText;            //Store a reference to the UI Text component which will display the 'You win' message.
```

Add the `SetCountText` method in ItemManager.cs like the one we have in PlayerController.cs:

```
    void SetCountText()
    {
        //Set the text property of our our countText object to "Count: " followed by the number
stored in our count variable.
        countText.text = "Count: " + score.ToString();

        if (score >= 50){
            //... then set the text property of our winText object to "You win!"
            // winText.text = "You win!";
            winText.enabled = true;
            // stop time
            Time.timeScale = 0;
            // Application.Quit();
        }
        //if there's nothing else to eat
        if (coin.amount == 0 && greenJewel.amount == 0 && yellowJewel.amount == 0 &
garbageJewel.amount == 0){
            winText.text = "You Lose!";
            winText.enabled = true;
            Time.timeScale = 0;
        }

    }
```

Write a method that invoke onHit event inside ItemManager.cs

Now we are all set for writing a method that will invoke onHit. Lets create a method called consumablesHit in ItemManager.cs that does the following:

```
public void consumablesHit(float speed, Consumable.items type, int index){
    Debug.Log("consumables Hit!" + " of type " + type.ToString());
    if (onHit != null){ //check if the event has any listeners
        onHit(type, index); //if yes, call them
    }
```

1. So the Debug is just there for sanity check.
2. When this method is called, it will check if the event has any listeners (which we know it has since the Jewels and Coin constructors are called when you instantiate them in Start, but this is good to prevent errors during runtime *in case*).
3. Then ItemManager invoke the event! This will trigger the hit() methods in the Jewel classes and Coin class

Write a method that invoke onHit event inside ItemManager.cs

ItemManager also manages your score in this scenario. So, we can add this code inside consumablesHit method too:

```
        //update score based on the types
        if (type == Consumable.items.COIN){
            if (score >= coin.cost)
                score += (int) coin.health;
            else score -= coin.cost;
        }
        else if (type == Consumable.items.GREENJEWEL){
            score += Mathf.FloorToInt(Mathf.Max(greenJewel.health,0));
        }
        else if (type == Consumable.items.YELLOWJEWEL){
            score += Mathf.FloorToInt(Mathf.Max(yellowJewel.health - speed * 0.1f *
yellowJewel.damage,0));
        }
        else if (type == Consumable.items.GARBAGEJEWEL){
            score += Mathf.FloorToInt(-1 *speed * garbageJewel.damage);
        }

        SetCountText();
```

Step 5: Polymorphism using Interface

We are not quite there yet. There has to be someone who calls consumablesHit inside ItemManager right, so that consumablesHit can trigger the event. This “someone” is none other than our pickups. Let’s do this by implementing an **interface to our pickup script**. You can only inherit ONE class at a time, but you can implement multiple interface.

Create a script BasicObjectInterface:

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

interface BasicObjectInterface{
    void getHit(float speed);
}
```

Step 5: Polymorphism using Interface

Create another script called BasicObjectScript and attach it to ALL four prefabs inside the prefab folder. This will give them the same *base type* (sort of)... so PlayerController.cs doesn't have to deal with so many tags anymore.

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

public class BasicObjectScript : MonoBehaviour, BasicObjectInterface
{
    public int index; // the object index inside Jewels' list, for coin this is just 0
    public Consumable.items type;
    public GameObject itemManager;

    //this is the method that one has to implement (because of the adopted BasicObjectInterface)
    public void getHit(float speed){
        itemManager = GameObject.FindGameObjectWithTag("Manager");
        // call the ItemManager's consumables hit method, and the item manager will notify
everybody
        itemManager.GetComponent<ItemManager>().consumablesHit(speed, type, index);

    }
}
```

Step 5: Polymorphism using Interface

Modify the OnTriggerEnter2D method inside PlayerController.cs to be the following:

```
    void OnTriggerEnter2D(Collider2D other)
    {
        BasicObjectInterface obj = other.GetComponent<BasicObjectInterface>();

