

Beetle Mania: Part 1 – Player and Shooting

Lesson Goals

1. Create a simplified version of the minigame Beetle Mania from Super Mario RPG.
 - a. We are choosing to omit UI and Menu elements from our game. This is primarily due to time constraints. However, the omitted elements have been covered in the previous lessons.
2. Create the player character controller.
3. Dive into some commonly used math concepts and terminology that are used frequently in the creation of games.
4. Learn how to create level bounds to keep the player inside of the level.
5. Give the player the ability to shoot bullets.
6. Learn about collision layers.
7. Create a trigger that is able to destroy objects.

Lesson Intro and Theory

For this lesson we are going to be making the game **Beetle Mania** scratch. This game appears as a minigame in the game, Super Mario RPG: The Legend of the Seven Stars.



[Here is a link to gameplay.](#)

First off, we are going to be making this game not only because of its simplicity but also because making a game that already exists can be a really good learning tool for beginner game developers.

Making games that already exists can help us step through the design process that the game designer went through when developing the game. We are able to see why they made certain design

decisions as well as allow us to change some of these decisions to see how it might affect the game.

Game Breakdown/Design Decisions

In preparation for making this game, I played Beetle Mania and figured out how various elements of the game worked. Doing so allowed me to formulate a plan on how the game should be developed but also allowed me to make sure that the game we made would be as close to the original as possible.

I was also able to glean a lot of design decisions that went into making this simple game. I highly recommend doing this as an exercise for one of your favorite games. The game you analyze might seem simple on the surface but might hold a fair amount of depth underneath.

Below are my findings on how each element of the game worked.

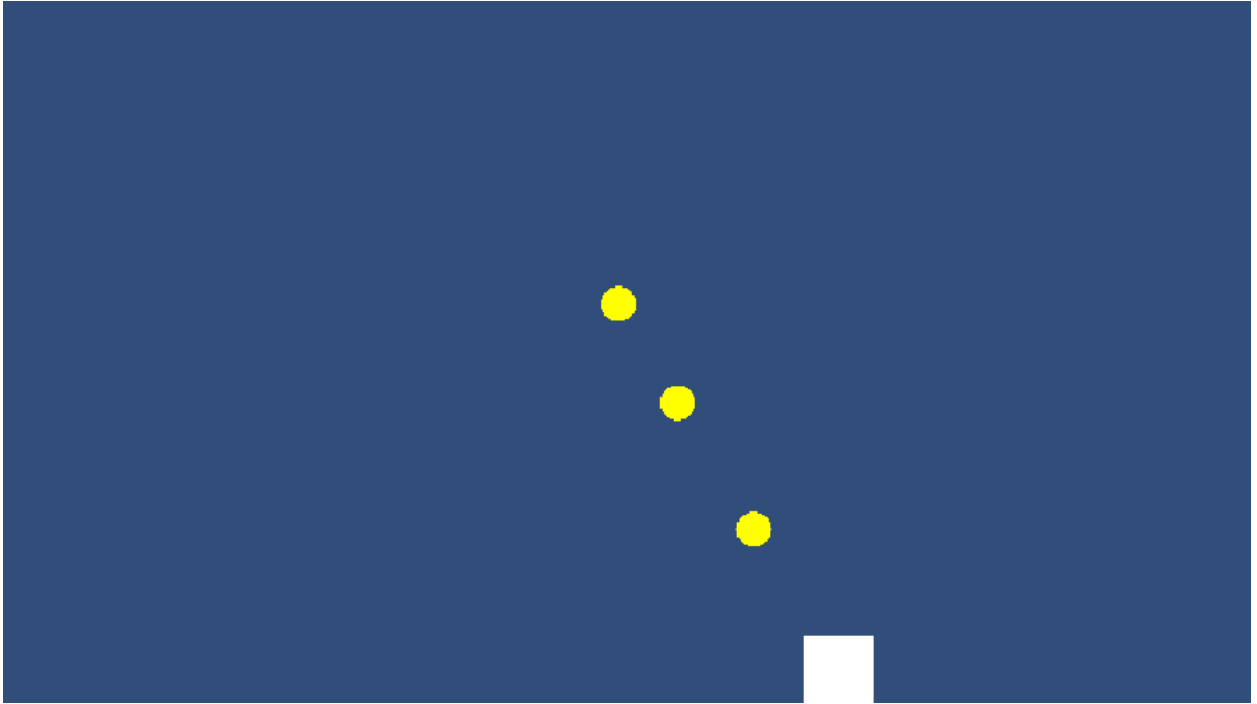
Beetle Mania Breakdown

1. Player moves left and right at the bottom of the screen.
 - a. Player cannot move past the boundaries of the level.
 - b. Player cannot move up or down.
 - c. There does not appear to be any acceleration or deceleration applied to the player.
2. The player is able to shoot out stars at a constant rate.
 - a. Can only shoot out five at a time.
 - b. Stars will de-spawn when they leave the screen or collide with a turtle shell.
3. Turtle shells can fall from anywhere at the top of the screen.
 - a. They fall in a random direction with some randomness to their speed.
 - b. Turtle shells bounce off the walls of the level.
 - c. Turtle shells hitting the ground will bounce off of it.
 - i. They also seem to speed up after hitting the ground.
 - ii. The speed of the turtle shells is capped.
 - d. The game will stop spawning turtle shells once around 30 shells are on screen.
 - e. The game spawns turtles shells about every 0.25 seconds.
4. When turtle shells are hit with a star, they:
 - a. Fire out 5 new stars in a random direction.
 - b. Output a score based on the chain of turtle shell hits.
 - i. Probably some chain number that just increases with every shell hit.
 - ii. After a certain amount of time, the chain drops back down to zero.

