Introduction to Unity Part 6 -- Endless & Parallax Scrolling, Basic FX, Application of Pooling

Week 4

Create a new 3D project, and import the assets from here.

Adapted from Unity Tutorial on Sound and Effects.

Background

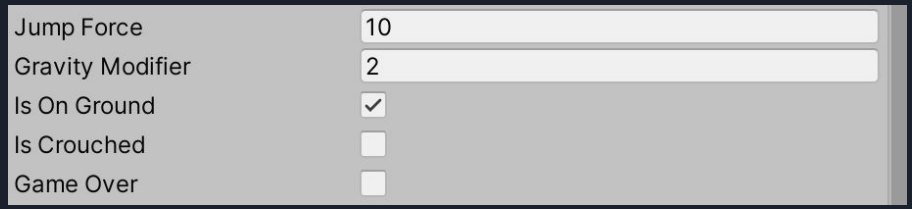
Open GameScene, and run it. You will see some crates spawning and moved to the left.

The character is T-posing. You can press space to make it jump over the small box, but it’s still T-posing nonetheless.

There’s two items spawned:

1. Normal crate -- you can jump over this
2. A stack of small crates -- you can’t jump over this

Background



Look at the Player GameObject, and its PlayerController Component. There's several public variables here:

1. Jump Force: how much we can bounce up the player GameObject when we press space
2. Gravity Modifier: how fast the player can drop back down on the ground
3. Three states: IsOnGround, IsCrouched, and GameOver.
    a. IsOnGround is true at first, and set to false when "Space" is pressed. The ground, has a collider component. Upon landing, the method `OnCollisionEnter` in the PlayerController script will be called. This is where we set IsOnGround back to false.
    b. IsCrouched is true for as long as 'c' button is pressed. Will be false when 'c' is no longer pressed.
    c. GameOver is true when player collide with "Obtacle" (normal crate), or "Destroyable" (stack of crates) when not crouching

The issue

1. The game is somehow… working
2. It will spawn obstacles endlessly, the normal crate and then the stack of small crates
3. The player can jump or crouch. The jump height is set such that he cannot jump over the small crates stack
4. If the player jumps over the normal crate, he survives
5. If the player IsCrouched state is true when colliding with the stack of small crates, he survives
6. Game over if condition (4) or (5) is violated

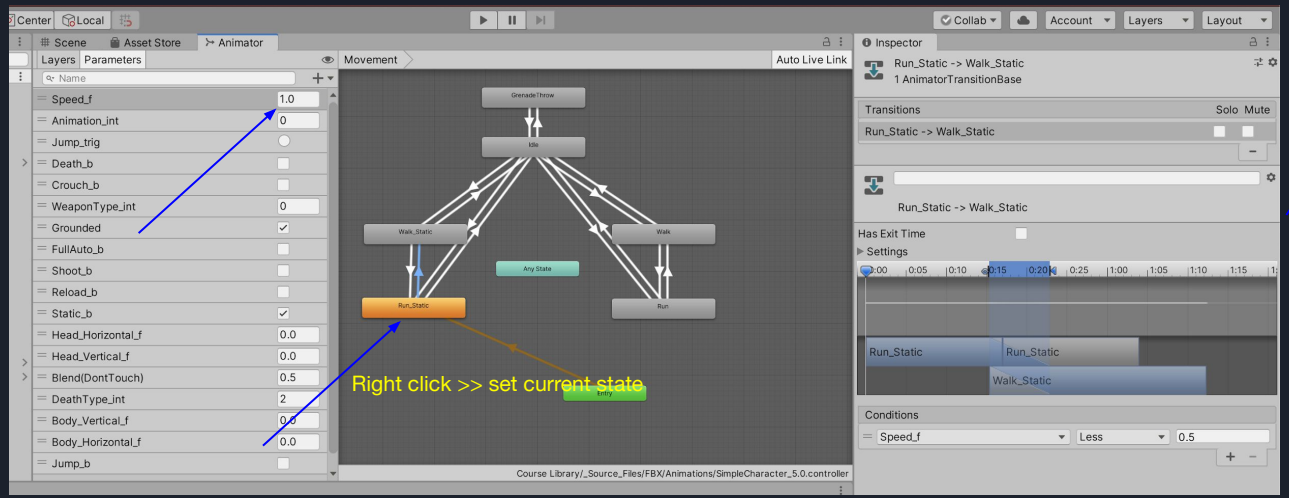
So the game is *functional,* however it is obviously poorly designed. Nothing wrong with endless scrolling -- survival -- kind of game idea. But there's no sound effect, no animation, no particle effects, and the background is boringly static.

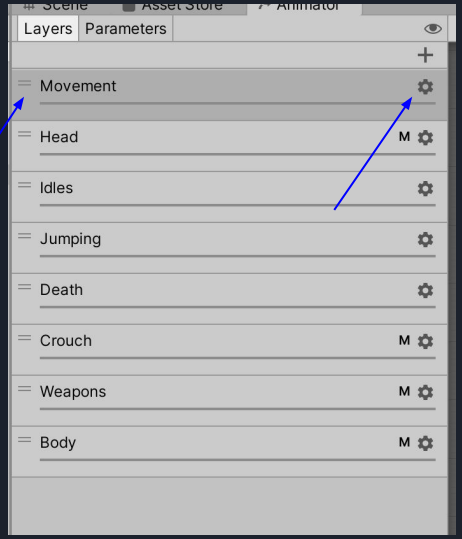
We are going to add all these and experience how small things can enhance the game greatly.

Step 1: Animate the Running

Open the Animator window and click on the player. There's tons of layers and tons of parameters in there. Take it as a good chance to study how complex animator state machine works. Click on the gear signs on the layers, and study about "mask" and blending mode. *Might be useful for your game in the future*. Anyway,

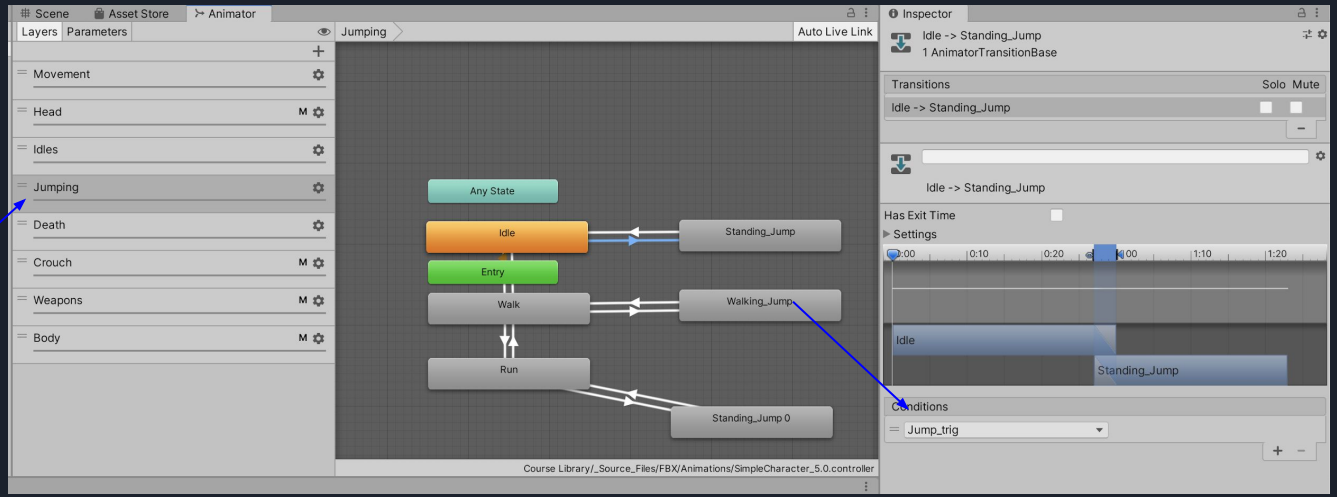
1. Modify the "Movement" layer to match the screenshot below. The parameter: Speed_f > 1 prevents moving to Walk_static state
2. Test run. The man should now be running-in-place all the time.





Step 2: Trigger the jumping animation

Go to the Jumping layer, and click the highlighted arrow as shown. Notice that this transition is triggered only if Jump_trig parameter is set to true. Let's do this in the script.



Step 2: Trigger the jumping animation

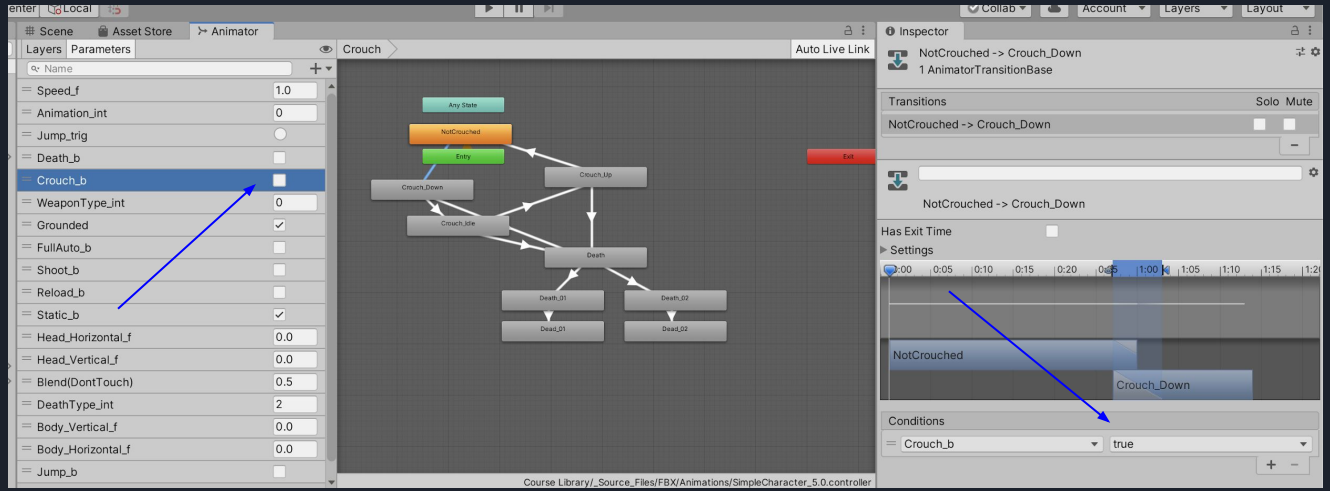
Inside PlayerController.cs, in Update() method, add the coloured line.

Test run and press ‘c’. You should see the jumping animation overrides the original running animation when the trigger is true, and the player goes back to running state.

```
        if (Input.GetKeyDown(KeyCode.Space) && isOnGround && !gameOver)
        {
            playerRb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
            isOnGround = false;
            playerAnim.SetTrigger("Jump_trig");
}
```

Step 3: Animate the Crouching

Click on the Crouch layer and observe the transition from NotCrouched to Crouch_Down. This happens only if the boolean parameter Crouch_b is “True”. Remember this is NOT a trigger parameter type. You need to set this to “False” when you want the player to get back to the NotCrouched state (after transiting to Crouch_Idle >> Crouch_Up).



Step 3: Animate the Crouching

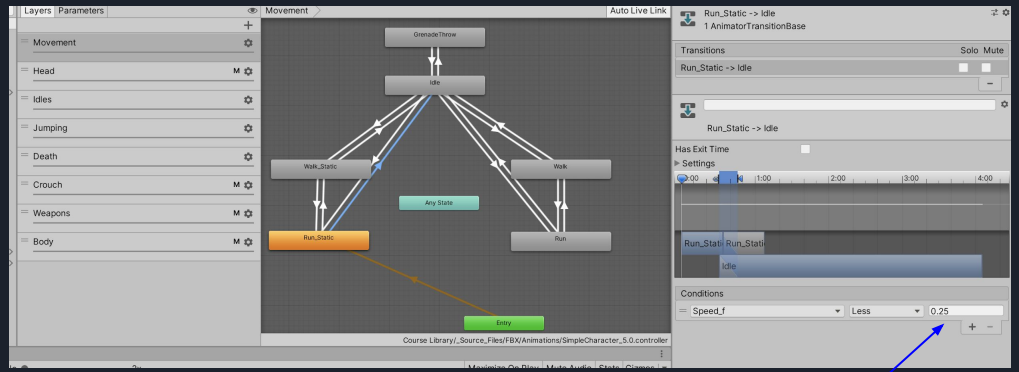
Add this coloured code to trigger the transition to Crouch_Down inside PlayerController.cs, and to stand back up when ‘c’ is no longer pressed. Test it and see the character crouch for as long as ‘c’ is pressed.

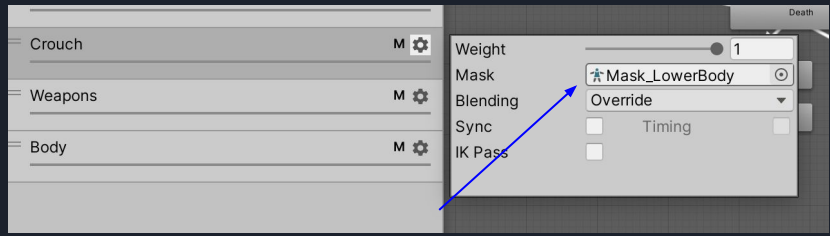
```
        if (Input.GetKeyDown(KeyCode.C) && isOnGround && !gameOver && !isCrouched)
        {
            playerAnim.SetTrigger("Crouch_b");
            isCrouched = true;
        }

        if (Input.GetKeyUp(KeyCode.C) && isOnGround && !gameOver && isCrouched)
        {
            playerAnim.ResetTrigger("Crouch_b");
            isCrouched = false;
        }
```

Step 4: Stop Hand Movement when Crouching

When we are crouching, we want the hand to stop swinging like as if our player is crouch...running. It is a little weird to do that. In order for the player to stop running by his hands, we need to change the “Movement” layer state into “Idle”

The Crouching Layer has a “Lower Body” as the mask, so the upper body is still affected by other layers. In order to go back to “Idle” state in the Movement layer, we need to temporarily set Speed_f to a small value (<0.25).





Step 4: Stop Hand Movement when Crouching

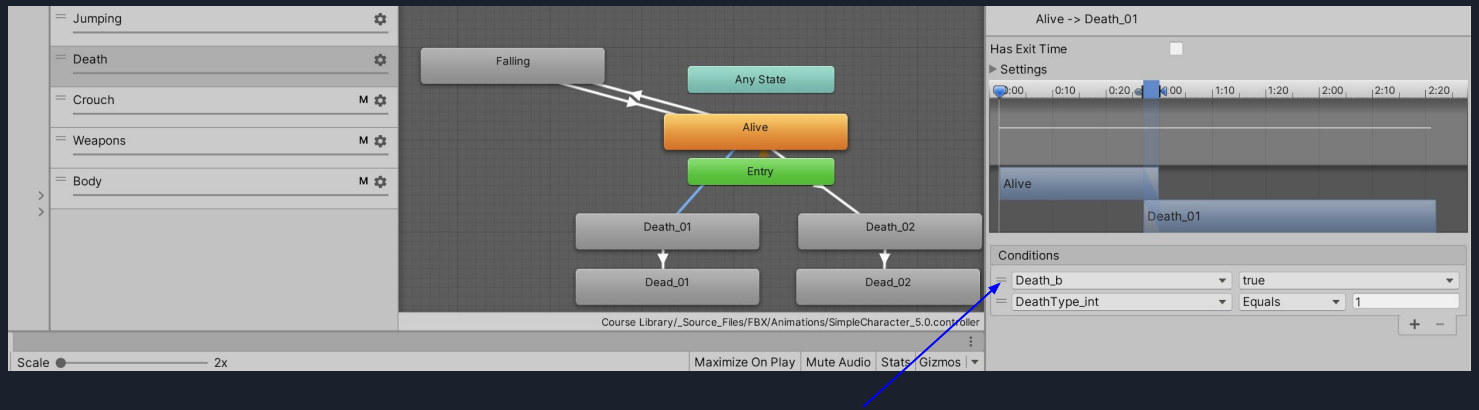
Add the colored code to PlayerController.cs. Test it and now you should see the player's hands stop moving while crouching.

```
        if (Input.GetKeyDown(KeyCode.C) && isOnGround && !gameOver && !isCrouched)
        {
            playerAnim.SetTrigger("Crouch_b");
            playerAnim.SetFloat("Speed_f", 0.1f);
            isCrouched = true;
        }

        if (Input.GetKeyUp(KeyCode.C) && isOnGround && !gameOver && isCrouched)
        {
            playerAnim.ResetTrigger("Crouch_b");
            playerAnim.SetFloat("Speed_f", 1.0f);
            isCrouched = false;
        }
```

Step 5: Add Death Animation

When death condition happens (`deathTasks` method is called), the game is over. We want to animate this as well. We can trigger the “Death” layer of animator to take over if we have the conditions. Add this line inside `deathTasks()` method in PlayerController.cs. Test that the animation happens when deathTasks() is called. You can add the debug line to test.

```
        //set death animation
        playerAnim.SetBool("Death_b", true);
        playerAnim.SetInteger("DeathType_int", 1);
```



Step 6: Add Sound Effects

We want to add various sound effects:

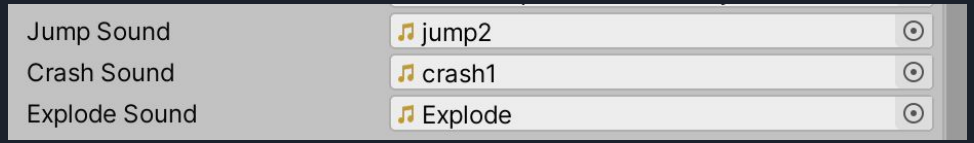
1. When the player is jumping
2. When collision with the small crate stack happens
3. When collision with the normal crate happens

We can just have **one** audio source attached to the Player, and various audio clips to be played during these events. Add these three public variables in PlayerController.cs:

```
    public AudioClip jumpSound;
    public AudioClip crashSound;
    public AudioClip explodeSound;
```

Step 6: Add Sound Effects

Link up the clips in the inspector. You can choose whichever sound clips you want though, or download your own.



Then play the sound effect by adding the code inside Update() for jumping sound:

```
        if (Input.GetKeyDown(KeyCode.Space) && isOnGround && !gameOver)
        {
            playerRb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
            isOnGround = false;
            playerAnim.SetTrigger("Jump_trig");
            playerAudio.PlayOneShot(jumpSound);
}
```

Using PlayOneShot in Unity

PlayOneShot works just like the standard Play function, but with a few key differences:

1. **PlayOneShot doesn't interrupt a clip that's already playing on the Audio Source**
   This makes it great for repeating sounds, like machine gun fire, footsteps, swings, hits, etc.
2. **PlayOneShot requires you to specify the Audio Clip to play when you call the function**
   This makes it easy to use different clips for variation, instead of swapping out the Audio Source's clip field.
3. **You can set the volume scale of the clip when calling PlayOneShot.**
   This will modify the volume of the clip (on a float scale of 0-1), great for randomising the sound without affecting the overall volume of the Audio Source.

Step 6: Add Sound Effects

For the collision sound effect with the normal crate, add this line inside `deathTasks()` method:

```
        playerAudio.PlayOneShot(crashSound);
```

For the collision sound effect with the stack of small crates, add this line inside `OnTriggerEnter` method. Test that all sound effects are working properly.

```
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Destroyable") && !gameOver)
        {
            if (!isCrouched)
            {
                deathTasks();
            }
            else
            {
                playerAudio.PlayOneShot(explodeSound);
                Debug.Log("Explode sound!");
            }
        }

    }
```

Step 7: Add Particle FX

The impact can be much better if you can add particle system as the effect of “collision” with the normal crater before the player collapses.

The particle system prefab is already made for you. We will learn about this in the later Parts of the lab. For now, lets just learn how to use it.

Add this line inside PlayerController.cs: `public ParticleSystem explosion;`

And add this line inside `deathTasks()`:

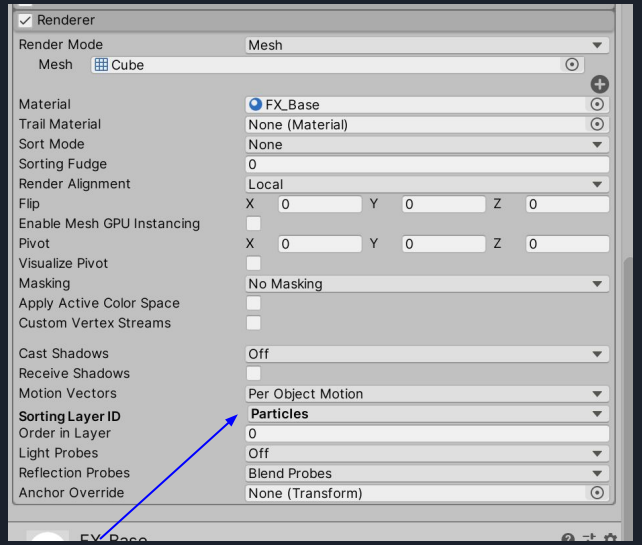
```
explosion.Play();
```

Test that the particle system is triggered during collision condition that caused gameOver.

About Particle System Renderer

If for some reason, the explosion particle is not shown properly, or hidden behind background, open the FX_Explosion_Smoke and in the ParticleSystem inspector open the Renderer tab.

See the Sorting Layer ID and set it to be “Particles”, or whatever that is the last because it’s the one that will be rendered “in front” of the rest.



Step 7: Add Particle FX

Finally it'll be fun to add tiny "dirt" particles when the player is running, to simulate "kicking" of something off the ground.

Add this line inside PlayerController.cs:

```
    public ParticleSystem dirtparticle;
```

Activate it at `Start()`:

```
        dirtparticle.Play();
```

You can stop it using:

```
        dirtparticle.Stop();
```

Add the Play() and Stop() instruction above in appropriate places in the code, i,e: when jumping in the air, when c is pressed, when `deathTasks()` happens, etc. As usual, dont forget to test it.

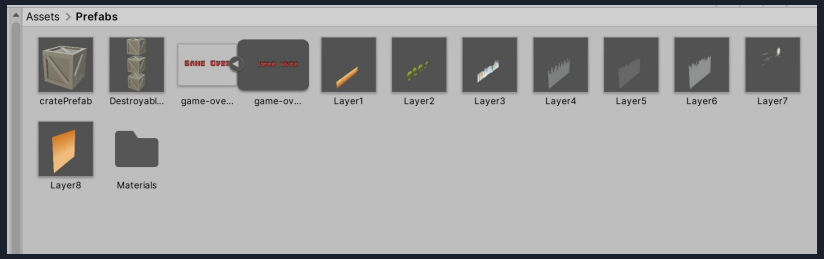
Step 8: Adding Parallax Scrolling Background

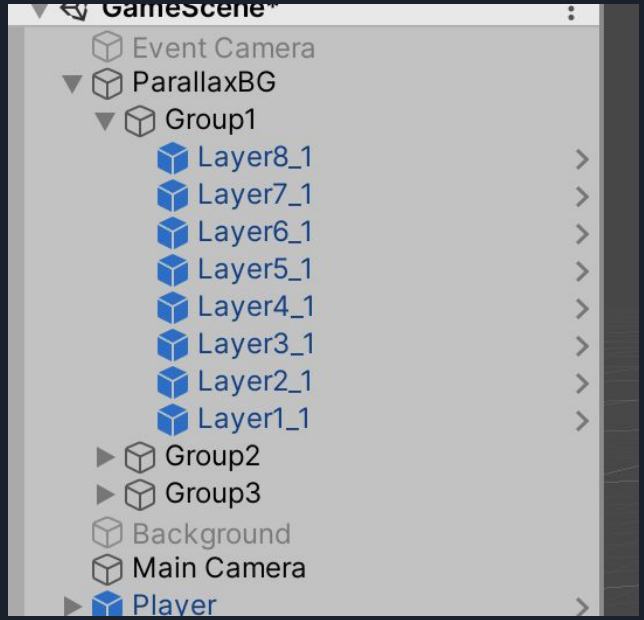
Parallax scrolling is a technique of moving layers of background at a different speed to give a sense of “3D effect” and “depth”. Deactivate “Background” and Activate ParallaxBG. You should see the following (or not, if its messed up, change the x-transform properly):

Parallax BG

Expand ParallaxBG and notice there's 3 groups of backgrounds. Each group is consisted of 8 layers.

Go to the Prefabs folder to observe each layer. The idea is to scroll each layer to the left with different speeds, and spawn it back on the right once its out of view, hence giving the endless scrolling effect.

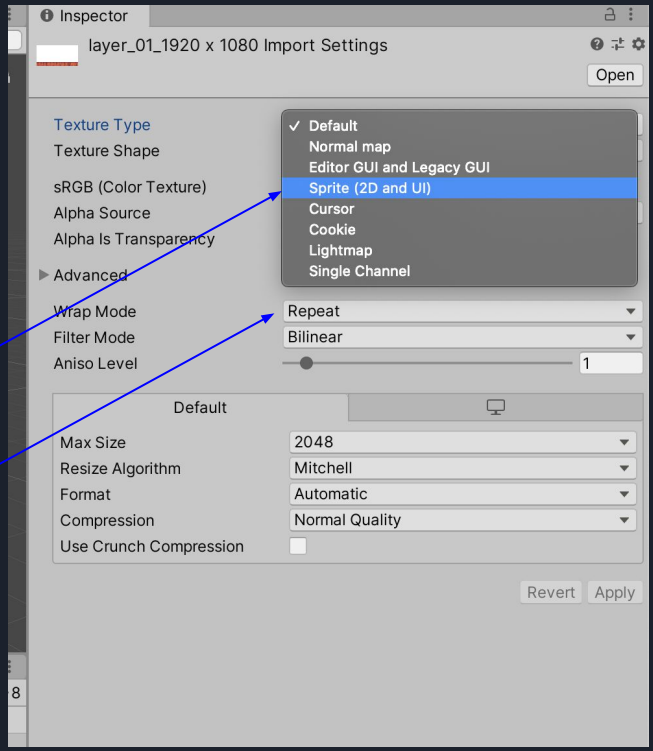




Info: Creating Textures

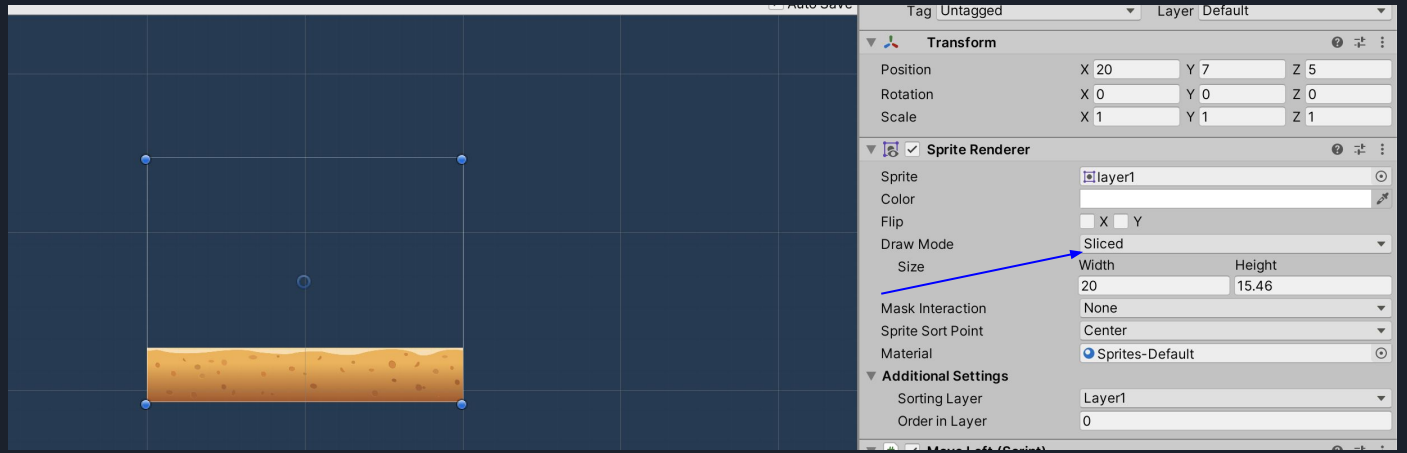
When you first drag PNG / JPEG file to Unity Assets folder, you need to change its inspector Texture Type value if you want it to be selectable as Sprite option.

You can also change its “Wrap Mode”, etc for more fine-grained tuning.



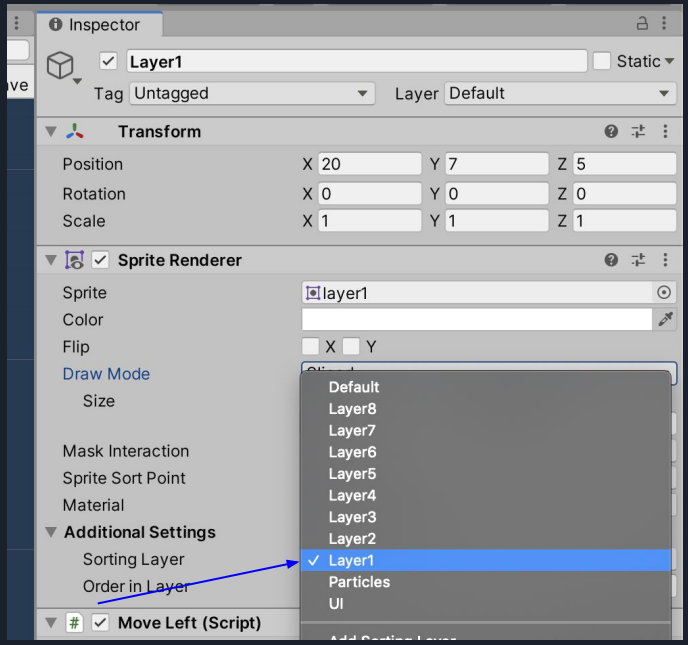
Info: Sprite Renderer Size

You might want to play around with the “Draw Mode” and “Mask Interaction” (if there’s Sprite Mask component attached to the object) to determine size and “viewable region”. Draw mode of “Sliced” will try to stretch the sprite with the size of the sprite. Right now we use “Slice” and the width of the sprite is set into 20. So the texture is fitted to The other option: Tiled, will repeat the sprite if the size of the sprite is bigger than the texture size, or cropped if otherwise. this width without getting cropped.



Info: Sorting Layer

To ensure that Layer 1 is rendered before Layer 2, and so on, and Layer 8 is at the back, make sure the sorting layer option of the Sprite Renderer is properly set for each prefab type. We have played with Sorting layer before in Part 1. This is the same, just that we have 8 layers instead.



Step 8: Adding Parallax Scrolling Background

The first step is to move the background to the left. Add the MoveLeft.cs script to ALL Layer1 to Layer8 Prefabs. Set the speed to 10 for ALL.