        if (obj != null){
            //notify
            obj.getHit(rb2d.velocity.magnitude);
            source.Play();
            // SetCountText();
        }
}
```

So no matter whether we hit green/yellow/garbage jewel, or coin, we just invoke the same method called ***hit*** in the BasicObjectScript component inside all prefabs. Its neater than the version before!

Bonus: Script Execution Order

The execution order for event functions within a script in Unity is deterministic, so you won't suffer from race condition. However, if you have more than 1 script *modifying the same object at runtime*, you might not know which script executes first.

You can set it at: `Edit >> Project Settings >> Script Execution Order`

Test it:

1. Create two gameobjects on the scene: `colorchange1, colorchange2`
2. Create two scripts, each changing the color of the player to Color.green and to Color.red
3. Attach each script to each game objects created in (1)
4. Set the order in Project settings and run the program. Switch the order and run again. Do you see the color of the player change to be the whatever color that the last script set?

Summary

Now when you run the game, this is what happens:

1. When player moves and collides with any jewels / coin, since they are Triggers, `OnTriggerEnter2D` method of PlayerController is invoked (actually the the `OnTriggerEnter2D` methods of any scripts in each prefab are also invoked, but we don't implement that so nothing happens!)
2. This checks if it collides with *other* that implements `BasicObjectInterface`, if yes, call the `getHit` method
3. The `getHit` method of each prefab (of each `BasicObjectScript` component in each prefab actually...) calls `consumablesHit` method of `ItemManager`, and pass its two arguments: `type` and `index` -- set during `create()` in Coin.cs and `add()` in Jewels.cs
4. The `consumablesHit` method of `ItemManager` invoke the event `onHit` to all subscribers (the 3 types of Jewels classes and Coin), and the correct type will render itself invisible
5. The `consumablesHit` method also deals with the scoring update and set the count text.

CHECKOFF

1. Create a new type of pickup with a new score rule. Give good variable names or document it properly so that the code is readable.
2. It has to subscribe to the onHit event and must inherit Consumable.cs
3. Edit Consumable.cs enum type to have one more type of pickup
4. The ItemManager must update score accordingly with this pickup type

Create a 1 minute video showcasing all these tasks. You can just do a simple voiceover when recording your screen. Upload it on Youtube/any other cloud services and upload the LINK on the edimension.

Introduction to Unity Part 3 - Animation

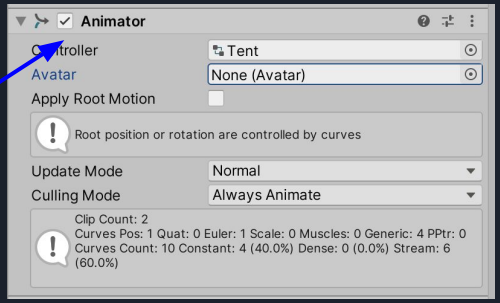
Lab 3 - Week 1

Learning Objectives for this Lab

1. Creating animator and animation clip, attaching it to GameObject
2. Transition between animations
3. Creating parameters and using it to trigger transition
4. Changing animation framerates
5. StateMachineBehavior scripting
6. Timing animation and creating events → to call a function when a particular event happens during the animation

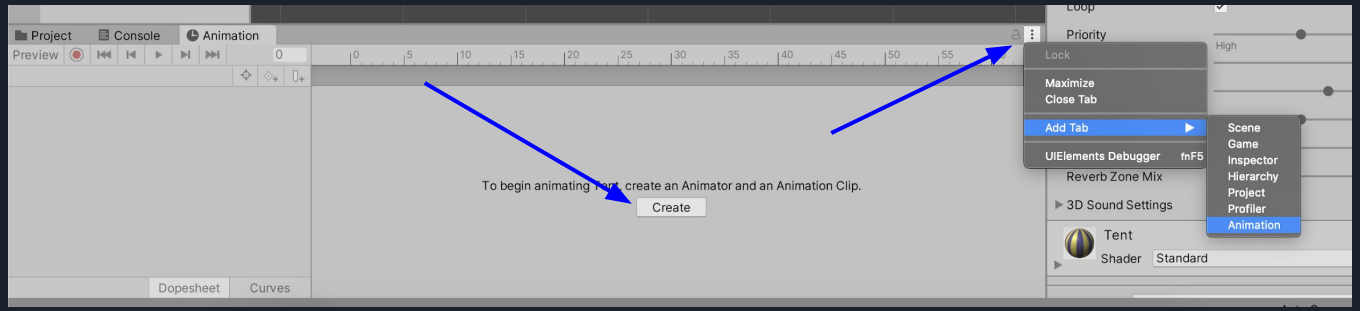
Create a new 3D project.
Download the asset from here and import it.

Step 1: Creating an Animator



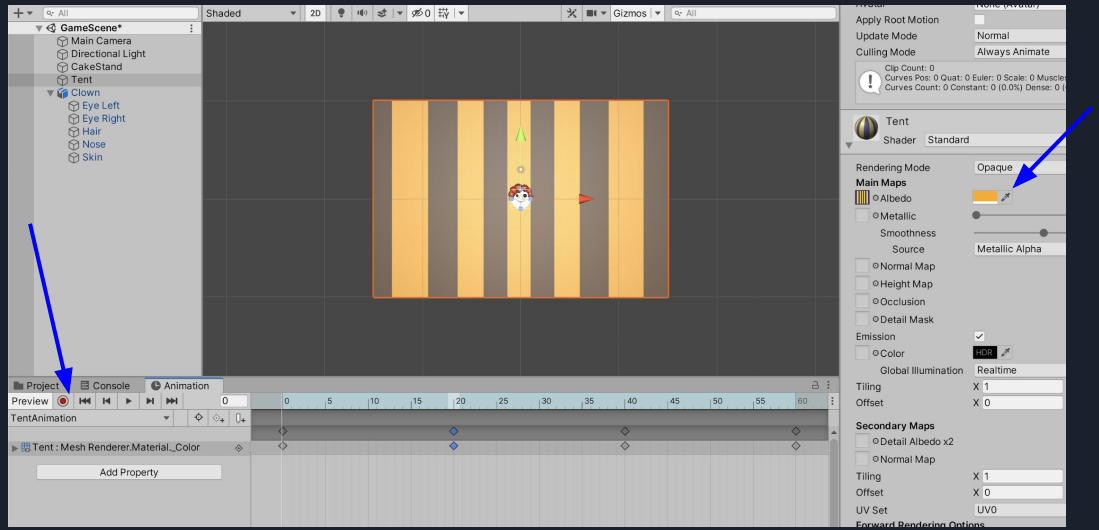
The goal of this lesson is to introduce you into a new element called *animator*, where you can create *animation clips* and play them in any order that you want. Open the GameScene.

Then you need to first open the Animation Tab. Then, click on any object on the scene. In this example, “Tent” is clicked. We can begin creating a Tent Animator. Click “Create” button below. This will add an animator component to your object like the screenshot in upper right hand corner.



Step 2: Creating Animation Clips

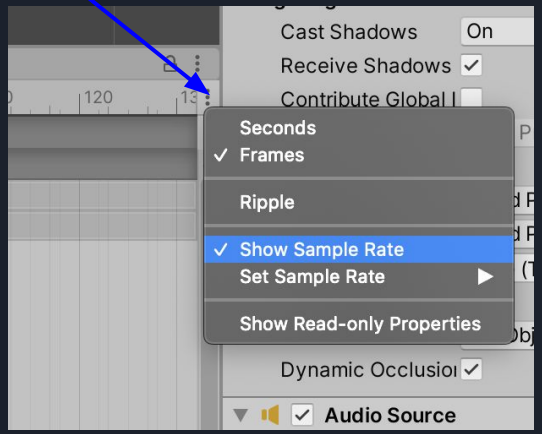
After clicking create, and a name for your first *clip*, you can then begin recording animation by clicking the record button and make changes to your tent in the scene. Click on the dopesheet do determine the time frame you want that new change to happen. For example, we change the color a few times here. Your dopesheet should look something like this:

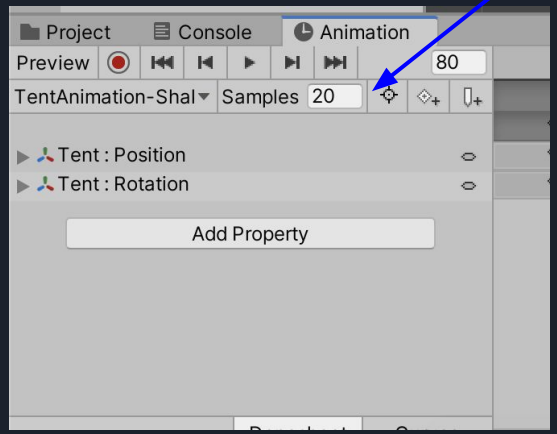


Note: the tent mistakenly had remains of old animator component from my answer copy. Please REMOVE it first from the Tent inspector before beginning. Unity need to link the animator with the gameobject instance at runtime.

Step 2: Creating Animation Clips

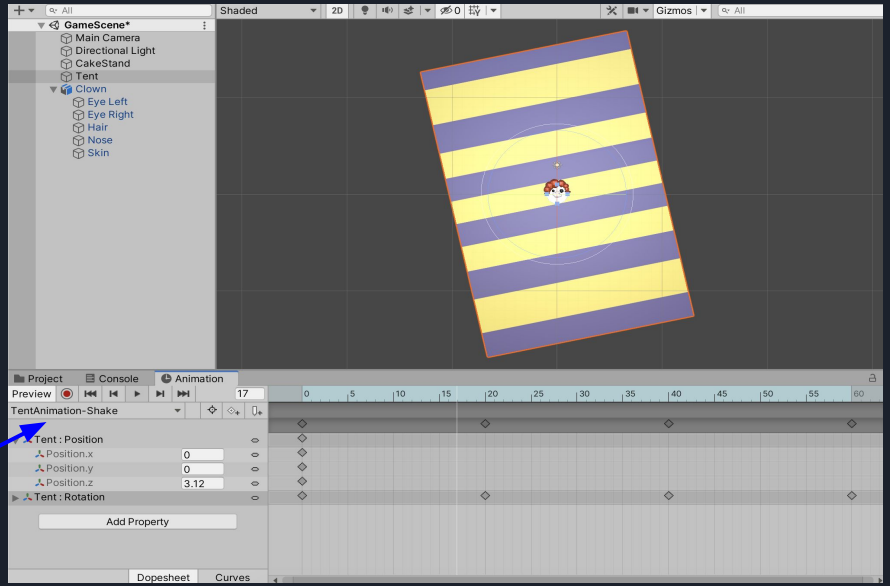
You can choose to show sample rate and set it to a value that you want. This shows the number of *frames per second*. Of course the lower the number you put, the slower your animation is.

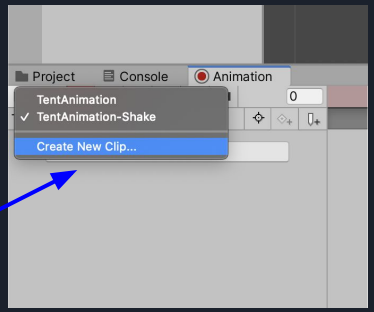




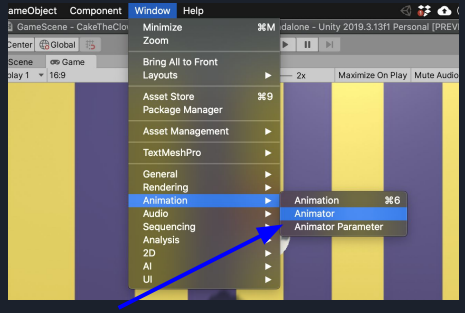
Step 2: Creating Animation Clips

You can create more clips and record more animation. In this example, we rotate the background around.



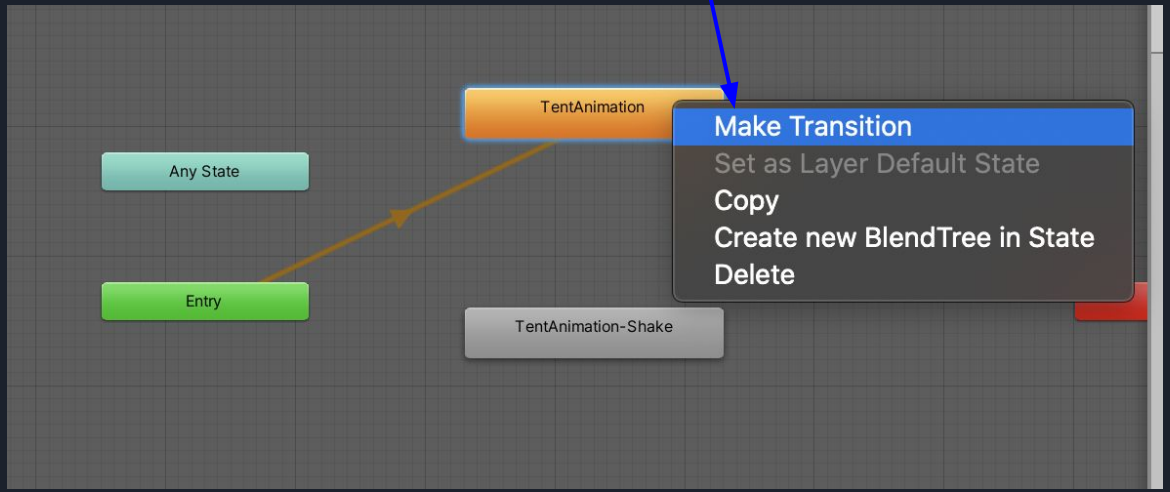


Step 3: Managing Animation Clips



Open the Animator window to now manage the two clips you created. Make a transition back and forth between the two clips. This is basically a state machine, where we define our transitions / triggers.

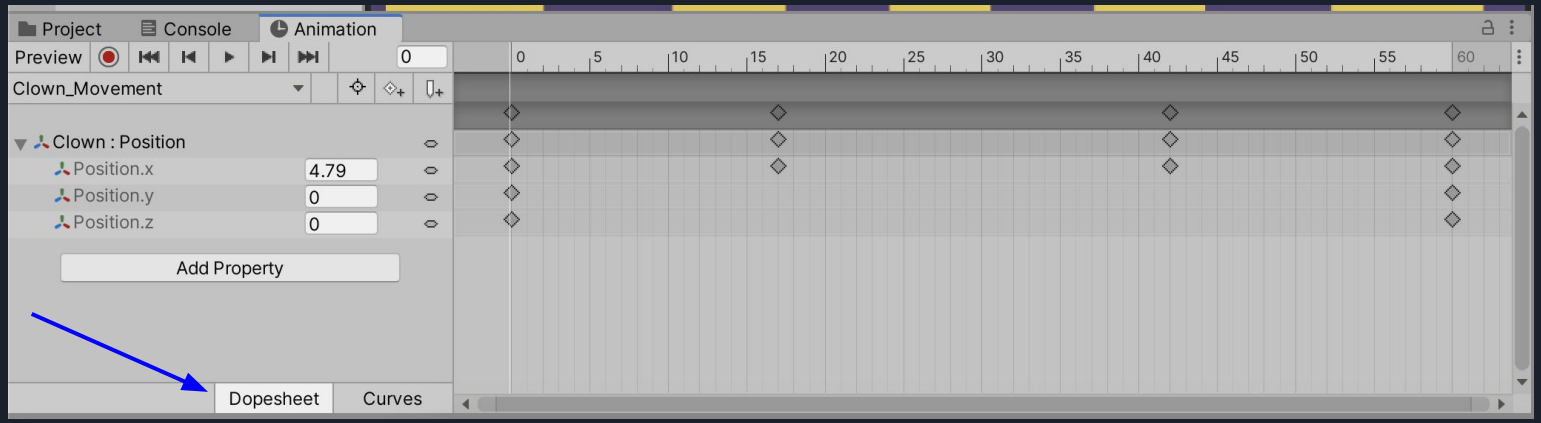
Play the game, you notice now the background moves back and forth between the two animations.



Step 4: Animate the Clown and *trigger animation*

Now that the background is animated, it is time to animate the Clown and learn how to trigger certain animation.

Click the Clown Object and in the Animation tab click Create. This will create an Animator controller for the clown. Create a few clips that move the clown around and rotate them, etc. Your dopesheet should more or less like this in some of the clips:



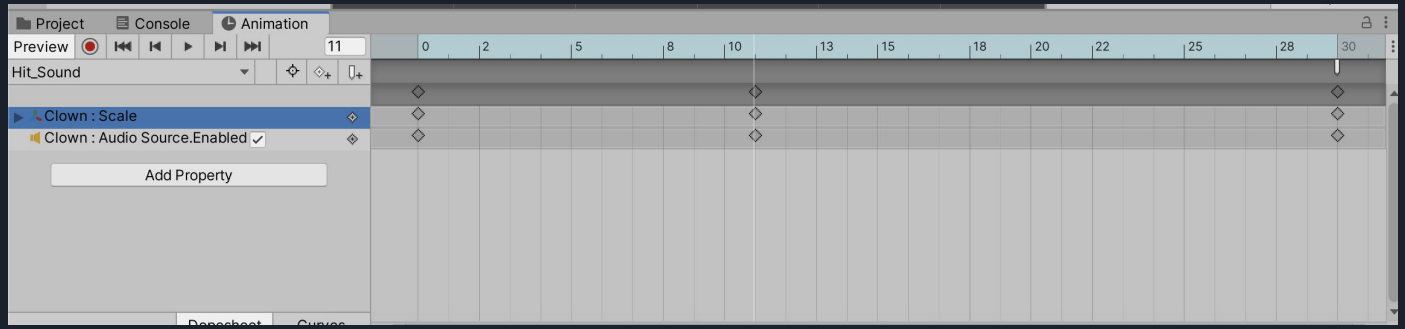
Step 4: Animate the Clown and *trigger animation*

Notice the Clown has an audio source component. Create an animation clip that:

1. Plays the hit sound and then turns it off again.

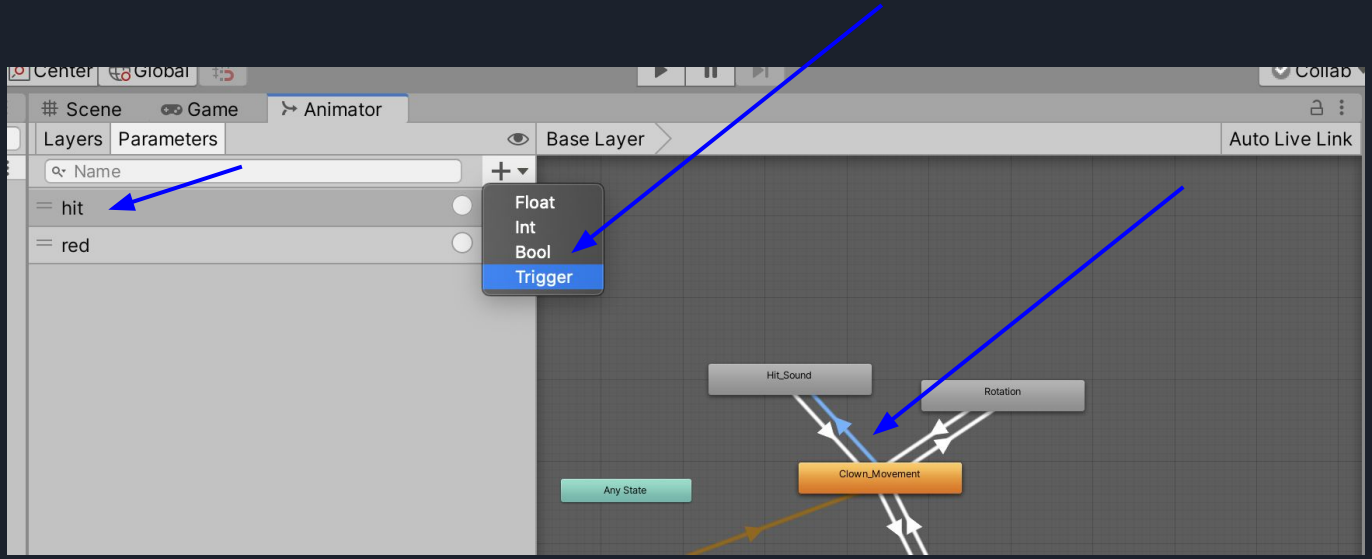
2. At the same time, reduce the scale to (0,0,0) and back to (1,1,1)

We will try to trigger animation (of this hit sound and scale out and back) when the clown is hit by the cake. The output of your dopesheet should look like this:



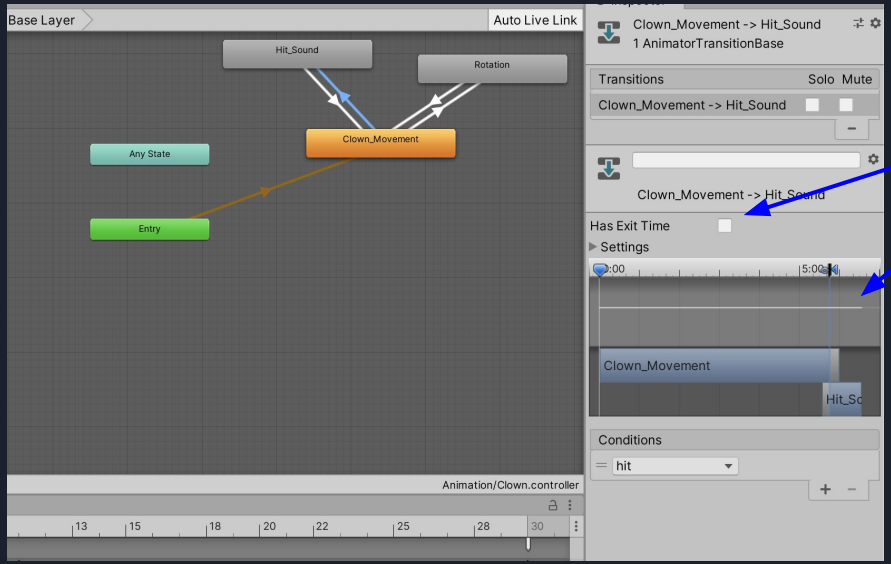
Step 5: Set Parameter

Now go to the Animator tab, and click on Parameter: create Trigger parameter and name it hit (ignore red, that's additional feature unreated to this lab). Also create transition arrows as shown in the picture (right click on the box >> Make Transition).



Step 6: Conditional Transition

Click on the transition arrow between movement and hitsound, and add the hit *Condition* there. This shows that when `hit == True`, then transition happens. You don't need condition for the other transitions. Also untick Has Exit Time -- you want the transition to happen immediately. You can edit the "transition" overlap by editing the sliders here.



Step 7: Scripting

So now what's left is of course triggering the hit parameter from Clown's Animator into true whenever the Clown collides with the Cake. We will use OnCollisionEnter() instead this time round.

OnCollisionEnter() and OnTriggerEnter() both are the methods of the Collider Class. Both are used to detect collisions when colliders enter the collision but both perform differently and cause different events.

OnCollisionenter() is used to create Collisions between objects. Gameobjects collide with each other and get repelled by their forces.

To use OnCollisionEnter(), a collider must be attached to the gameobject and its IsTrigger property should be unchecked and in order to receive a physical collision event one of the colliding gameobject must have rigid body attached to itself.

Step 7: Scripting

OnTriggerEnter() is used to detect collision but it does not act as solid body , rather allows the gameobjects to pass through them. To use OnTriggerEnter() , the IsTrigger property of the collider should be checked. To get trigger event to work properly a rigid body must be attached to one of the colliding gameobject.

We want the cake to *repel* and *not go through the Clown*, so we want the Physics to work properly. However, we want to *do something* when the bodies collide, therefore we use OnCollisionEnter() instead.

Open Collision Behavior Script. It is already attached to Clown >> Skin.

Step 7: Scripting

You need to first get the Clown's animator to access its hit parameter. Add this line and hook it up from the inspector.:

```
    public Animator clownAnimator;
```

And then, trigger the parameter:

```
    void OnCollisionEnter(Collision collision) {
                clownAnimator.SetTrigger("hit");
    }
```

Press play and you will notice that your clown will play the hit sound and disappear...kinda… But if you spam the cake you will see that the clown respawn and immediately got hit and disappear again many times.

Step 8: Timing Animation Event

We do not want our clown to get hit again *before the hitSound* animation ends. To do this, we need to trigger a **function** that allows the clown to get hit again *at the end of hitSound*.

Open CollisionBehavior.cs and add a public variable:

```
    public bool collided;
```

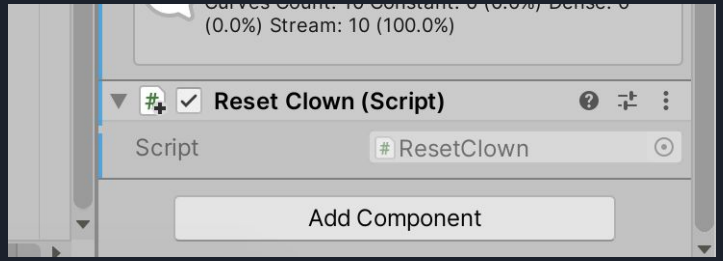
This serves as a “state” to remind us on whether the “hitSound” has been triggered yet.

Then, open ResetClown.cs and add a method there that resets this collided bool:

```
    void ResetClownOnHit() {
        gameObject.GetComponentInChildren<CollisionBehavior>().collided = false;
    }
```

Step 8: Timing Animation Event

Attach ResetClown.cs as a script to Clown object.



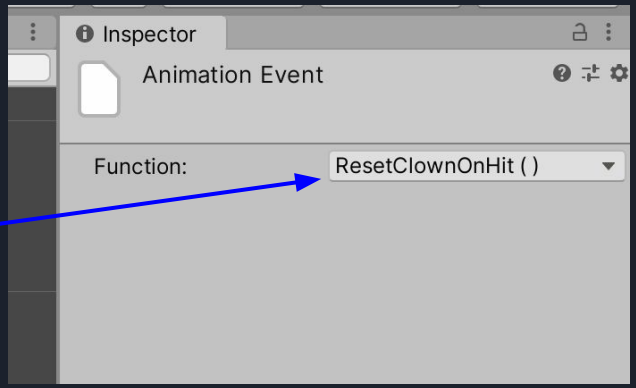
Modify the OnCollisionEnter() method in CollisionBehavior.cs to trigger the *hit* only when it hasn't collided yet:

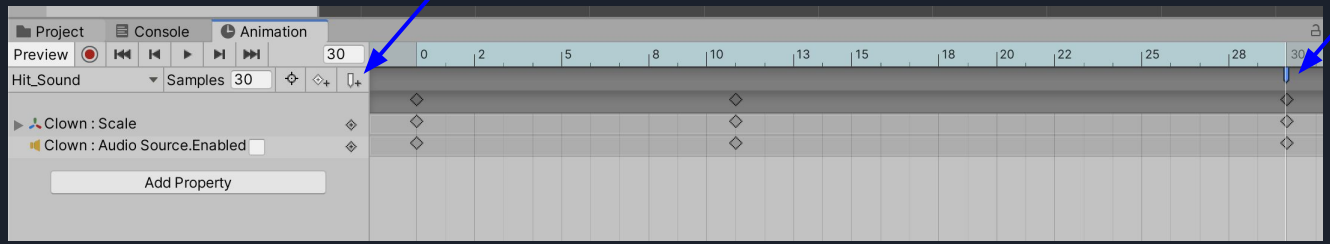
```
    // 2
    void OnCollisionEnter(Collision collision) {
        if (!collided) {
            collided = true;
            clownAnimator.SetTrigger("hit");
        }
    }
```

Step 8: Timing Animation Event -- add Event

Now we need to call this ResetClownOnHit() method in ResetClown.cs when the hitSound animation *ends*.

Click the dopesheet and press “add event”. Then in inspector, select the function: ResetClownOnHit(). This function appears because its one of the custom functions you have in the GameObject -- Clown -- where this Animation element is.





Step 8: Timing Animation Event -- summary of chain of events

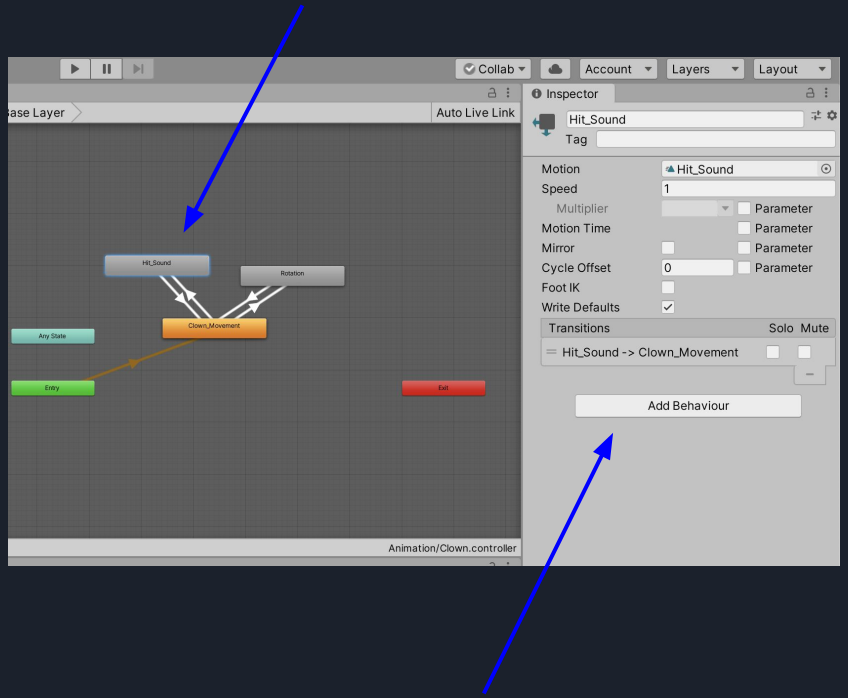
A summary of what just happened:

1. When the cake hits the clown, onCollisionEnter is called, setting collided into *true* and *hit parameter* into true, thus
2. Triggering the hitSound animation, where it plays the sound and scale down the clown and back up
3. When hitSound animation **ends, *the event triggers*** ResetClownOnHit() function that turns the *collided* parameter into false, thus allowing step (1) to happen again

Step 9: Add State Machine Behavior

We can also *trigger a behavior* (i.e: basically run a script) when in a particular state of animation. Click on hitSound state on the Animator Tab, and click Add Behavior. Name your script ClownHitBehavior. You'll notice that a new script is created that inherits the type StateMachineBehavior.

```
public class ClownHitBehavior :
StateMachineBehaviour
```



Step 9: Add State Machine Behavior

The methods inside the script is explanatory. Let's say you want to do something with the Clown upon entering this state. You can add the following code to get the gameobject (thats attached to this animator)

```
    private GameObject clown;
    private float angleRotated;
     // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
        clown = animator.gameObject;
        angleRotated = 0.0f;
    }
```

Step 9: Add State Machine Behavior

Fill up the other two methods as such. We can see the effect right away when the game is run.

```
    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
{
        clown.transform.Rotate(0,0,10, Space.Self);
        angleRotated += 10;
    }