- c. Score chain doubles each time.
 - i. $1 > 2 > 4 > 8$ and so on.
 - ii. Has a cap of 9999.
- 5. When the beetle is hit by a shell it:
 - a. Stops in place and needs to be revived.
 - b. Reviving the beetle is based on a number of button presses.
 - i. The beetle needs 5 button presses at first.
 - ii. Each subsequent down, increased the amount of button presses needed.
 - 1. Maybe go up 2/3/4 each time, could randomize it.
 - c. If the button presses needed to revive the beetle is not met within 3 seconds, the game is over.
 - i. The amount of seconds the beetle has left is displayed above the beetles head.
- 6. Hearts
 - a. Hearts fall from the sky if the chain counter is high enough.
 - i. Hearts can fall anywhere from the top of the screen.
 - b. Maybe after a chain of multiples of 20, a heart is dropped.
 - c. When a player collects a heart, the amount of button presses needed to revive is reduced 5.
 - d. If the beetle collects a heart when it is downed, it is instantly revived.
- 7. Game Over
 - a. Beetle explodes into five stars when the game is over, to try and get last minute points.
 - b. When the chain is over, or **when there are not more stars on the screen.**
 - i. Score is displayed at the center of the screen.
 - ii. There is and option to retry the game
 - iii. There is an option to quit the game.
 - c. All the turtle shells in the game stop in place.

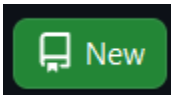
Demo

- 1. Here is what our game will look like by the end of class. It is not the most visually interesting thing but there is a lot going on in the background.




Set Up



1. For this project we are going to be creating a new GitHub repository. Go to GitHub and click the green **New** button to create a new repository.



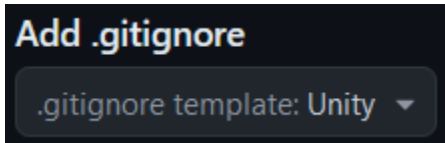
2. Set yourself as the owner and name the repository **BeetleMania**.

Owner *	Repository name *
 AsleepInTheBreakroom ▼	/ BeetleMania
	✔ BeetleMania is available.

3. Set the repository to private:

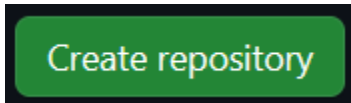
		Private
You choose who can see and commit to this repository.		

4. Be sure to add the Unity .gitignore file.



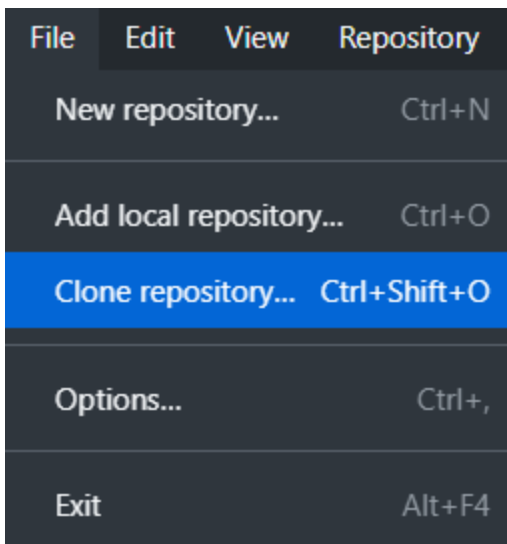
If you do not, you will not be able to upload your project to GitHub.

5. Once all the options are set, click the green **Create Repository** button.

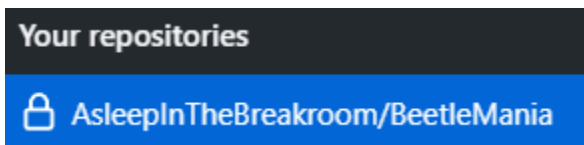


Cloning the Repository

1. Open **GitHub Desktop** and clone the Beetle Mania repository that was just created. Go to **File > Clone Repository**.



2. Once selected, GitHub Desktop will return a list of all of your repositories. Select the Beetle Mania repository.



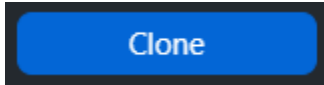
3. Lastly, choose a location that you want the repository to appear at. I recommend putting the repository on the desktop.

Local path

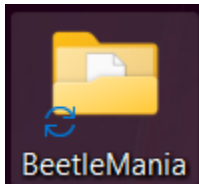
C:\Users\Rossin\OneDrive - Ohio University\Desktop\BeetleMania

Choose...

- Once set, click the blue **Clone** button.

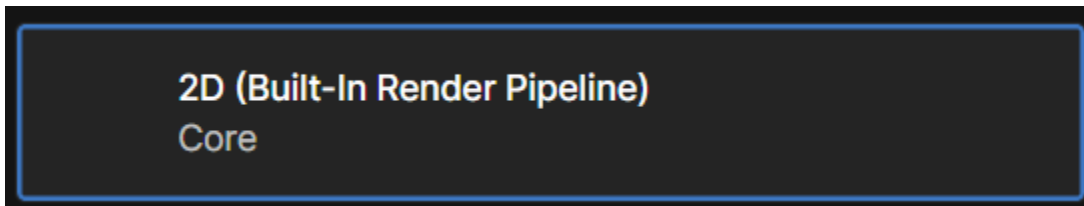


- You should now see the Beetle Mania repository on your computer.