Test run and observe all layers moving at the same speed.

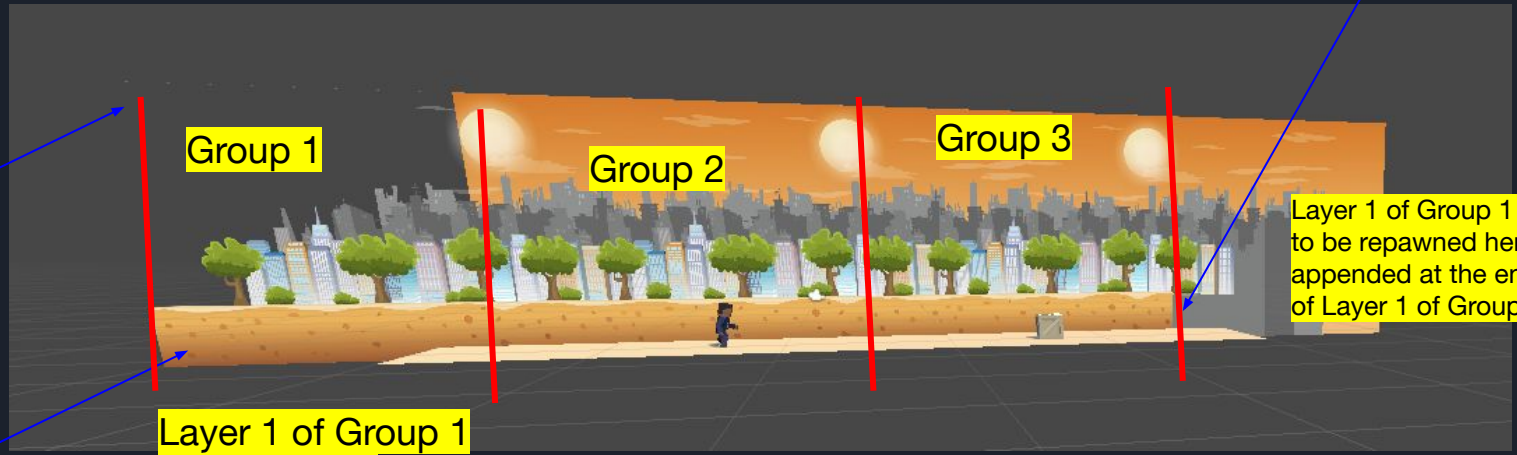
Now, set different speed for each LayerX prefab. Speed = 10 for Layer1, Speed = 9 for Layer2, etc and finally Speed = 3 for Layer 8.

Test run and observe the 8 layers moving at different speed.

Step 9: Respawn the background

One of the last steps to finetune is spawning of the background when it is out of view.

What we want is to RESPAWN the object layer X of the leftmost group that is out of view and APPEND it at the back of the (corresponding object Layer X) in the rightmost group. For example, Layer1 as below. This is VERY tideous to do, so just read the next slides and skip to slide 30 for a better way to actually implement.



Create a new script called RepeatBackground.cs, with the following variables:

1. `startPos` is a Vector3 type that dictates where we need to respawn
2. `endBoundaryX` dictates the x-value of objects that are out of camera's view
3. `reference` is the GameObject of which x-value we need in order to re-spawn this GameObject

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

public class RepeatBackground :
MonoBehaviour
{
    private Vector3 startPos;
    private float endBoundaryX = -20;
    public GameObject reference;
}
```

In the Update() method, add the following code. Let's do some math on where we want to respawn this object. Remember the sprite width is 20, and anchored at the center. So if we want to respawn the object at the back of the rightmost layer, then we need to respawn it at the reference's position + 20 (derived from 10 + 10 because transform.position is counted from the center).

However if we +20, even though the math is right, this will only be rendered in the "next" cycle of which the transform.position.x of the reference object might have been moved by 1 unit. Therefore we need to take this into account, and +19 instead of 20. If you try to +20 here, you will see a unit gap. You can fix this the proper way by stating the order of script execution (MoveLeft.cs and this RepeatBackground.cs).

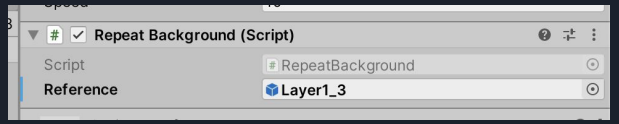
```
    // Update is called once per frame
    void Update()
    {
        if (transform.position.x < endBoundaryX)
        {
            if (reference != null)
            {
                startPos = new Vector3(reference.transform.position.x + 19, transform.position.y,
transform.position.z);
                transform.position = startPos;
            }
        }
    }
```

Attach RepeatBackground.cs to ALL LayerX PREFABS. Then the tedious work is to link up the “Reference” gameobject for each layer and for each group in the scene.

Here’s the idea:

1. For each LayerX_1 GameObject in Group1, your reference is the corresponding LayerX_3 GameObject in Group 3.
2. Similarly, Group3’s reference is object in Group 2
3. Group2’s reference is object in Group 1

You must manually link up the Reference for ALL layers in each group, so in total there’s 24. Then test run, and you should see the parallax scrolling and respawning, forming a seamless endless background.

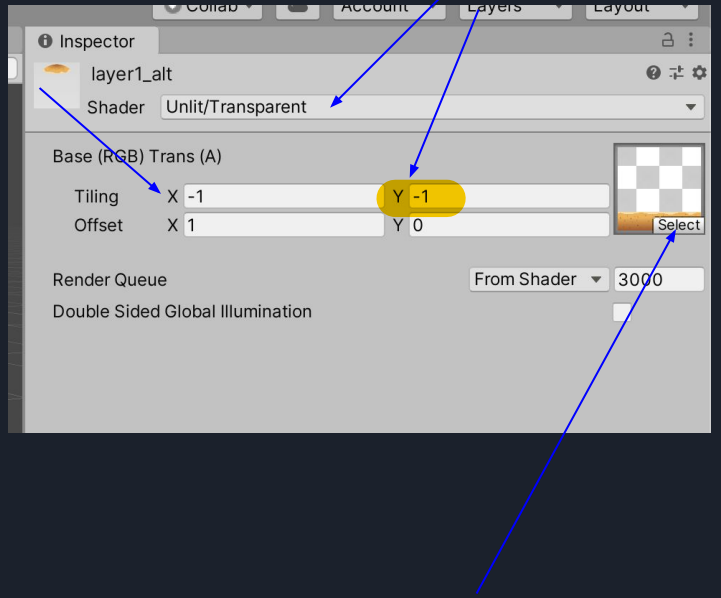


A better method to create parallax background

The method that's just taught to you might be *easier* to understand what is a parallax background and how to make it work quick at this point (assuming you have zero knowledge in Unity in the beginning of the semester).

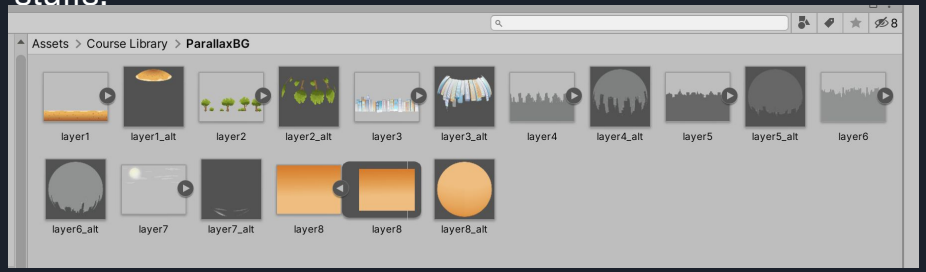
However, a better way instead is to "scroll the material" of the background. We can't do this using Sprite Renderer though, we need to do this using Mesh Renderer (the 3D version of SpriteRenderer) instead.

Let's do this quickly by creating the materials for Mesh Renderer. Go to Assets >> Course Library >> ParallaxBG, and create layer1_alt to layer8_alt materials from the Textures as such.

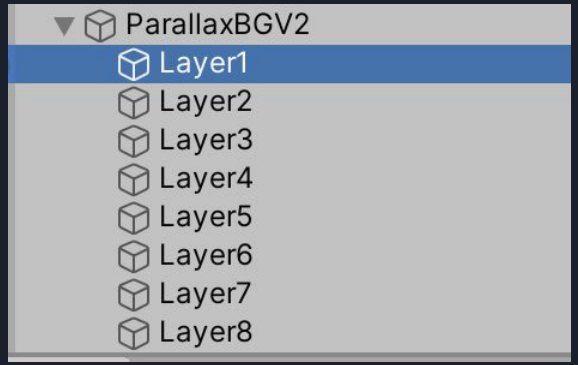


A better method to create parallax background

Once you create all 8, you should have these stuffs:

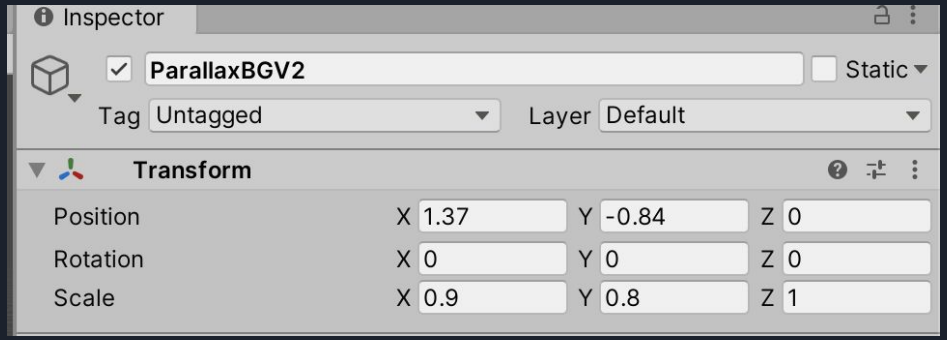


Then Create an Empty GameObject: ParallaxBGV2, and create 8 children (3D >> Plane)



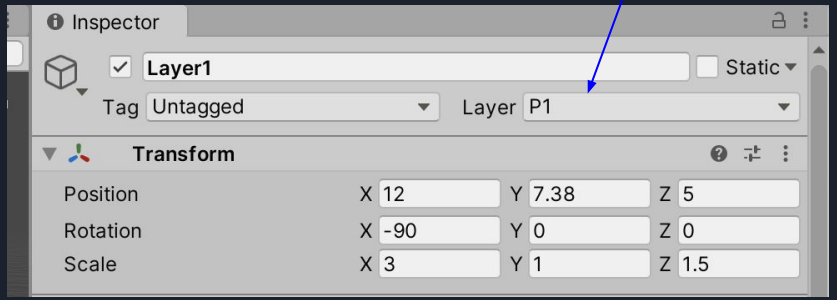
A better method to create parallax background

Adjust the transforms:



And here's transform for each Layer child object.

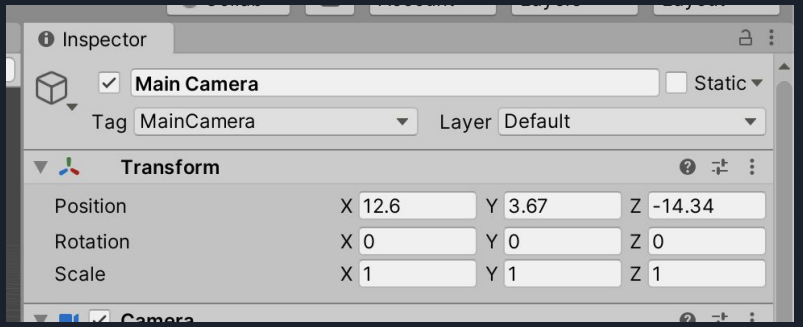
They're all set to the same value. Create 8 layers called P1 to P8 and set them accordingly.

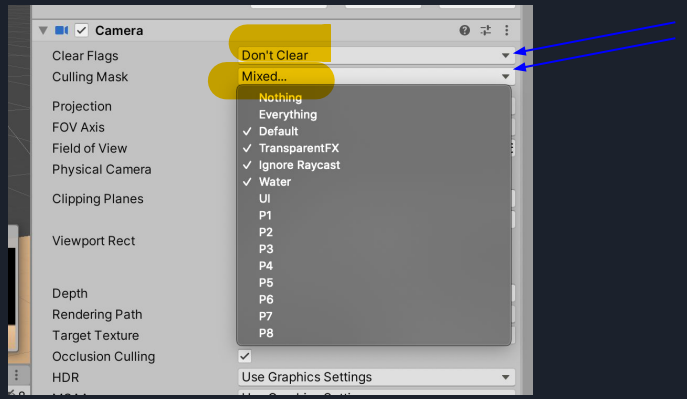


A better method to create parallax background

If it doesn't fit well with the main view, it helps to adjust the main camera. Here's the coordinate that's used for the demo.

Set the Main Camera's Clear Flags and Culling Mask. You should see this for now in your game preview, assuming you diable the previous ParallaxBG version.



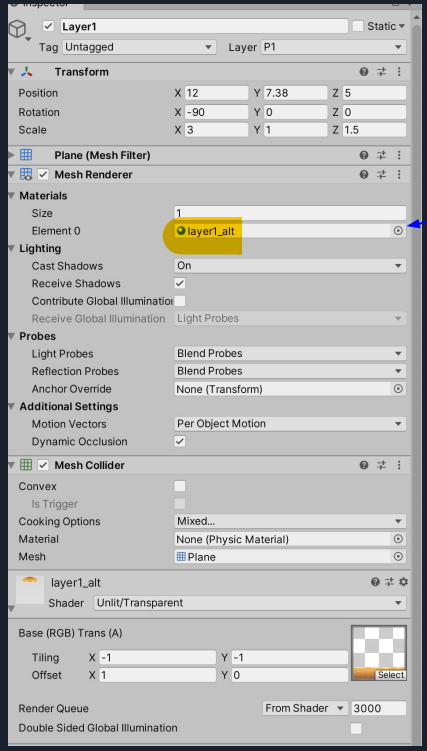


A better method to create parallax background

Now for each LayerX GameObject under ParallaxBGV2, set the Element0 of MeshRenderer Component into layerx_alt that you create earlier.

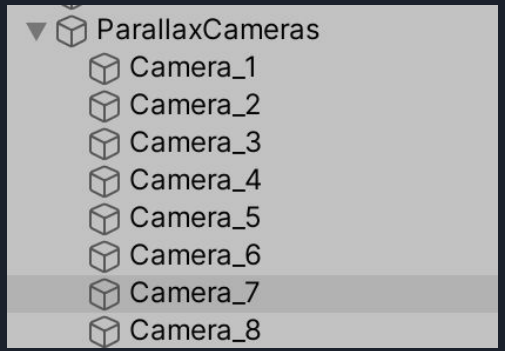
We still cant see anything, that's OK.

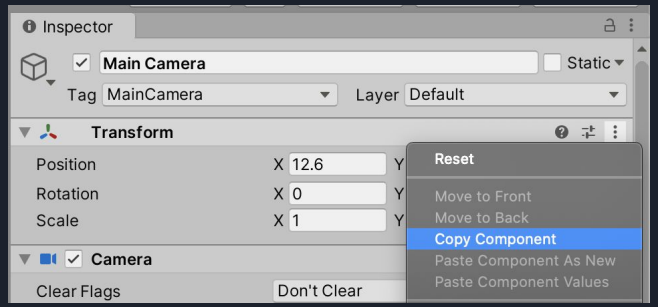
We need new cameras to see each layer.



A better method to create parallax background

Create a parent gameobject ParallaxCamera at Transform location 0,0,0. Then, create 8 Cameras as its child objects (delete the AudioListener components).



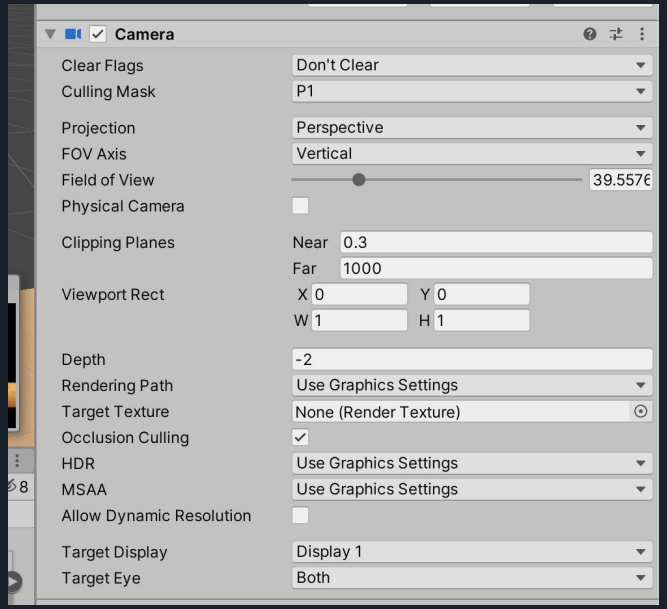


For each Camera_X, copy the Transform component values AND Camera component values from the MainCamera. This is to ensure that ALL CAMERAS are on the same location, just that each camera will be rendering a different thing.

A better method to create parallax background

Edit the Camera Component of each Camera_X. Here's the idea:

1. The culling mask for each Camera_X has to be ONLY PX layer. This is to ensure that each Camera_X sees LayerX.
2. The "DEPTH" field tells us what's rendered in front of what. Bigger values means "in front", smaller values means "behind". Camera_1 starts at Depth = -2, and Camera_8 is at Depth = -9. Decrease them by 1 for each Camera.



A better method to create parallax background

At this point, you should see this in the Game tab. Now our job is to scroll the background at different speed.

A better method to create parallax background

Create a new script called ParallaxScroller.cs

Put these variables:

```
    public Renderer[] Layers;
    public float[] speed;

    public float offset = 0.0f;
```

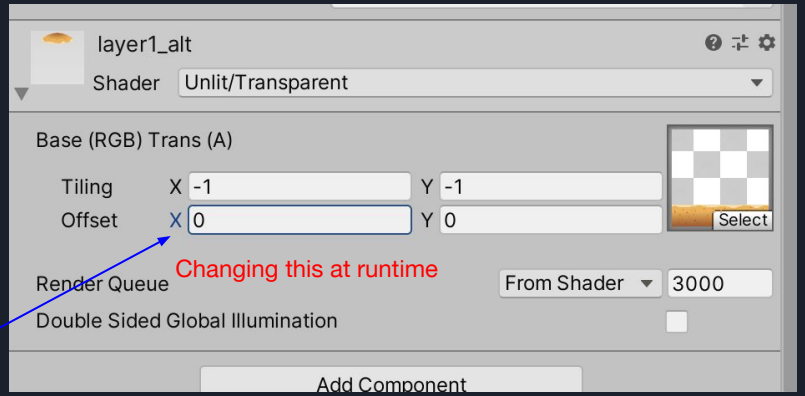
And edit the update method:

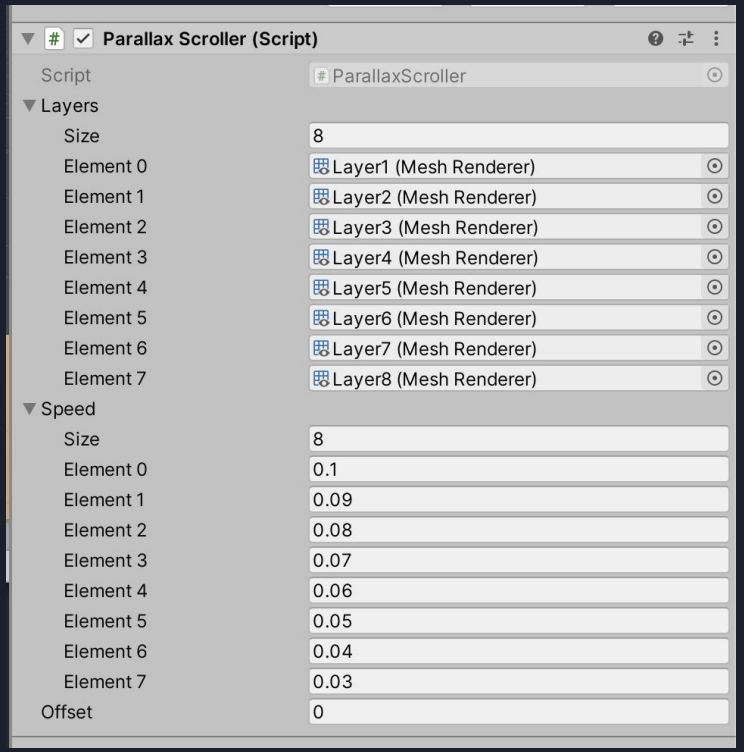
```
    private void Update()
    {
        if (offset == 1.0f) offset = 0.0f; //reset offset

        for(int i = 0; i< Layers.Length; i++){
            float layerOffset = offset * speed[i];
            Layers[i].material.mainTextureOffset = new
Vector2(layerOffset, 0);
        }
        offset += 0.01f;
    }
```

A better method to create parallax background

The idea is to “shift” the offset of the material in MeshRenderer during runtime, thus giving it an illusion of being scrolled.

Attach the script to ParallaxBGV2 GameObject. Fill up the fields as shown on the right





A better method to create parallax background

You can edit the speed in the inspector of ParallaxScroller script for faster effects. It is entirely up to you.

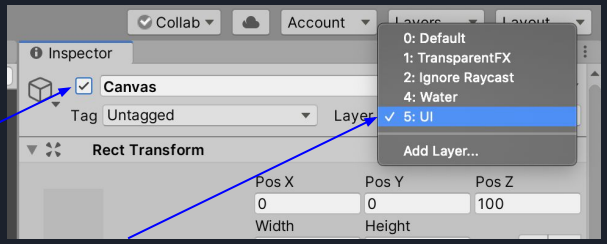
This way, we simply scroll the material of the MeshRenderer at runtime at different speeds to give the illusion of the player going forward, when actually it is in place. If you want to actually move the player (changing its x-value transform), this method works too, because the cameras are independent. This is what making this method better than the previous one.

Why can't we do this using SpriteRenderer, i.e: changing offset at runtime? Well, simply because SpriteRenderer isn't meant to change its texture during runtime, so it doesn't have this functionality. You need to use MeshRenderer for this.

Step 10: Adding GameOver Overlay

We all know how to create a Canvas, with SpriteRenderer in it and we can enable/disable it when the time comes like we did in Part 1 and Part 4. Let's do it differently now with multiple cameras.

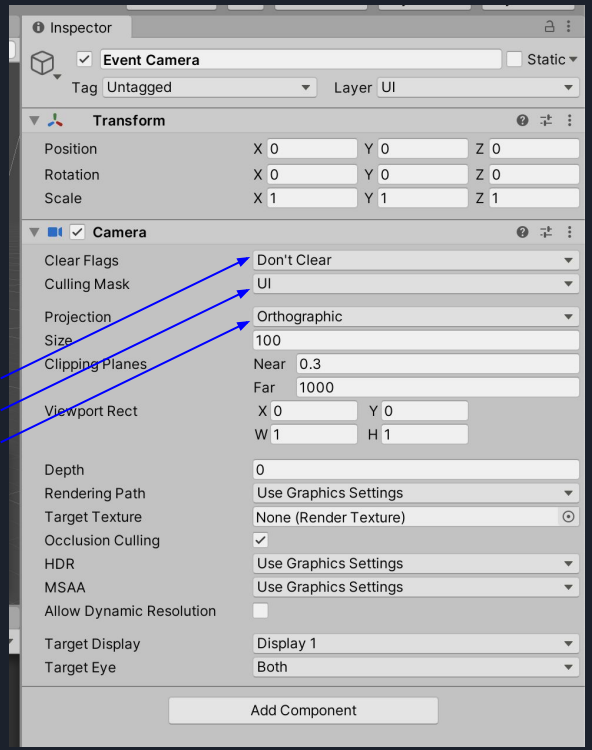
Enable the Canvas Object and change its GameObject Layer to UI (Add layer if its not already there).



Step 10: Adding GameOver Overlay

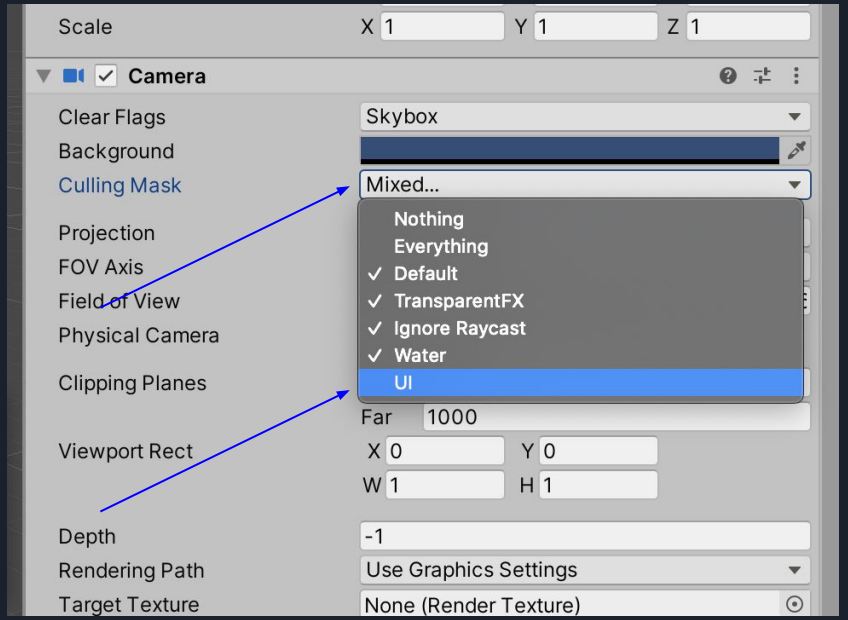
Create a new Camera GameObject in the scene and name it EventCamera:

1. Remove AudioListener Component. We just want the game to have 1 listener per scene.
2. Edit the Culling Mask to just include UI. This means it ONLY RENDERS the UI Layer set in the GameObject.
3. The Camera Preview must be as shown.
4. Set Clear Flags to be “Don’t Clear” to make it have black (no background) at all. This allows us to “add” the view of this camera + the Main Camera when game is over.
5. Set to ORTHOGRAPHIC projection. We don’t need the effect of perspective for the game over sprite.



Step 10: Adding GameOver Overlay

Change the Culling Mask of the MAIN CAMERA to NOT include the UI layer. This way it wont render the Game Over canvas anymore.



Step 10: Adding GameOver Overlay

Add this variable in PlayerController.cs:

```
    public GameObject EventCamera;
```

Then enable it inside `deathTasks()` function:

```
        EventCamera.SetActive(true);
```

This way the camera's view is only rendered out to Display1 (our game screen) when the game is over.

Others

To tell the SpawnManager to stop spawning, we create an event in PlayerController.cs of which SpawnManager subscribes to. Upon death, the code

```
        // notify if there's subscriber
        if (notifyDeath != null) notifyDeath();
```

is run. This cast the handler subscribed in SpawnManager:

```
    void updatePlayerDeath(){
        gameOver = true;
    }
```

And in turn, the `SpawnObstacle()` will no longer spawn anything else since the state gameOver is true. Of course SpawnManager can just check the PlayerController's gameOver state in `SpawnObstacle()` over and over. It is up to you to design your code, whether you want everybody to check a centralised state, or you want the change in state to notify everybody that's dependent on it.

Summary

In this Part, we have learned:

1. How to add basic FX (audio and visual) to enhance the game experience
2. Study the use of Animator in a more complex way using various layers, parameters, mask, and blending mode
3. How parallax background work
4. Respawning of background without create/destroy
5. Basics about SpriteRenderer Draw Mode and Texture Types
6. Adding another camera, setting Culling Mask and setting layers in GameObject to allow rendering on different cameras

CHECKOFF

1. Implement Object Pooling mechanism for the spawned objects. Right now in SpawnManager.cs and MoveLeft.cs, we repeatedly Instantiate and Destroy them at runtime, which can be very resource consuming (1.5%).
2. BONUS: Implement SINGLETON pattern for the SpawnManager (see Part 5 for details) -- 1.5%. SHOW the advantage of using this in your code, so probably you need 2 scenes to demonstrate the persistence. This will add bonus to your OVERALL grade, in whatever you are lacking. So take this opportunity wisely.
3. Vary the speed at runtime to increase the difficulty, OR randomise the distance of spawning. WARNING: this will mess up the effect of parallax background speed as well if you chose to settle with the “amateur” method, so do it with caution when modifying the `float speed` variable in MoveLeft.cs attached to cratePrefab and DestroyableCrate prefab (1.5%)

As usual, record a video showing your code and you providing quick explanations. Perhaps we need 2-3 minutes for this. Then upload a README file explaining how you did the bonus part if you do and WHY singleton pattern is useful for manager-type object. This should be obvious if you’ve been following the lessons / googling it (Failure to provide readme file will result in voided bonus marks). Give the video link as submission in e-dimension. Refer to edimension for due date.

Introduction to Unity Part 7 - Physics Engine Basics

You can only have one effector per GameObject. Meaning that, if a GameObject (with a Collider2D) already has an AreaEffector2D, then it cannot have PointEffector2D within it as well.

Week 5

Outline

Adapted from
https://www.raywenderlich.com/6184-unity-2017-2d-techniques-build-a-2d-pinball-game

Create a new 2D project and download the starter assets here. There's two items in there: usual asset package and ProjectSetting (to retain layers, and tags). Create a NEW 3D project, and paste the ProjectSettings folder into the project to overwrite it. And then, import the asset.

Our goal is to make a working pinball game. In a classic pinball game, we have many surfaces that add different types of **forces** to the ball. There's also "areas" that can boost / dampen the ball's movement.

We can do all this easily by defining the types of components in a GameObject, define its physics material, and let Unity does the numerical computation.

We will also add some basic animations using pre-defined textures, as well as adding impactful sound effects to enhance the game.

Open Sample Scene

Run the game. You quickly will notice THREE weird things happening here. Firstly, the right flipper falls off.

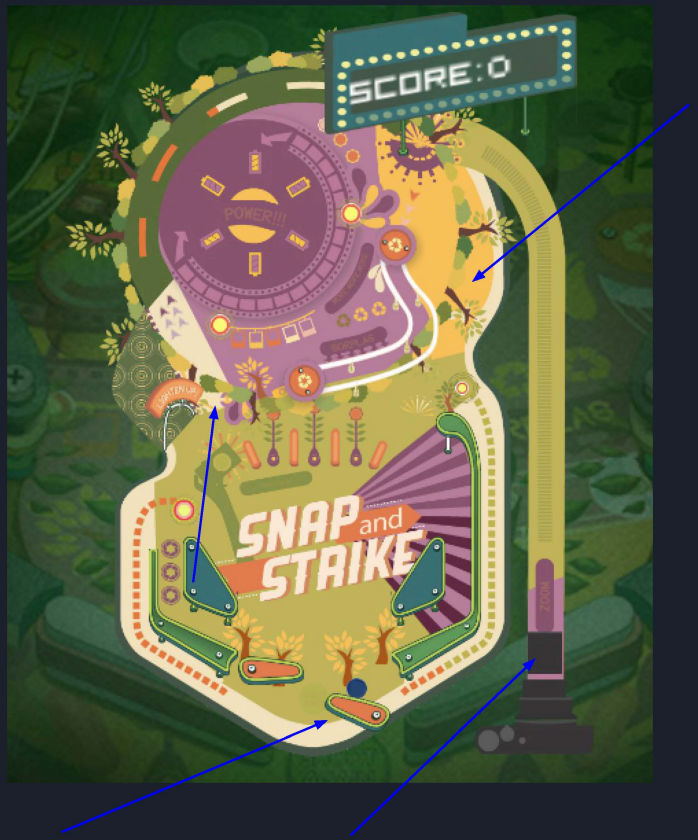
Secondly, the plunger floats weirdly.

Thirdly, the trees seem out of place.

If you restart the game by placing the ball elsewhere, you notice that there's no "bouncy" parts in this game either.

Notice also some error message in the console about many NullReferenceExceptions.

In this lab, we are going to fix these bugs.

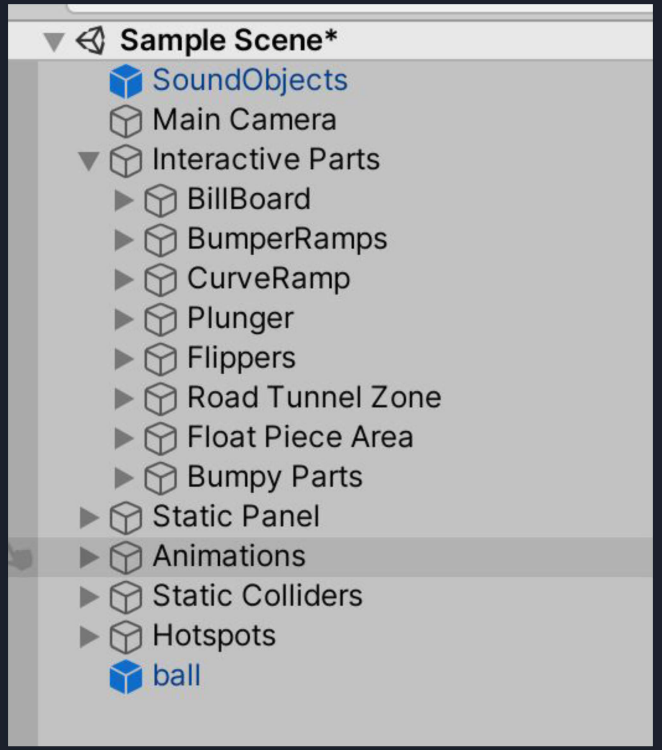


Open SampleScene

Look under Interactive Parts

Here exist all the game objects that can “interact” with the ball

We are going to add more functionalities to them one by one, and create their effects

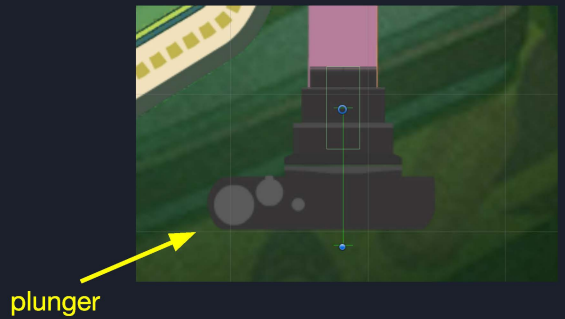


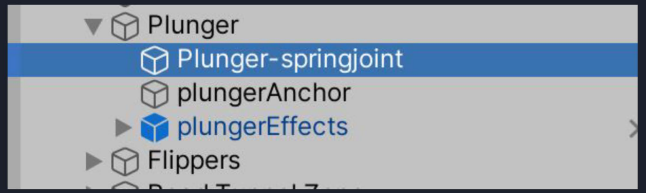
Step 1: Adding the Plunger using SpringJoint2D

A Plunger is the part where you launch the ball. The idea is that if you press “space”, it pull down the *spring* inside it, and when you release, the ball will be pushed upwards.

We do this by adding a SpringJoint2D component to the GameObject: Plunger-springjoint.

Let’s learn about how springJoint works first before editing to to achieve the picture



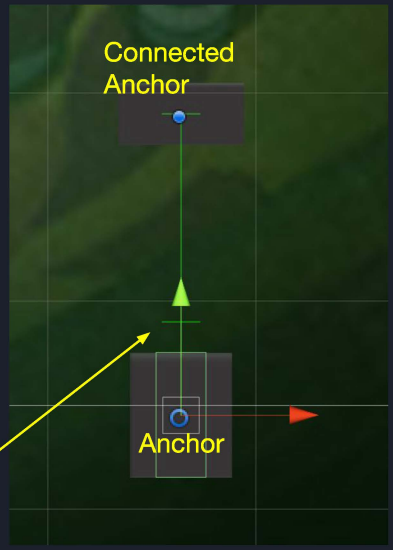


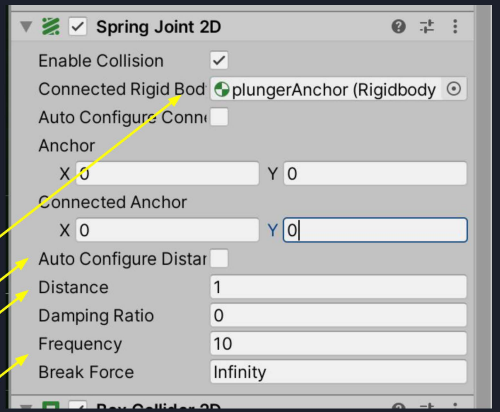
Anchor

SpringJoint2D

1. Drag Plunger-springjoint and plungerAnchor to the side and place it at some distance away as shown (anchor above).
2. Add spring joint 2D component to Plunger-springjoint.
3. Run the game, go to the scene tab and drag around Plunger-springjoint. You will notice the object swinging around due to the spring connected to the top object.
4. Play around with value of *Anchor* and *connected anchor* in the inspector to get the common sense on how spring works. You can also change distance, damping ratio, and frequency to get the perfect spring.

If you enable collision, it will collide with the object on the top where it is connected to.



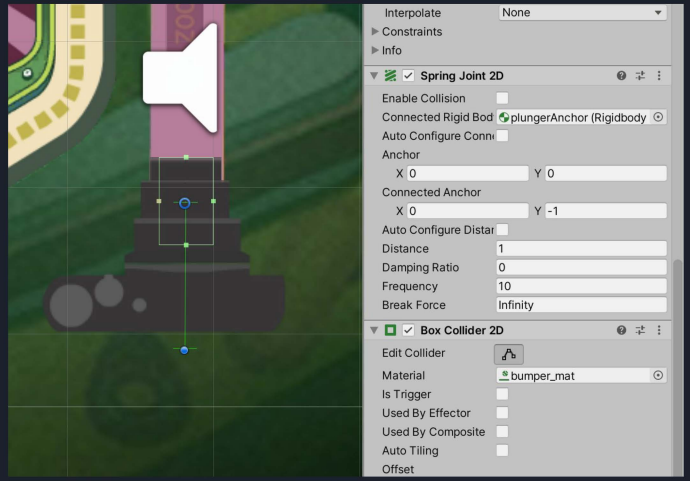


Now return the Plunger-springjoint and plungerAnchor back to its original position, and modify the SpringJoint2D component in Plunger-springjoint as shown.

Notice we also have the “BoxCollider2D” component inside Plunger-springjoint.

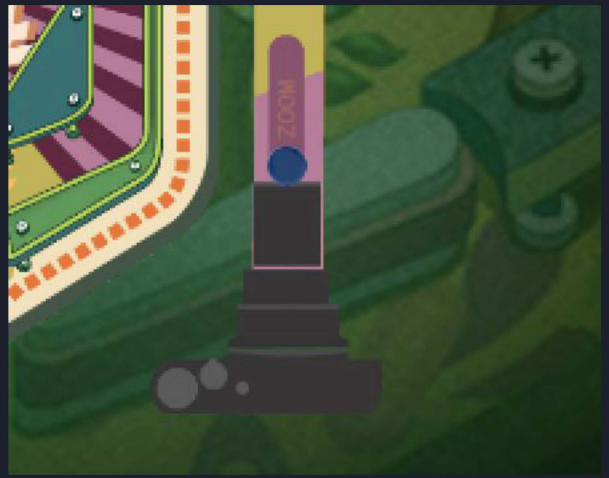
SpringJoint2D always require a RigidBody, and collider is handy to prevent the ball from falling through the object. Edit it so it fits the sprite. Notice the material inside the collider: to add the bouncy effect.

Test: Place the ball above the plunger and press run. It should fall on the plunger and has the springy effect.

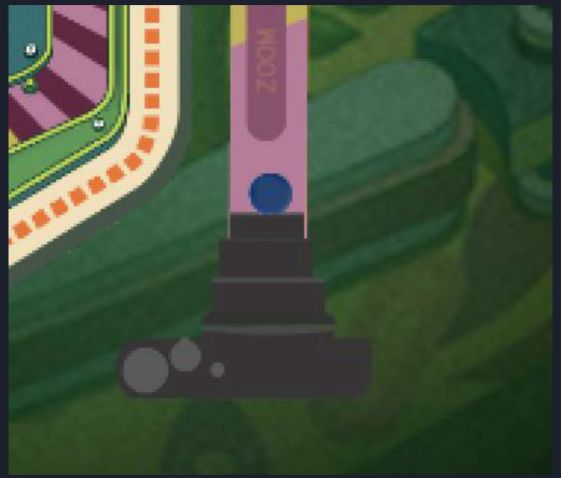


Note: be careful to not place colliders overlapping each other especially if they’re static objects. It will result in simulation error (they’ll vibrate)

Without SpringJoint2D, the colliders separate the two objects, resulting in one above the other. With the SpringJoint2D, the ball rests on the springy component



Before



After

Basic Terms Review: RigidBody vs Collider

- A RigidBody component makes the object a "physics object" -- it has mass, speed and can be knocked around. The physics engine moves RigidBodies, and checks against colliders (if they have any).
- TYPE OPTIONS (Dynamic? Kinematic? Static?) in RigidBody component determine whether the object is simulated or not.
- Having RigidBody without Colliders is OK, but then a few of these simulated RigidBodies won't affect each other's motion, although they can be still moved around
- A Collider component without any RigidBody in a GameObject is alright for non-movable objects like Walls, Obstacles, etc. HOWEVER, To detect collisions, a minimum of one of the two objects in the collision need a rigidbody.

Examples:

1. If you have a bullet, it needs a RigidBody, or it will just stay stationary where you fired/summoned it. Of course, you can write a script to edit its transform manually, but you can give this tedious task to Unity Physics Engine instead.

2. Moving platforms are typically just colliders moved via script without RigidBody, because their movements are simple. Anything that hits them will treat them as immobile walls, bouncing right off them without affecting or shaking these platforms. By conservation of energy, the actual results of a collision with a non-rigidbody will generally give more bounce, since we can't knock it backwards. However can set the bounce amount in physics materials of the Colliders too.

Basic Terms Review: RigidBody vs Collider

Note also in order for OnCollisionEnter and OnTriggerEnter to fire, you need:

1. Both GameObjects must contain a Collider component.
2. One must have Collider.isTrigger enabled, and contain a Rigidbody.
3. If both GameObjects have Collider.isTrigger enabled, no collision happens.
4. The same applies when both GameObjects do not have a Rigidbody component.

For OnCollisionEnter, it is called when this collider/rigidbody has begun touching another rigidbody/collider.

Do an experiment: If you have two objects -- one having RigidBody and Collider, the other just a Collider, will OnCollisionEnter be called on BOTH objects or only the one with RigidBody?

Please read the documentation first for more information and be certain what is called under what condition.

Read this for more information

Collision action matrix

When two objects collide, a number of different script events can occur depending on the configurations of the colliding objects' rigidbodies. The charts below give details of which event functions are called based on the components that are attached to the objects. Some of the combinations only cause one of the two objects to be affected by the collision, but the general rule is that physics will not be applied to an object that doesn't have a Rigidbody component attached.

Collision detection occurs and messages are sent upon collision

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Y				
Rigidbody Collider	Y	Y	Y			
Kinematic Rigidbody Collider		Y				
Static Trigger Collider						
Rigidbody Trigger Collider						
Kinematic Rigidbody Trigger Collider						

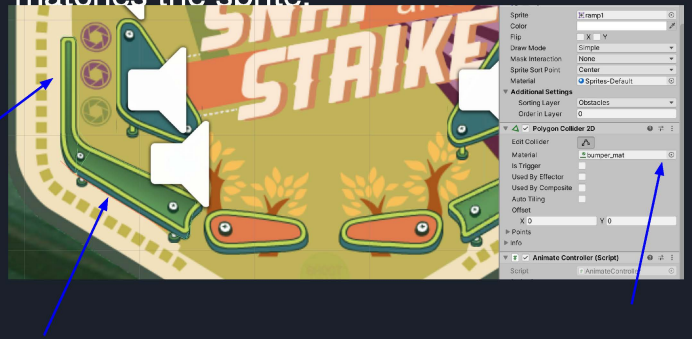
Trigger messages are sent upon collision

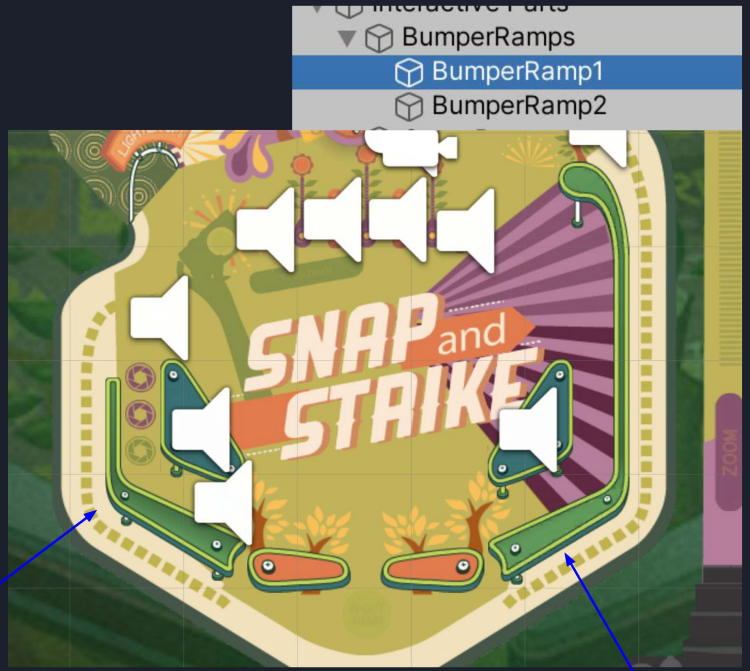
	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider					Y	Y
Rigidbody Collider				Y	Y	Y
Kinematic Rigidbody Collider				Y	Y	Y
Static Trigger Collider		Y	Y		Y	Y
Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y
Kinematic Rigidbody Trigger Collider	Y	Y	Y	Y	Y	Y

Step 2: Adding PolygonCollider2D to Bumper Ramps

BumperRamp1 (left) and BumperRamp2 (right) are two common items on the pinball game where the ball can collide to, and slightly bounced of it.

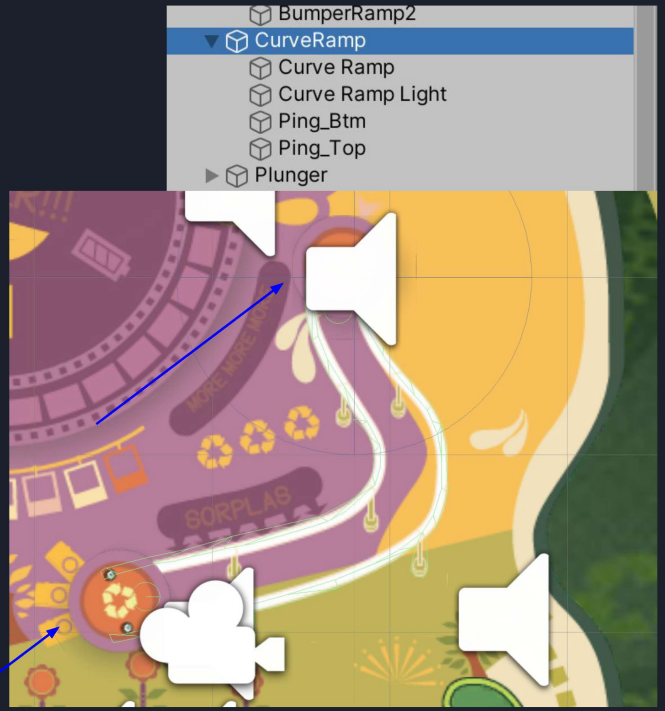
Add PolligonCollider2D on each GameObject. Adjust so that the shape matches the sprite.





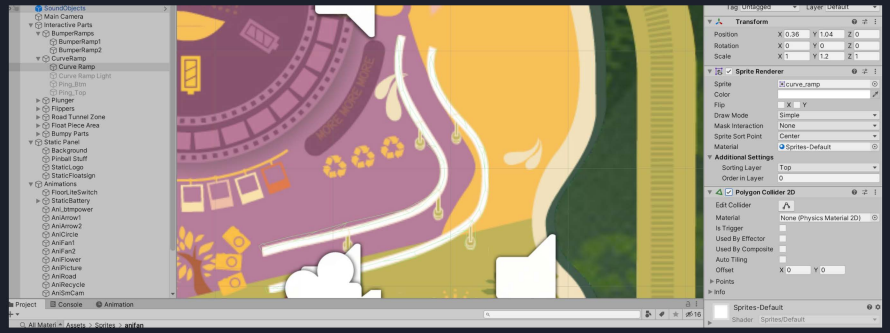
Test: Place the ball above one of the bumpers and press run. It should collide with it. Note that the bumper doesn't have RigidBody because its meant to be a static object, so a Collider will suffice

Step 3: Perfect the CurveRamp



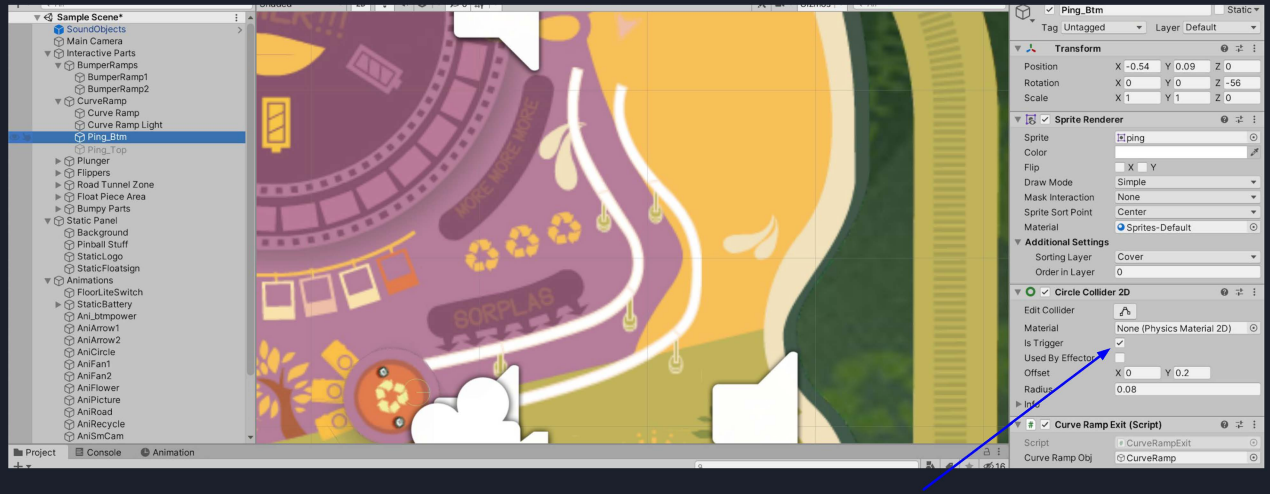
The CurveRamp allows the ball to enter from the top and exit at the bottom. There's four components of the CurveRamp, activate and deactivate each component to identify them.

1. Deactivate the other 3 objects except Curve Ramp. Add PolygonCollider2D and edit the shape to follow the white curves only.



Step 3: Perfect the CurveRamp

2. Add a circle collider with the specified radius and offset as shown for Ping_Btm. Select IsTrigger. This is because we don't want to "block" the ball, but simply triggers something.



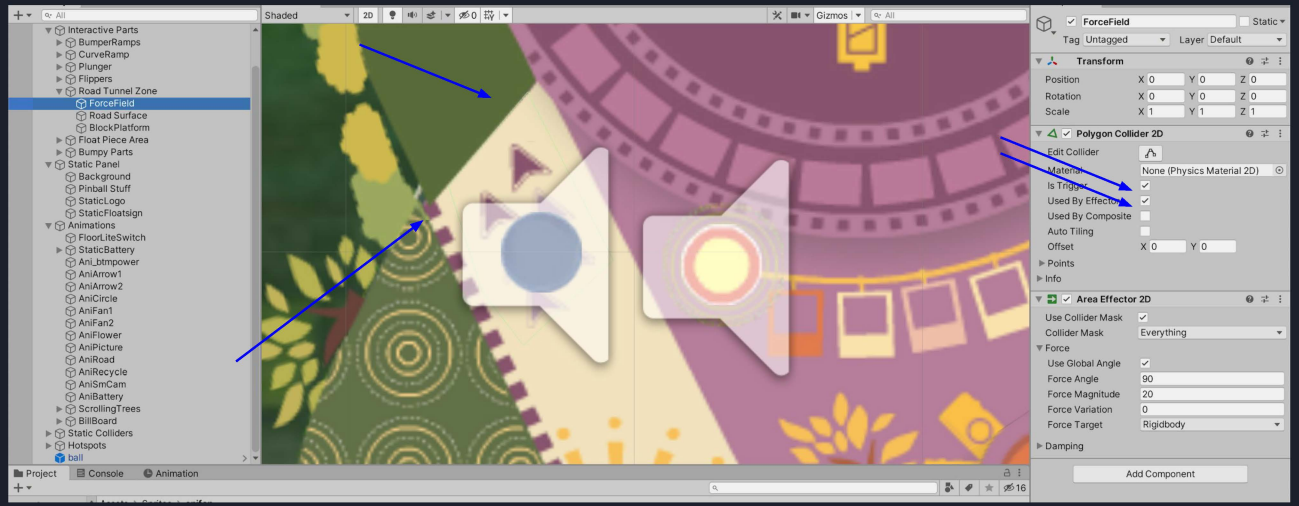
Step 3: Perfect the CurveRamp

3. Add a circle collider with the specified radius and offset as shown for Ping_Top. Also select IsTrigger. Test it by placing the ball somewhere within the CurveRamp, you should see it sliding round the ramp and exit it below swiftly.

used by effector isTrigger ForceField AreaEffector2D Collider Mask

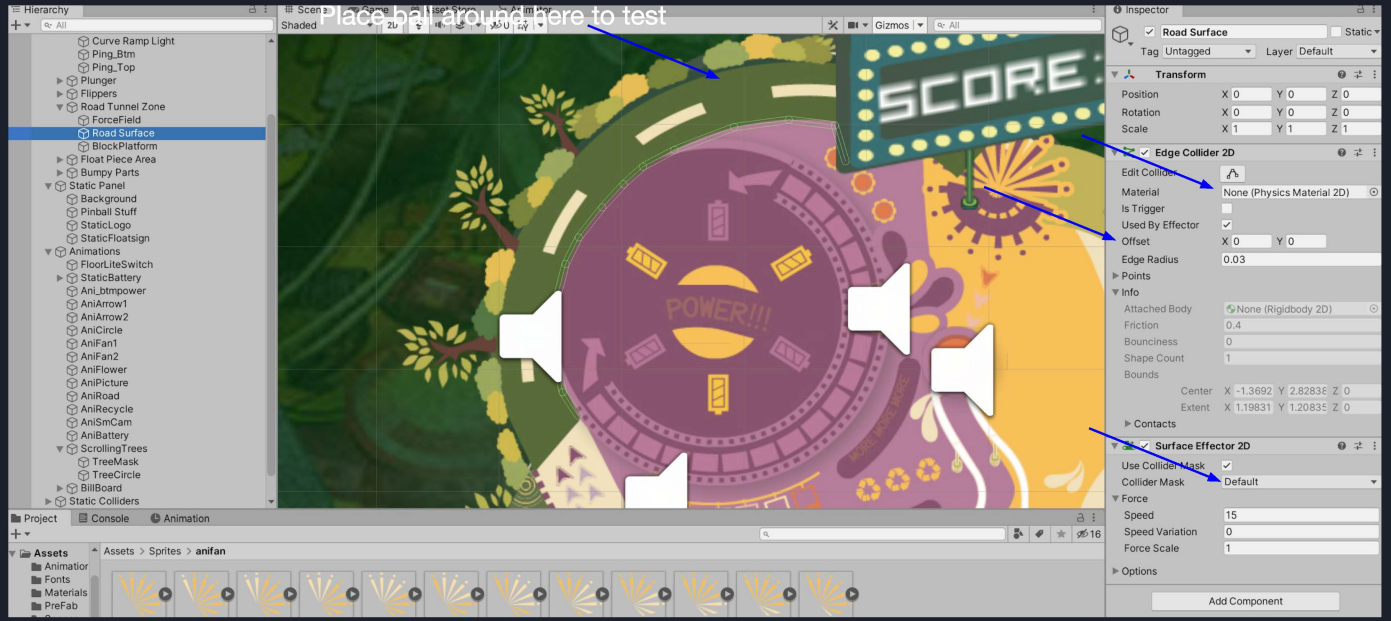
Step 4: Edit Road Tunnel Zone with Effectors

1. Select ForceField and add PolygonCollider2D on it. Edit the shape such that it wraps the little arrows at the ForceField arena. Then, add an AreaEffector2D on it. Ensure the Polygon is “Used By Effector” + “Is Trigger” is on (to allow ball to pass through). This adds *additional* force allowing the collider to be “passed” through by the ball and push it upwards! Test by placing the ball at the ForceField arena and see it being pushed upwards.



2. Select Road Surface and add EdgeCollider2D and SurfaceEffector2D on it. This added like a “speed boost” on the ball that touches the surface. You can edit the EdgeCollider2D if its not very fitting. Press shift while moving the points, or edit the control points directly. Test by placing the ball at the Road Surface arena and see it being accelerated forward (and back entering plunger)

Can only add one effector for each collider (for each gameobject)!
If you want many effect on same gameobject, just create empty gameobjects and add effects



EdgeCollider2D PlatformEffector2D

3. BlockPlatform is a GameObject that's job is to prevent the ball from going out back to the plunger. If we just simply add an EdgeCollider2D, then the ball cannot exit the plunger. We should allow "one way" collision by ALSO adding a PlatformEffector2D (obviously with Use One Way ticked). Enable "Used By Effector" in EdgeCollider2D. Test by placing the ball at the Road Surface arena and see it being accelerated forward but not entering the plunger.

BuoyancyEffector2D PolygonCollider2D density

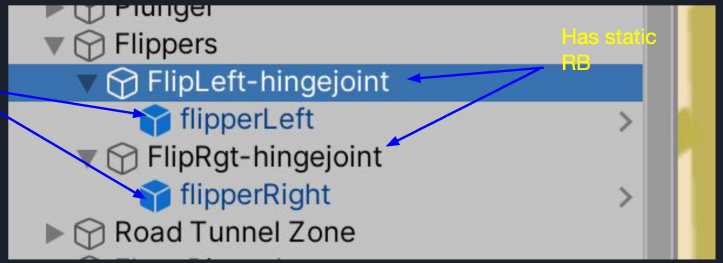
Step 5: Use Buoyancy Effector to the Polygon Collider in Float Area

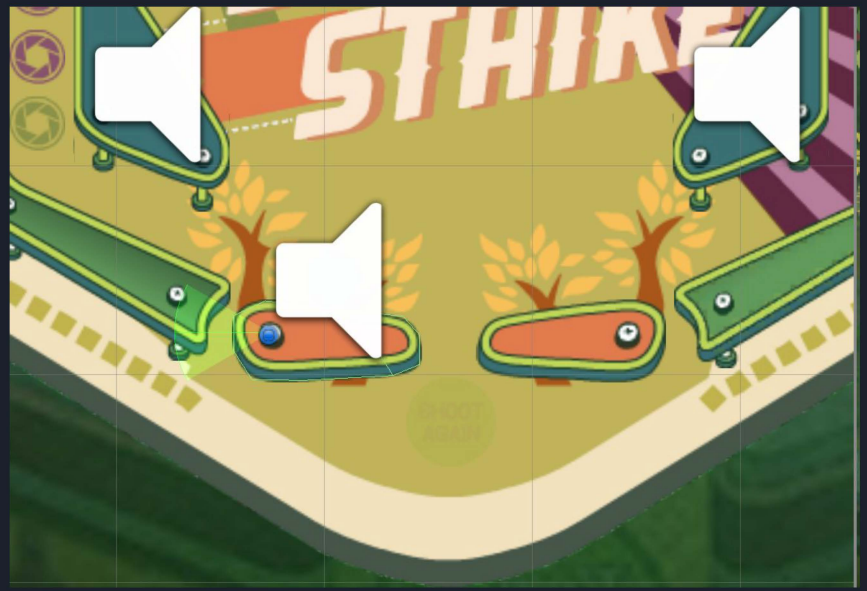
We want an area where a GameObject with RigidBody can “float” around and not just exit or bounce off right away. Add PolygonCollider 2D and edit its shape surrounding the Float Area sprite, then add BuoyancyEffector2D. Test by placing the ball as such and watch it “float around” before eventually exiting the area.

Step 6: Using HingeJoint to Move the Flippers

Click On the Flippers >> FlipLeft-HingeJoint. Add a Hinge Joint 2D component to it. Connect it to the flipperLeft (not FlipLeft-HingeJoint!) Rigidbody.

Notice each component has a RigidBody: the prefabs with STATIC type and the normal GameObjects with KINEMATIC type.





The reason there's two types of RB is because we want to MOVE the flippers following the Hinge limits when LEFT or RIGHT button is pressed. Therefore we want to simulate some Physics there. However, we do NOT want it to FALL off when a ball hits it.

It'll be a tragedy if this happens to your flippers…

dynamic RB static RB

Therefore we need to wrap the "inner" GameObject (flipperLeft) as a child object with Dynamic RB under the FlipLeft-HingeJoint with Static RB and HingeJoint2D attached to it. Note: A HingeJoint typically needs to be connected to TWO rigidbodies. If not, then the "connected anchor" its attached to is a point in space of which it can rotate around it.

Study how HingeJoint2D works

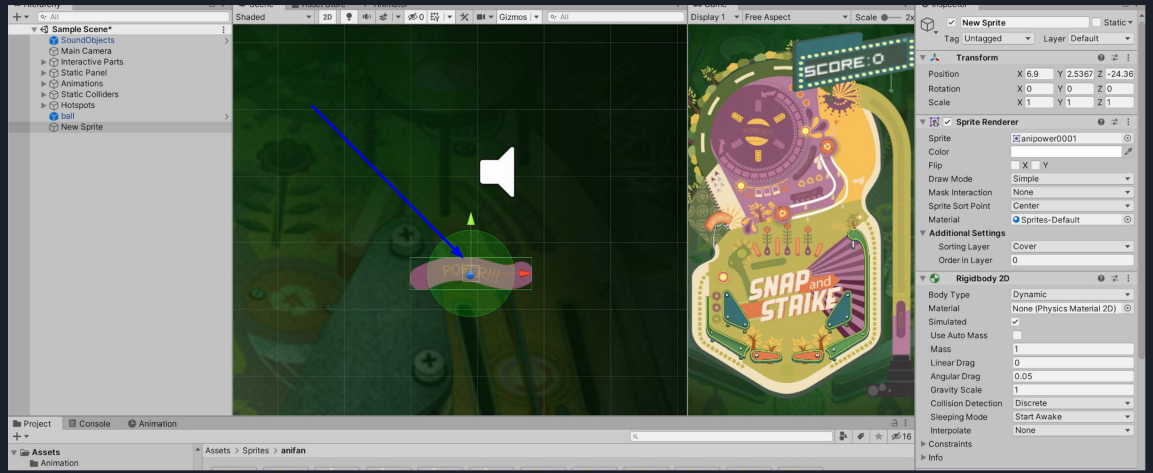
Create a new GameObject >> 2D >> Sprite. Add some sprite to it, a Rigidbody2D (dynamic, set gravity to 0) and a Box Collider2D. Then, add a HingeJoint2D to it. Place the ball above it and play around with the HingeJoint2D parameters, especially Connected Anchor and Anchor.

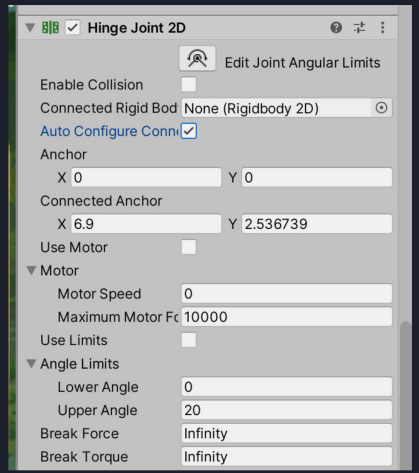
“Connected Anchor” is a point of which the ENTIRE gameobject will Rotate. It is defined in the world space if you don’t have any RigidBody connected.

Connected Anchor

“Anchor” is in local space, which is the real anchor of the GameObject. Typically by default they’re placed together.

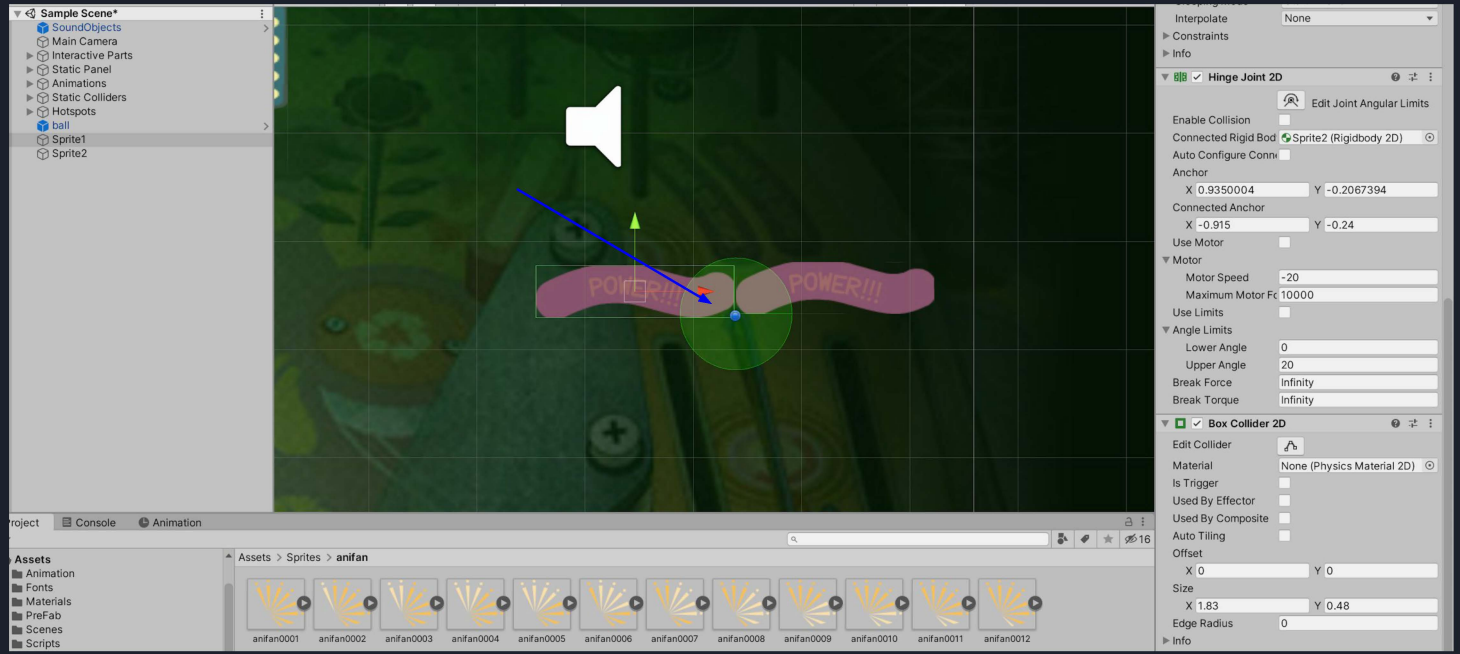
When this is run, the ball falls and the object will rotate around its center.





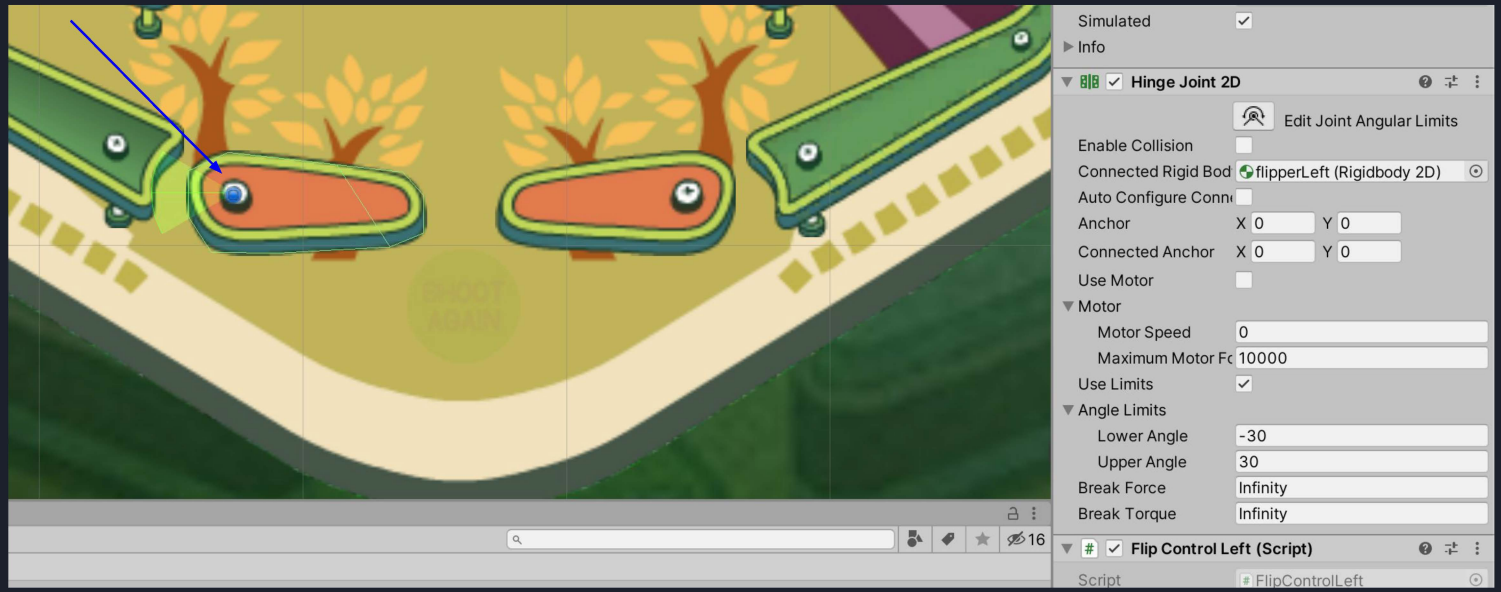
Study how HingeJoint2D works

Edit the location of Anchor and Connected Anchor and study its effect. Add another GameObject (2D Sprite, with Static Rigidbody and BoxCollider2D. Then, edit the HingeJoint to be similar to the picture below. When the ball falls, you can see the first Sprite rotating around the second one as reference. Now, the Connected Anchor, since its connected to Sprite2, has its coordinate system with respect to Sprite2 (no longer the world). You can play with Enable Collision and Motor to see its effect.



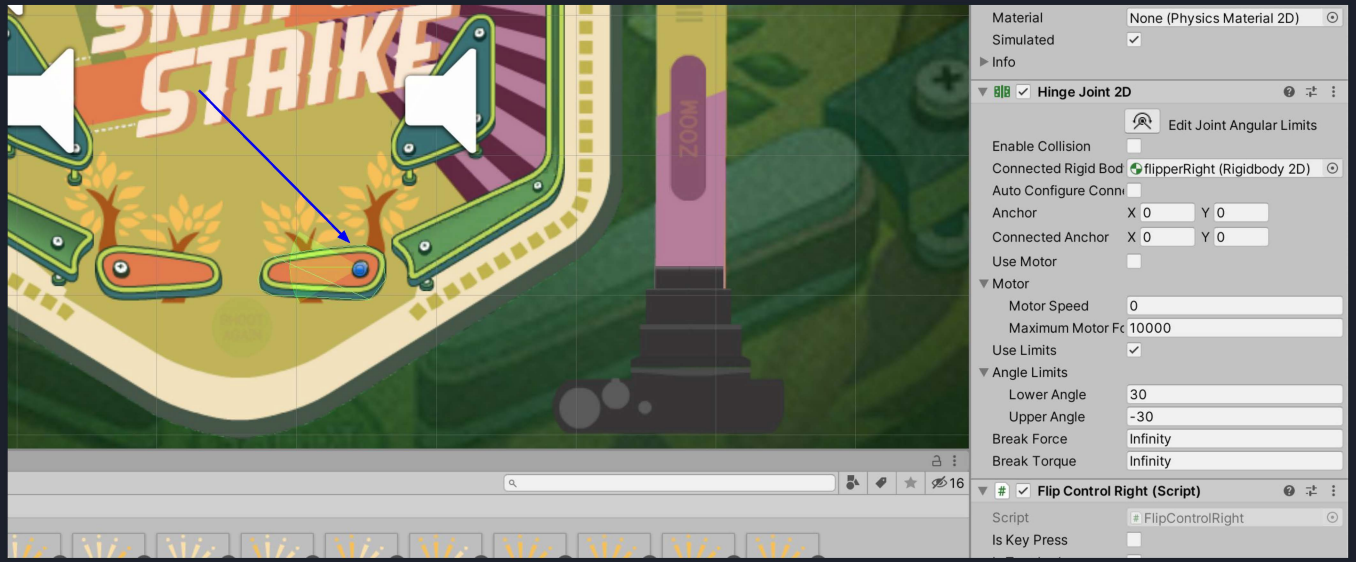
Set FlipLeft-hingejoint HingeJoint2D component

Test run by pressing left arrow button. Notice how Motor is enabled when its run.



Set FlipRgt-hingejoint HingeJoint2D component

Test run by pressing right arrow button. Notice how Motor is enabled when its run (at an opposite speed as opposed to what we saw in the left flipper).



Step 7: Set Bumpy Parts

Now we are left with setting up the Bumpy Parts: super bouncy components of the game. There's 4 bumpy parts, find out who they are in the scene.

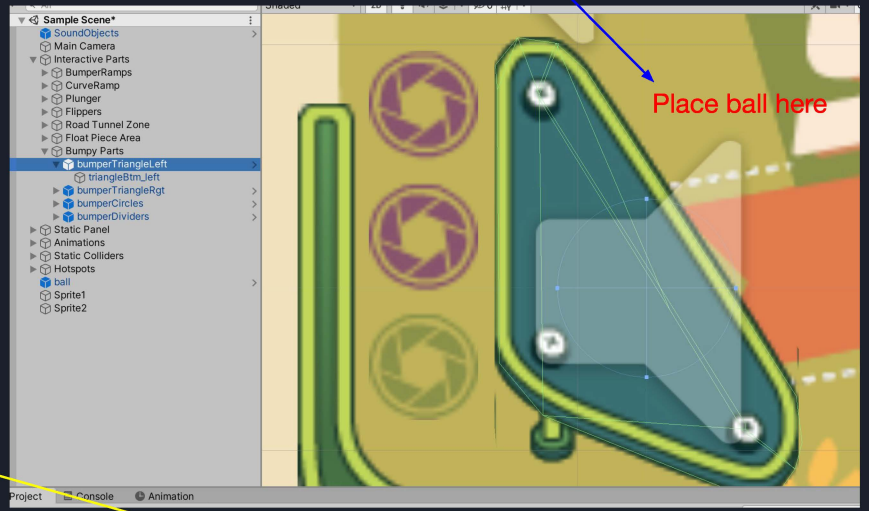
The red arrow points to parts of bumperDividers, the yellow arrow to one of the objects under bumperCircles, the teal arrow to bumperTriangleLeft

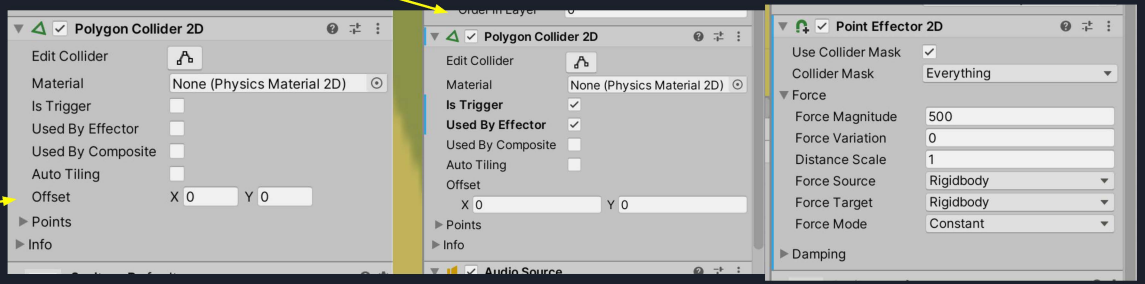
bumperTriangleLeft

PointEffector2D

Force Magnitude

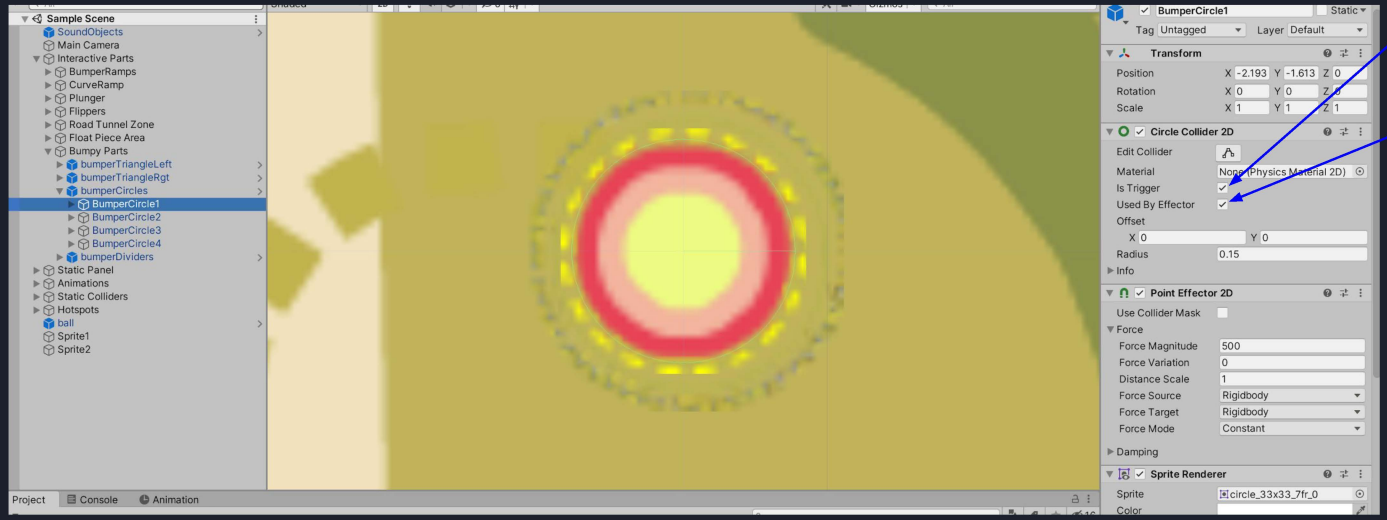
1. We want to have a super bouncy part at the hypotenuse of the triangle, but normal collider at the two other sides.
2. We have PolygonCollider2D in bumperTriangleLeft just at the hypotenuse side. Activate Is Trigger and Used By Effector. This is already done for you.
3. Add PointEffector2D in bumperTriangleLeft as shown (this results in violent force off the hypotenuse side when the ball hits it) -- far right screenshot
4. Add PolygonCollider2D for the other two side inside triangleBtm_left
5. Do (3) and (4) the same way for components in bumperTriangleRgt
6. Test by placing the ball on either side and observe the effect.





bumperCircles

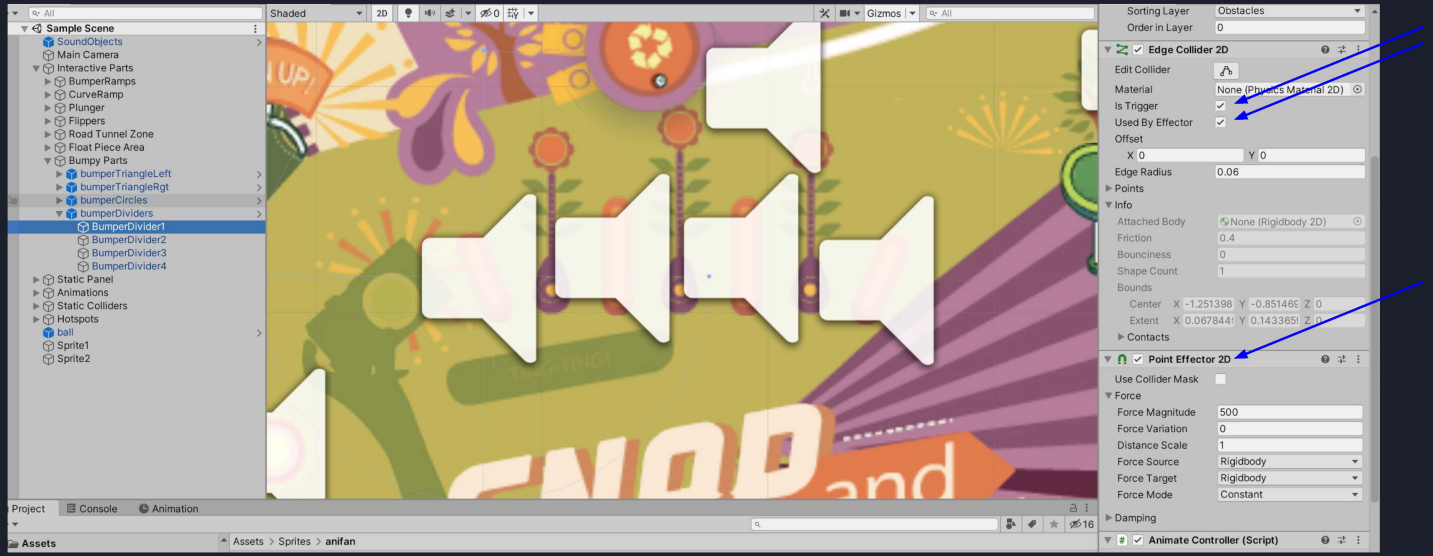
For each BumperCircleN, add circle collider and PointEffector2D as well. Don’t tick “Use Collider Mask”.



Test by placing the ball near this object so that the ball collides with it.

bumperDividers

For each BumperDividerN, we have edge collider and PointEffector2D as well. This is ALREADY done for you.

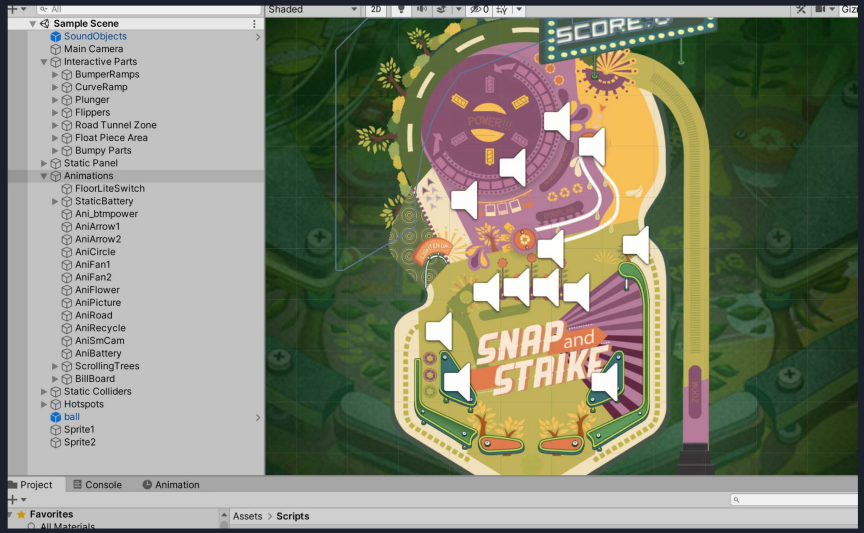


Test by placing the ball near this object so that the ball collides with it. If all is well, then we are done with editing the Physics interaction between the ball and the objects. Let's move on to handling animations.

“Background” Animation vs Controlled Animation

All components under Animations are responsible for visual effects on the game. Some of them however are just background animation, meaning that they just loop over and over for as long as the game is run. The Animation Controller handles this and they’re set to *loop*. For example: AniCircle, AniFan1, AniFan2, etc.

The others are controlled animation, meaning that they are launched by these scripts: AnimateController.cs and ReactionController.cs, and possibly other additional scripts when certain things happened, e.g: hit by the player’s ball.



Sprite Mask

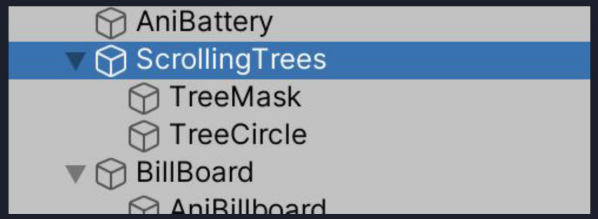
Sprite Mask

SpriteMask

Notice that the trees can be seen even on the board itself. We want to make it showing only on the road tunnel zone side.

We can “cover” sprite using Sprite Mask.

Create a new GameObject called TreeMask:



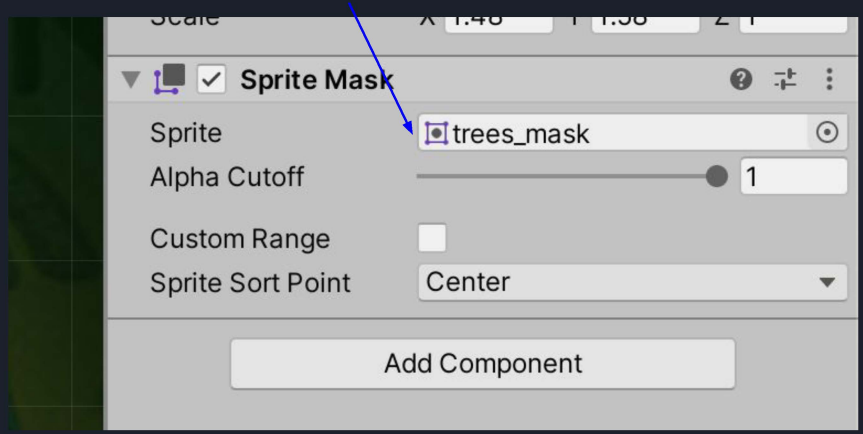
Sprite Mask

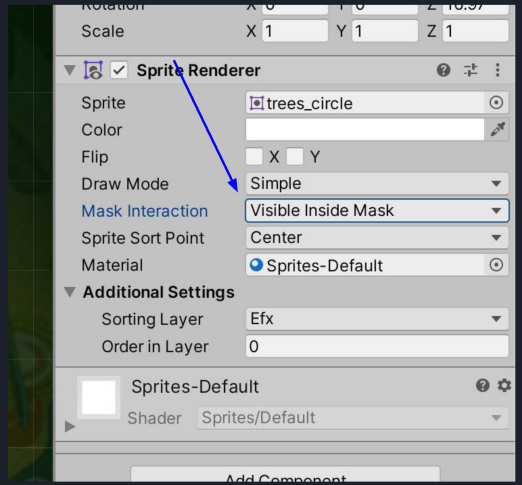
Add a SpriteMask component inside the Tree Mask.

Then, inside TreeCircle's Sprite Renderer, change the Mask Interaction to be "Visible Inside Mask".

This allows us to "see" the tree only when it is inside the mask region.

By default (without any further setting, etc), a SpriteMask will affect any sprite in the scene that has their mask interaction set to Visible / notVisible under Mask.

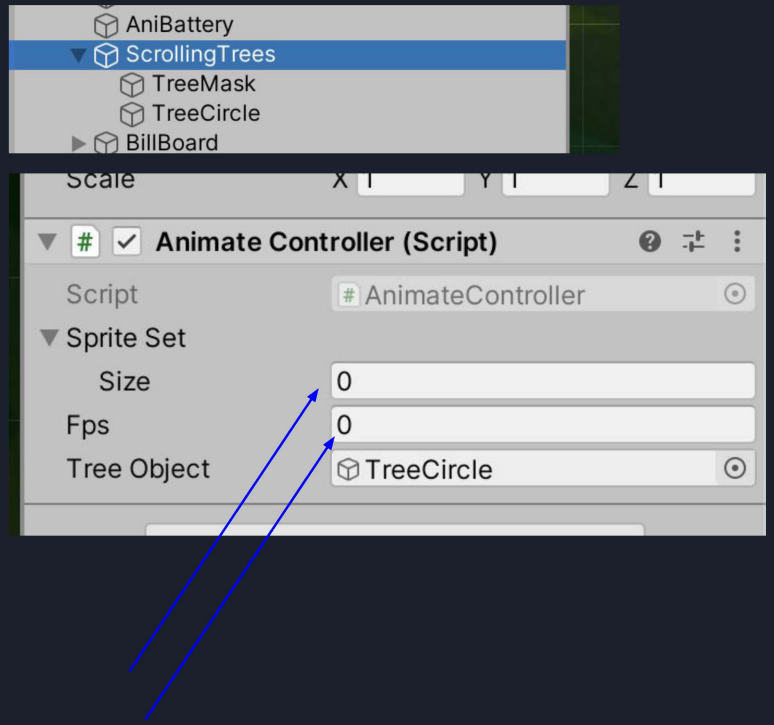




Animating the Trees

We want the Trees to now *rotate* around the Road Tunnel Zone. Attach the AnimateController.cs script to ScrollingTrees object, and hook up the TreeCircle child object to it.

Notice that this script is used to “store” a list of Sprites, which will be used by ReactionController (if any is attached) to play these sprites at an interval. However for the tree, we simply want to just rotate it, so we don’t care about the Sprite, setting the Size to 0 and Fps to 0 as shown.



Refactor the code

RotateTree

This code inside AnimateController.cs seems irrelevant for other objects having the script, except for the ScrollingTree object. Let's refactor it. Remove this chunk from AnimateController.cs:

```
    public GameObject treeObject;
    private float turnspeed = 30f;

    void FixedUpdate()
    {
        // if current instance has a treeObject
        if (treeObject != null)
        {
            treeObject.transform.Rotate(new Vector3(0, 0, turnspeed * Time.deltaTime));
        }
    }
```

Then create a new script called RotateTree.cs which we will attach to ScrollingTree GameObject whose job is to solely….rotate them trees.

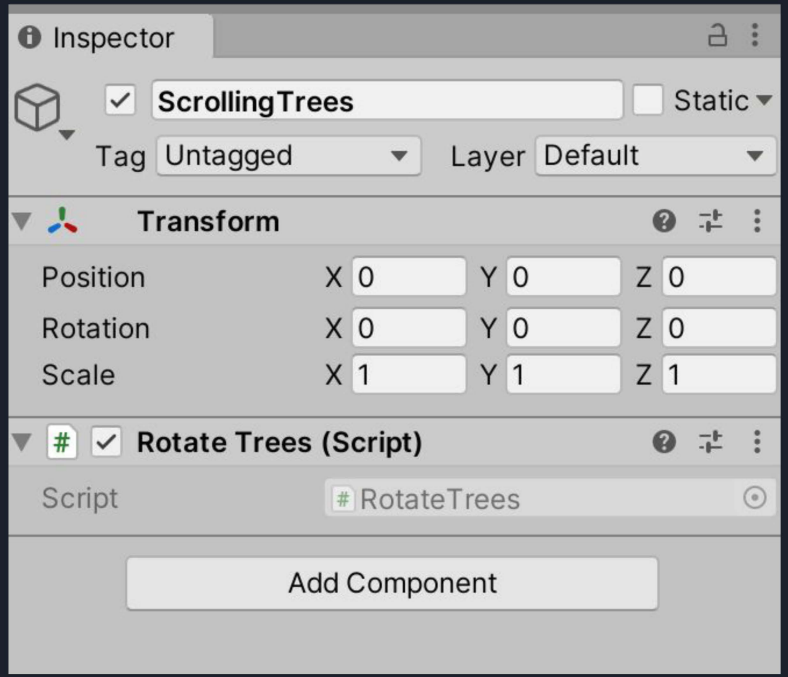
Write this code inside RotateTree.cs. Notice the workaround to get the *child* of Scrolling trees using its **transform**. An Object's transform is basically in some kind of *tree-like* structure, because affine transformations are applied from the parents and propagated to its children (its a series of matrix multiplications in essence). The index refers to the order of GameObject that you define in the project hierarchy. You can learn more about parent-child transformations that's very handy for hierarchical objects -- moving joints, etc in 50.017 :)

```
public class RotateTrees : MonoBehaviour
{

    private float turnspeed = 30f;
    private GameObject treeObject;

    void Start(){
        treeObject = gameObject.transform.GetChild(1).gameObject; // get the child from transform (transform is a
hierarchy)
    }

    void FixedUpdate()
    {
        // if current instance has a treeObject
        if (treeObject != null && treeObject.name == "TreeCircle")
        {
            treeObject.transform.Rotate(new Vector3(0, 0, turnspeed * Time.deltaTime));
        }
    }
}
```

Refactor the code

Finally, attach RotateTrees.cs inside ScrollingTree GameObject.

Test: The trees should now turn anticlockwise around the Road Tunnel Zone.

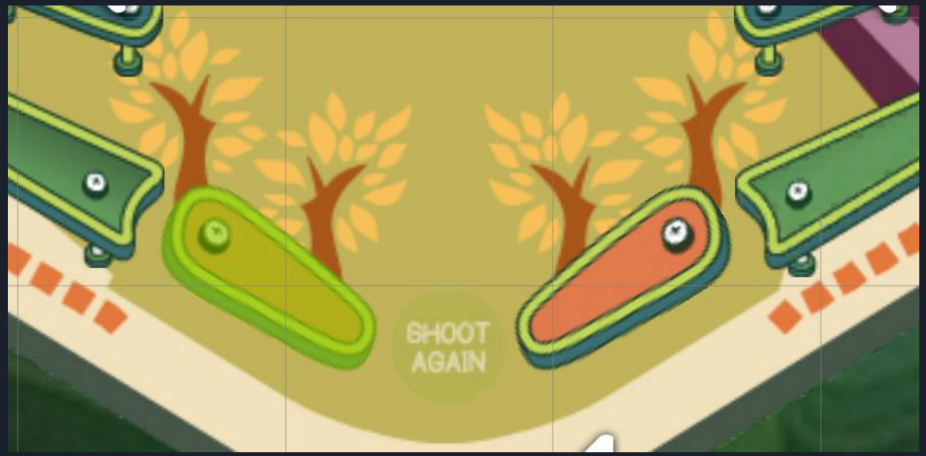


Animating a series sprites

If you notice, flipperLeft has this blinking animation when the ball touches it and flipperRgt doesn't have it.

While we can create animationController to each and every of these things and "trigger" it like how we learned it with the CakeTheClown game, we can also control animation manually from script.

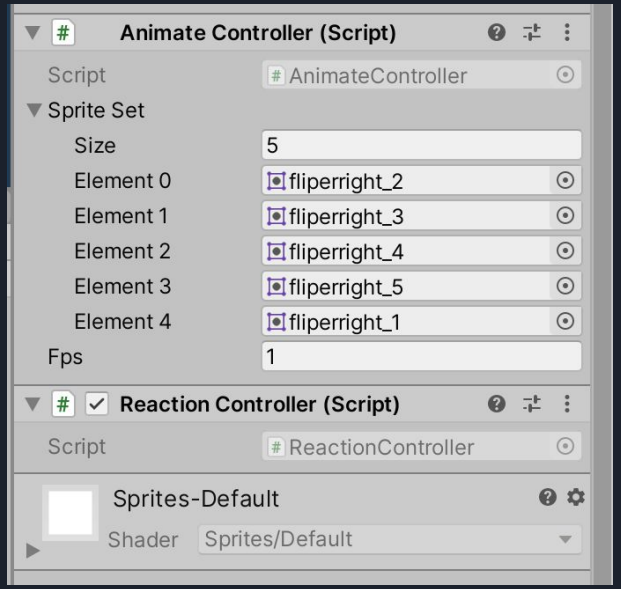
Attach the scripts AnimateController.cs and ReactionController.cs to flipperRight PREFAB (in prefabs folder).



Set the elements of Animate Controller

Then, set the clips of flipperRight that you want to have. These are a list of sprites thats simply added to the public Sprite list. The value of fps indicate how “fast” we want the sprite changes be.