    // OnStateExit is called when a transition ends and the state machine finishes evaluating this
state
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex) {
        clown.transform.Rotate(0,0,-angleRotated, Space.Self);
    }
```

Summary

1. Creating **animator** and **animation clip**, attaching it to GameObjects
2. Editing animation clips + recording in the dopesheet
3. Making **transition** between animation clips
4. Creating **parameters** and using it to trigger transition
5. Changing animation frame rates
6. **StateMachineBehavior** scripting
7. Timing animation and creating **events** → to call a function when a particular event happens during the animation
8. Demo (if have time): quickly adding another trigger, demo Bunny.fbx & Psyduck.fbx

CHECKOFF

1. Create another interesting throwable item prefab -- (0.5%)
2. Create a new animation for the Clown when hit by this object -- (0.5%)
3. Use the Clown Animator Controller to dictate this trigger and transition to the animation in (2) -- (0.5%)
4. You're free to add **ONE (you can do more**) modification to the game to make it more interesting, taking into account that you have understood what it means by *meaningful play* by now (this accounts for 1.5%). Of course be reasonable, it doesn't have to be very fancy/difficult but perhaps it is something that takes around 20-30 minutes to do (assuming you're no longer a total beginner by now).

As usual, record your screen and just voiceover the explanation to make a short 1-2 minute video. Upload it on Youtube/other online platform and submit the LINK on edimension.

See submission link for due date.

Introduction to Unity
Part 4 -- 3D Basics

Week 3

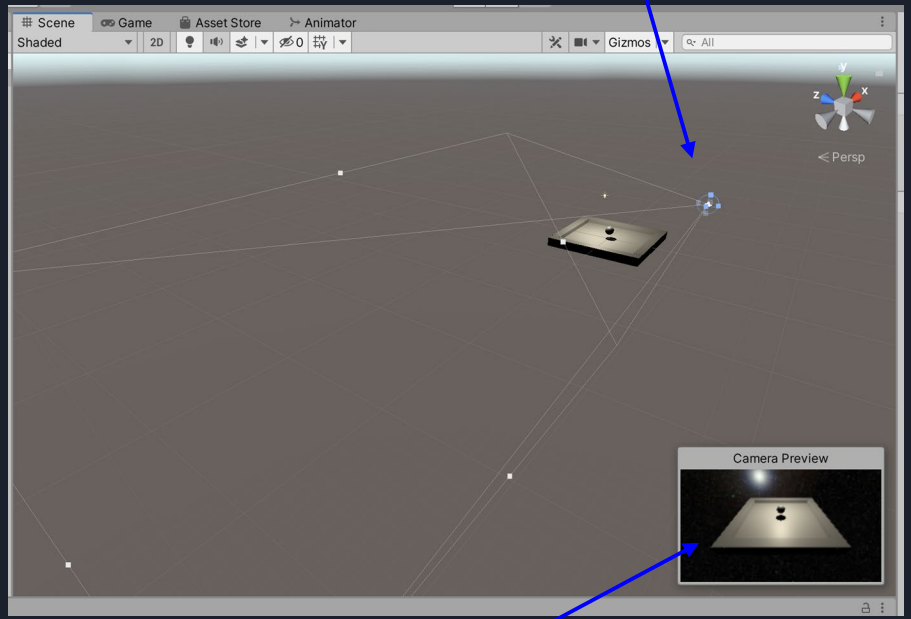
Create a new 3D project

- Download the assets here
- You can also prepare your own JPEG/PNG files for textures
- Download some *normal maps*, you can use: https://cpetry.github.io/NormalMap-Online/
- A reminder: Things that we are about to learn (and have been learning) isn't necessarily the *best practice*, in fact some of the ways are convoluted and they won't be done exactly this way by experienced coder. However the contents are made to **stress** some learning point and for conditioning -- getting used to the terms, etc. If you realise that some parts are troublesome or convoluted, then congratulations :) it means that you understand the essence, and from now on you can embark on the journey to customize it to a more fitting way / simpler / better / more efficient.

Adapted from Unity tutorial: https://learn.unity.com/project/roll-a-ball-tutorial

Introduction

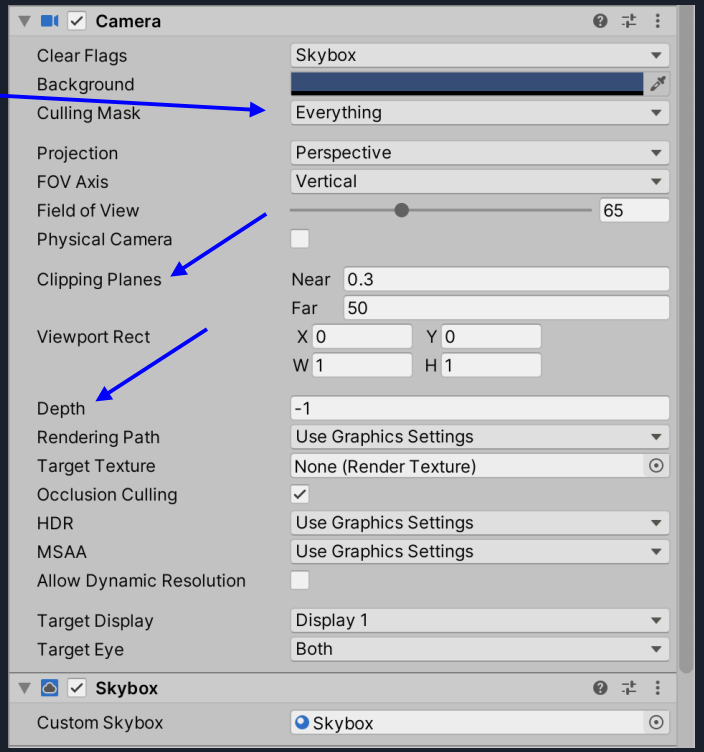
- Navigating in 3D takes time and practice
- Right click hold to rotate view
- Mouse scroll to zoom in and out
- Hand tool to translate scene
- Be mindful of your gameobject's coordinate and Main Camera position



Clear Flags

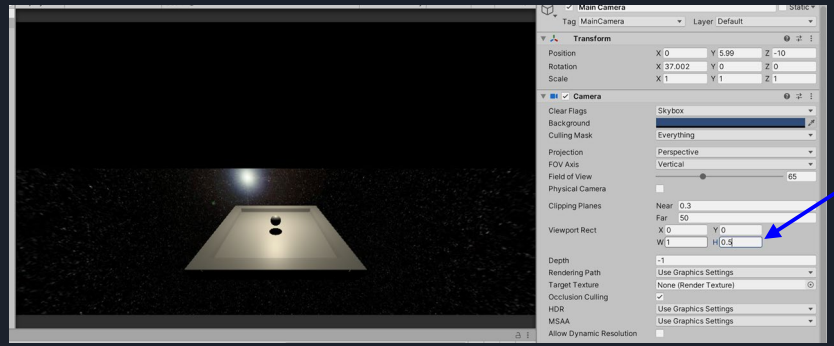
Objects that are placed outside of the Camera's view frustum will not be visible in the Game window.

Main Camera

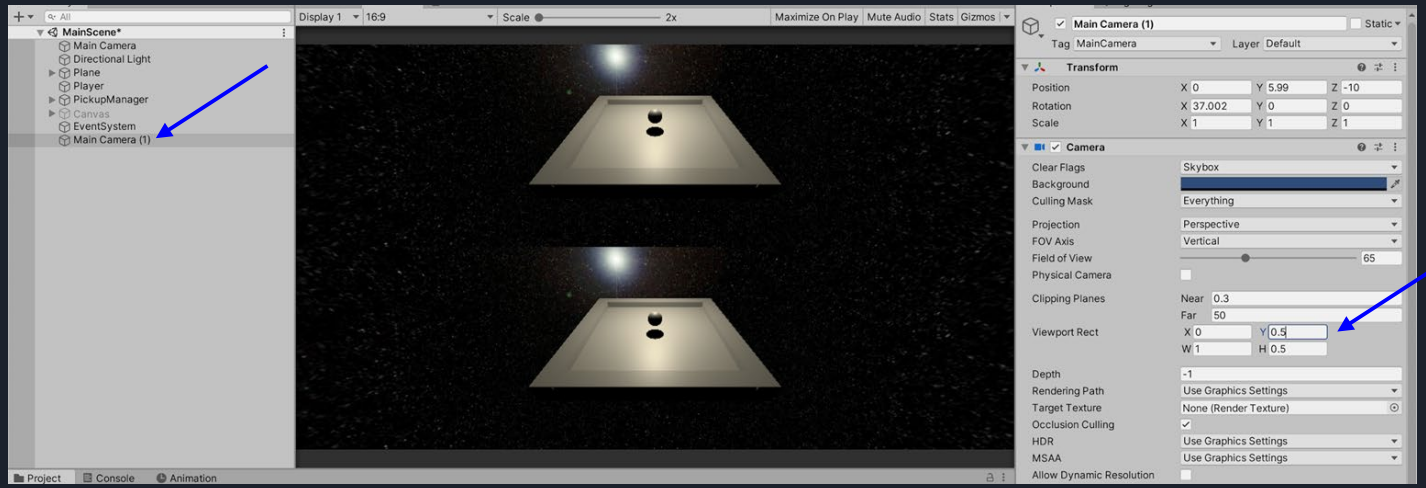


- You can choose projection: Perspective of Orthogonal (no sense of depth), depending on the kind of "feel" you want to have for the game
- Clipping planes: e.g: anything beyond '50' units away from the camera isn't rendered
- Depth value: smaller value will be placed "behind". You need to use this with the *culling mask* if you want various objects to be rendered together.

Main Camera

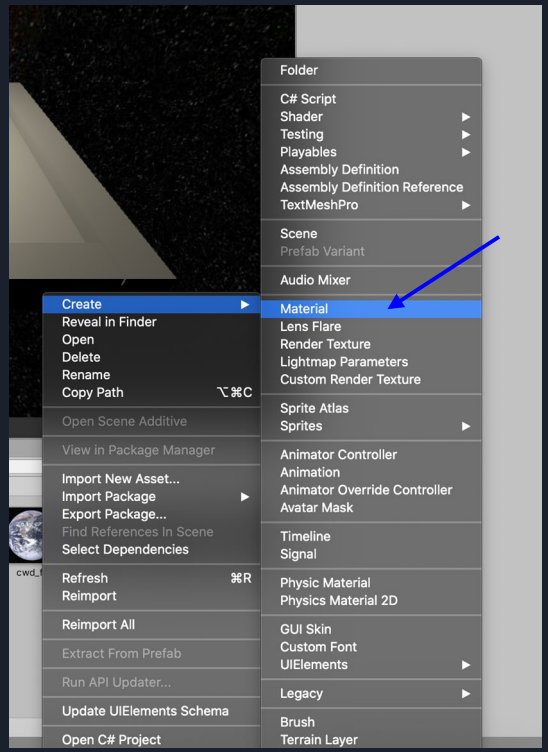


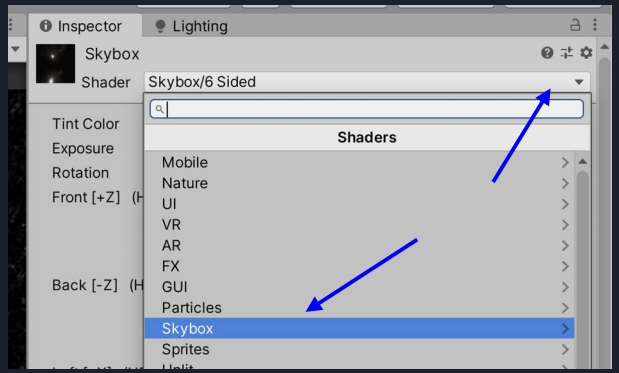
- Editing “Viewport” allows you to “split screen”
- You can add another camera in the scene



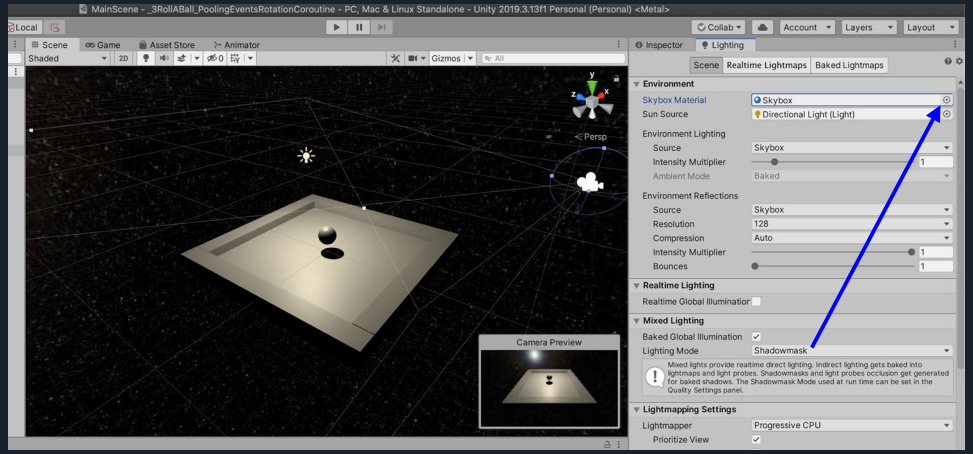
Step 1: Create Skybox

- Just a background to beautify your scene
- You can download skybox online, or make one yourself: you need a “6-sided”
- Inside materials folder, right click Create >> Material, name it “Skybox” and change the Shader material to Skybox/6 Sided. Load the images: cwd_x accordingly.



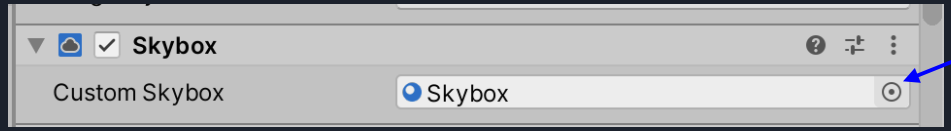


Attach Skybox to camera

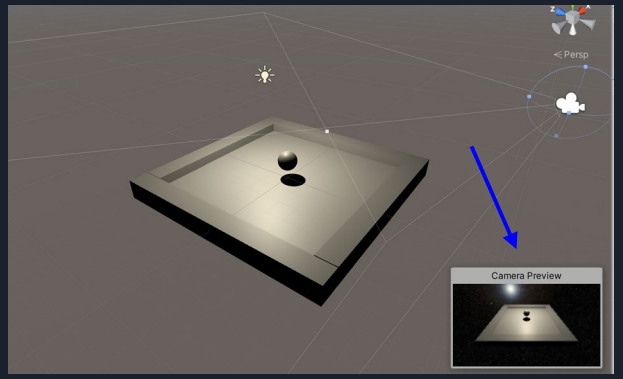


Click main camera, in the inspector Add Component >> Skybox

Load the skybox you just made



Then you can see your skybox in the preview.

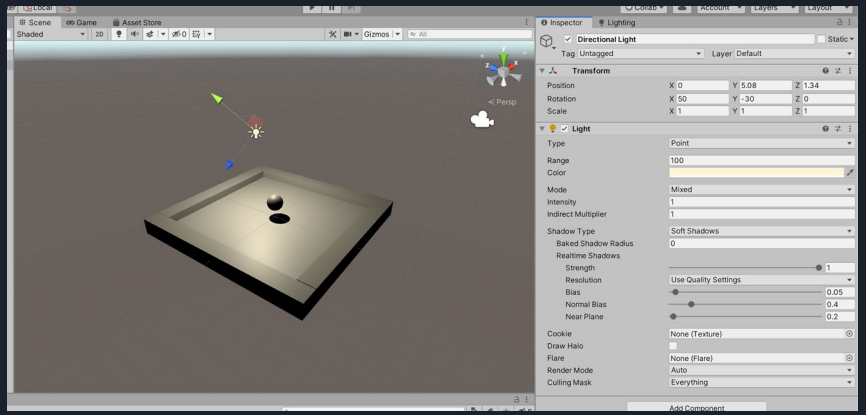


You can also change the scene skybox. Go to the Lighting tab (if you don't have it, go to Window >> Rendering >> Lighting Settings). Then load your skybox at the skybox material field in the Inspector. You will see the scene now will also show the skybox of your choice.

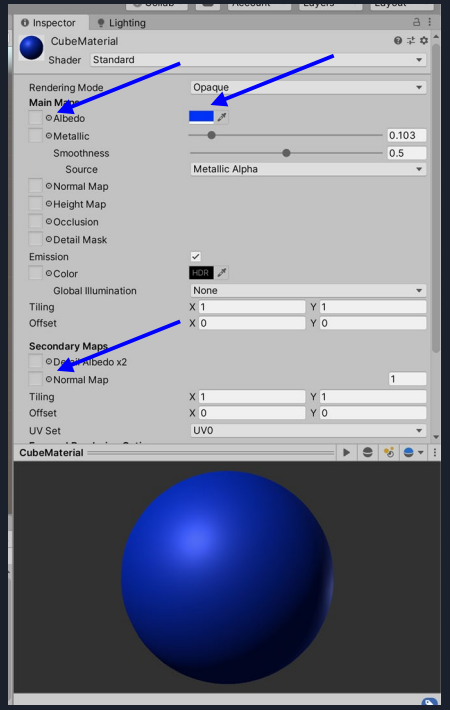
Adjust the light

Click on the Directional Light object, and modify its color, position, intensity, and type to your liking

This helps to set up the “environment” of your game. Take some time trying out other light types: spotlight, point light



Step 2: Create pickup prefabs, create & assign Material

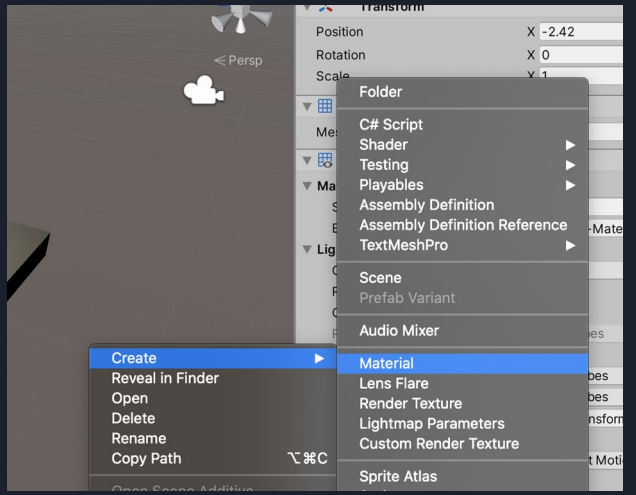


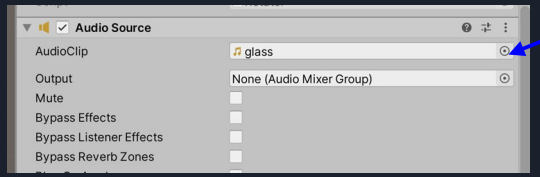
This step is similar to Part 1 -- you can create a 3d Object (Cube) and place it on the scene. Name it “PickUp”

Add an AudioSource to it, and attach the “glass” clip to it

Then go to the Materials folder, and create new Material. Edit the color to your liking. You can also add *normal maps*, secondary maps, etc. You can add any image to add texture to the object as well. After you’re done, drag it to the cube.

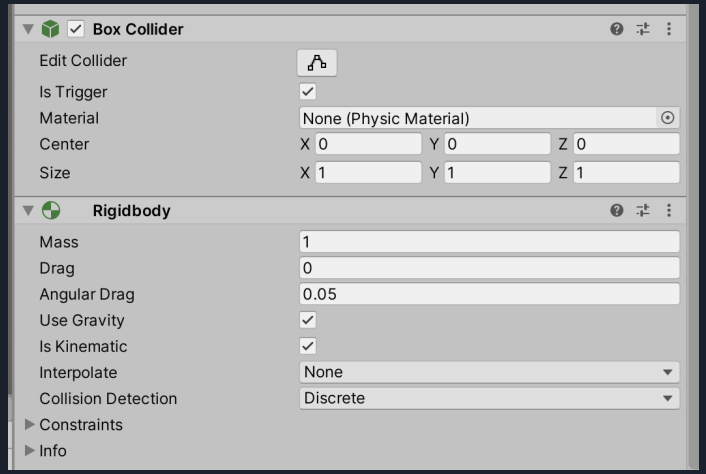
You should see your cube having the new material you just created.

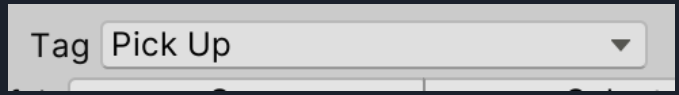




Step 2: Create pickup prefabs, create &assign Material

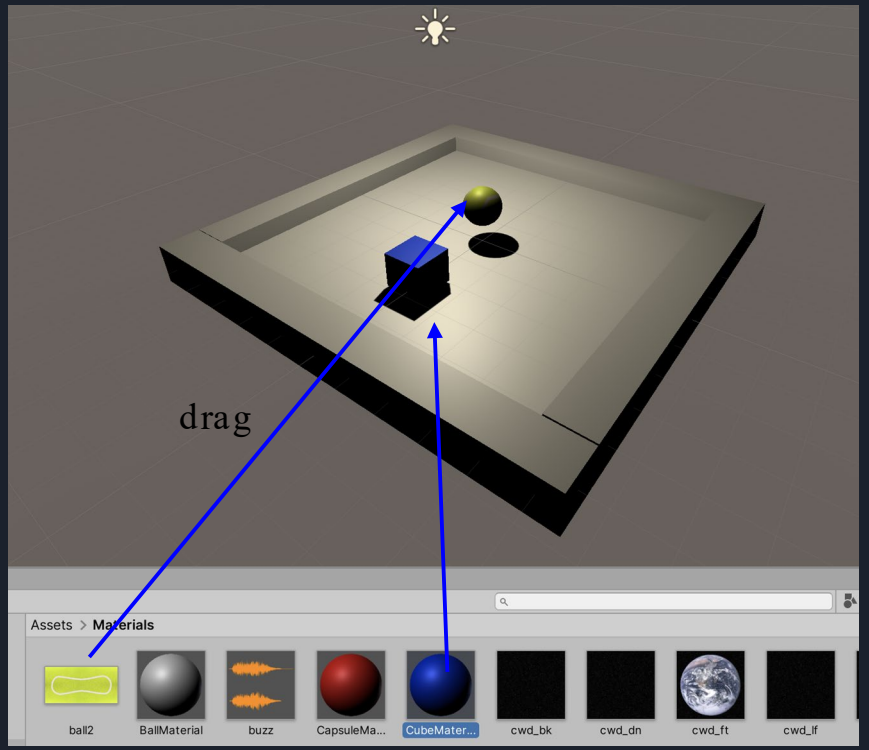
Don’t forget to add the Rigidbody and Box Collider components with these settings to allow collision with the Player. Also, make a new tag and assign it.





Step 2: Create pickup prefabs, create & assign Material

You can also assign material to your player directly by dragging the ball2 image to the player (sphere)



Once you are satisfied with the cube's color, edit its transform to make it smaller by changing the "scale", and also change the "rotation" to make it look slightly fancier:

Rotation	X	31.899	Y	-24.424	Z -40.675
Scale	X	0.3	Y	0.3	Z 0.3

Finally drag the cube to the "Prefabs" folder.

Step 3: Move the Player

Open PlayerController.cs. Under FixedUpdate(), you can add the code to dictate the movement of the player with a button press:

```
        // Set some local float variables equal to the value of our Horizontal and Vertical Inputs
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        // Create a Vector3 variable, and assign X and Z to feature our horizontal and vertical float variables above
        Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

        // Add a physical force to our Player rigidbody using our 'movement' Vector3 above,
        // multiplying it by 'speed' - our public player speed that appears in the inspector
        rb.AddForce(movement * speed);
```

Step 3: Move the Camera to follow the Player

Open CameraController.cs, declare a `Vector3 offset, GameObject player`. Under, Start() method, store the offset vector between camera's starting position and player's starting position:

```
        offset = transform.position - player.transform.position;
```

Then under Update() method, move the camera depending on where the player is:

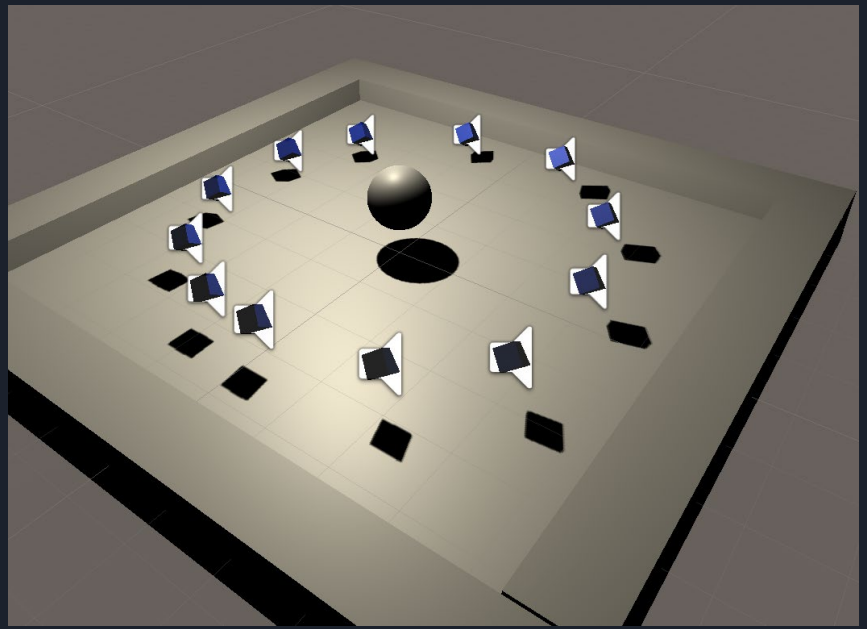
```
        transform.position = player.transform.position + offset;
```

Step 3 is very similar to what we did in Part 1, where we adjust the camera to follow the UFO.

Step 4: Place Pickup Prefabs on the scene.

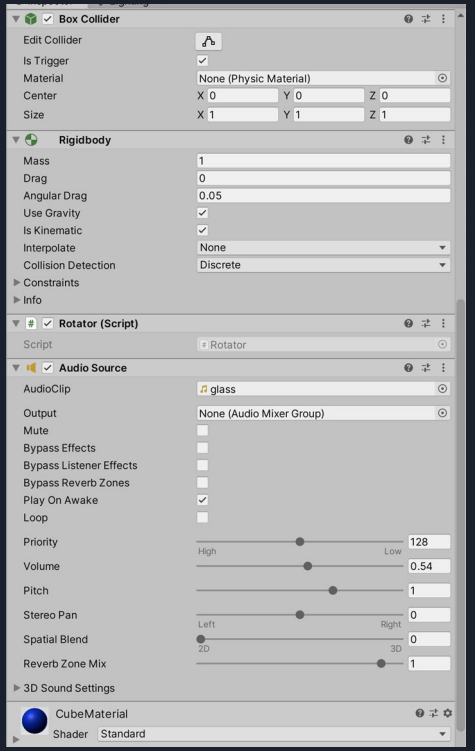
Create a new empty gameobject in the scene, call it PickupManager. Drag a few “PickUp” prefabs to be the children of this PickupManager. Place them on the scene wherever you want it.

Test run: When you run the program, you should be able to move your player around, but nothing happens yet when you collide with the cubes.



Step 5: Add script to the Pickup prefab

Attach Rotator.cs to the pickup prefab. This “rotates” the cube to add some kind of visual effects.



Step 6: Detect collision in Rotator.cs and perform action using Coroutine

Now we want to try to *detect collision* and do something about it from Rotator.cs (instead of PlayerController.cs like we did in Part 1). Firstly, ensure that you have added box collider to the Cube pickup prefab.

When the player collides with the cube, the OnCollisionEnter method on BOTH scripts are triggered. At first we may think that we can just add similar code to Rotator.cs:

```
  void OnTriggerEnter(Collider other)
    {


        //when player collides into this object
        if (other.gameObject.tag == "Player")
        {
            GetComponent<AudioSource>().Play();
            gameObject.SetActive(false);
        }
    }
```

Step 6: Detect collision in Rotator.cs and perform action using Coroutine

However if we did that, the *audio file will not play properly* (although sometimes it might). You face a problem because you didn't know whether the next line is called *after the audio HAS finished playing*. There's no time for the Audio to finish playing BEFORE the parent object is set to False. The only *guarantee* that you have is that when the game object's SetActive is set to false, audio *has started* playing (just that its not finished or you might not hear it yet, only played for a few frames). If you set the object to False, it will disable everything, including you AudioSource attached to the object.

So you **can SetActive(false) ONLY after the audio has finished playing. You can do this using using a kind of scheduled function called the Coroutine** .

```
IEnumerator waitForSound()
{
    //Wait Until Sound has finished playing
    while (GetComponent<AudioSource>().isPlaying)
    {
        yield return new WaitForSeconds(GetComponent<AudioSource>().clip.length);
    }

    //Audio has finished playing, disable GameObject
    gameObject.SetActive(false);
}
```

A coroutine is *like* a function that has the ability to pause execution (yield and WaitForSeconds) and return control to other tasks but then to continue where it left off when it runs again. Instructions in Coroutines are CONCURRENT but NOT parallel though, and it has nothing to do with Threads. So please don't mix up the two. We don't need so much details about it but you need to read up more about it if you'd like to use it for more advanced stuff, i.e: efficiency.

Coroutines vs Threads

Coroutine methods can be executed piece by piece over time, but all processes are still done by a single Main Thread executing the object's script. In other words, coroutines are executed in the main thread where Update, FixedUpdate, and all other Unity's stuff runs If a Coroutine attempts to execute time-consuming operation, the whole application will freeze for the time being.

What is the Main Thread: One thread runs at the start of a program by default. This is the "main thread". You can define the *order of script execution,* to determine which script is run first. The main thread creates new threads to handle tasks. These new threads run in parallel to one another, and usually synchronize their results with the main thread once completed.

CREATION of Threads are different. Its creation is COSTLY and it can run on different CPUs if you have more than 1 core (and if your OS allows). You also need to care about synchronizing your computed data with the Main Thread, so it is dangerous to create new Threads if you don't know what you are doing. We do not learn multi-threading in this course. *If you're interested, you can ask outside of class hours.*

All in all, you need to remember that Unity API is NOT THREAD SAFE. All calls to Unity API MUST BE DONE in the Main Thread. To make it easy, just don't create new threads unless you're sure with what you are doing.

Step 6: Detect collision in Rotator.cs and perform action using Coroutine

Then call it in the OnTriggerEnter method. To render the object “invisible” upon collision and prevent double collision, you can set its renderer and collider to false first. The StartCoroutine function will call waitForSound:

```
    void OnTriggerEnter(Collider other)
    {

        //when player collides into this object
        if (other.gameObject.tag == "Player")
        {
            GetComponent<AudioSource>().Play();
            gameObject.GetComponent<MeshRenderer>().enabled = false;
            gameObject.GetComponent<Collider>().enabled = false;
            StartCoroutine(waitForSound());
        }
    }
```

Test: Now you can move your player and collide with the cube. A sound will play and the cube will disappear. It will be inactive only moments after.

Step 7: Count scores and show it at Canvas

We want the score to be updated each time player collides with the object. Let's use events again for this.

In Rotator.cs add:

```
    public delegate void UpdatePlayerScore(int amount);
    public static event UpdatePlayerScore updatePlayerScore;
    private int score_updater = 1;
```

Recall that Rotator.cs is attached to the pickup prefab. The score updater is a variable to indicate how much score should be added when a Player collides with this object.

Step 7: Count scores and show it at Canvas

Add the method in PlayerController.cs that fit the signature of this delegate:

```
    void scoreUpdate(int score)
    {
        // Add one to the score variable 'count'
        count = count + score;

        // Run the 'SetCountText()' function (see below)
        SetCountText();
    }
```

Subscribe to the event in start():

```
        //subscribe to the event
        Rotator.updatePlayerScore += scoreUpdate;
```

Step 7: Count scores and show it at Canvas

Invoke the event when the object collides with player in this method inside Rotator.cs:

```
    void OnTriggerEnter(Collider other)
    {

        //when player collides into this object
        if (other.gameObject.tag == "Player")
        {
            GetComponent<AudioSource>().Play();
            //update player's who's listening to this event
            updatePlayerScore(score_updater);
            gameObject.GetComponent<MeshRenderer>().enabled = false;
            gameObject.GetComponent<Collider>().enabled = false;
            StartCoroutine(waitForSound());
        }
    }
```

Step 7: Count scores and show it at Canvas

Check that these variables in Player object are hooked up properly:

```
    public Text countText;
    public Text winText;
```

Summary

In this short lab, we have done:

1. Creation of 3D project
2. Navigation in the 3D scene editor
3. Setup of skybox, camera, and lighting
4. Create materials and applying them to prefabs / gameobjects
5. Schedule functions using coroutines
6. Moving 3D player from input
7. Detect collision (from object) not player
8. Update score to canvas

There’s no checkoff for this lab, but you need to complete it in order to continue with Part 5 -- where there will be checkoff.

Introduction to Unity Part 5a - Object Pooling and 3D Rotations

Week 3

Materials

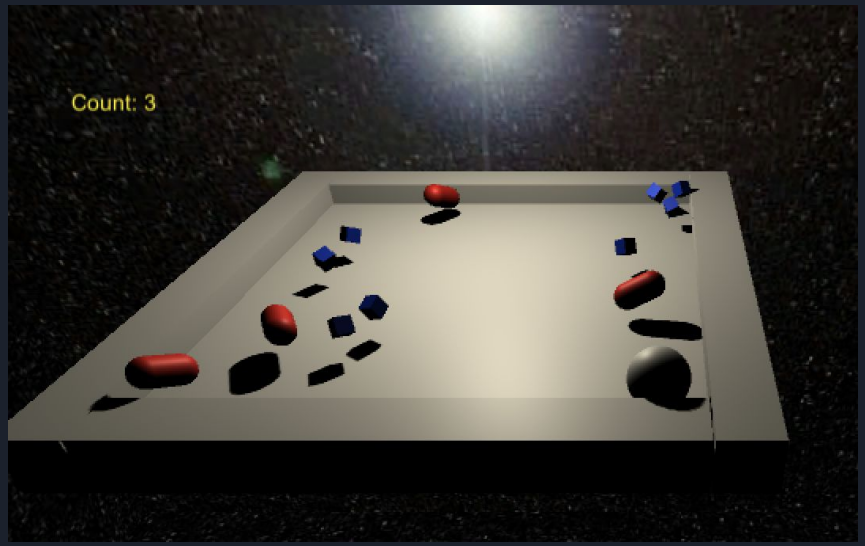
Continue with what you have completed in Part 4

There’s no need to download additional materials for this part

Adapted from the series of Unity tutorials on Rotations, and
https://www.raywenderlich.com/847-object-pooling-in-unity

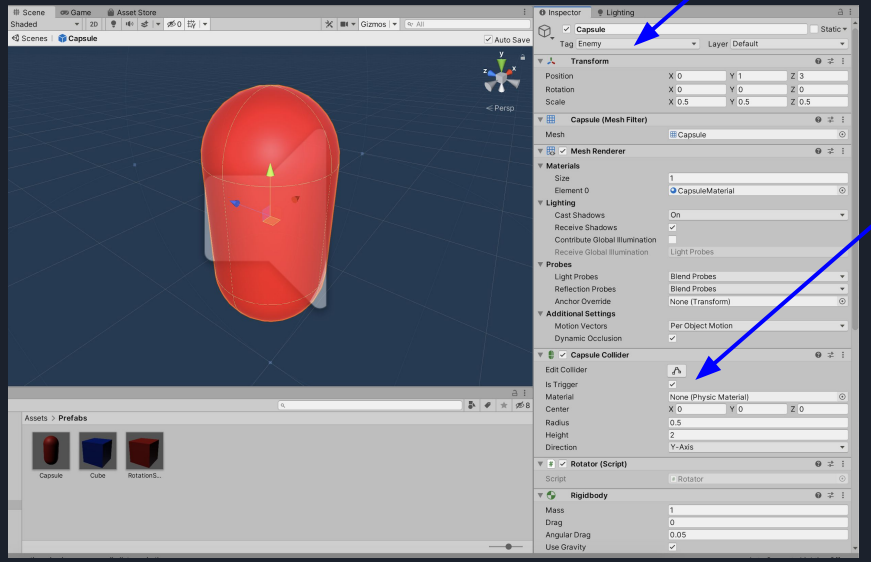
Objectives

1. Spawn different types of pickup from script
2. Write scripts to *pool* objects so there's no need for GameObject instantiation and destroy at runtime
3. Basics of 3D rotations using Quaternion



Before we begin...

1. Quickly another prefab and name it with a different tag like “Enemy”.
2. Add the collider, rigidbody, and rotator script to it.
3. Set RigidBody to IsKinematic as usual.
4. Add audiosource and drag the buzz soundclip to it.
5. Add the Enemy tag.



Step 1: Creating Object Pooler

The idea is to spawn N items in the beginning, and reuse them throughout the game (by setting it to be active/inactive). Create a script ObjectPooler.cs.

Add a custom class in it:

```
[System.Serializable]
public class ObjectPoolItem
{
    public int amount;
    public GameObject prefab;
    public bool expandPool;
}
```

And declare these variables:

```
    //declared as a static variable, so other classes can actually access it
    public static ObjectPooler SharedInstance;
    public bool expandPool = true;


    public List<ObjectPoolItem> itemsToPool;
    public List<GameObject> pooledObjects;
```

Step 1: Creating Object Pooler

Add the following in the Awake() method:

```
    void Awake()
    {
        SharedInstance = this;
        pooledObjects = new List<GameObject>();

        foreach (ObjectPoolItem item in itemsToPool)
        {
            for (int i = 0; i < item.amount; i++)
            {
                GameObject pickup = (GameObject)Instantiate(item.prefab);
                pickup.SetActive(false);
                pooledObjects.Add(pickup);
            }
        }
    }
```

So the idea is to set “this” instance as the static variable. This means there can only be **one** object pooler in the game. We instantiate as many gameobjects as dictated in the itemsToPool.

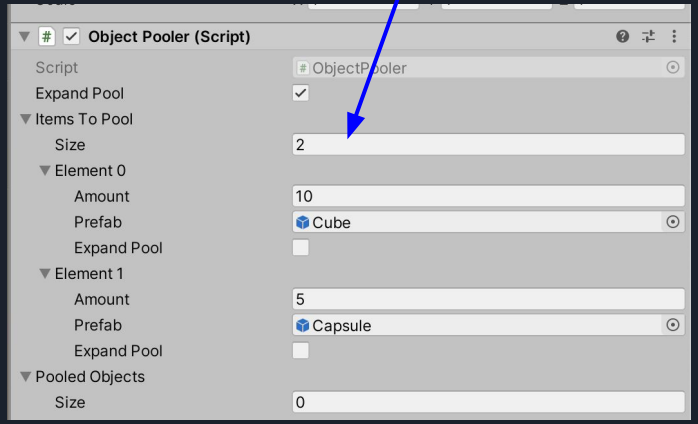
Step 1: Creating Object Pooler

ItemsToPool is a list of ObjectPoolItem,
Which we set to be serializable.

Therefore we can set its values in the editor later on.

Then add this method too to obtain one object from the pool, given a *tag*:

```
    public GameObject GetPooledObject(string tag)
    {
        for (int i = 0; i < pooledObjects.Count; i++)
        {
            if (!pooledObjects[i].activeInHierarchy &&
pooledObjects[i].tag == tag)
            {
                return pooledObjects[i];
            }
        }
        return null;
    }
```



Step 1: Creating Object Pooler

What if there's no available object (inactive and matching the tag) in the Pool? Well, we can choose to *expand* our pool at runtime as well. Add this code *before* the return null:

```
        foreach (ObjectPoolItem item in itemsToPool)
        {
            if (item.prefab.tag == tag)
            {
                if (item.expandPool)
                {
                    GameObject pickup = (GameObject)Instantiate(item.prefab);
                    pickup.SetActive(false);
                    pooledObjects.Add(pickup);
                    return pickup;
                }
            }
        }
```

Now we will return null IF and only IF the tag isn't matching with what is defined in the itemsToPool.

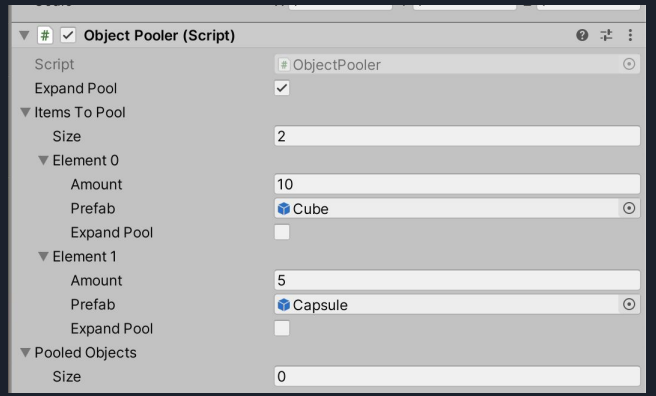
Step 2: Using the Object Pooler

Create an empty GameObject, call it PickupManager. Attach the script ObjectPooler.cs to it. Also, create another script PickupManager.cs and attach to it.

In the Start() method of PickupManager.cs, we use the object pooler *statically, note: change the tag accordingly that match your pickup prefab*:

```
        for (int i = 0; i < 5; i++)
        {
            GameObject pickUp = ObjectPooler.SharedInstance.GetPooledObject("Pick Up");
            pickUp.transform.position = new Vector3(Random.Range(-4.5f, 4.5f), 0.6f,
Random.Range(-4.5f, 4.5f));
            pickUp.SetActive(true);
        }
```

In the inspector, update the Object Pooler values:

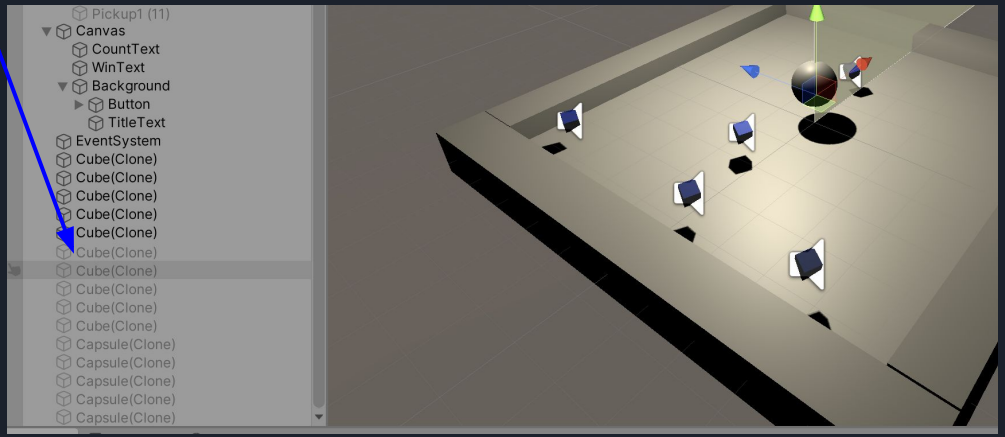


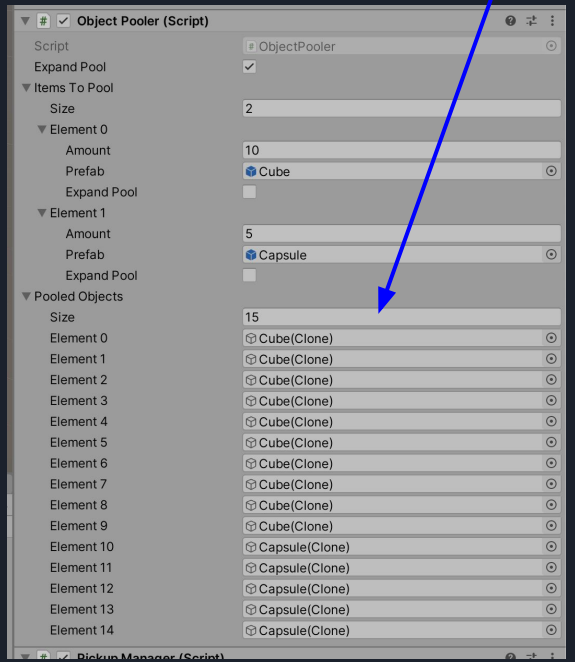
Run the program and right now we have
5 cubes spawned.

Step 2: Using the Object Pooler

However if you hover to the inspector, there's actually 10 cubes in the scene and 5 capsules, its just that they are "hidden".

Move the player to hit some cubes, you'd see that some cubes are rendered invisible, but are still there in the game.





Step 3: Spawn the other type of pickup

Let’s say the capsule looking prefab is supposed to “Reduce” your score, and the cube is supposed to “Add” to your existing score. Also, lets say we want to “spawn” 1 new cube and 1 new capsule for each cube eaten.

In Rotator.cs, add this line at the `Start()` method:

```
    void Start()
    {
        if (gameObject.tag == "Enemy")
        {
            score_updater = -1;
        }
        else if (gameObject.tag == "Pick Up")
        {
            score_updater = 1;
        }
    }
```

Step 3: Spawn the other type of pickup

In PickupManager.cs, write this method to spawn one cube and one enemy when called. Just like what we did in Start(), we simply fetch available item from pool, set its new location and made it active.

```
    public void notifySpawn()
    {
        //spawn one cube and one enemy
        GameObject pickUp = ObjectPooler.SharedInstance.GetPooledObject("Pick Up");
        if (pickUp != null){
            pickUp.transform.position = new Vector3(Random.Range(-4.5f, 4.5f), 0.6f,
Random.Range(-4.5f, 4.5f));
            pickUp.SetActive(true);
        }

        GameObject enemy = ObjectPooler.SharedInstance.GetPooledObject("Enemy");
        if (enemy != null){
            enemy.transform.position = new Vector3(Random.Range(-4.5f, 4.5f), 0.6f,
Random.Range(-4.5f, 4.5f));
            enemy.SetActive(true);
        }

    }
```

Step 3: Spawn the other type of pickup

In PlayerController.cs, refer to the PickupManager and hook it up in the inspector.

```
    public PickupManager manager;
```

In ScoreUpdate method in PlayerController.cs, add this line before SetCountText:

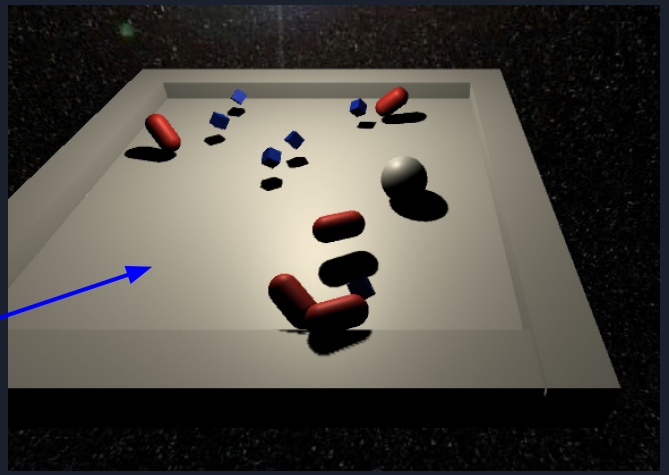
```
        if (count < 12)
        {
            manager.notifySpawn(); // if game isn't over yet, spawn a new one
        }
```

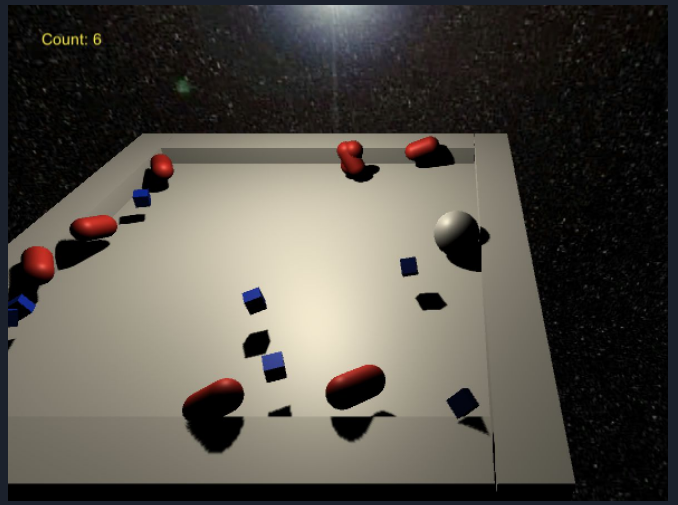
Therefore, if game isn’t over yet (count < 12), then we tell the PickupManager to spawn one more cube and the capsule enemy prefab.

Step 4: enable “expandPool”

Run the game, and collide with 5 cubes. You notice that no more enemy capsule prefabs will be spawned after 5 is present.

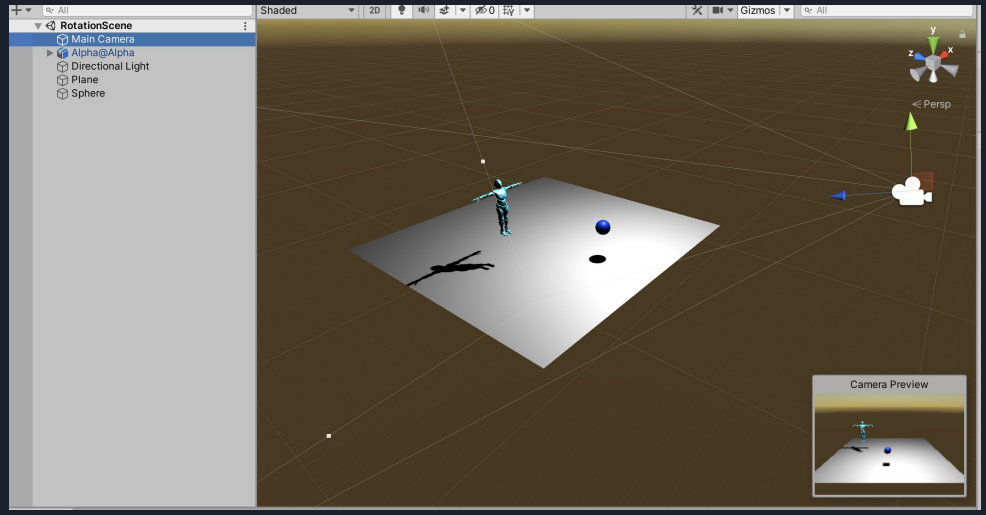
You can tick the Expand Pool under inspector of the PickupManager, and watch more enemy prefabs spawned even though initially it was set to just 5.





3D Rotation Basics

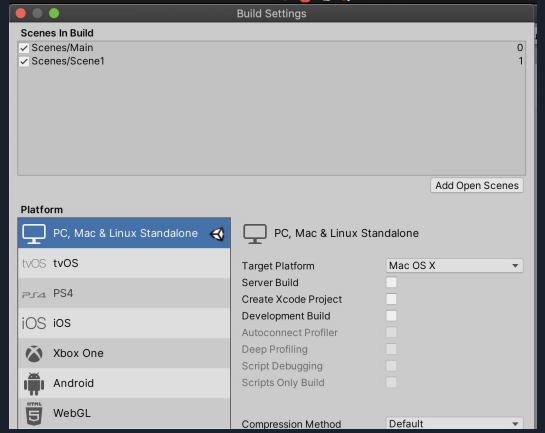
Create new scene and add objects to your scene as shown in the picture.



Note: you can switch between scenes through script by using the code:

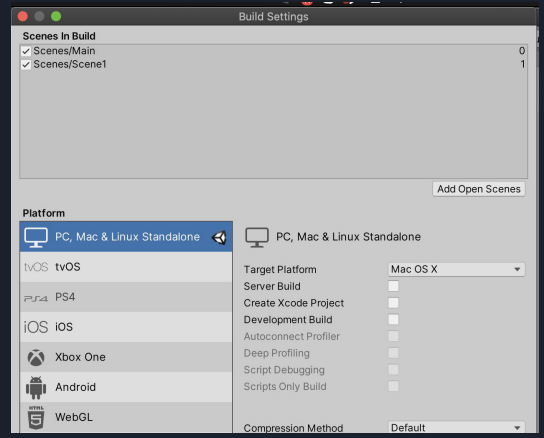
```
using UnityEngine.SceneManagement;