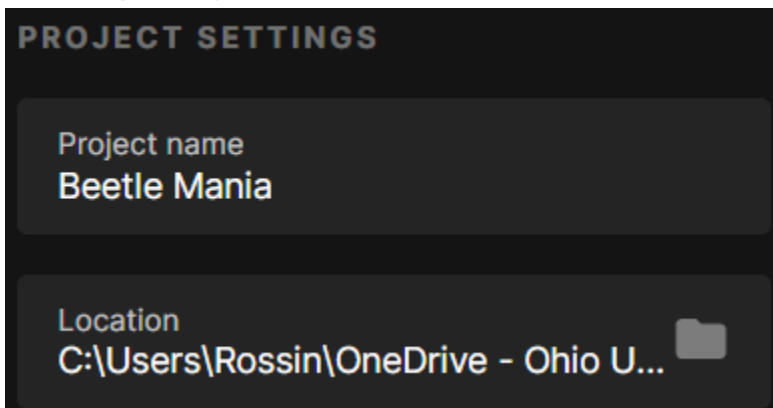


Unity Project

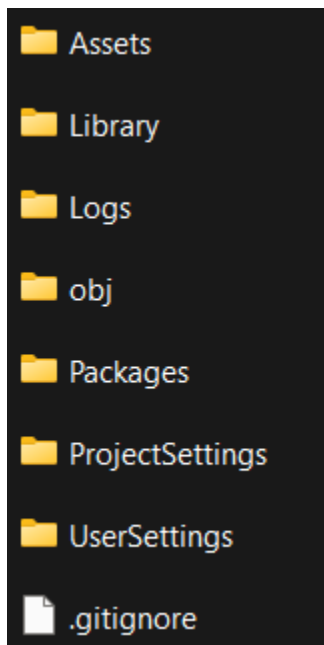
- Next open **Unity Hub** and create a new **2D Core** Unity project.



- Next create a new 2D core Unity project. **Be sure it is being created in the local Beetle Mania repository.**



- Once the project is created, we just need to move the .gitignore file into the right place. First find the .gitignore file and then move it into the same folder the houses all the files for the project.

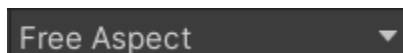


We need to make sure the **.gitignore** is in this spot for our project. If not, we will not be able to commit the project to GitHub. We will get a large file size error.

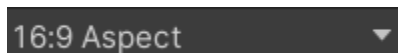
4. The only thing we need to do before we start making our game is adjust the aspect ratio. To do this switch over to the **Game** tab at the top of the game window.



5. There should be a box labeled **Free Aspect**,



Click it and from the drop down select **16:9 Aspect**.



There should now be black bars on the sides of the game window.

6. Be sure that **Toggle Gizmos** is active. In the top right-hand corner of Unity you should see the icon.

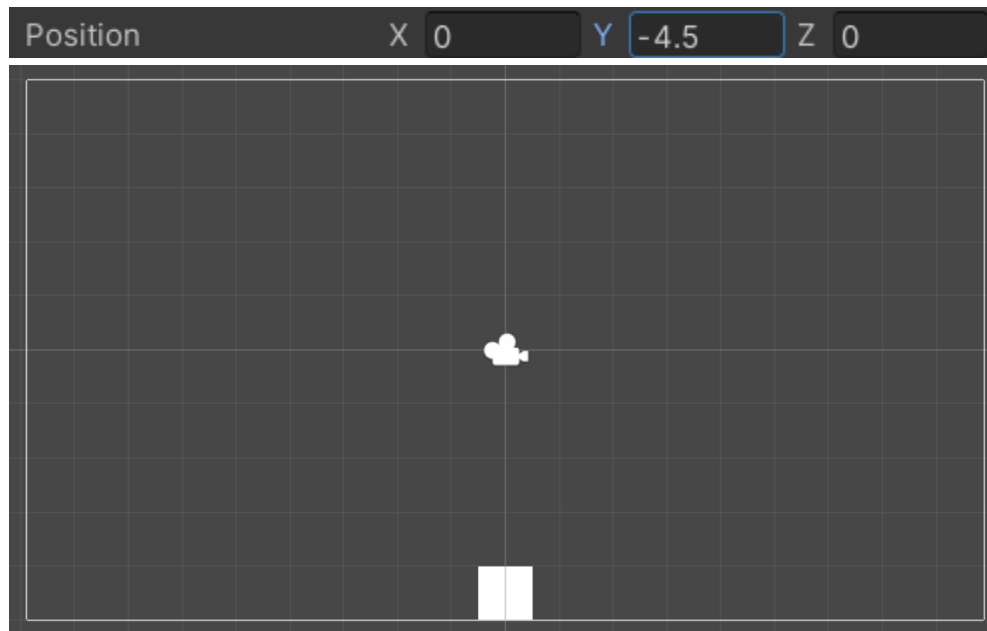


Clicking it will make the icon turn blue. This means you should be able to see game gizmos in the scene.



Player

1. The first thing we are going to add to our game is the player. To do this, add a square sprite to the scene. Name it **Player** and then position it at the bottom of the screen.



2. Create a new script and rename it **Player**. Attach the script to the player game object.



3. Next, we are going to give the player a **Rigidbody2D** component. Select the player, then in the inspector click the **Add Component** button. Search for the **Rigidbody2D** component and then once found, click it to add the component to the player.