```
public Sprite[] spriteSet;
public float fps;
```



Using Coroutines for Simple Animations

`Coroutines` `ReactionController`

Then open the ReactionController.cs

It is like a “standard” script thats used to trigger and show the list of sprites at “fps” when this object is colliding with the ball. At first, it attempts to get these components if they exist.

```
    private AnimateController animateController;
    private SpriteRenderer thisRenderer;
    private AudioSource sound;

    // Use this for initialization
    void Start()
    {
        animateController = GetComponent<AnimateController>();
        thisRenderer = GetComponent<Renderer>() as SpriteRenderer;
        sound = GetComponent<AudioSource>();
    }
```

OnTriggerEnter2D recap

```
void OnTriggerEnter2D(Collider2D obj)
{
    if (sound != null && obj.name == "ball")
    {
        sound.Play();
    }
    StartCoroutine(playAnimation());
}
```

onTriggerEnter2D

The method above is called when it collides with other objects with IsTrigger enabled or if this object has a collider with IsTrigger ticked (either way it will cast this method, but this method wont be called if BOTH has IsTrigger enabled) -- and that the other object is the “ball”.

Remember that in order to invoke this method, you need to have:

1. Colliders in both game objects
2. At least one of them with a **rigidbody** AND
3. With at least 1 of them with isTrigger set to **true**.

When all conditions are fulfilled, OnTriggerEnter is called on all game objects involved if you have the OnTriggerEnter() function written in the scripts attached to them.

It doesn't matter with one has the rigidbody.

OnCollisionEnter recap

```
void OnCollisionEnter2D (Collision2D evt)
{
    if (sound != null && evt.gameObject.name == "ball")
    {
        sound.Play();
    }
    StartCoroutine(playAnimation());
}
```

The method above is called when it collides with other objects and one of them have RigidBodies attached (see collision Matrix to observe which kind of RigidBody), and that the other object is the “ball”.

Just like OnTriggerEnter, OnCollisionEnter is called on any object involved in a Collision event

This method will NOT be called if this object has IsTrigger enabled (but instead OnCollisionTrigger will be called).

onCollisionTrigger

A collider configured as a Trigger (using the Is Trigger property) does not behave as a solid object and will simply allow other colliders to pass through. And then when a collider enters its space, a trigger will call the OnTriggerEnter function on the trigger object’s scripts.

The Couroutine

```
IEnumerator playAnimation()
{
    for (int i = 0; i < animateController.spriteSet.Length; i++)        {
        thisRenderer.sprite = animateController.spriteSet[i];
        yield return new WaitForSeconds(animateController.fps / 10);
    }
}
```

As the code shows, it will set this' sprite as the N-th sprite in the animateController component attached to "this", paused for a few moments, and continue the for-loop until the instruction ends.

Now test by placing the ball above the right flipper, and run the game. It should shows the right flipper blinking properly when colliding with the ball. Note that in this particular case, the animation coroutine is called from OnCollisionEnter2D because both ball and flipper do not have "IsTrigger" enabled on its colliders.

Summary

In this part, we have learned:

1. How Collider and Rigidbody works
2. Using Different types of Colliders
3. Using Effectors: Area effector, Point effector, etc
4. OnTriggerEnter and OnCollisionEnter calls
5. Collision Matrices
6. Briefly about how to search for child objects using *transform*
7. Sprite Mask
8. Animation from script (manually) and running it using Couroutines

CHECKOFF

Click the item FloorLiteSwitch, and notice that it “blinks” when the ball exits the arena.

Find out:

1. **How this FloorLiteSwitch animation works**, i.e: where is it called and how?
2. How **ball respawn** is done
3. Whether the “old” ball is destroyed or reused for the next round?

Write your concise answer and export as pdf. Your answer should NOT exceed one page. Upload it on edimension.

Introduction to Unity Part 8 - Scriptable Objects

Week 6

Create a new 3D Unity project and import the assets from here.

Objectives

1. Create and use scriptable objects as event and data container
2. Create generic event listener
3. Glow FX

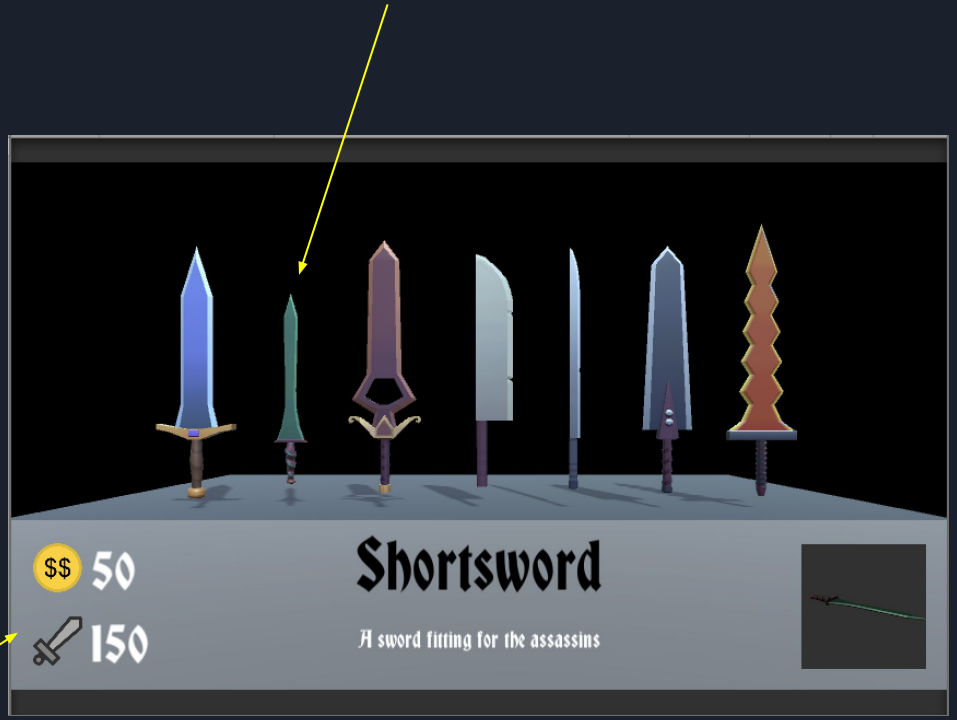
This tutorial was originally obtained from one of Ray Wenderlich's tutorial here.

Context

Imagine being a swords merchant, where you have many different swords and many different variants of the same sword family.

We need to store its name, description, price, damage, etc. This description is most likely *persistent*, meaning that it doesn't vary with time unless we change the object altogether.

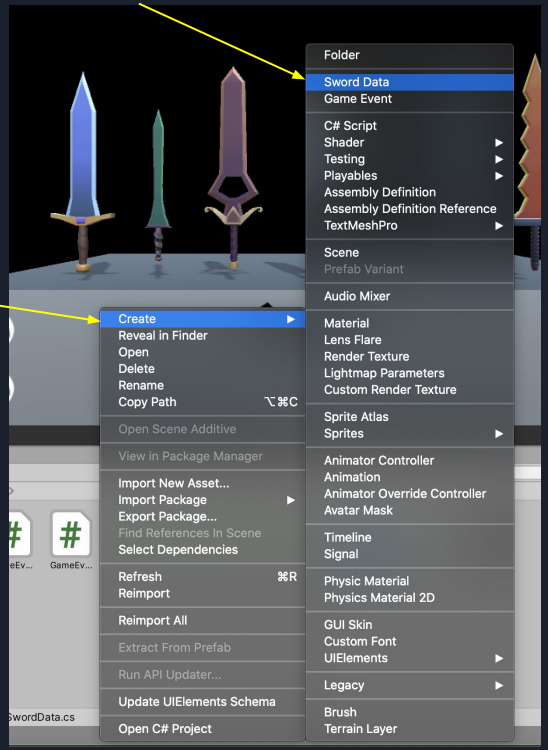
Scriptable object can be used to make our lives easier when transporting data around, hence decoupling the GameObject of Sword with its data.



Step 1: Create scriptable object script to store sword data

- Go to Assets >> Scripts and create new C# script.
- Instead of inheriting MonoBehaviour, we inherit ScriptableObject
- The first line “CreateAssetMenu” allows us to create a special type of object from this script later on

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

[CreateAssetMenu(fileName="New SwordData", menuName="Sword Data",
order=51)] // order=51 puts in the second grouping
public class SwordData : ScriptableObject
{
}
```



Step 2: Add data variables to the script using properties

Under the class declaration, add the following.

By now you know SerializeField allows us to expose private data to the inspector.

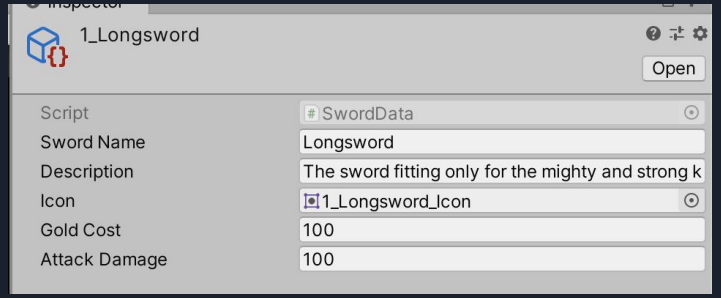
```
    [SerializeField]
    private string swordName;
    [SerializeField]
    private string description;
    [SerializeField]
    private Sprite icon;
    [SerializeField]
    private int goldCost;
    [SerializeField]
    private int attackDamage;
```

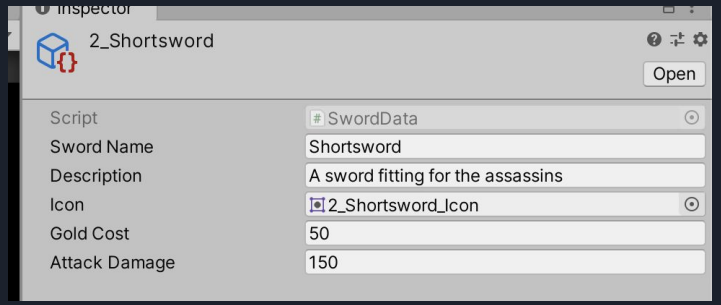
Then, create getter and setter for each of the private variables as follows:

```
    // creating getter and setter for the private variable
    public string SwordName
    {
        get
        {
            return swordName;
        }
    }
```

Step 3: Create SwordData

- Create a new folder called SwordScriptableObjects under Asset
- Create 7 SwordData (right click, Create >> SwordData
- Name each of them as follows, link up the sprite icon, and fill up the fields at the inspector with whatever you want.
    - 1_Longsword
    - 2_Shortsword
    - 3_Claymore
    - 4_Falchion
    - 5_Katana
    - 6_Broadsword
    - 7_Edgesword





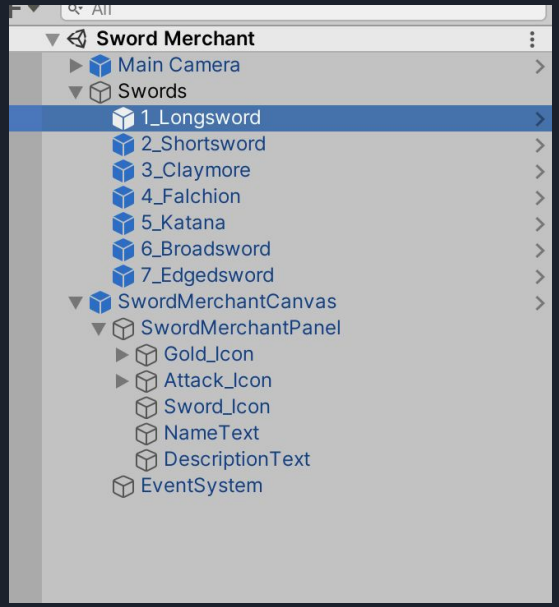
Step 4: Use the ScriptableObject

On each sword GameObject, notice that the script Sword.cs is already attached to it. Modify Sword.cs so that it has this public variable declared:

```
    [SerializeField]
    public SwordData swordData; // The data container for this sword's data
```

Then, lets check if it works by adding this method:

```
    private void OnMouseDown()
    {
        Debug.Log(swordData.SwordName);
        Debug.Log(swordData.Description);
        Debug.Log(swordData.Icon.name);
        Debug.Log(swordData.GoldCost);
        Debug.Log(swordData.AttackDamage);
}
```

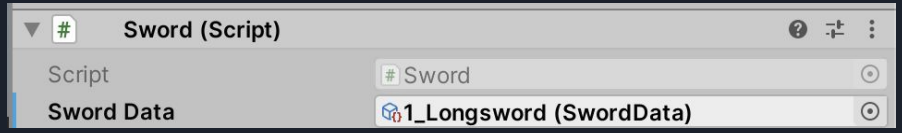


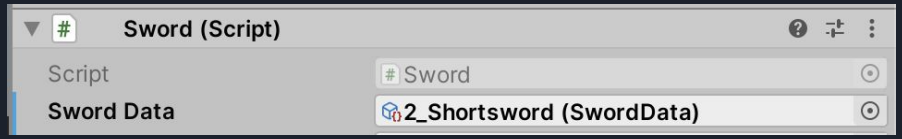
Link the respective ScriptableObject you created earlier on in step 3 to each of the Sword.cs attached on each sword GameObject.

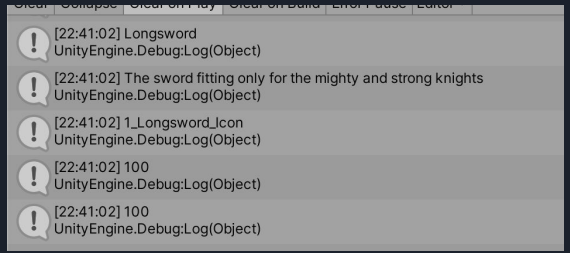
For example:

1. Top: Screenshot on 1_Longsword GameObject inspector
2. Bottom: Screenshot on 2_Shortsword GameObject inspector

Test run and you should see a bunch of things printed out on the console when you click the sword







Step 5: Managing Events

Now that we have a bunch of swords and its data, we want to show it at the Canvas whenever we click a sword. To do this more effectively, we can use ScriptableObjects to manage event for each sword.

The basic idea is to have:

1. `GameEventListener` (script), that will invoke a pre-determined response when *something* happens
2. `GameEvent` (scriptable object), that maintains a list of GameEventListeners. Will invoke all the listeners in this list when told.

Under Asset >> Scripts, create two scripts: GameEvent.cs and GameEventListener.cs

GameEventListener.cs

The first script is the listener (inheriting standard MonoBehaviour). There's two fields:

1. GameEvent (the scriptable object we will create later) that it listens to
2. UnityResponse (this is a method that we need to set that will be called when the game event is *raised -- using OnEventRaised method below*)

OnEnable & OnDisable API calls that will be invoked when the GameObject where this script is attached is enabled or not.

OnEventRaised() method that will invoke the response. Will be called when that *something* happens

```
using UnityEngine;
using UnityEngine.Events;

public class GameEventListener : MonoBehaviour
{
    [SerializeField]
    private GameEvent gameEvent;
    [SerializeField]
    private UnityEvent response;

    private void OnEnable()
    {
        Debug.Log("On enable called");
        gameEvent.RegisterListener(this);
    }

    private void OnDisable()
    {
        Debug.Log("On disable called");
        gameEvent.UnregisterListener(this);
    }

    public void OnEventRaised()
    {
        Debug.Log("Invoke response!");
        response.Invoke();
    }
}
```

GameEvent.cs

We add the CreateAssetMenu command so we can create them later on similar to what we did in Step 3.

The Raise() method goes through all GameEventListener registered in the list *listeners*, and raise them all.

The last two methods are of course, self-explanatory.

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

[CreateAssetMenu(fileName = "New Game Event",
menuName = "Game Event", order = 52)] // 1
public class GameEvent : ScriptableObject
{
   private List<GameEventListener> listeners = new
List<GameEventListener>();

   public void Raise()
   {
       Debug.Log("Listener count : " +
listeners.Count.ToString());
       for (int i = 0; i < listeners.Count; i++){
           listeners[i].OnEventRaised();
       }
   }

   public void RegisterListener(GameEventListener
listener)
   {
       listeners.Add(listener);
   }

   public void UnregisterListener(GameEventListener
listener)
   {
       listeners.Remove(listener);
   }
}
```

Write the Event handler in SwordMerchant.cs

When we click the sword, we want the data of the sword to be displayed. Therefore now we need to write a method that accepts a SwordData as argument in `SwordMerchant.cs`

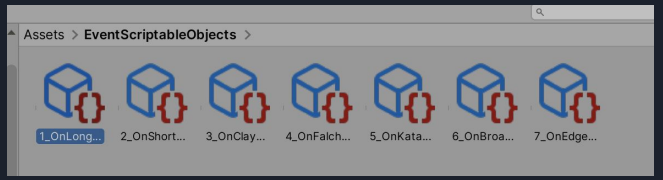
```
    public void UpdateDisplayUI(SwordData swordData)
    {
        Debug.Log("Response invoked");
        swordName.text = swordData.SwordName;
        description.text = swordData.Description;
        icon.sprite = swordData.Icon;
        goldCost.text = swordData.GoldCost.ToString();
        attackDamage.text =
swordData.AttackDamage.ToString();
        state.swordState = swordData.SwordName;
    }
```

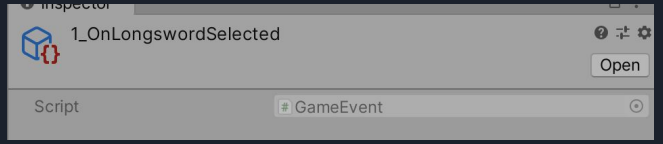
Step 6: Create GameEvents

Create a new folder called EventScriptableObjects under Assets. Create 7 instances of GameEvent (right click Create >> GameEvent), and name them each:

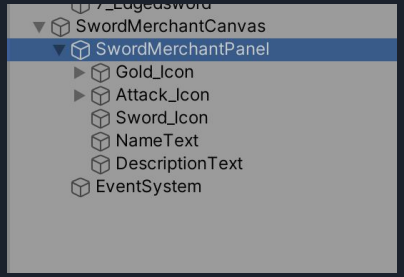
1. 1_OnLongswordSelected
2. 2_OnShortswordSelected
3. 3_OnClaymoreSelected
4. 4_OnFalchionSelected
5. 5_OnKatanaSelected
6. 6_OnBroadswordSelected
7. 7_OnEdgeswordSelected

The inspector looks just like the screenshot, you don't need to add anything in.



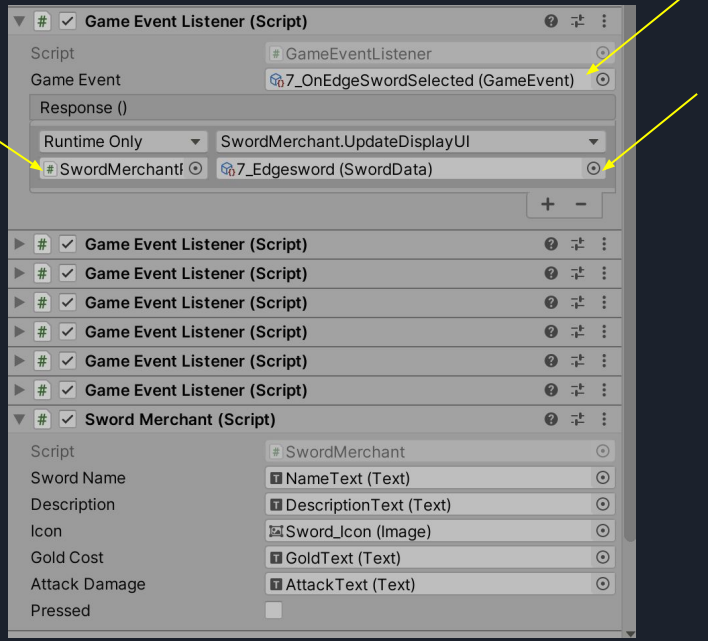


Step 7: Add GameEventListener script to SwordMerchantPanel



Drag GameEventListener script to SwordMerchantPanel. Do this 7 times, because each listener is listening to each of the 7 GameEvent. The idea is that when a sword is clicked, we need to tell the SwordMerchantPanel to display the data of the sword being clicked.

Its inspector should look like something on the right. Hook up each GameEvent field, SwordMerchant Script in the scene, and the corresponding SwordData.



Step 8: Raise the event in Sword.cs

Now what is left is to call the Raise() method of the right GameEvent when each sword is clicked. We can do this inside Sword.cs.

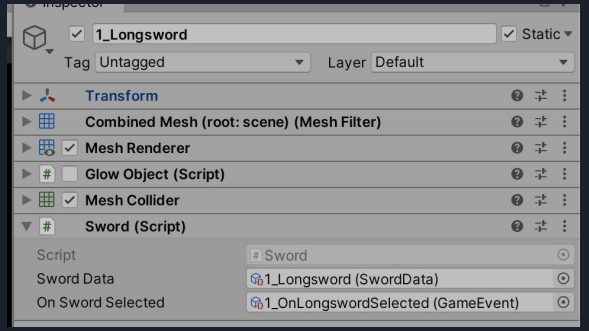
Add the following field in Sword.cs:

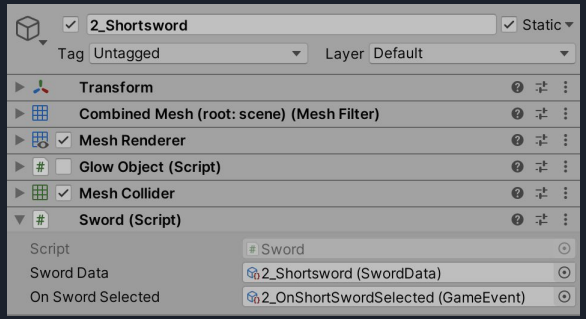
```
    [SerializeField]
    private GameEvent OnSwordSelected;
```

And the following instruction under OnMouseDown() method:

```
        OnSwordSelected.Raise(); //raise its own event when onMouse
down
```

At the inspector of each Sword GameObject, link up the right GameEvent. The screenshot shows the sample for the first two swords. Do the same for the rest of the swords.





Step 8: Raise the event in Sword.cs

When you click on a sword therefore, it will raise the corresponding GameEvent and then pass the sword data to it, therefore you can see it at the canvas.

However for now you don’t have the “glow” when you click the sword. We will do it in the next step.

Step 9: Highlighting the Sword

Just for a simple visual effect, create a new script GlowObject.cs. It has four private variables and one public field, glowColor that we can set in the inspector later on.

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

public class GlowObject : MonoBehaviour
{
    public Color glowColor;

    private float lerpMultiplier = 12;
    private List<Material> materials = new List<Material>();
    private Color currentColor, targetColor;
    private Renderer[] renders;
}
```

Step 9: Highlighting the Sword

Then we add the Start method that stores all materials of this GameObject’s children (and itself too, thats how GetComponentsInChildren works).

```
    private void Start()
    {
        renders = GetComponentsInChildren<Renderer>();
        for (int i = 0; i < renders.Length; i++)
        {
            materials.AddRange(renders[i].materials);
        }
    }
```

Step 9: Highlighting the Sword

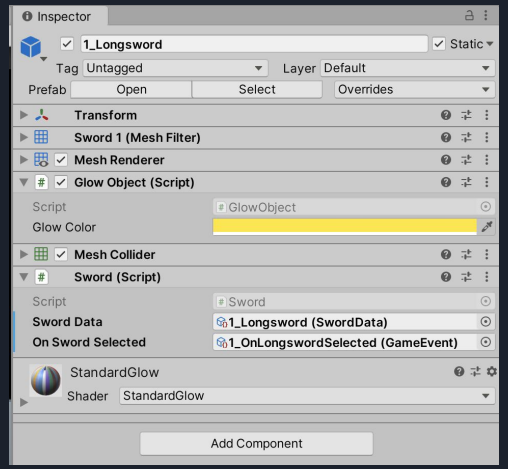
The Update() method should constantly check whether or not the currentColor == targetColor. To save abit of time, we disable the script from running certain methods as written below once the color target is reached. To allow a smooth transition, we use linear interpolation with Color.Lerp. This color setting: SetColor tells the shader of the amount of color to use for the sword.

```
private void Update()
{
    if (currentColor == targetColor)
    {
        // Enabled is a default property that allows us to enable or disable the script
        // The enabled property only affects the calling of Start, Update, FixedUpdate and OnGUI.
        enabled = false;
    }
    else
    {
        currentColor = Color.Lerp(currentColor, targetColor, Time.deltaTime * lerpMultiplier);
        for (int i = 0; i < materials.Count; i++)
        {
            materials[i].SetColor("_GlowColor", currentColor);
        }
    }
}
```

Step 9: Highlighting the Sword

Now, we simply add two more methods to make the sword glow when we hover our mouse. These are called automatically when we hover over the object.

```
    private void OnMouseEnter()
    {
        targetColor = glowColor;
        enabled = true;
    }

    private void OnMouseExit()
    {
        targetColor = Color.black;
        enabled = true;
    }
```



Finally, attach this script to each of the sword GameObject. Set the Glow Color to be whatever you like. Test run and you should see the object glows when we hover the mouse over it, and show the stats when we click on it.

Step 9: Highlighting the Sword

Notice that this simple step is possible due to some preset that's been done for you: Shaders (in Shaders folder) and two scripts: GlowComposite.cs and GlowPrePass.cs, and the GlowPrePass Camera. If you are interested and relevant to your subject, you can study how these work. It is not required to know about how shaders work in this course.

Summary

In this part, we have learned:

1. What is a scriptable object
2. How to create and set them up
3. Using scriptable objects as data container that’s persistent
4. Using scriptable objects to create custom events
5. Adding some glow effect using linear interpolation

CHECKOFF

For the remaining of the class time, you're required to modify the code such that its able to do the following at runtime:

1. Detect if there's a key press, e.g: 'a' is pressed
2. If a particular key is pressed, it will change the attackDamage stats of one of the swords, e.g: if 'a' is pressed, the attackDamage of Longsword is added by 10
3. The modification in step (2) must be immediately visible at the canvas if that sword modified is currently highlighted / clicked, otherwise it remains at other sword's stats and it will be available later when that sword is clicked.
4. Just do (1-3) to ONE sword to demonstrate your understanding.
5. You need to modify SwordData such that the attackDamage property can be modified/set (and not just read/get)
6. You can create new scripts / scriptable objects.