SceneManager.LoadScene (sceneName:"YourSceneName",
LoadSceneMode.Single);
```

And set up the “start scene” + other scenes to build in the Build Settings before running your game. Notice the number on the right side dictates the order. Scene 0 is the starting scene.



Extra: Loading Scene Asynchronously

When you want to load from Scene A to Scene B, sometimes you want to let the scene loads in the background first *before* presenting it. You can use instead LoadSceneAsync method. This requires a coroutine:




```
StartCoroutine(LoadYourAsyncScene("Scene1")); // call this where you want to trigger scene change
```

```
IEnumerator LoadYourAsyncScene(string sceneName)
{
    // The Application loads the Scene in the background as the current Scene runs.
    // This is particularly good for creating loading screens.
    // You could also load the Scene by using sceneBuildIndex. In this case Scene1 has
    // a sceneBuildIndex of 1 as shown in Build Settings.
    AsyncOperation asyncLoad = SceneManager.LoadSceneAsync(sceneName, LoadSceneMode.Single);
    // Wait until the asynchronous scene fully loads
    while (!asyncLoad.isDone)
    {
        yield return null;
    }
}
```

Extra: Keeping one script between many scenes using Singleton

When you load a new scene, typically all instances of scripts in the previous scene will be destroyed.

The singleton pattern is a way to ensure a class has only a single globally accessible instance ***available at all times***. There's pros and cons of using singleton pattern, but ultimately it's up to you and your use case. The skeleton code for a script where you want to implement the Singleton pattern is as such shown on the right.

Of course alternatives to Singleton are ScriptableObject and static variables. You know how to use the latter, but it has to be set manually through scripts. The former is more flexible and we can set it in the inspector. We will learn the former in the later part of the semester.

```
public class Someclass : MonoBehaviour
{
    private static Someclass  instance;
    public static Someclass Instance
    {
        get { return _instance; }
    }

    private void Awake()
    {
        if (_instance != null && _instance != this)
        {
            Destroy(this.gameObject);
            return;
        }

         instance = this;
        DontDestroyOnLoad(this.gameObject);
    }
}
```

Extra: Generic Singleton Class

Of course the issue with adding the singleton pattern in the previous slide to any script you want is boilerplate code (repeated code).

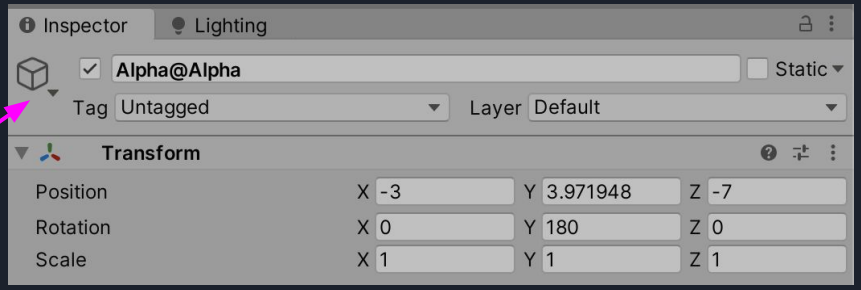
We can create a generic singleton class instead, and use them as such. You are recommended to use this for ALL MANAGER-TYPE scripts.

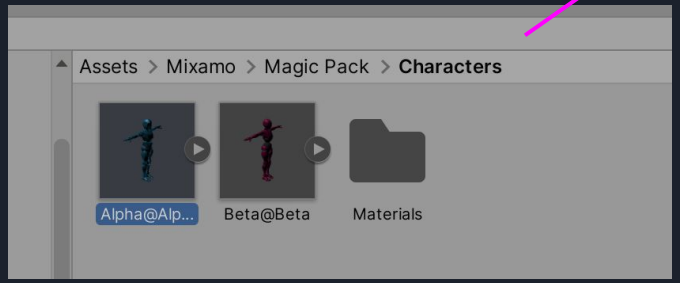
```
public class MyGameManager :
GenericSingletonClass<MyGameManager>;
{
 // your code here
 public override void Awake(){
     base.Awake();
     //code here if necessary

 }
```

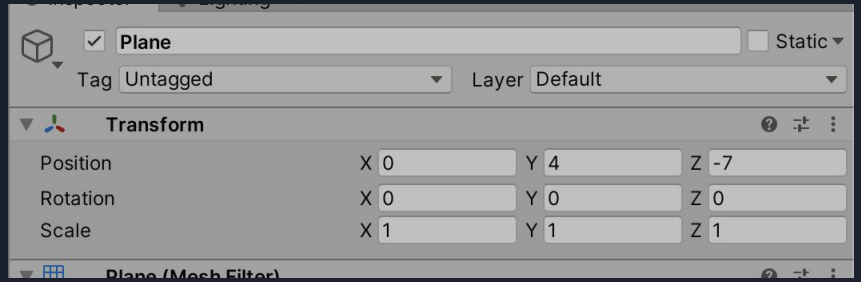
```
public class GenericSingletonClass<T> : MonoBehaviour
where T : Component
{
   private static T instance;
   public static T Instance {
     get {
       if (instance == null) {
          instance = FindObjectOfType<T> ();
          if (instance == null) {
            GameObject obj = new GameObject ();
            obj.name = typeof(T).Name;
            instance = obj.AddComponent<T>();
          }
       }
      return instance;
     }
   }
   public virtual void Awake ()
   {
      if (instance == null) {
        instance = this as T;
        DontDestroyOnLoad (this.gameObject);
   } else {
     Destroy (gameObject);
   }
 }
}
```

Note: referenced from http://www.unitygeek.com/unity_c_singleton/
You can find many other exapmples of generic singleton class that you can adapt, some is more complex even with locks, etc.
You might also need to refresh your knowledge about virtual methods (easily understood when compared withs abstract methods)

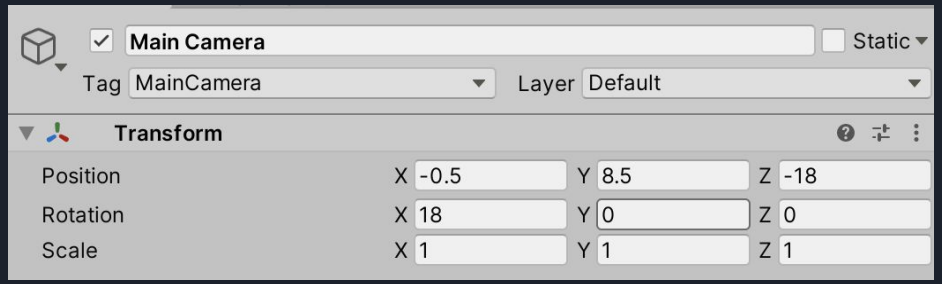
Setup scene transform:

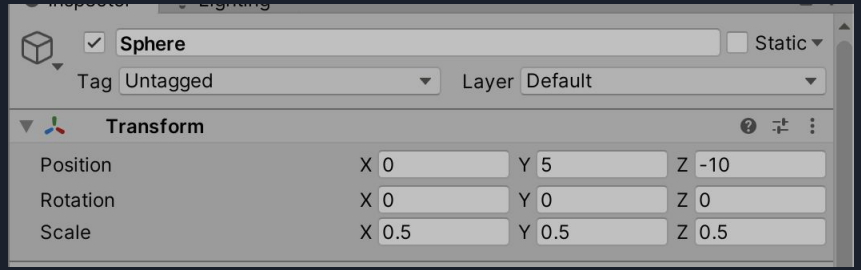




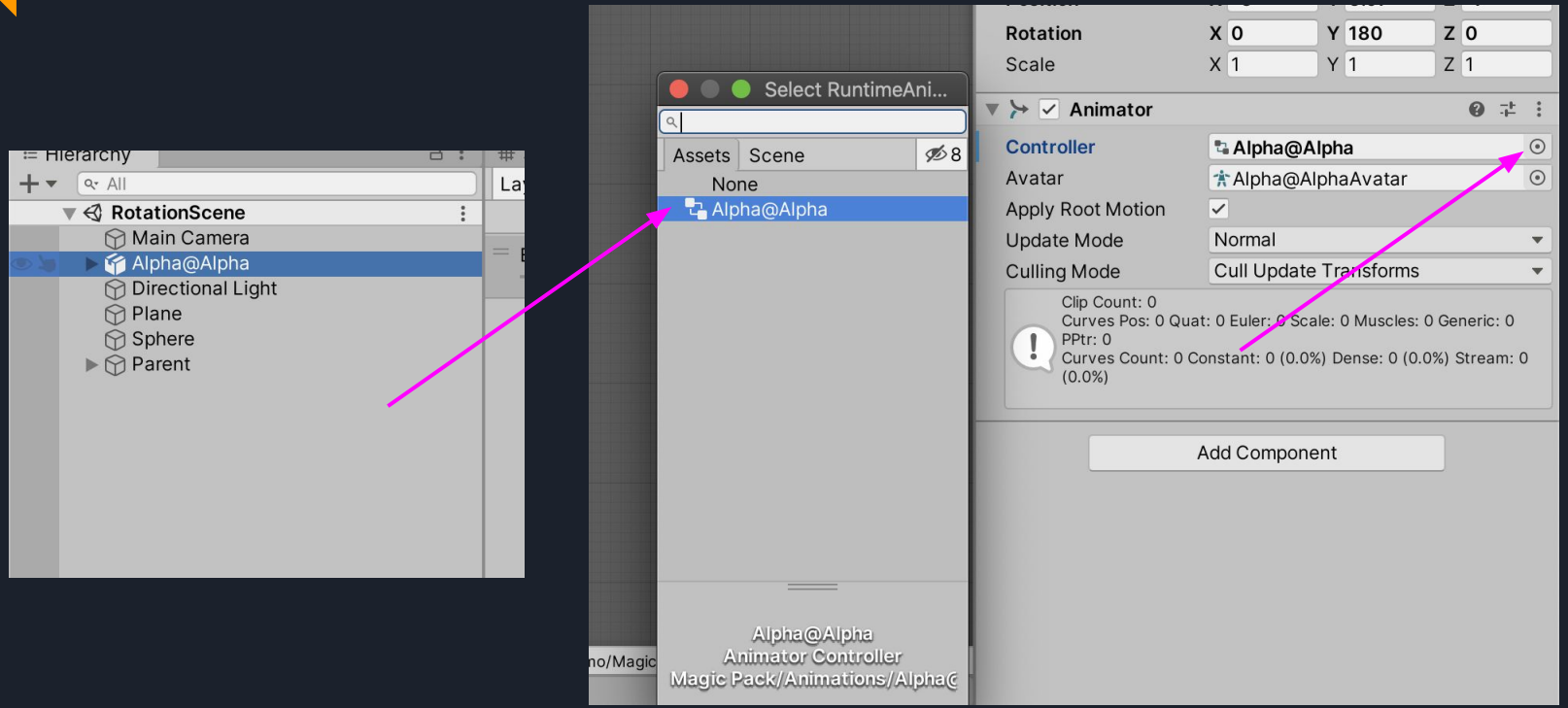


Add materials of your liking to the Plane and Sphere





Add Animator to the Avatar



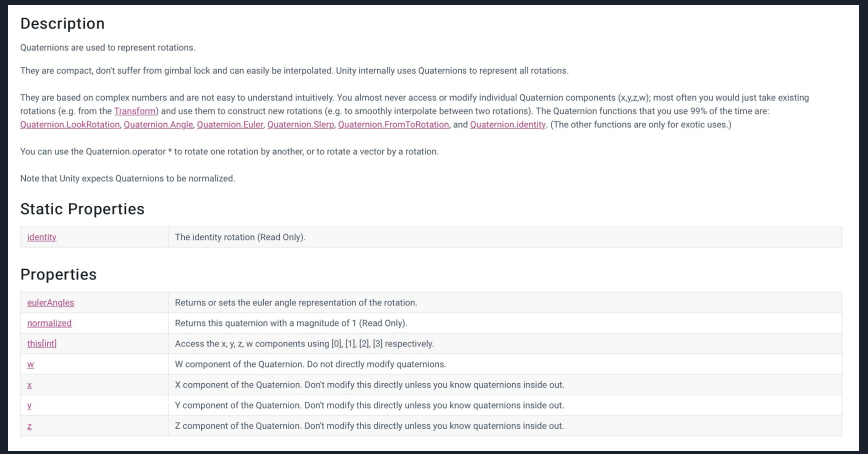
Quaternion: an important concept so you can *rotate things* properly

Unity uses quaternion to prevent a phenomenon called **gimbal lock**. The math behind it is complicated, though you're welcome to ask me outside class hours if you're interested.

You can use it *without* understanding how it works, as long as you know *how to use it*.

To define an object's rotation, you can use:

```
Quaternion rotation = Quaternion.Euler(0, 30, 0);
```



Description

Quaternions are used to represent rotations.

They are compact, don't suffer from gimbal lock and can easily be interpolated. Unity internally uses Quaternions to represent all rotations.

They are based on complex numbers and are not easy to understand intuitively. You almost never access or modify individual Quaternion components (x,y,z,w); most often you would just take existing rotations (e.g. from the Transform) and use them to construct new rotations (e.g. to smoothly interpolate between two rotations). The Quaternion functions that you use 99% of the time are: Quaternion.LookRotation, Quaternion.Angle, Quaternion.Euler, Quaternion.Slerp, Quaternion.FromToRotation, and Quaternion.identity. (The other functions are only for exotic uses.)

You can use the Quaternion.operator * to rotate one rotation by another, or to rotate a vector by a rotation.

Note that Unity expects Quaternions to be normalized.

Static Properties

identity	The identity rotation (Read Only).

Properties

eulerAngles	Returns or sets the euler angle representation of the rotation.
normalized	Returns this quaternion with a magnitude of 1 (Read Only).
this[int]	Access the x, y, z, w components using [0], [1], [2], [3] respectively.
w	W component of the Quaternion. Do not directly modify quaternions.
x	X component of the Quaternion. Don't modify this directly unless you know quaternions inside out.
y	Y component of the Quaternion. Don't modify this directly unless you know quaternions inside out.
z	Z component of the Quaternion. Don't modify this directly unless you know quaternions inside out.

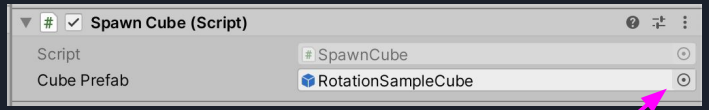
Test the Quaternion

```
//These two are NOT the same even though they look like they are if you are no
transform.rotation = transform.rotation * Quaternion.Euler(axis2, axis1, 0);
//or
transform.rotation = Quaternion.Euler(axis2, axis1, 0) * transform.rotation;
```

Create a new script called SpawnCube.cs. Add the variable `public GameObject cubePrefab`. Under Update() method add:

```
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Instantiate(cubePrefab, new Vector3(0, 5, 0),
Quaternion.Euler(-50.0f, 22.5f, 0)); // 22.5 degrees
rotation around the global y-axis
        }
```

Returns a rotation that rotates z degrees around the z axis, x degrees around the x axis, and y degrees around the y axis; applied in that order.



Attach the script to the Main Camera and load the given prefab in the assets you downloaded.

Then run the script and press the spacebar. You should see a cube spawned, rotated 22.5 degrees around global y-axis, and *then* -50 degrees around global x-axis -- rotation is NOT commutative

Test the Quaternion

When you press space, a cube shall appear.

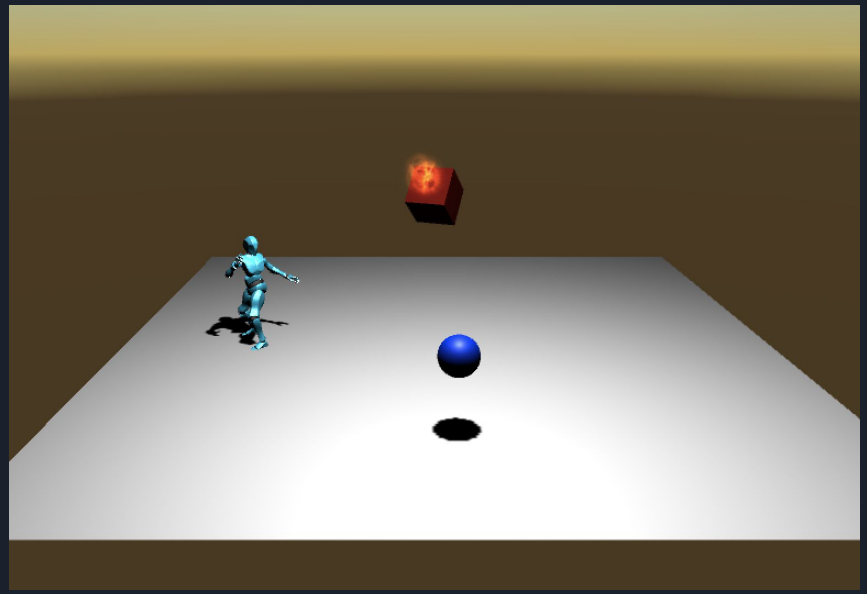
Initial rotation is based on *global rotation*

Always use `Quaternion.Euler(arg)` when defining rotation, where arg is a 3D vector as we know it in linear algebra.

DO NOT ATTEMPT to change the transform rotation component directly unless you know Quaternion inside out!

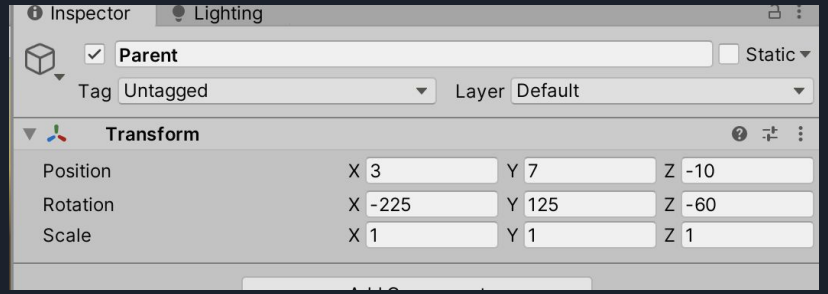
```
transform.rotation.x = X; //THIS IS NOT RECOMMENDED!
```

THESE VALUES are NOT Euler vector x, y, or z components.

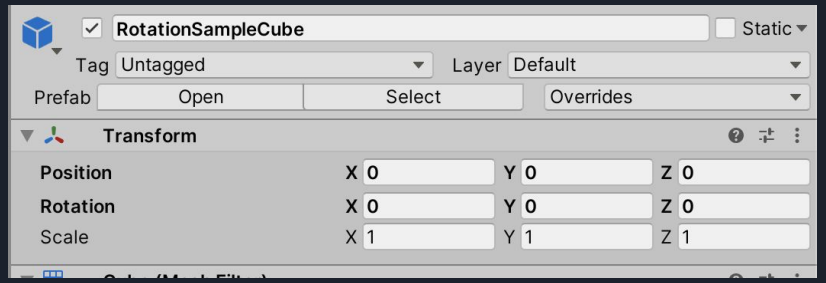


Global vs Local Coordinate

Create a gameobject called Parent with transform values:



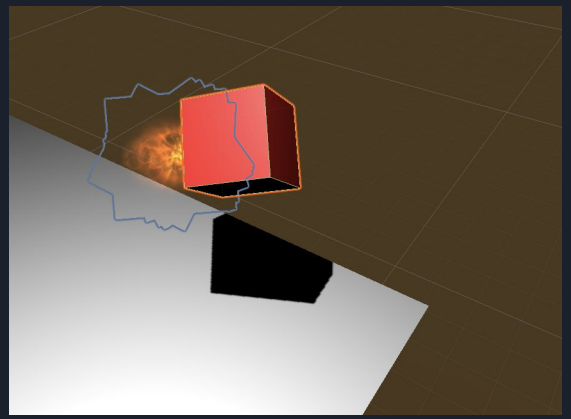
Drag RotationSampleCube prefab to be the child object under ‘Parent’:

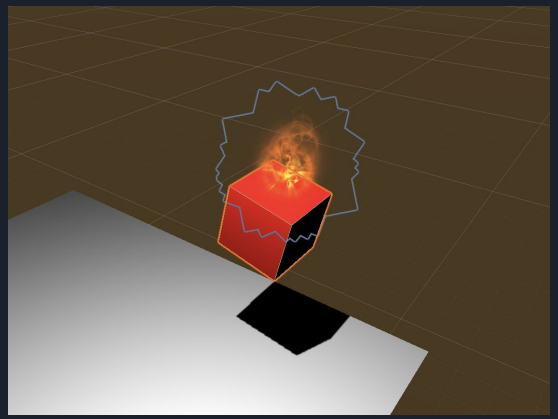


Global vs Local Coordinate

Attach the script PrefabRotationDemo.cs to the RotationSampleCube. Please analyse the script first and observe the output::

1. Observe the cube rotating in z-axis and then x-axis of *its local frame* at first
2. Then press ‘s’ button. It’s rotation is with respect to *global* z-axis then x-axis instead





Rotating an Object to “look” at a destination

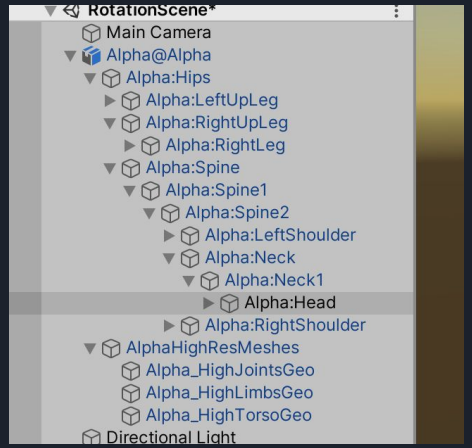
Our goal now is to drag the ball, and then make the avatar’s head rotate and “look” at wherever the ball is. Create a new script called DragObject.cs and attach it to the ball in the scene. Paste the code on the right.

Study the script carefully to understand how this works. Basically OnMouseDown() is triggered when you click on the ball. You will then find the z-coordinate (because screen is 2D, you dont want your mouse drag to change the depth), and make it constant as you drag left-and-right.

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

public class DragObject : MonoBehaviour
{
    private Vector3 mOffset;
    private float mZcoord;

    private Vector3 GetMouseWorldPos(){
        Vector3 mousePoint = Input.mousePosition;
        mousePoint.z = mZcoord;
        return Camera.main.ScreenToWorldPoint(mousePoint);
    }
    void OnMouseDown(){
        mZcoord = Camera.main.WorldToScreenPoint(gameObject.transform.position).z;
        mOffset = gameObject.transform.position - GetMouseWorldPos();
    }
    void OnMouseDrag(){
        transform.position = GetMouseWorldPos() + mOffset;
    }
    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Rotating an Object to “look” at a destination

Create a new script called Aim.cs with the following content. Then attach it to the avatar’s head. Run the program and the avatar will “look” at the ball.




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

public class Aim : MonoBehaviour
{
    [SerializeField]
    private Transform  sphere;
    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(StopAnimation());
    }

    IEnumerator StopAnimation()
    {
        yield return new WaitForSeconds(.1f);
        gameObject.GetComponentInParent<Animator>().enabled = false;
        yield return null;
    }


    // Update is called once per frame
    void Update()
    {
        // formula calculating direction to look at
        // direction = destination - source
        Vector3 directionToFace = _sphere.position - transform.position;

        //access current rotation to be = Quaternion Look Rotation
        //this is immediate, choppy
        transform.rotation = Quaternion.LookRotation(directionToFace);
}
}
```

Smooth “look”

Use Quaternion.Slerp instead for smoother rotation. What is “Slerp”?

In computer graphics, ***Slerp*** is shorthand for spherical linear interpolation, introduced by Ken Shoemake in the context of quaternion interpolation for the purpose of animating 3D rotation.

Again, the math behind this is kind of complicated. You’re welcome to consult me after class if you’d like to know (or consult your fellas in 50.017).

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

public class Aim : MonoBehaviour
{
    [SerializeField]
    private Transform  sphere;
    // Start is called before the first frame update
    void Start()
    {
        StartCoroutine(StopAnimation());
    }

    IEnumerator StopAnimation()
    {
        yield return new WaitForSeconds(.1f);
        gameObject.GetComponentInParent<Animator>().enabled = false;
        yield return null;
    }


    // Update is called once per frame
    void Update()
    {
        // formula calculating direction to look at
        // direction = destination - source
        Vector3 directionToFace = _sphere.position - transform.position;

        //access current rotation to be = Quaternion Look Rotation
        //this is immediate, choppy
        //transform.rotation = Quaternion.LookRotation(directionToFace);

        //to control better we use SLERP
        Quaternion targetRotation = Quaternion.LookRotation(directionToFace);
        transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation,
Time.deltaTime);

    }
```

Extra: Using RayCast to see if you clicked a particular object

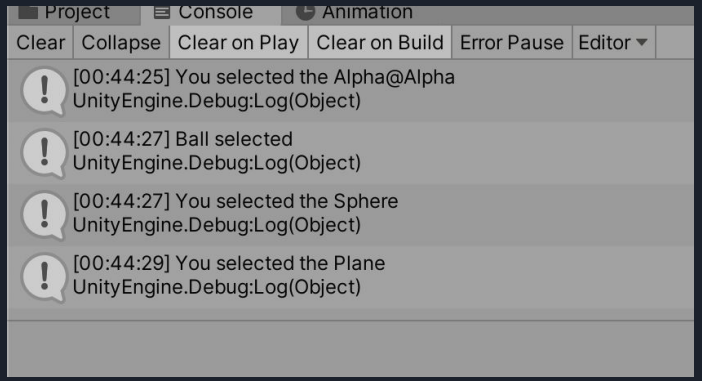
Open SpawnCube.cs and add this method inside Update()

```
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(ray, out hit, 100.0f))
            {
                if (hit.transform.tag == "Player"){
                    StartCoroutine(ScaleMe(hit));
                }
                Debug.Log("You selected the " + hit.transform.name); // ensure you
picked right object
            }
        }
```

And this Coroutine:

```
    IEnumerator ScaleMe(RaycastHit hit)
    {
        hit.transform.localScale *= 1.2f;
        yield return new WaitForSeconds(0.5f);
        hit.transform.localScale /= 1.2f;
    }
```

Run the program and you see the cube “expand” when you selected it. Otherwise it won’t. The raycast method casts a ray from the screen point and find the **FIRST** gameobject it intersects. It will print the name of gameobject intersected, but only add the effect on the RotationSampleCube prefab due to its tag.



Summary

In this Lab, we have learned how to:

1. Use object pooling so to not overload your CPU during runtime
2. Expand object pool as we go
3. Serialize selected fields
4. 3D rotation basics using Quaternion
5. Selecting an object in the scene and drag them across without changing the depth (z-value)
6. Quaternion.Slerp
7. Extra: raycast

CHECKOFF

Make your game *reasonably more fun* by adding **TWO** more simple features, for example timer, more pickup items, bigger map, etc. One of the features have to implement ANY Quaternion API. Justify the CORE DRIVE behind each of your added feature.

Note: The checkoff regarding object pooling will be done in Part 6. So this checkoff only involve basic Rotations.

Record your screen while playing the game in 1 minute, showing which script uses any of the Quaternion API. Upload your video to Youtube/any other online platform and SUBMIT THE LINK on edimension.

Refer to edimension for due date.