A rigidbody is what will allow us to move the player character around. There are three types of bodies in game development: **Rigid Bodies**, **Kinematic Bodies** and **Static Bodies**.

Rigid Bodies are controlled by Unity's physics engine, this means that they will have forces such as gravity or friction acting upon them resulting in the body's velocity changing. We can apply forces or set the velocity of the object directly to move these

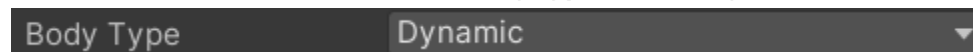
types of objects around.

Kinematic Bodies do not use Unity's physics engine and do not move by forces. Instead, we have to program physics of these bodies ourselves.

Static Bodies are bodies that will not move under any circumstance. Typically, we want to use them for the walls and floors of our game. Rigid Bodies and Kinematic Bodies will be unable to move these bodies in any way.

4. Now that we have the rigid body attached to our object lets change a few of its settings.

First, we need to make sure that the **Body Type** is set to **Dynamic**.

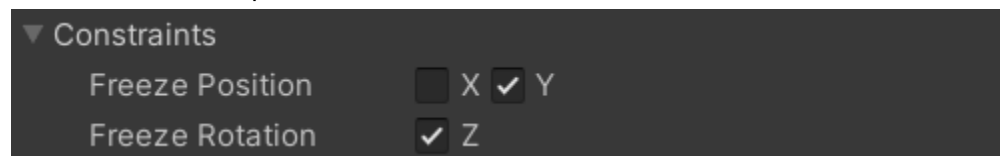


This will allow the rigidbody to collide with other objects.

We set the **Gravity Scale** to **Zero** since we do not want the player to fall.



We then freeze the position and rotation.



This makes it so the player can only move left and right and will not be able to rotate if they collide with another object.

5. Next give the player a box collider.



Colliders are responsible for allowing objects to interact with each other.

6. Now let's open the **Player** script and add the code that allows the player to move.

At the top of the script, let define a few variables.

```
//Movement
private Rigidbody2D rb;
private Vector3 input_vec = Vector3.zero;
public float speed = 300;
```

RB will hold a reference to the rigidbody component that is on the player.

Input_Vec is a vector that will store the **direction** that the player want to go.

Speed is how fast the player will move.

7. In the **Start()** function we write the code:

```
//Get Rigidbody Component  
rb = GetComponent<Rigidbody2D>();
```

Here we are just getting the rigidbody component and storing it in the rb variable.

8. In the **Update()** function lets then write:

```
//Set Input Vector  
input_vec = new Vector3(Input.GetAxisRaw("Horizontal"), 0, 0);
```

Here we are creating a new Vector and storing it inside the input_vec variable. We use the Input class's [GetAxisRaw](#) function to return either a 0, 1 or -1. 0 is if the player is not holding any key. -1 is if the player is holding the A key and 1 is if the player is holding the D key.

Note: As a side note we are getting the player's input in the Update function because we want to let the game know when the player did something as soon as possible. This is an okay method of getting input for right now but in the future we are going to use Unity's [Input System](#) package to handle how we get the input for our game.

9. Lastly we are going to add the fixed FixedUpdate() function to our code. We can add this function under the Update() function. Inside we write the code:

```
void FixedUpdate()  
{  
    //Set the velocity.  
    rb.velocity = input_vec * speed * Time.fixedDeltaTime;  
}
```

Here we are setting the velocity of the rigidbody.

10. If we save the script and run the game the player should now be able to move left and right.

Math Terminology – Vectors, Direction, Speed and Velocity

As a short aside let's talk about some math terminology that is frequently used when programming games. This should hopefully give some context as to what our code in our game is doing so far.

Vectors

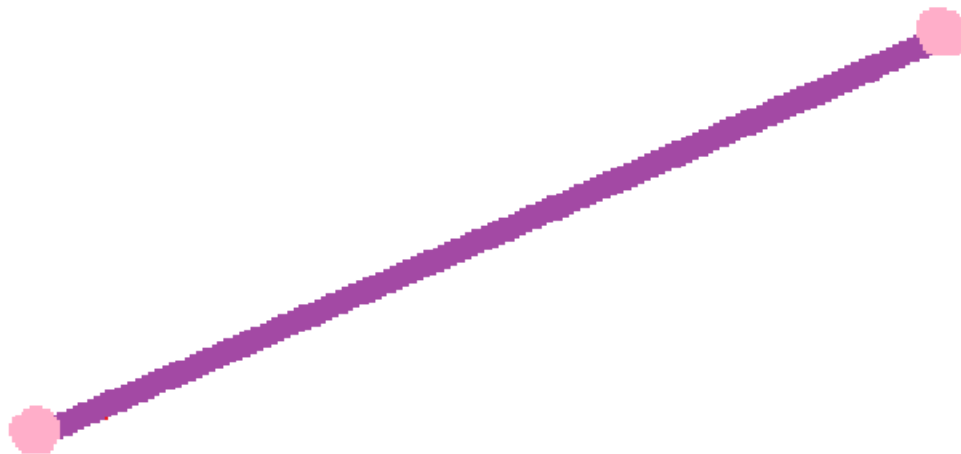
1. Firstly, before I even define what a vector is, I am talking about the variable type as well as the mathematical concept. As far as we are concerned, they are the same thing. We can derive the same information from them.
2. As we should already know from talking about variable types, **Vectors** contain multiples values. If the vector is a Vector2, it contains an X and a Y value. If the vector is a Vector3, it contains an X, Y and Z value.