Show your script that does this, show where do you use it in the scene, and record the output into a video. Upload the link to your video on edimension. For due date, refer to edimension as well.

Introduction to Unity Part 9 - Finite State Machine

Week 6

Download the asset and ProjectSetting from here.

Similar to Part7, overwrite your new project's ProjectSetting with this one, and then import the asset.

Materials adapted from Unity tutorial: https://learn.unity.com/project/tanks-tutorial

Background

The goal of this lab is to fix the game by implementing a state machine such that the two tanks: green (scanner) and red (chaser) are able to identify you in the map and perform actions accordingly.

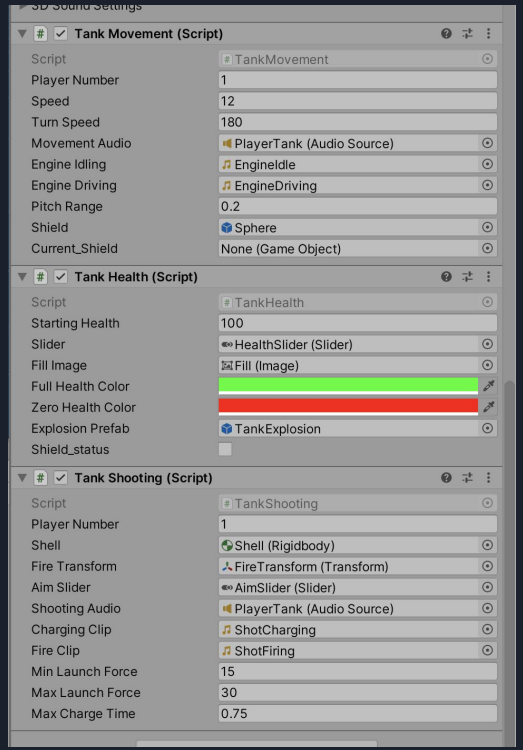
We will use ScriptableObjects that we have learned in the previous part to implement the state machine.

Controlling the Player Tank

Go to Prefab >>> PlayerTank and open the prefab

Note that there's **3 scripts** that's controlling the Tank:

1. The first script is **TankMovement.cs**, as you know it, it get the keyboard input and perform update of tank location under the `FixedUpdate()` method
2. The second script is called **TankHealth.cs**. It contains all the methods responsible to manage the amount of health this *instance* of player tank has. *Task: Find out which script calls these health update methods such as TakeDamage and SetHealthUI.*
3. The third script is **TankShooting.cs**. This is the script that controls how fast a bullet is shot out from the tank, i.e: the longer u press space, the more force the bullet has. Take a look at line 73 - 75 — it creates a new **instance of a bullet,** and gives it an initial velocity in the "forward" direction (local to the tank). Physics engine took care of the rest. **Remember you need to clean the unused bullet in the end.** *Task: Find out how to create the Shell prefab when fire button is pressed, and which script cleans up fired bullets.*



AITankChaser

This is one of the enemy tanks, *the “less” advanced tank* — its the **red** tank in the scene.

Our idea for the Tank is as follows:

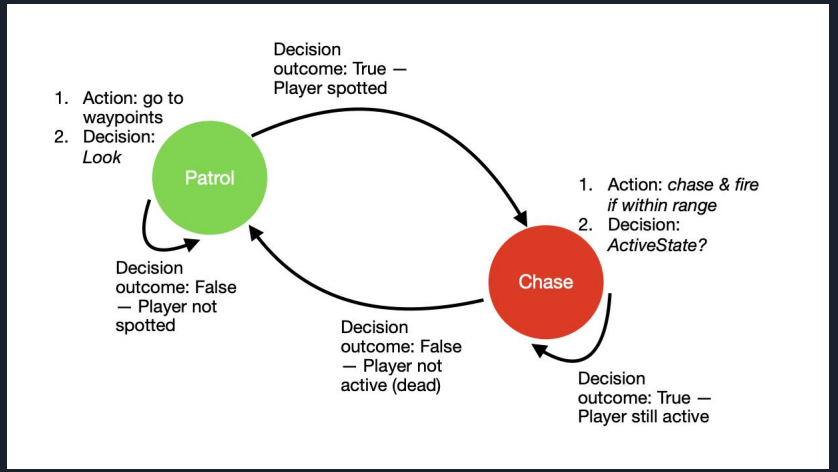
1. The enemy tank begin at a StartState
2. Upon entering any **state**, you can perform **action**(s).
3. Then afterwards, make a **decision**, to compute what is the next state is to go to.

Decision, Action, and State are all made using Scriptable object, a more lightweight form of Unity Object that doesn’t require any rendering or advance component as our usual objects.

AITankChaser State Transition Diagram

What this AI do in a nutshell is:

1. Begin at **state** *Patrol* and perform **action** of circulating around waypoint 1, 2, then 3 in that order. Afterwards, always perform **decision** called *Look*: cast a ray and see if it hits anybody with a tag of *Player*.
    1. If yes, set the chase target as the Player object and store its location, and go to *Chase* **state**
    2. If no, remain in **state** *Patrol* and repeat action and decision *Look*

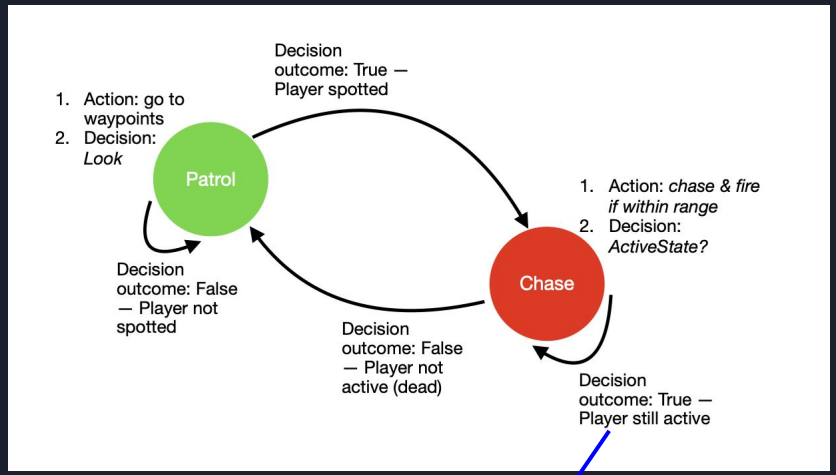


AITankChaser State Transition Diagram

2. At *chase* **state**, perform two **actions**. Action 1: *Move to the last chase target location of Player,* and then Action 2: *Attack — fire bullet if player is within range — otherwise this attack motion does nothing*.

Then, to perform **decision** called *ActiveStateDecision*: check if player is still active:

1. If player is still active: The tank will remain in *chase* **state** and perform *chase* and *attack* **action** until the player until the player dies.
2. If player object is no longer active: — means dead — The tank will go back to patrolling the 3 waypoints (the action in *patrol* **state**).



At each time step, this sets the navmesh destination as the player's last destination — yes even when you're out of sight, the Tank has gotten the address of your Player object so it knows where you are regardless of whether there's obstacle within line of sight or not, so you will keep running around on the scene.

AITankScanner

AITankScanner is a more advanced AI, in the sense that it will NOT just chase you to your death even when you're out of sight.

This is the green tank in your scene.

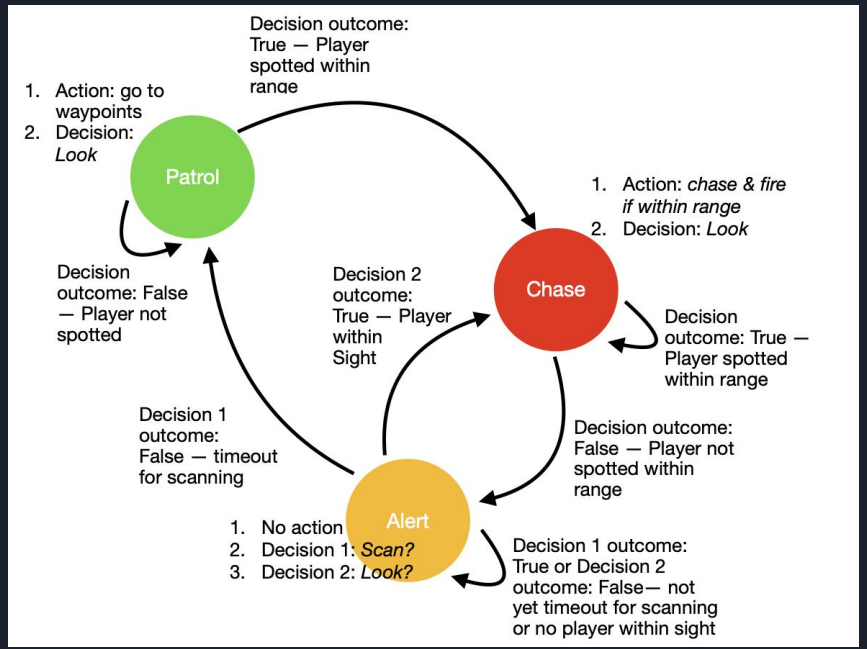
We will implement the same structure of state machine for this Tank too:

1. The enemy tank begin at a StartState
2. Upon entering any **state**, you can perform **action**(s).
3. Then, make a **decision**, as to what is the next state is to go to.

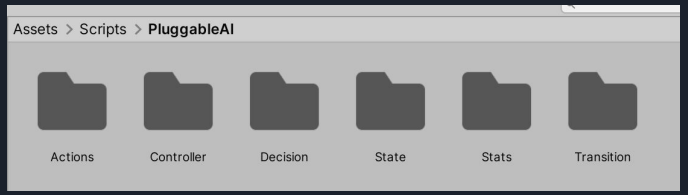
AITankScanner State Machine Diagram

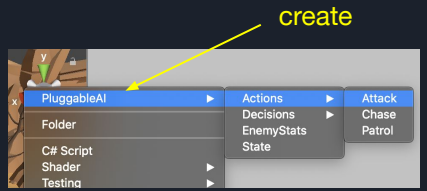
In essence:

1. An AITankScanner will start in the *patrol* **state,**
2. Perform the **action** of going around waypoints if there's no player in sight.
3. It will also perform the Look **decision**. If there's a player in sight, it will go to Chase **state** (chase and fire).
4. If player's no longer within sight then it will go to an alert state where it will **perform two decisions — one after another:**
   a. *Scan* (rotate in place to search for player) and then *Look* **Decision**. In *Scan* **decision**, there's a timeout involved. *Task: Find out how to keep track of the time elapsed while scanning. Which scriptable objects / files are involved?*
   b. The tank will go either to patrol **state**, remain in scan **state**, or to chase state accordingly depending on the outcome of these two decisions.



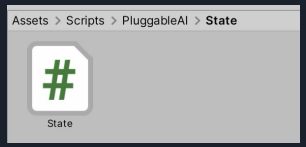
Step 1: State, Action, and Decision

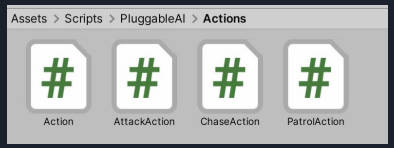


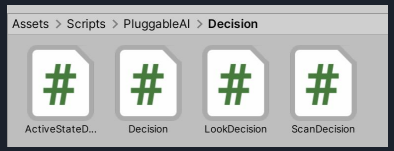


Inside PluggableAI folder, there's three folders containing our important ScriptableObjects:

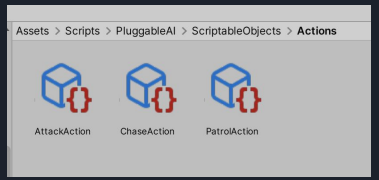
1. **State** (as the name says)
2. **Action**: all other inheriting from the base Action.cs
3. **Decision**: all other inheriting from the base Decision.cs

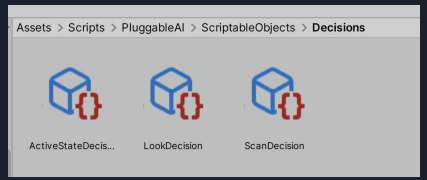






Create scriptable objects (action, decision) out of each:





Step 2: Read how State.cs works

State inherits ScriptableObject, and has several public attributes: actions, and transitions.

It has an important method: UpdateState that is going to be called periodically, and perform two other methods:

1. DoActions and
2. CheckTransitions

The former executes a series of actions specified, and then *decide* what is the next state to go to.

```
using UnityEngine;

[CreateAssetMenu(menuName = "PluggableAI/State")]
public class State : ScriptableObject
{

    public Action[] actions;
    public Transition[] transitions;
    public Color sceneGizmoColor = Color.grey;

    public void UpdateState(StateController controller)
    {
        DoActions(controller);
        CheckTransitions(controller);
    }
}
```

Step 3: Transition.cs

Take a look at the script Transition.cs to understand what constitute this instance.

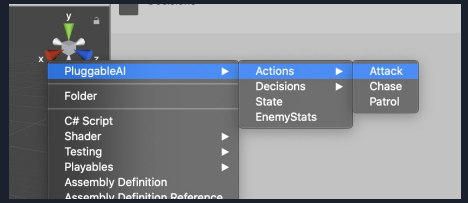
This means that this instance needs 3 components to be set up at the Inspector, which is the Decision, and two states that constitutes True or False.

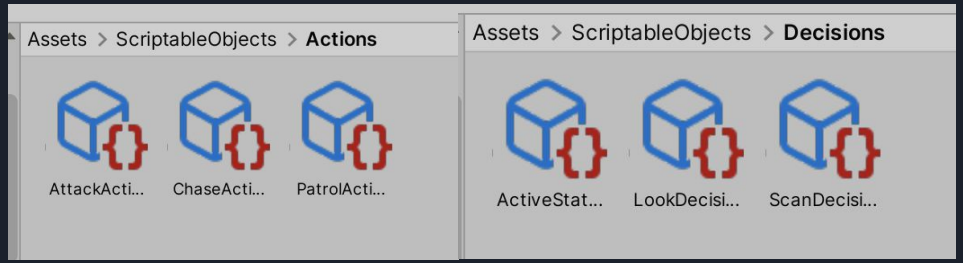
```
[System.Serializable]
public class Transition
{

    public Decision decision;
    public State trueState;
    public State falseState;

}
```

Step 4: Create Actions, Decisions, and States

Create all the following scriptable objects for each Actions, and Decisions.

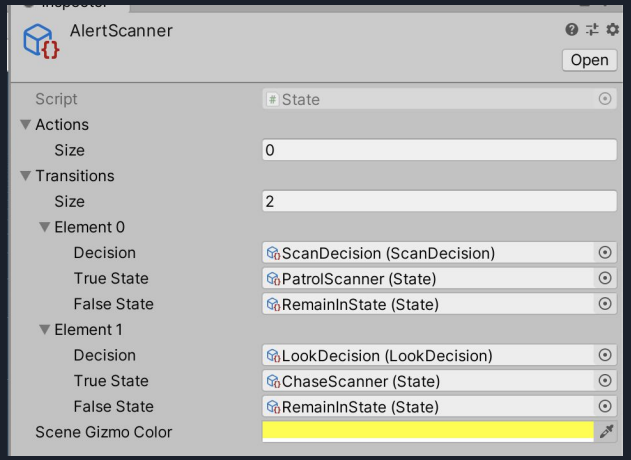
We are going to plug it later on to our States.

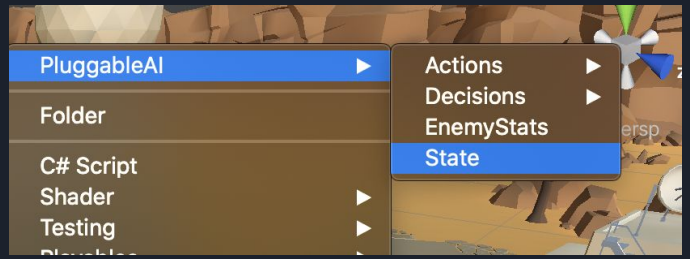


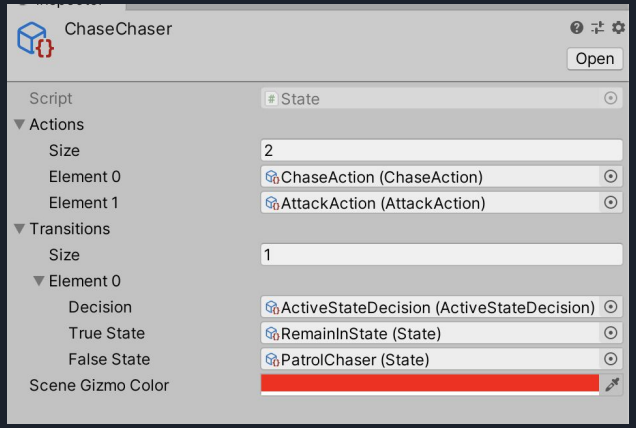


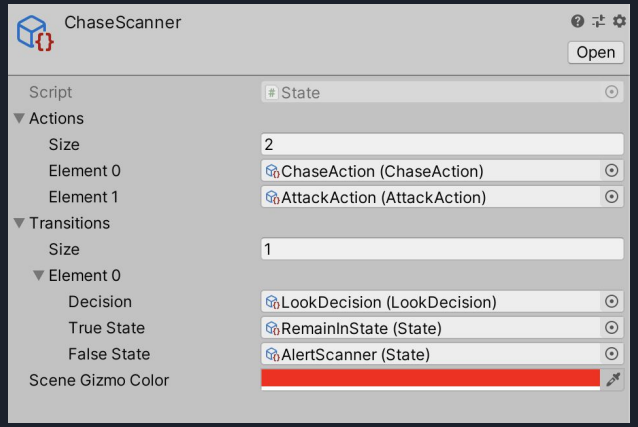
Step 4: Create Actions, Decisions, and States

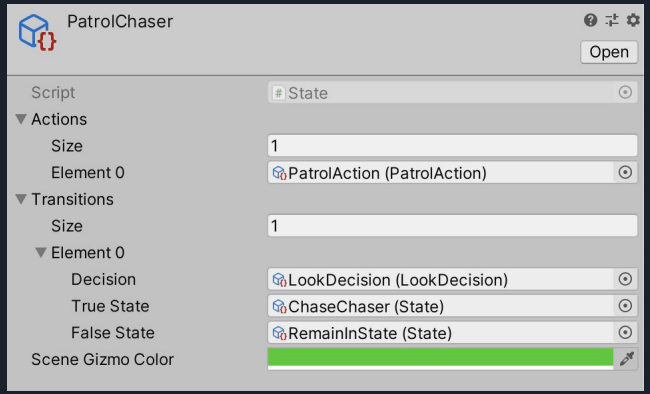
Create the following States ScriptableObjects. Fill up the inspector value as follows for each state:

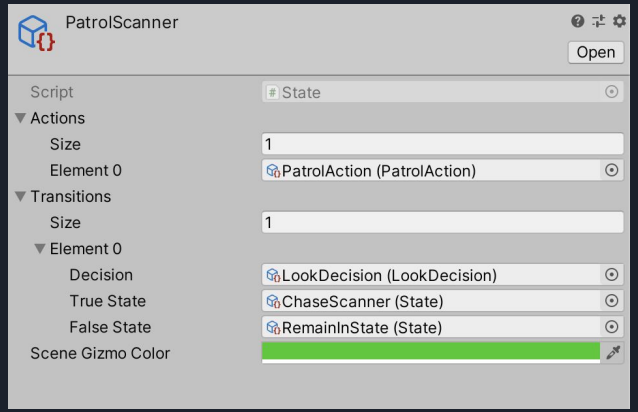





ChaseChaser
Open
Script
State
Actions
Size
2
Element 0
ChaseAction (ChaseAction)
Element 1
AttackAction (AttackAction)
Transitions
Size
1
Element 0
Decision
ActiveStateDecision (ActiveStateDecision)
True State
RemainInState (State)
False State
PatrolChaser (State)
Scene Gizmo Color


ChaseScanner
Open
Script
State
Actions
Size
2
Element 0
ChaseAction (ChaseAction)
Element 1
AttackAction (AttackAction)
Transitions
Size
1
Element 0
Decision
LookDecision (LookDecision)
True State
RemainInState (State)
False State
AlertScanner (State)
Scene Gizmo Color


PatrolChaser
Open
Script
State
Actions
Size
1
Element 0
PatrolAction (PatrolAction)
Transitions
Size
1
Element 0
Decision
LookDecision (LookDecision)
True State
ChaseChaser (State)
False State
RemainInState (State)
Scene Gizmo Color


PatrolScanner
Open
Script
State
Actions
Size
1
Element 0
PatrolAction (PatrolAction)
Transitions
Size
1
Element 0
Decision
LookDecision (LookDecision)
True State
ChaseScanner (State)
False State
RemainInState (State)
Scene Gizmo Color

RemainInState State

Of course you can transit to the same state if you want to, instead of creating a “RemainInState” state.

However for compactness, we can create a State that remains in itself, meaning that it will tell the StateController to execute back the current state as the next state

The method CheckTransitions in State.cs makes use of the StateControllers’ TransitionToState method.


```
private void CheckTransitions(StateController controller)
{
    for (int i = 0; i < transitions.Length; ++i)
    {
        bool decisionSucceded = transitions[i].decision.Decide(controller);

        if (decisionSucceded)
        {
            controller.TransitionToState(transitions[i].trueState);
        }
        else
        {
            controller.TransitionToState(transitions[i].falseState);
        }
    }
}
```

RemainInState State

StateController inherits from MonoBehavior, is attached to a gameObject in the scene.
Responsible for transition of states & set up AI tanks.
(And to checkIfCountDownElapsed)

In StateController.cs script, we have this method:

```
public void TransitionToState(State nextState)
{
    if (nextState == remainState) return;
    currentState = nextState;

    if(currentState.name == "ChaseScanner" || currentState.name ==
"ChaseChaser")gameObject.GetComponent<Animation>().Play();

    OnExitState();
}
```

,

It is clear that if the nextState given equals to remainState (which is a public variable with RemainInState scriptable object linked up), then we don't need to do the double work of setting currentState = nextState in the next line anymore.

Of course the difference between using RemainInState and just setting the nextState as currentState separately depends on whether OnExitState() is called. We will investigate what this OnExtiState() method does soon.

Skip checking for other transition once a transition leads to a NEW next state

Open StateController.cs and add the following variable:

```
    public bool transitionStateChanged = false;
```

And modify `TransitionToState` method into the following:

```
    public void TransitionToState(State nextState)
    {
        if (nextState == remainState) return;
        currentState = nextState;
        transitionStateChanged = true;

        if(currentState.name == "ChaseScanner" || currentState.name ==
"ChaseChaser")gameObject.GetComponent<Animation>().Play();

        OnExitState();
    }
```

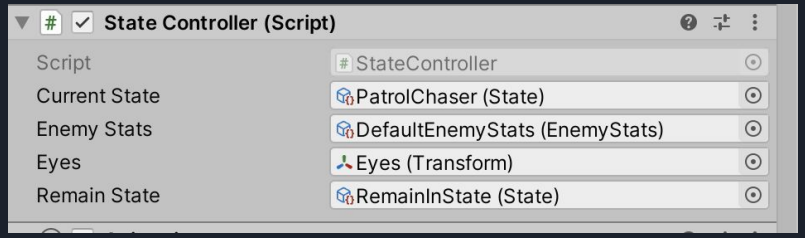
Skip checking for other transition once a transition leads to a NEW next state

Open State.cs and modify the code for the method `CheckTransitions` to the following:

```
    private void CheckTransitions(StateController controller)
    {
        controller.transitionStateChanged = false; //reset
        for (int i = 0; i < transitions.Length; ++i)
        {
            //check if the previous transition has caused a change. If yes, stop. Let Update() in
StateController run again in the next state.
            if (controller.transitionStateChanged){
                break;
            }
            bool decisionSucceded = transitions[i].decision.Decide(controller);
            if (decisionSucceded)
            {
                controller.TransitionToState(transitions[i].trueState);
            }
            else
            {
                controller.TransitionToState(transitions[i].falseState);
            }
        }
    }
```

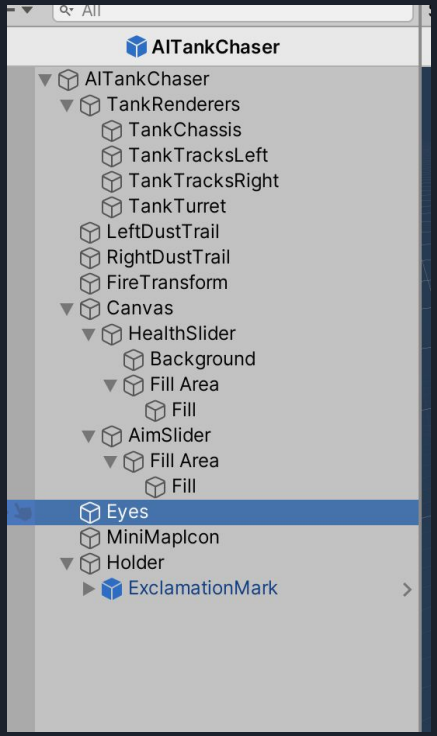
Step 5: Plug in the States for AITankChaser

Go to AITankChaser Prefab, and add the StateController.cs script to it.



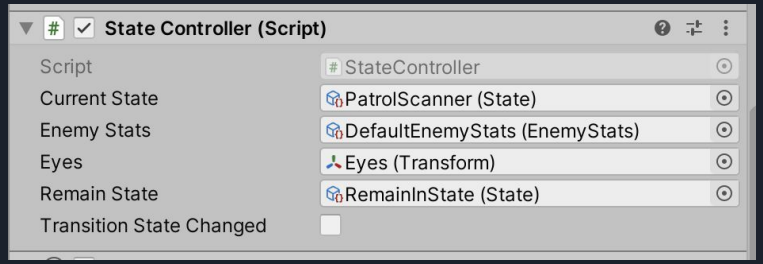
Fill in the inspector as shown. The “Eyes” is just the Eyes GameObject of this prefab.

Test run and you should see the Red Tank try to chase you when it notices where you are.



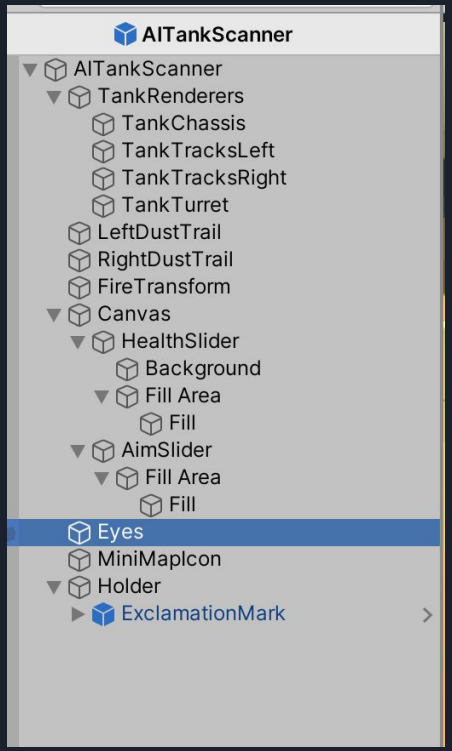
Step 6: Plug in the States for AITankScanner

Go to AITankScanner Prefab, and add the StateController.cs script to it.



Fill in the inspector as shown. The “Eyes” is just the Eyes GameObject of this prefab.

Test run and you should see the Green Tank try to chase you when it notices where you are.

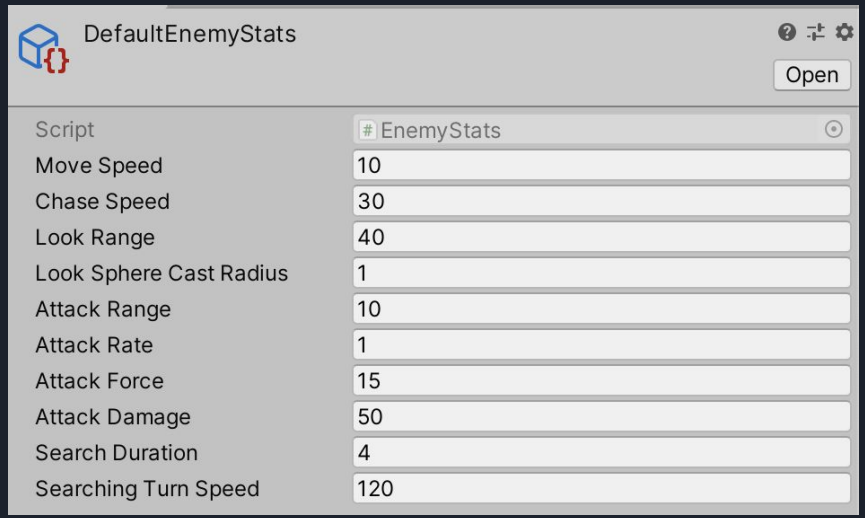


Step 7: Varying movement speed

There's also another ScriptableObject : DefaultEnemyStats that's used by the Decision and Action scripts. This DefaultEnemyStats dictates basic stuff like Look Range, etc.

Now we want to vary the movement speed of the tank when it is Patrolling vs when it is Chasing. Go to EnemyStats.cs and add the modify moveSpeed, and add chaseSpeed variable:

```
public float moveSpeed = 10;
public float chaseSpeed = 30;
```



Step 7: Varying movement speed

Then, go to ChaseAction.cs and modify the chase method to add the following line:

```
    private void Chase(StateController controller)
    {
        controller.navMeshAgent.destination = controller.chaseTarget.position;
        controller.navMeshAgent.isStopped = false;
        controller.navMeshAgent.speed = controller.enemyStats.chaseSpeed;
    }
```

Open PatrolAction.cs and add the following line too:

```
    void Patrol(StateController controller)
    {
        controller.navMeshAgent.destination =
controller.wayPointList[controller.nextWayPoint].position;
        controller.navMeshAgent.isStopped = false;
        controller.navMeshAgent.speed = controller.enemyStats.moveSpeed;
...
}
```

Test run and witness the tank chasing you with speed once they have seen you.