When we write out a vector like the image below.

The image shows a vector written as a coordinate triplet: (X, Y, Z). The 'X' is red, the 'Y' is green, and the 'Z' is blue. The parentheses and commas are black. This represents a 3D vector.

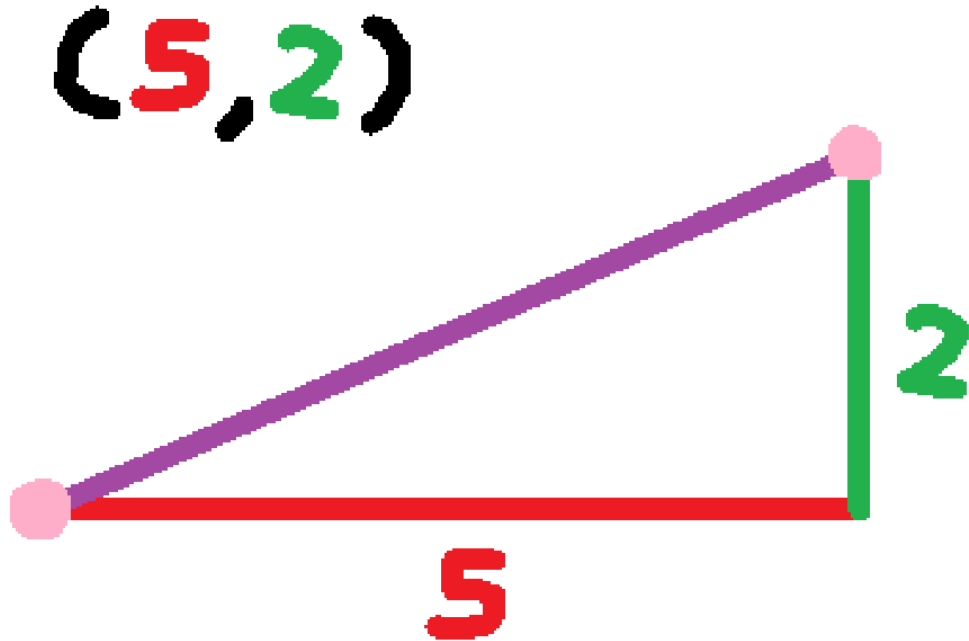
The X value is first, the Y value is second and the Z value is last. This is a Vector3 but if it were a Vector2 it would just be the X and Y values. I've also color coded each value to match the color of each axes in Unity.

Let's take a look at how we can think about vectors.



A vector can be thought up as a line that connects two points in space. The positions of these points don't really matter, however the distance between these two points are how we can gather information about the vector.

3. A much better way to represent vectors would be a triangle.

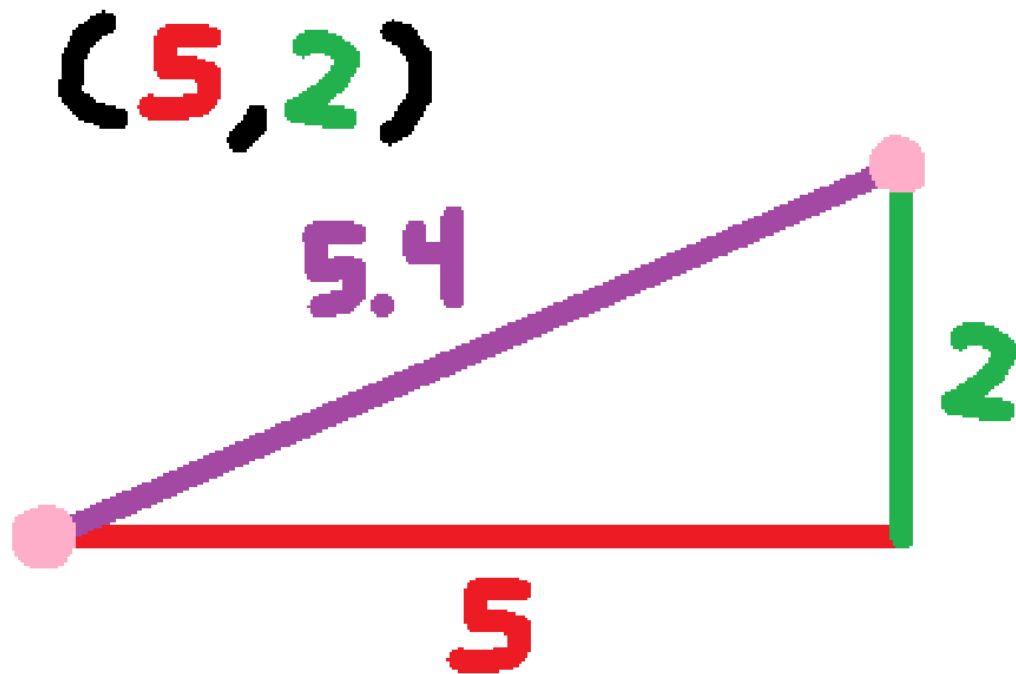


Here we have the **X component** of the vector represented in **red** with a value of **5**. We also have the **Y component** of the vector in **green** with a value of **2**. So now if we were to write out this vector, we can write it as **(5,2)**.

It is important to note that the **horizontal distance** between the two points is X value of the vector, while the **vertical distance** between the two points is the Y value.

4. One other thing we may need to know about our vector its length. This is otherwise called the **magnitude**. We can find the magnitude of the vector by using the **Pythagorean Theorem, $A^2 + B^2 = C^2$** . It is important to note that we can only do this since we have a right triangle.

So in our example above we would do the calculation, $5^2 + 2^2$, this gives us a value of 29. We would then take the square root of 29 to get the **5.4**.

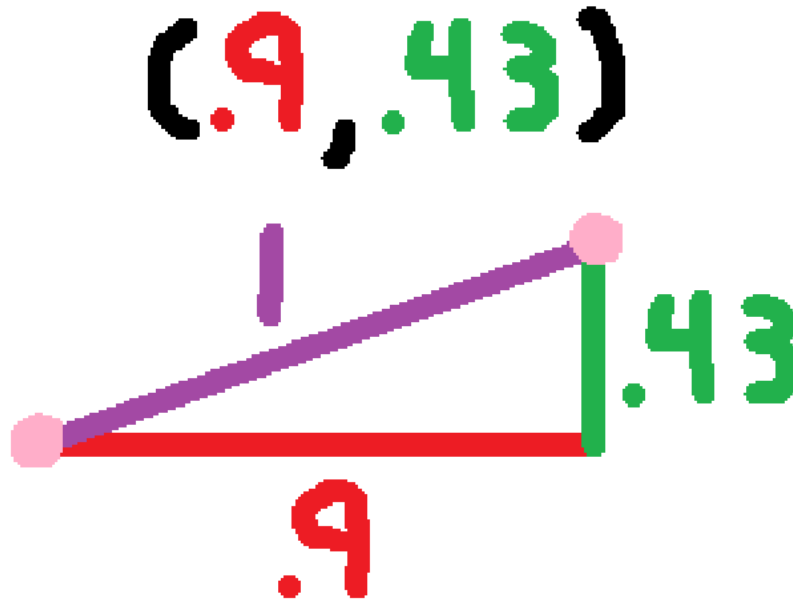


For right now, a good use for magnitude might be to determine the speed of a character.

Direction

1. When it comes to moving objects in games, we can represent directions as vectors. Typically, we want the vector that represents our direction to be something called a **Unit Vector** or a vector that has a **magnitude of one**.

Let's take a look at an example of a unit vector.



We can see that the magnitude of the vector is 1 as well as the x and y values do not exceed 1.

2. A good way to turn a vector into a unit vector is to **normalize** it. Normalizing a vector is great for us because it allows the vector we normalized to maintain its direction.

The process of normalizing a vector is fairly simple. We take vectors X and Y values and then divide them by magnitude. The results will then give us the X and Y components of the normalized vector.

3. We can use unit vectors in conjunction with some speed variable to determine how far an object would move along the X and Y axis. **Think of it this way, the .9 for the X value and the .43 for the Y value is the percentage to move along the X and Y axes.**

So if we had a speed variable of 25. The distance to move along the X axis would be $.9 * 25$, resulting in 22.5. The distance to move along the Y axis would be $.43 * 25$, resulting in 10.75.

4. An even better example would be to look at the code we wrote. Back in our **FixedUpdate()** function we wrote a line that reads:

```
rb.velocity = input_vec * speed * Time.fixedDeltaTime;
```

Lets bring emphasis to:

```
input_vec * speed
```

Our `input_vector` is acting as the direction vector for our game. It can have the values of `(-1, 0)` and `(1, 0)`, which are also unit vectors. By multiplying it by the speed value, 300, our player would move 300 units across the X axis.

Speed

1. Speed is just a value. It represents how fast an object is traveling. We do not know what units the speed is being measured in nor do we know direction of the speed.

Velocity

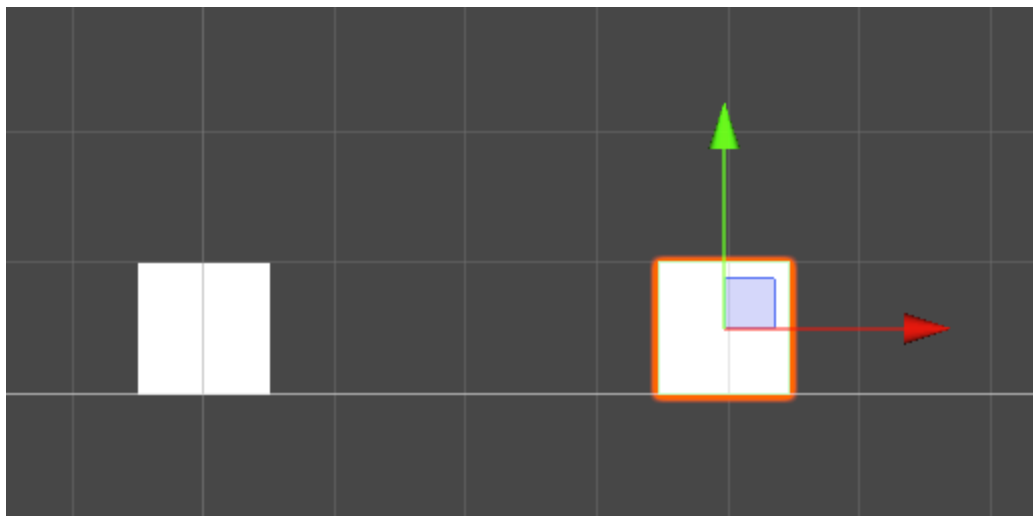
1. **Velocity is speed in a given direction.** It is the change of position from one location to another. Let's take a look at our movement code one last time.

```
rb.velocity = input_vec * speed * Time.fixedDeltaTime;
```

Our **direction** is the `input_vec` and our **speed** is `speed * Time.fixedDeltaTime`. By multiplying the two together we are figuring out the change in position along the X and Y axis.