How it works in a nutshell

Here's the steps of whats happening after adding this script to the Tank:

1. The `Update()` method of *StateController* instance in this Chaser tank will call *currentState's* `UpdateState()` method.
2. This `UpdateState()` method will perform `DoActions` and `CheckTransitions`
3. Inside `DoAction(),` the idea is that for each *Action* stored in the *State*, the instance of this *Action* will use the *StateController* instance and perform what needs to be done with the particular Tank. For example, in ChaseAction:

```
0 references
void Update()
{
    if (!aiActive) return;

    currentState.UpdateState(this);
}
```

```
1 reference
public void UpdateState(StateController controller)
{
    DoActions(controller);
    CheckTransitions(controller);
}
```

```
1 reference
public override void Act(StateController controller)
{
    Chase(controller);
}

1 reference
private void Chase(StateController controller)
{
    controller.navMeshAgent.destination = controller.chaseTarget.position;
    controller.navMeshAgent.isStopped = false;
}
```

```
1 reference
private void DoActions(StateController controller)
{
    for (int i = 0; i < actions.Length; i++) actions[i].Act(controller);
}
```

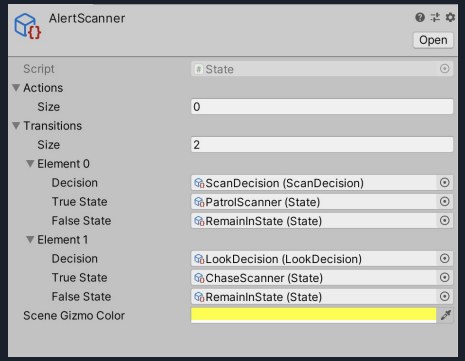
How it works in a nutshell

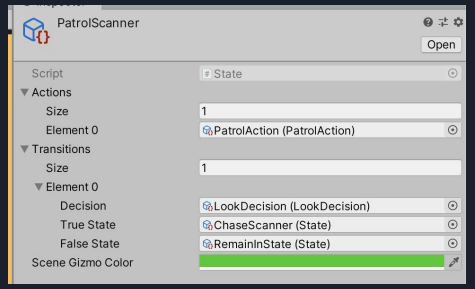
For each transition in this State (arrow out of a state in the state machine diagram), we decide which state to go next:

a. If we remain in state, and IF we have >1 transition, we proceed to check the *next* transition

b. If we go to a different NEXT state, and IF we have >1 transition, we no longer check the second transition. So which transition we check first depends on the *order* we put in the `Transitions` array in State ScriptableObject.

c. For example,
   - after we plug AlertScanner to StateController instance in the AIScannerTank,
   - for each Update(), we will compute transition using ScanDecision first.
   - If it results in True, we go to patrolScanner and perform its PatrolAction and then LookDecision. This will set `controller.transitionStateChanged = true` in StateController.cs (see slide 17)
   - Hence, the loop breaks and the second transition using LookDecision (that's coming from the SECOND element in AlertScanner) is no longer executed.






```
1 reference
private void CheckTransitions(StateController controller)
{
    controller.transitionStateChanged = false; //reset
    for (int i = 0; i < transitions.Length; ++i)
    {
        //check if the previous transition has caused a change. If yes, stop. Let Update() in StateController run
        again in the next state.
        if (controller.transitionStateChanged){
            break;
        }
        bool decisionSucceded = transitions[i].decision.Decide(controller);
        if (decisionSucceded)
        {
            controller.TransitionToState(transitions[i].trueState);
        }
        else
        {
            controller.TransitionToState(transitions[i].falseState);
        }
    }
}
```

Summary

In a nutshell, we have learned how to use ScriptableObject to create a simple Finite State Machine:

1. How to create various types of ScriptableObject: Action, Decision, and State
2. How to use them along with a generic StateController
3. How to trace back the usage of (1) and (2) thats called by StateController

CHECKOFF

It is recommended for you to try and see how GameManager.cs works: to spawn player tank, two AI tanks, and run the main loop, keep the score, etc.

It is also helpful to see TankHealth.cs and TankShooting.cs to see how we keep track of the Tank's stats in the game. The camera (CameraControl.cs) also has done a decent good job to keep you in the third person view.

For checkoff, you're simply required to CREATE A NEW AI TANK TYPE (it doesn't have to be a tank if you choose so, it can be other enemy, etc but you can simply stick with a Tank of a different color if you don't have time) with a NEW state transition behavior. You are free to be as creative as possible. In order to do this, you need to edit the inspector of GameManager too to accommodate your new prefab.

See edimension for due date.

Introduction to Unity Part 10 - Navmesh Basics

Week 6

Download the assets from here.
Create a new 3D project and import them.

The original tutorial on Navmesh is taken from UnityLearn: https://learn.unity.com/tutorial/unity-navmesh

Background

In the *Tanks* game (Part 9), our AI Tank seems to be able to somewhat navigate through predetermined *waypoints* automatically.

Notice how we can conveniently set the NavMesh Agent component of the Tank to go to a particular location, dictate its speed, etc (can be found inside ChaseAction.cs or PatrolAction.cs).

NavMesh is a part of Unity.AI module, and is extremely useful to perform **pathfinding** in pre-determined (baked) maps automatically.

```
private void Chase(StateController controller)
{
    controller.navMeshAgent.destination = controller.chaseTarget.position;
    controller.navMeshAgent.isStopped = false;
    controller.navMeshAgent.speed = controller.enemyStats.chaseSpeed;
}
```

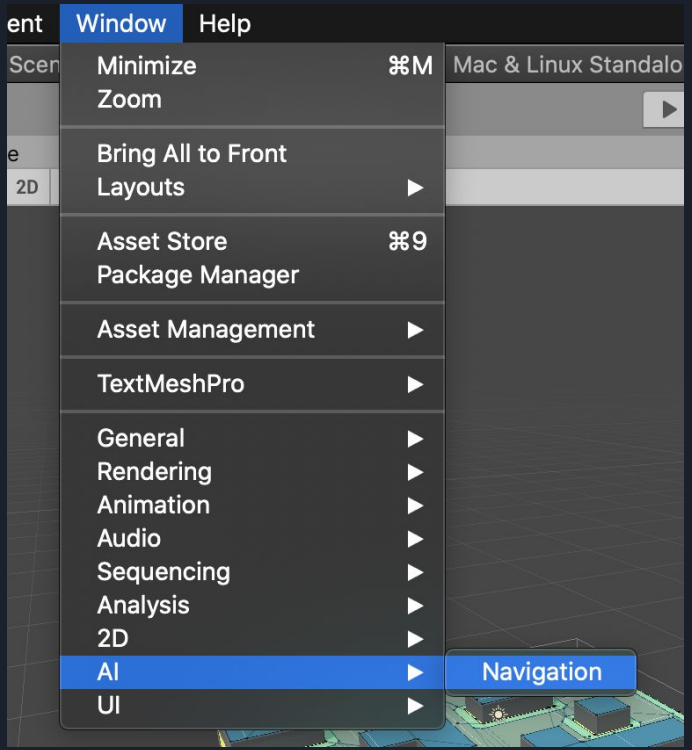
```
1 reference
void Patrol(StateController controller)
{
    controller.navMeshAgent.destination = controller.wayPointList[controller.nextWayPoint].position;
    controller.navMeshAgent.isStopped = false;
    controller.navMeshAgent.speed = controller.enemyStats.moveSpeed;
    if (controller.navMeshAgent.remainingDistance <= controller.navMeshAgent.stoppingDistance && !controller.
    navMeshAgent.pathPending)
    {
        controller.nextWayPoint = (controller.nextWayPoint + 1) % controller.wayPointList.Count;
    }
}
```

Step 1: Open the Navigation Window

Before we begin doing anything, open the Navigation window.

This will help us to set the properties of the agent, and bake areas.

We can also define “areas” to customise certain floor / walkable area more later on.

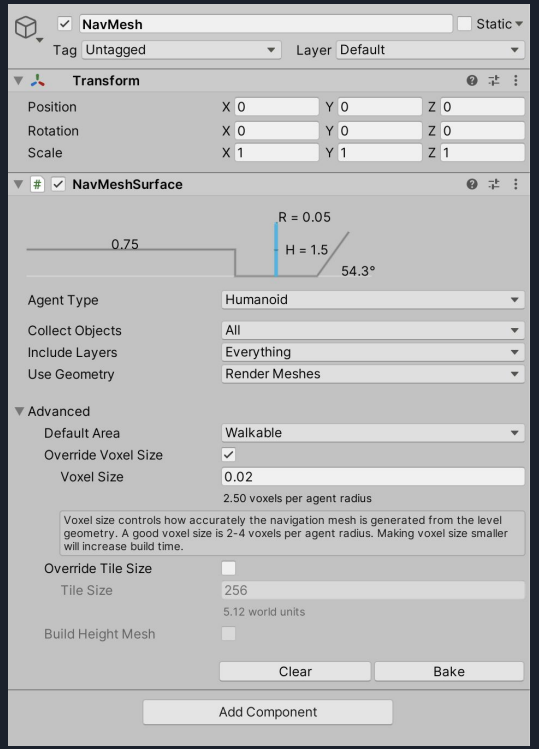


Step 2: Creating Basic Navmesh in the Scene

Open Scene1. In the scene, we see several things already placed for you such as the Level (map), camera, and light.

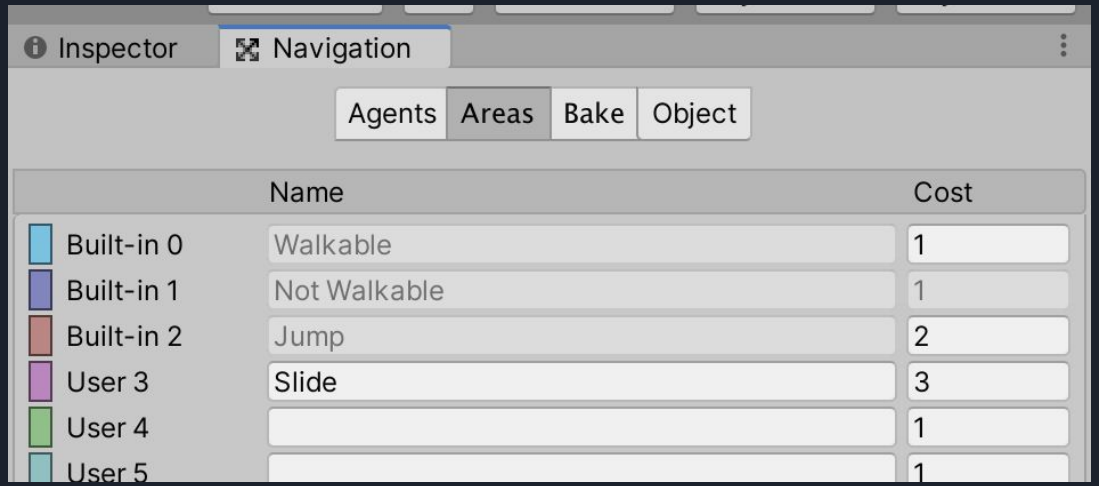
The Level is consisted of a rectangular platform (floor) and a series of taller shapes that act as a wall.

Add an empty GameObject in the scene. Name it “NavMesh” and add a NavmeshSurface component to it with these values.



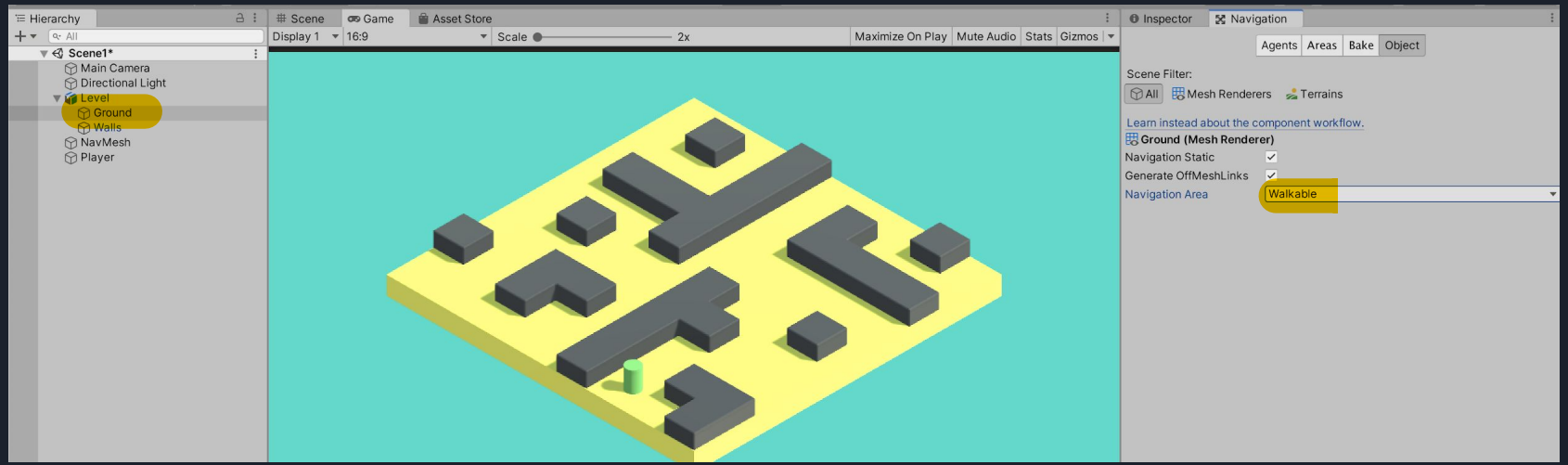
Step 3: Setting NavMesh Areas

You can define areas in the scene and its “cost”. The Agent will try to go path of least cost. The first three areas are built in though, you cannot modify them.



Step 3: Setting NavMesh Areas

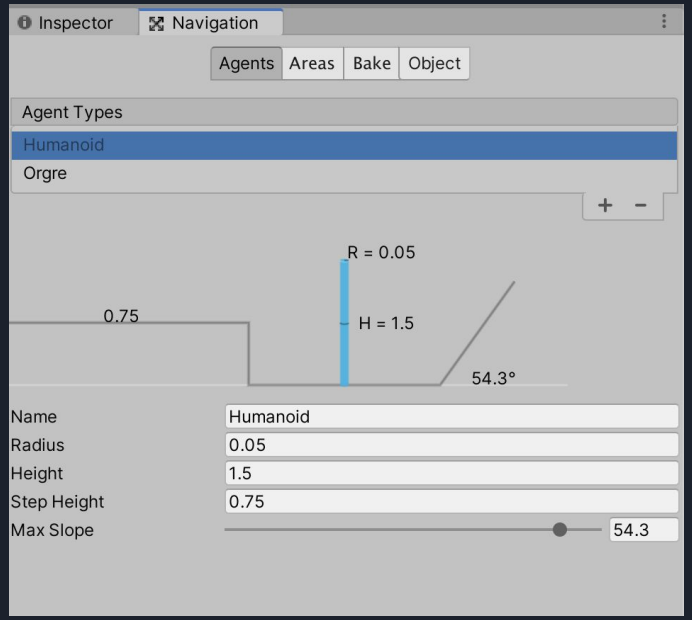
Afterwards click on any GameObject in the scene that has MeshRenderer component, and click on the Object tab in the Navigation Window. Tick Navigation Static and set the area. You can ignore the meaning of OffMesh link for now. Do this for every object which area you want to define.



Step 4: Defining NavmeshAgent

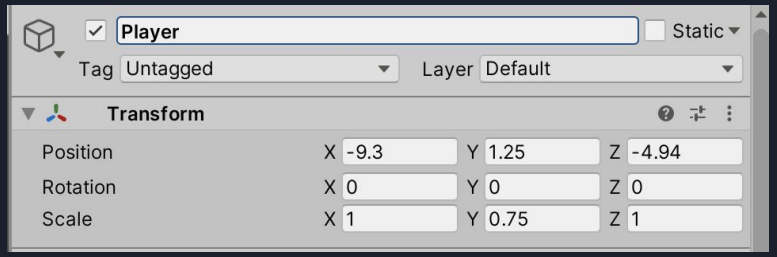
In the Navigation Window, we can create new agent and define them, such as:

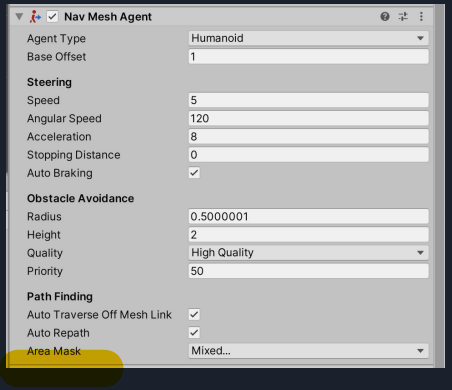
1. The max slope: Slope higher than this degree will not be climbable
2. Step height: objects with this height will no longer be walkable (like its a wall you cannot overcome)



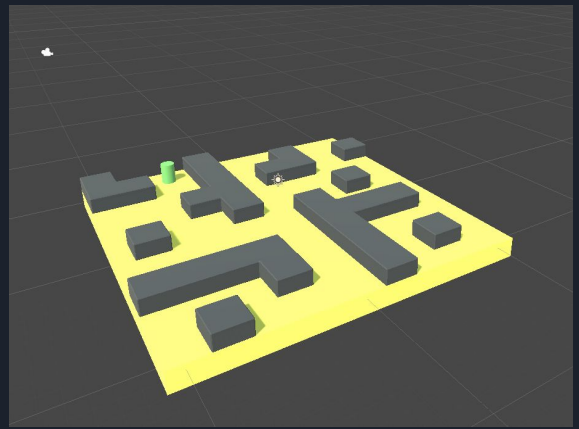
Step 5: Creating NavmeshAgent

Add a Cylinder to the scene and place it at the shown location. This will be your player. Add the NavMesh Agent component. You can modify its speed, turn rate, etc as you wish. Select the agent type that you have set in Step 3.





Area mask



Step 6: Moving the Agent

Create a new script called PlayerController1.cs

At the Start method, get the Scene's main camera and the NavMeshAgent component.

```
using UnityEngine;
using UnityEngine.AI;

public class PlayerController1 : MonoBehaviour
{
    private Camera cam;

    private NavMeshAgent agent;
    // Start is called before the first frame update
    void Start()
    {
        cam = Camera.main;
        agent = gameObject.GetComponent<NavMeshAgent>();
    }
}
```

Step 5: Moving the Agent

Create a new script called PlayerController1.cs

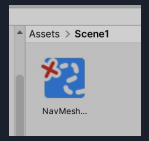
At the Start method, get the Scene's main camera and the NavMeshAgent component.

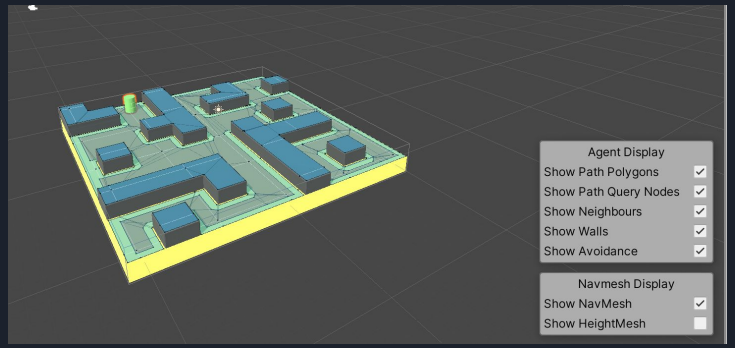
Then add the shown code in the Update() method. We are using the Raycast method to obtain the actual intersection coordinate of where we are clicking in the scene, and ask the agent to go to the location.

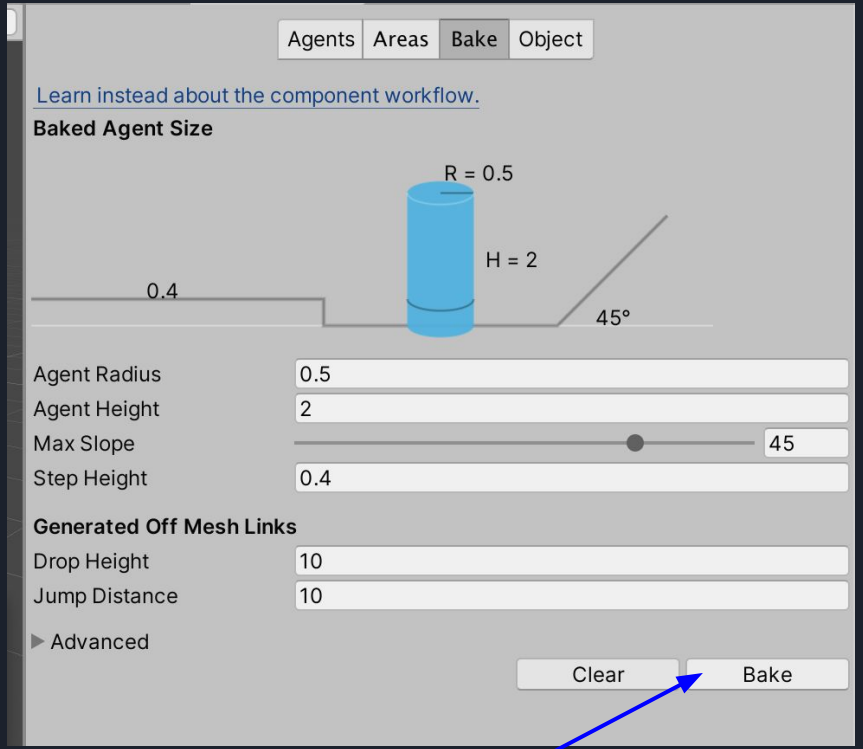
```
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Ray ray =
cam.ScreenPointToRay(Input.mousePosition);
            RaycastHit hit;

            if (Physics.Raycast(ray, out hit))
            {
                agent.SetDestination(hit.point);
            }
        }
    }
```

Step 7: Bake the NavMesh

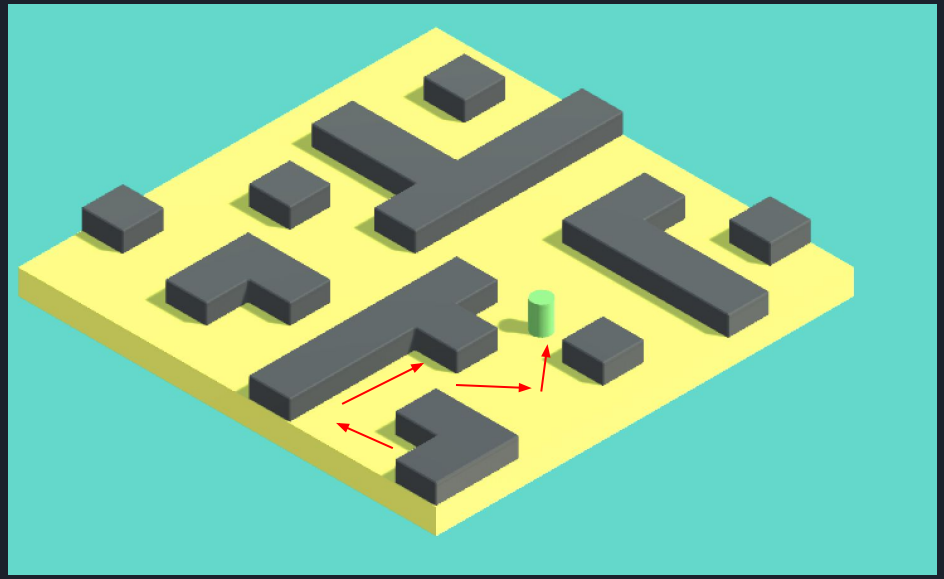
Go to the Navigation window, and click “Bake” in the Bake tab. This will pre-compute the “graph” that the agent can traverse. You can see it in the scene. The pre computed map is stored in the hierarchy.







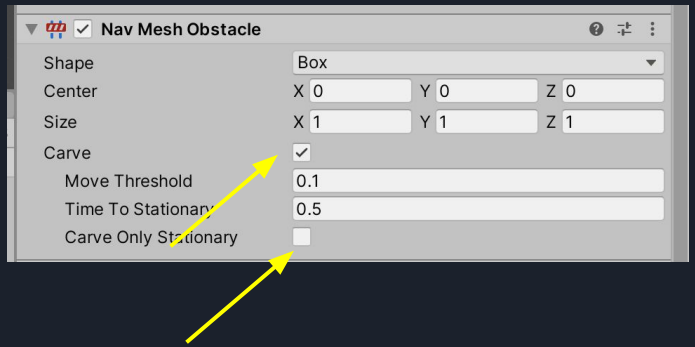
Test: run and click on the map to watch the Player move to your desired location.



Step 8: Moving Obstacles at Runtime

As you have noticed, the NavMesh map so far is generated *before* the program is run, meaning tha tit is a static map. We can also choose to create *obstacles* at runtime, hence modifying the map as the program runs instead of only once in the beginning.

Open Scene2 and add PlayerController1 Script to the GameObject Player. For each Obstacle in the scene, add the NavMeshObstacle component to it and set it with the following parameters:

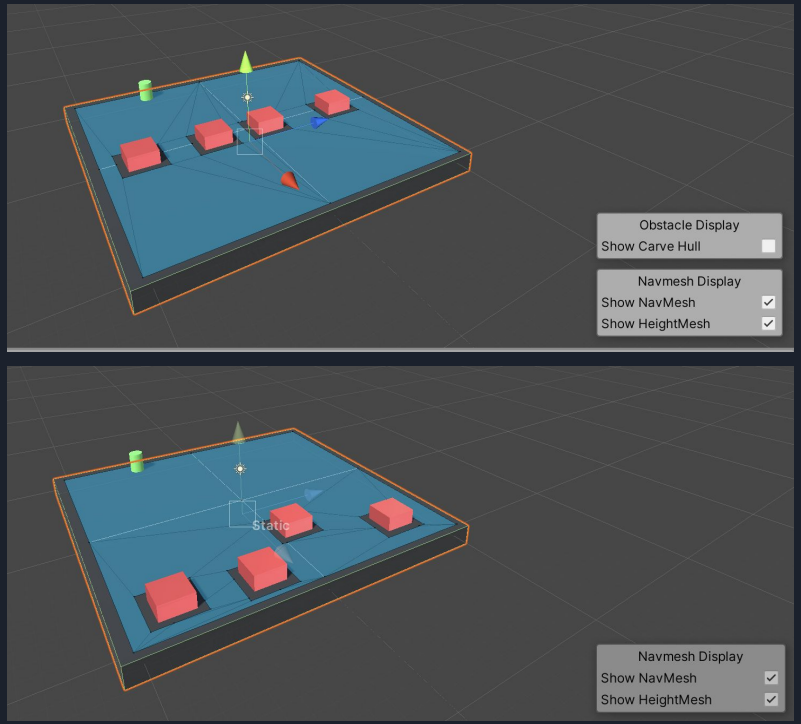


Carve

Step 8: Moving Obstacles at Runtime

Bake the NavMesh, and you shall see the obstacles to be not highlighted in blue (means not considered as a legal area by the Agent).

Test run and you will see the navmesh edited at runtime with these obstacles moving.



Step 9: Generating Obstacles at Runtime

You can also create obstacles at runtime. Create a new script called Obstacle.cs, and add the following variables:

```
using UnityEngine;
using UnityEngine.AI;

public class Generate : MonoBehaviour
{
    public NavMeshSurface surface;
    public int width = 10;
    public int height = 10;

    public GameObject wall;
}
```

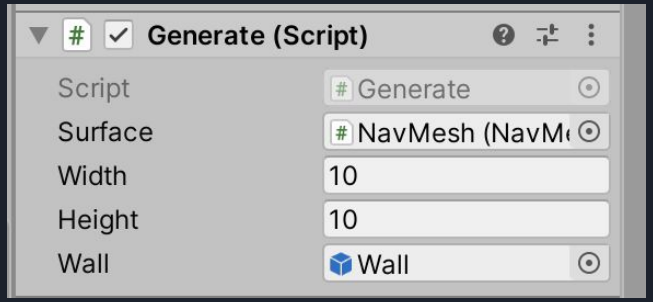
Add this method too:

```
    void GenerateWall()
    {
        float x = (Random.value - 0.5f) * 20.0f;
        float z = (Random.value - 0.5f) * 20.0f;
        Vector3 pos = new Vector3(x, 1f, z);
        GameObject instance =  Instantiate(wall,
pos, Quaternion.identity, transform);

instance.AddComponent(typeof(NavMeshObstacle));

instance.GetComponent<NavMeshObstacle>().carving
= true;
    }
```

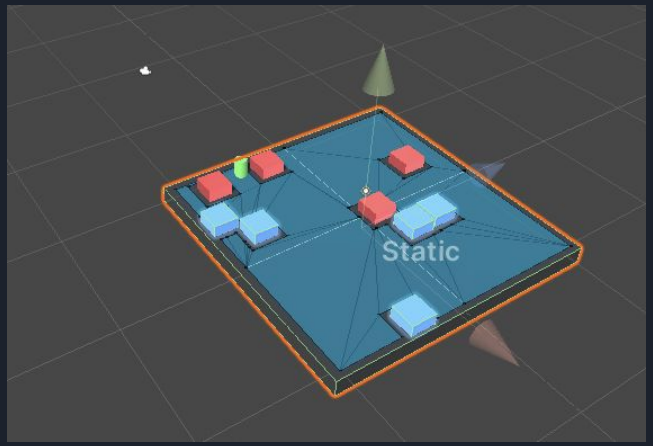
Step 9: Generating Obstacles at Runtime

And call it in Update() when the key ‘g’ is pressed:

```
void Update()
{
    if (Input.GetKeyDown("g"))
    {
        GenerateWall();
    }
}
```



Add this script to Ground object in Scene2, and Run it. You should see the map being updated as you press g. Its ok to see it overlap with other obstacles. We just want to demonstrate spawning obstacles and didn’t handle them from overlapping with one another.



Step 10: Removing Obstacles at Runtime

You can also for instance, erase everything that was generated in Step 9 at runtime. Add this code inside the Update() method of Generate.cs

Press ‘g’ several times after running the program to spawn obstacles, and then ‘r’ to reset them.

```
        if (Input.GetKeyDown("r"))
        {
            int childs = transform.childCount;
            Debug.Log("Children count : " +
childs.ToString());
            for (int i = 0; i < childs; i++)
            {

Destroy(transform.GetChild(i).gameObject);
            }
        }
```

This is a hack

Step 11: Removing Static navmesh object at Runtime?

If you want to remove static obstacles, you need to erase the existing data and rebuild the NavMesh. Of course by definition, static means STATIC, and you should use OBSTACLE if you'd like to modify stuff at runtime.

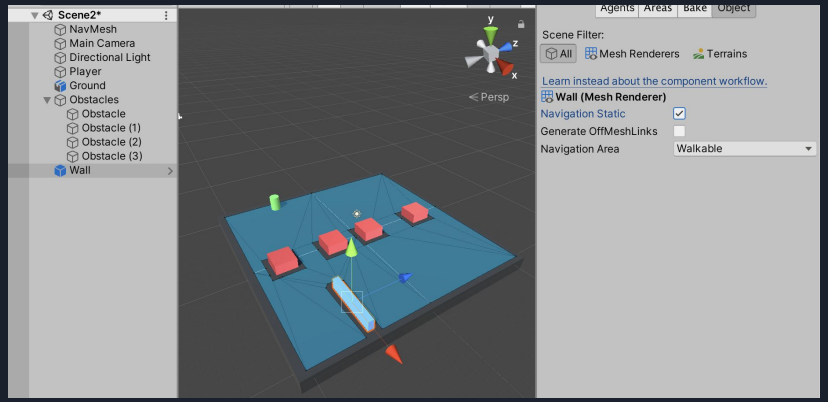
To test: Add the wall prefab to the scene and bake it.