Level Bounds

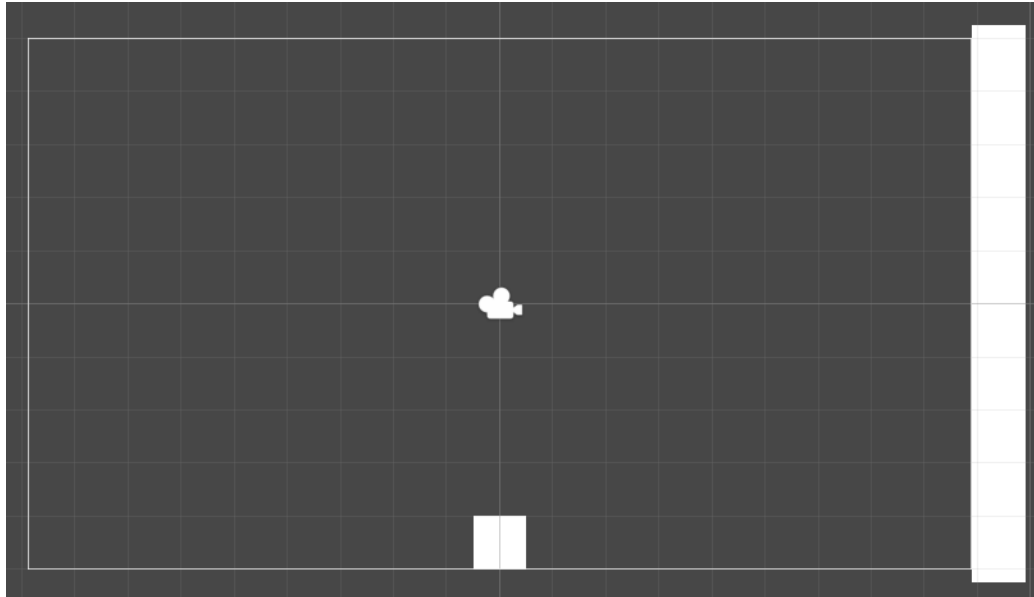
1. Next, we need some way of keeping our player inside the bounds of the camera. To do this we add another square sprite and name it **Wall**. Position it so that it is next to the player.



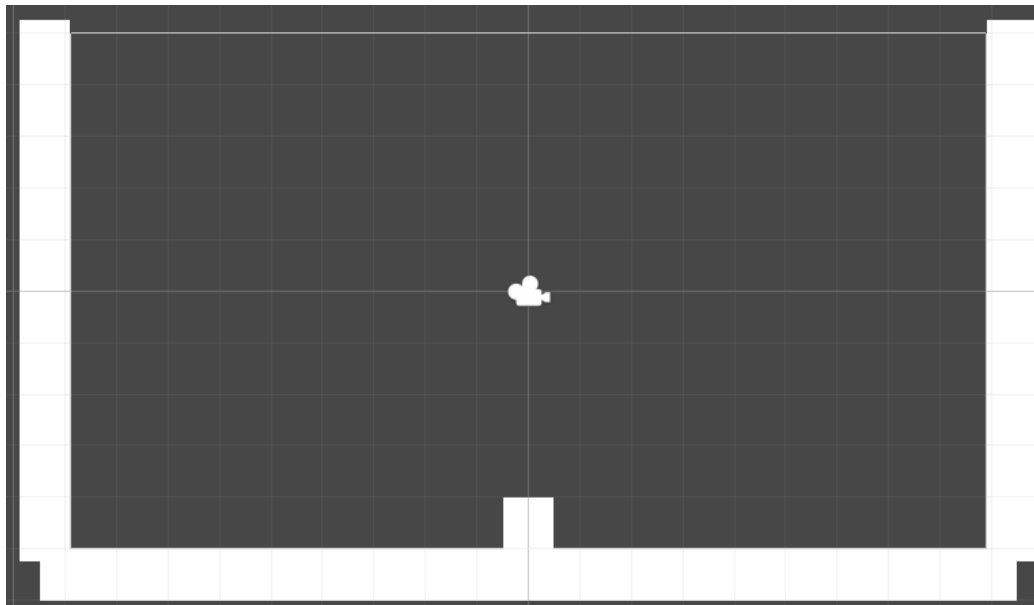
2. Give the wall a **Box Collider 2D** component.



3. Run the game and move the player into the wall. The player will now be able to collide with the Wall. Now we just need to position the wall on the outside of the camera and scale it so that it fully covers the side of the game window.



We can then repeat this process so that our game window will be surrounded by walls. However, we want to make sure to leave the top open, so that we can drop objects into our level.



Here are exact positions and scale each wall.

Right Wall:

Transform						
Position	X	9.4	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	10.5	Z	1

Left Wall:

Transform						
Position	X	-9.4	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	10.5	Z	1

Bottom Wall:

Transform						
Position	X	0	Y	-5.5	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	19	Y	1	Z	1

4. If we play the game now the player will stop at the edges of the level.

Shooting

1. Now that the player can only move within the bounds of the level, let's get to work on making the player shoot.
2. To do this let's first add a circle sprite and name it **Bullet**.

Set its scale to **0.5**.

Scale	X	0.5	Y	0.5	Z	0.5
-------	---	-----	---	-----	---	-----

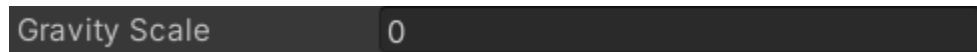
Give it a **Circle Collider 2D** component.