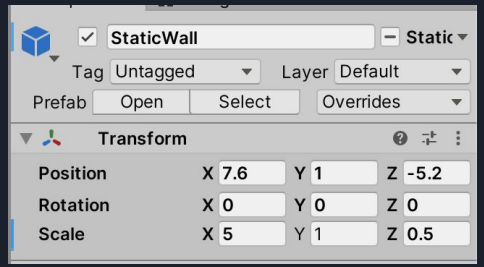
Add the variable on Generate.cs:

```
public GameObject staticWall;
```

And this line inside the Update() method and if key-pressed is key 'r' clause:

```
Destroy(staticWall);
```

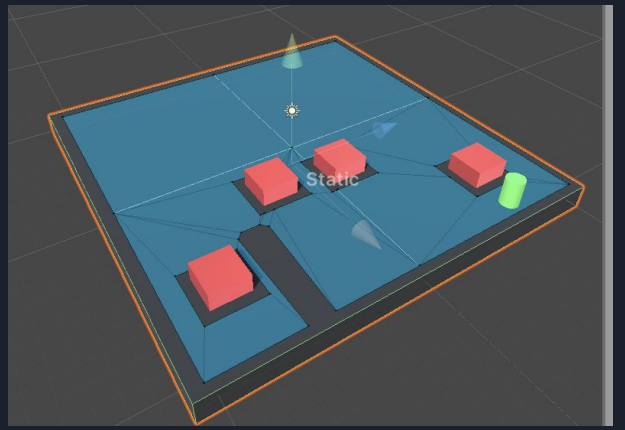




Step 11: Removing Static navmesh object at Runtime?

Run the game and press ‘r’. The wall disappears but the map isn’t updated.

The Navmesh that we baked previously is kept the same. Although we are *not* supposed to change static (baked) objects at runtime, of course theres *workaround*.



Step 11: Removing Static navmesh object at Runtime?

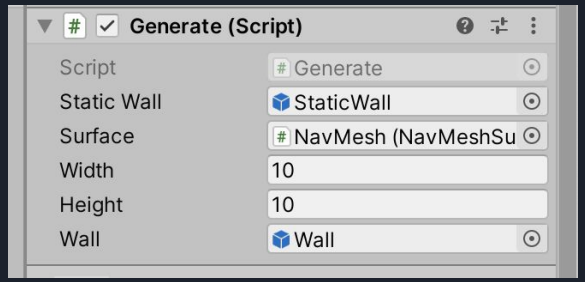
First thing first, delete this baked NavMesh.

Then, under Start() in Generate.cs, build the navmesh from the script:

```
    void Start()
    {
        surface.BuildNavMesh();
    }
```

```
    public GameObject staticWall;
```

Add reference of the staticWall to the script for easy access for this sake of testing.



Step 11: Removing Static navmesh object at Runtime?

Under Update(), remove existing staticWall and rebuild the navmesh after removing the staticWall. However you don't really know when it is actually deallocated though, so you need to force wait for a little bit using a Coroutine before removing the existing navmesh data and rebuilding it. *This is too longwinded, and that's its best to let static navmesh obstacles to remain...well...static.*

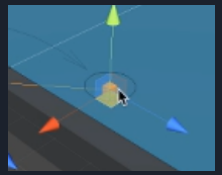
```
    IEnumerator resetNavmesh()
    {
        while (staticWall != null) yield return new
WaitForSeconds(1);
        surface.RemoveData();
        surface.BuildNavMesh();
    }
```

```
    void Update()
    {
        if (Input.GetKeyDown("g"))
        {
            GenerateWall();
        }

        if (Input.GetKeyDown("r"))
        {
            int childs = transform.childCount;
            Debug.Log("Children count : " +
childs.ToString());
            for (int i = 0; i < childs; i++)
            {
                Destroy(transform.GetChild(i).gameObject);
            }
            Destroy(staticWall);

            StartCoroutine(resetNavmesh());

        }
    }
```

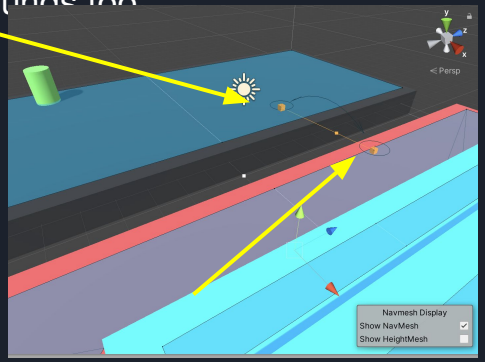
Step 12: Linking Platforms using NavMeshLink

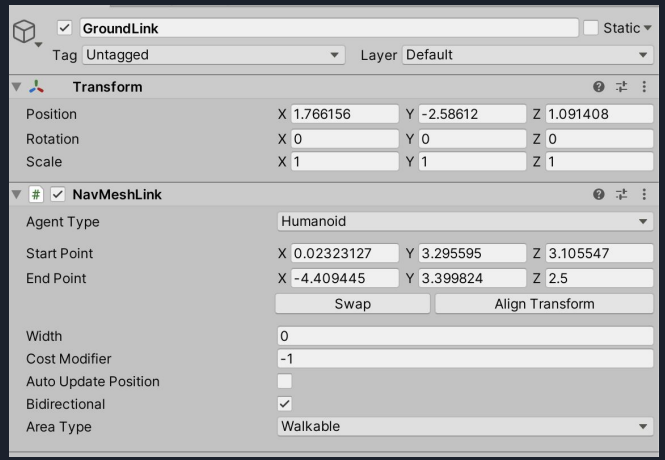
We can allow our agent to “traverse” between platforms using NavMeshLink component.

Open Scene3, and create a new gameobject called GroundLink with the following component. You can manually move the links in the editor until it snaps (has the circle) to both grounds too



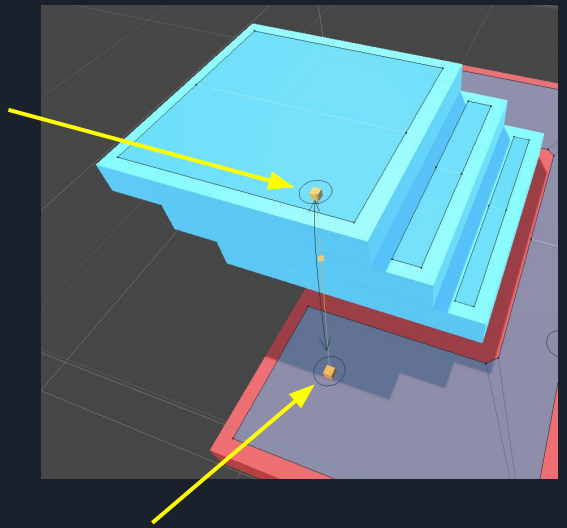
It is a bit hard to do this manually as you dont have a sense of “depth” from the screen. You need to rotate the view in the Scene to ensure its on the grounds

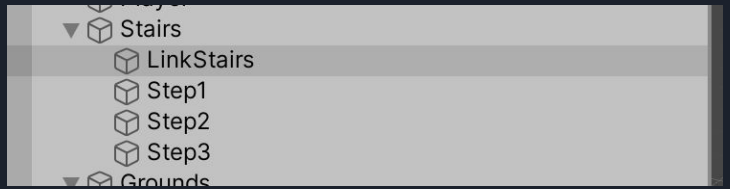


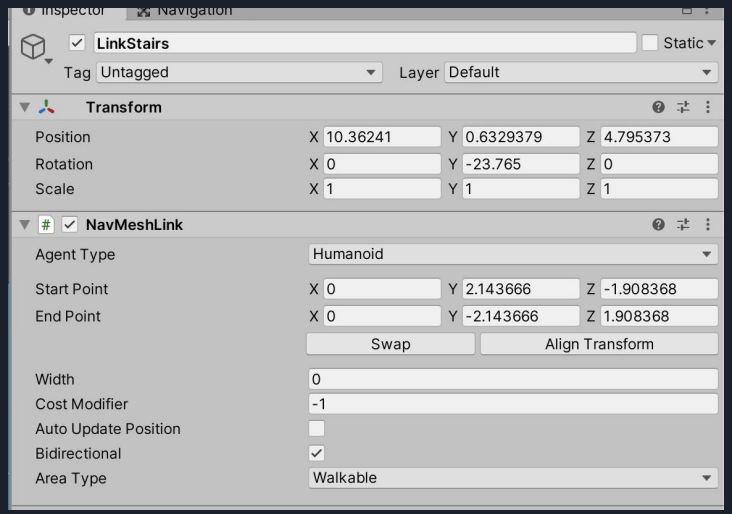


Step 12: Linking Platforms using NavMeshLink

Create another gameObject called stairLink and add the NavMeshLink component too with these settings.







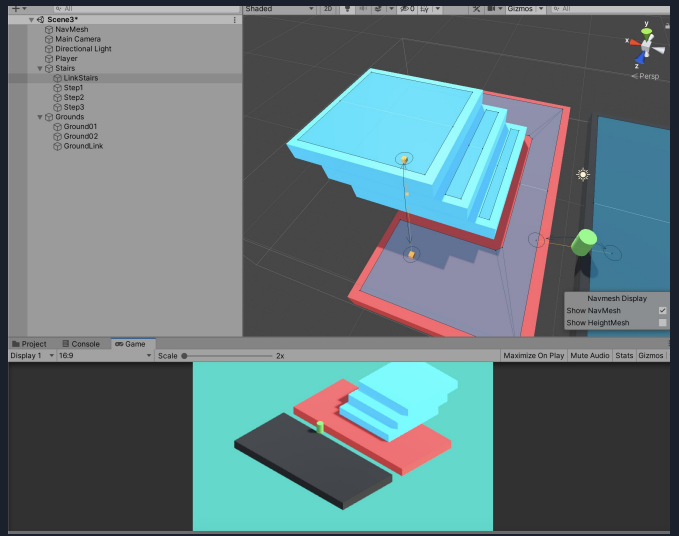
Step 12: Linking Platforms using NavMeshLink

Add PlayerController1.cs to the Player gameobject in Scene3 and test run.

Click on the other side of the platform and see the player traverse to the other side using the NavMeshLink. The stairs are initially not climbable due to the specification of the Humanoid agent (step height must be <0.75), but now it can teleport to the top through the link.

If you met any bug, check that the agent type of the Player is Humanoid, and that the same Humanoid type is allowed on the NavMeshLink setting. Also, bidirectional is ticked for the NavMeshLink.

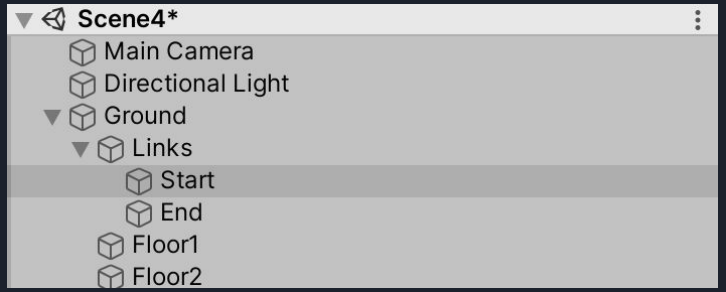
Don't forget to check that each object in the scene that's affecting navigation has NavigationStatic ticked in Navigation >> Object menu tab, and "Bake" is clicked (in the Navigation >> Bake tab)

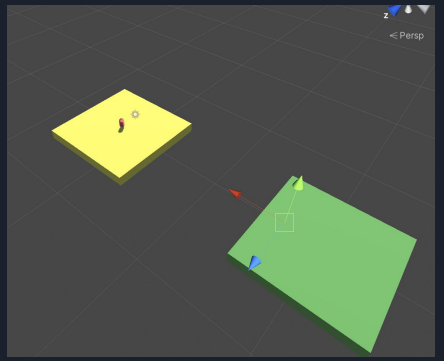


Step 13: Linking Platform using OffMesh Link

Open Scene4 and notice the following GameObjects. “Start” and “End” are simply GameObject with transforms.

We are going to use them as a “portal” for our Player to teleport over between the platforms.



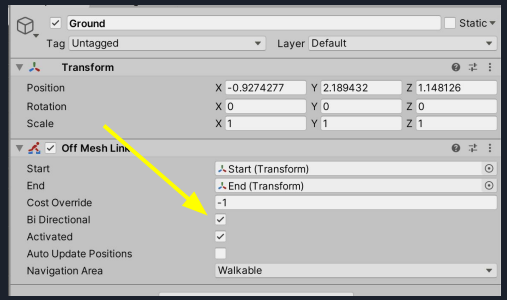


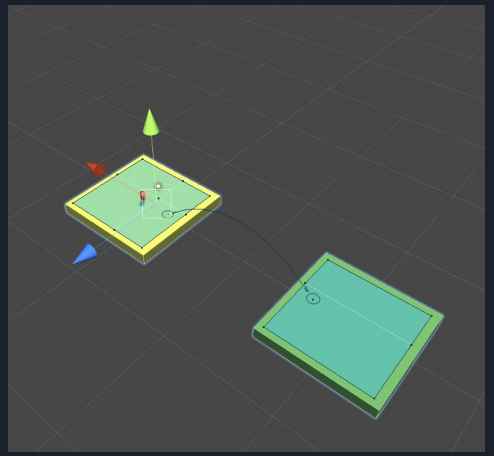
Step 13: Linking Platform using OffMesh Link

We also can connect between walkable platforms using OffMeshLink (instead of NavMeshLink).

Using this, we need two other Transforms as its start and end point (as opposed to the NavMeshLink where just need to specify the coordinates).

Add the OffMeshLink component to the Ground object, and link up the Start and End gameobject from the scene. Bake the NavMesh and you should see the link generated shown on the Scene.



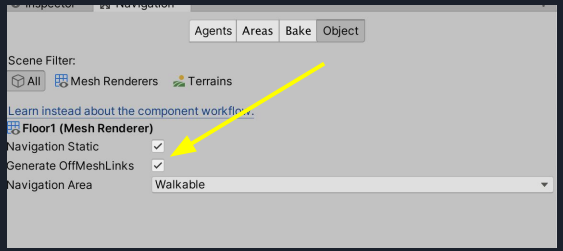


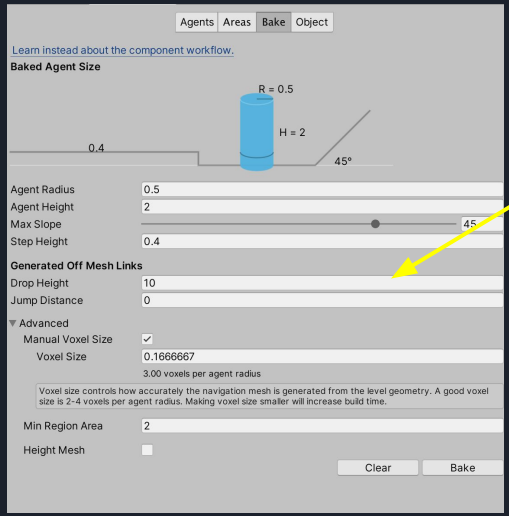
Step 14: Allowing auto-generated OffMesh Link

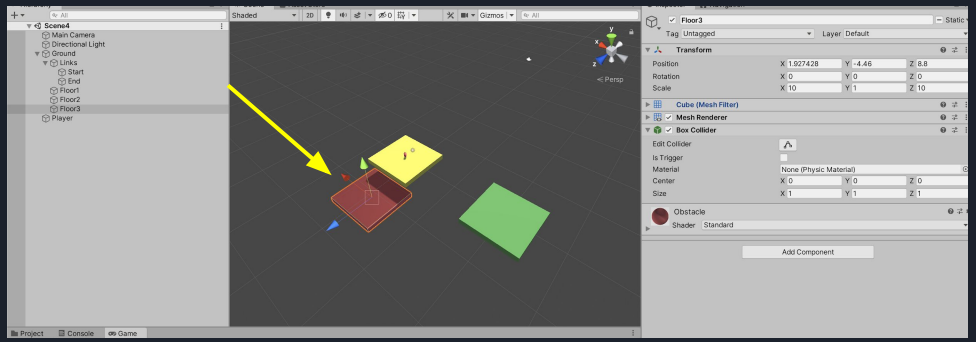
Create another floor object and place it below the yellow one.

Click the yellow floor (Floor 1) and enable GenerateOffMeshLink. Do the same for the new Floor 3.

Go to the Navigation tab and set the Drop Height.







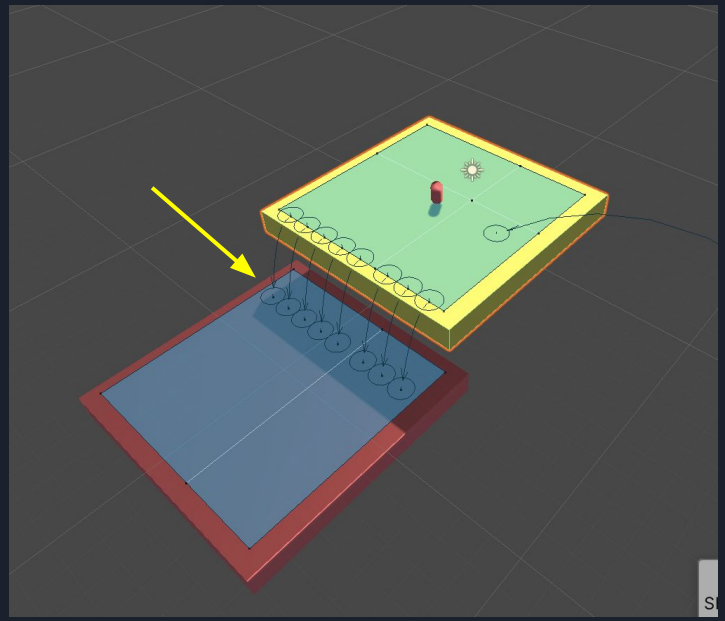
Step 14: Allowing auto-generated OffMesh Link

Click Bake now, and you should see auto generated links as such. Note that these are only *one way*.

Then, create another script called PlayerController2.cs.

Copy the content of PlayerController1 to it.

For now just run it, and see how the player traverse through the OffMeshLink.

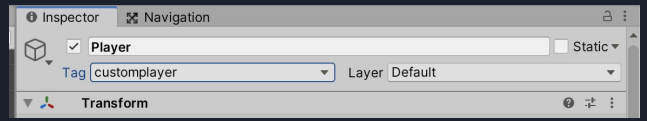


Step 15: Animate Agent GameObject when on OffMeshLink

Add a new tag to the Player. *If you hate tags, you're just going to have to bear with it. Its the fastest thing to use just to demonstrate this point.*

Attach an Animation component (not animator!) and add the jump_anim clip on it.

Now what we need to do is to play this animation while we are on the OffMeshLink.



Add the following private variable, and code inside the Update() method of `PlayerController2.cs`. Of course there's better way instead of constantly checking for these conditions under Update() but you can figure that out later on.

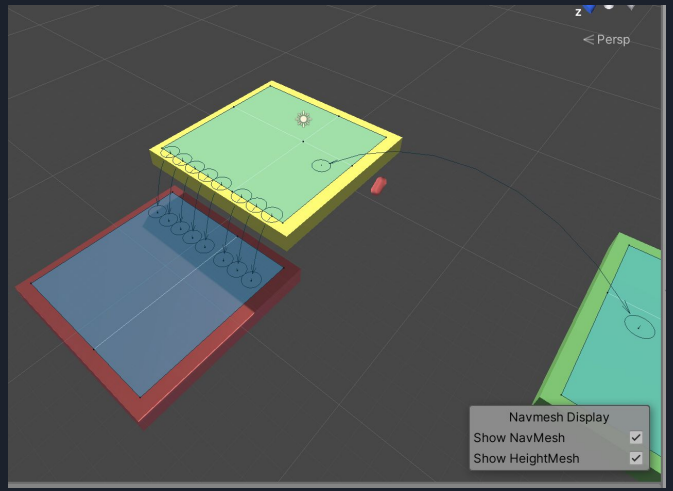
```
private bool isPlaying;
```

```
        if (agent.isOnOffMeshLink && gameObject.tag == "customplayer" && !isPlaying)
        {
            gameObject.GetComponent<Animation>().Play();
            isPlaying = true;
        }

        else if (!agent.isOnOffMeshLink && gameObject.tag == "customplayer" && isPlaying){
            if (isPlaying){
                isPlaying = false;
                gameObject.GetComponent<Animation>().Stop();
            }
        }
```

Step 15: Animate Agent GameObject when on OffMeshLink

Attach PlayerController2.cs to the Player GameObject of Scene4 and watch the agent rotate around while traversing the OffMesh link.

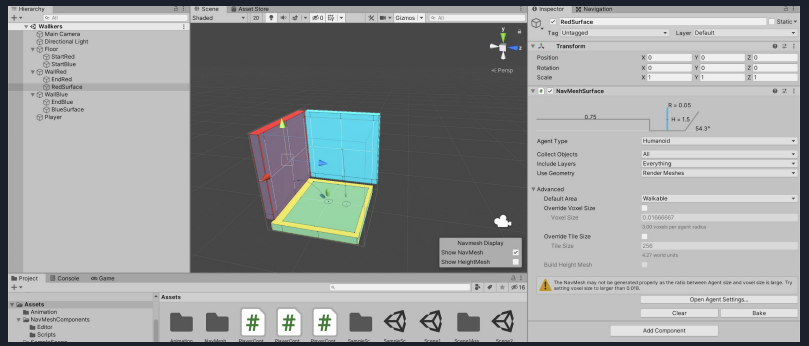
Check it yourself: can you animate this too when using NavMeshLink?



Step 16: Allow Agent to Walk on Walls

Remember in Step 2 we create and define NavMesh surface Object and added NavMeshSurface script to it? And then in other scenes we sort of..never did it anymore but somehow *it works*?

That's because by default, the global positive y-axis direction will be considered automatically as Navmesh generated area. If we want our agent to walk on walls, then we need to redefine the y-axis (local) orientation of the object where the NavMesh surface script is attached on.



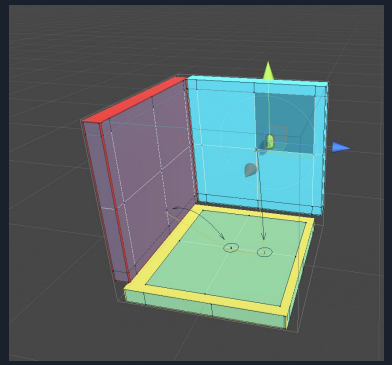
To test this, open the scene named Wallkers.

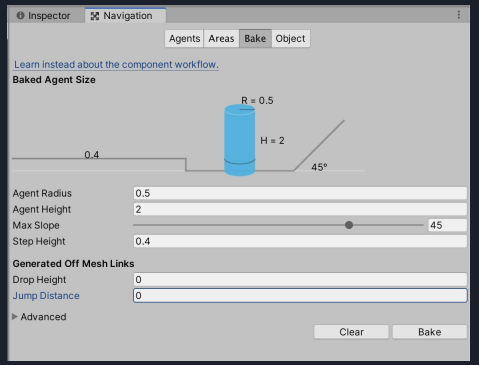
1. Click on RedSurface GameObject. Since it is a child object of WallRed, its local y-axis orientation is actually aligned with the global z-axis.
2. Add the NavMeshSurface component and click "Bake".
3. Do the same to BlueSurface GameObject.

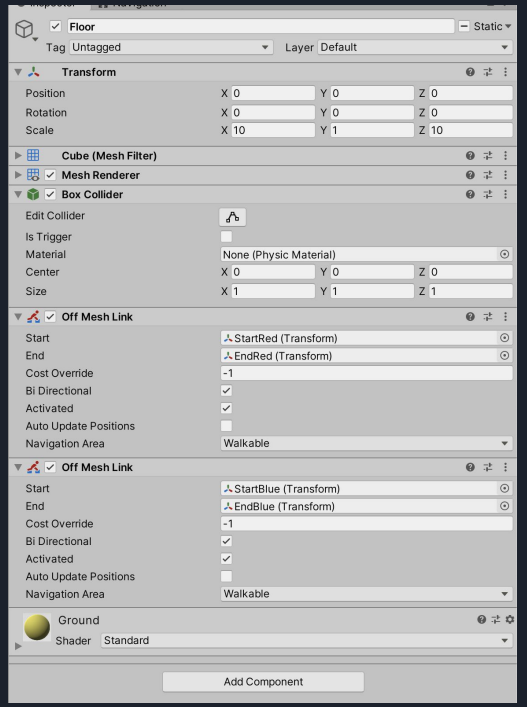
Bake!

We have already added the OffMeshLink to each wall on the Floor GameObject (or you can add it anywhere on a GameObject on the scene if you'd like some neat rearranging). Then Bake it.

Attach PlayerController2 collider to the Player GameObject and watch it stick to the walls.







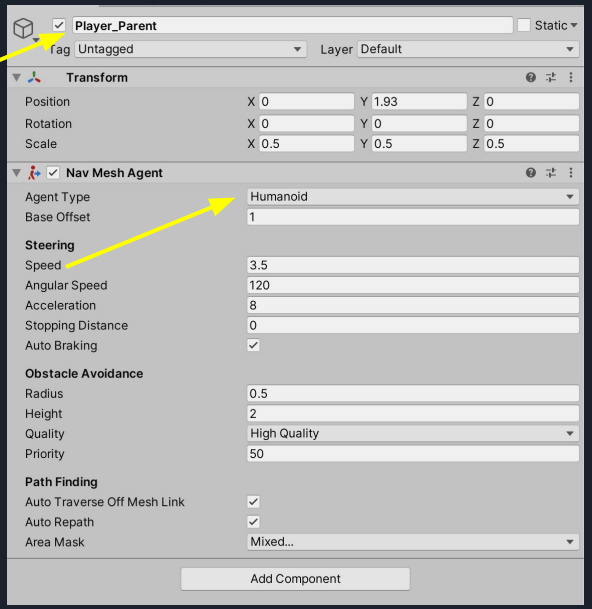
Step 17: Align the Vector.Up of Player

Do this step only if somehow your player's up vector doesn't align with the walls' "up" vector yet.

All we need to do is to manually update the "up" vector of the Agent to match that of the NavMeshSurface it is currently on.

To do this properly follow the steps below:

1. Create a new gameObject called Player_Parent.
2. Set its transform to be equivalent to the Player.
3. And then move the Player as the child of Player_Parent
4. Remove NavMeshAgent component in the Player
5. Add NavMeshAgent component in the Player_Parent
6. Create a new Script PlayerController3.cs and copy off PlayerController2.cs content to it.

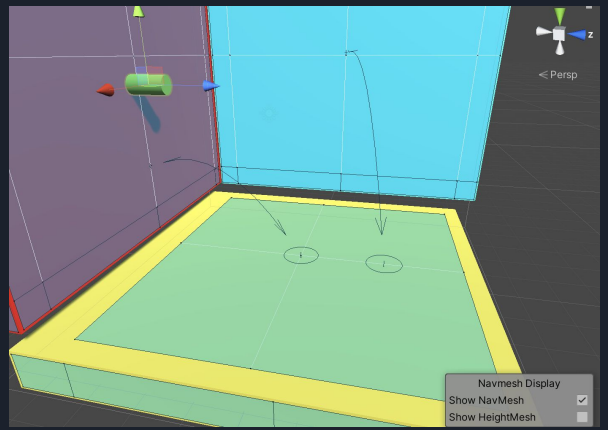


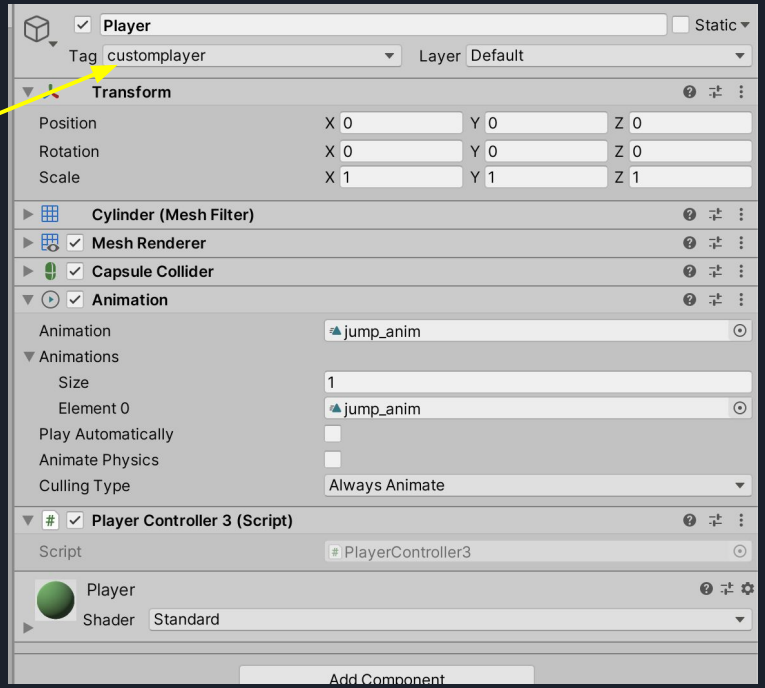
Modify the Start() method of PlayerController3.cs into:

```
void Start()
{

    cam = Camera.main;
    agent = gameObject.GetComponentInParent<NavMeshAgent>();
    agent.updateUpAxis = true;
}
```

And attach this script to the Player in the scene, add the customplayer tag to it, then bake and run it.

As the parent Player gameobject “up” axis is updated, the child’s transform is also updated, hence we see the Player Cylinder aligning with the “up” vector of the Surface its on.





CHECKOFF

The checkoff for this lab is very simple.

1. Create ONE NEW scene where you utilize almost all the points learned in this session: Navmesh Surface, Navmesh Agent, and manual generation of Offmesh / Navmesh Link
2. You need to allow your agent to stick to a wall (doesn't have to be vertical, angled is fine too) and align its up vector as in Step 16
3. You need to animate the agent when traversing certain Offmesh link. Create a new animation.
4. Generate and remove an Obstacle at runtime
5. Generate some OffMesh link automatically (the easiest is to allow agent to jump downwards)
6. Leave STATIC objects as is, please do not remove them at runtime. Step 11 is just a hack.

See edimension for due date.