<input checked="" type="checkbox"/>	Circle Collider 2D
-------------------------------------	--------------------

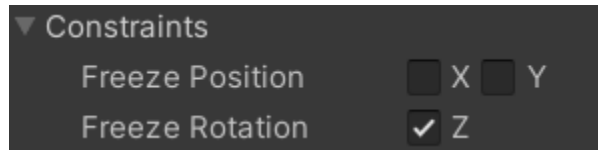
Give it a **Rigidbody** component.

<input checked="" type="checkbox"/>	Rigidbody 2D
-------------------------------------	--------------

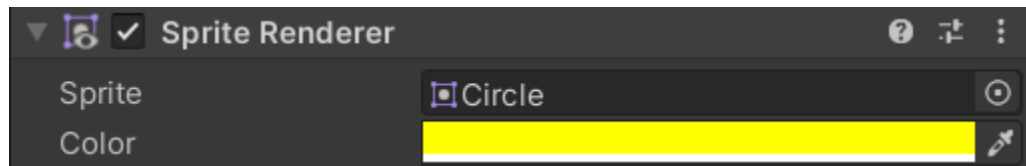
Disable gravity.



Freeze Rotation Constraints



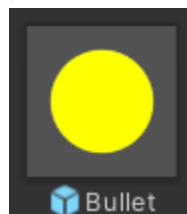
Change its color to Yellow.



3. Create a script and name it **Bullet**. Attach it to the bullet object.



4. We then need to turn the bullet into a prefab. Take the bullet object that is in the scene hierarchy and drag and drop it into the assets folder. This will then create the bullet prefab.



Once made, delete the bullet that was already in the scene.

5. Next let's open the player script and add the code that will allow the player to spawn bullets:

Lets add the variables:

```
//Shooting  
public GameObject bullet;
```

Bullet is the bullet prefab the that player will spawn when shooting.

In the Update() function lets write the code:

```
//Shooting
if (Input.GetKeyDown(KeyCode.Space))
{
    SpawnBullet();
}
```

We are just checking to see if the space bar is pressed. If so we call the SpawnBullet() function.

Since the SpawnBullet() function does not exist, under the FixedUpdate() lets create the function SpawnBullet.

```
void SpawnBullet()
{
    //Instance a new bullet.
    GameObject new_bullet = Instantiate(bullet);

    //Move the bullet to the player position.
    new_bullet.transform.position = transform.position;
}
```

We create a new variable that will hold a reference to an instantiated bullet object. We then move the bullet to the center of the player.

6. Let's then hop over to the **Bullet** script. Let's add some code that will allow the bullet to travel up towards the top of the screen.

Let's add the variable:

```
//Variables
private Rigidbody2D rb;
```

RB stores a reference to the rigidbody on the bullet.

Then in the Start() function we write the code:

```
//Get Components
rb = GetComponent<Rigidbody2D>();

//Set starting velocity to make the bullet shoot upwards.
rb.velocity = Vector3.up * 500 * Time.fixedDeltaTime;
```

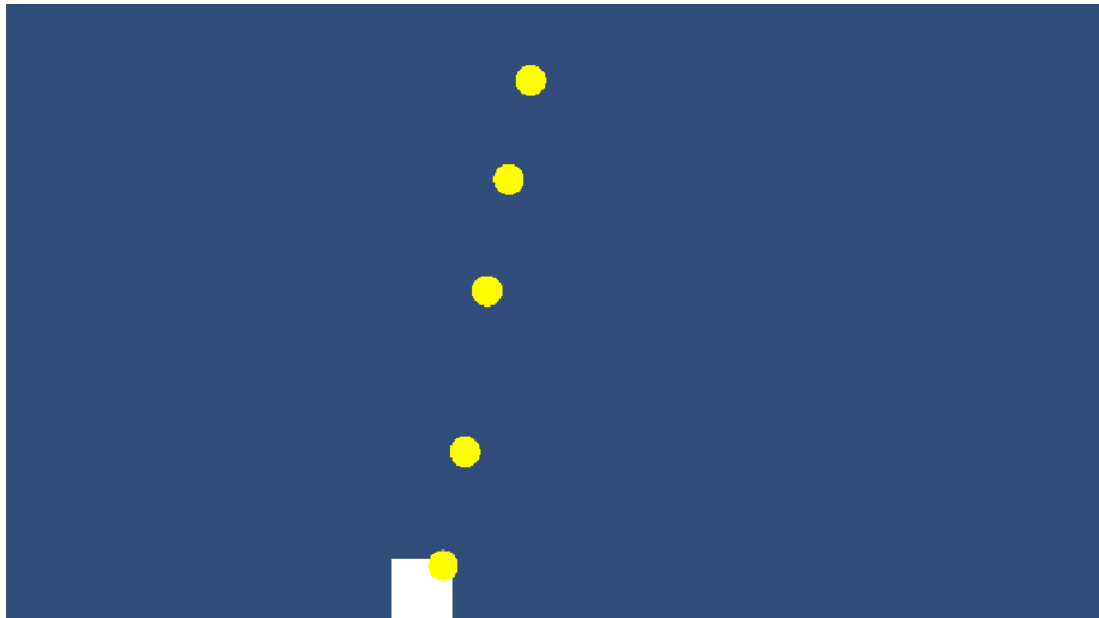
We get the rigidbody component and store it in the rb variable.

We then set the velocity of the Bullet to travel upwards.

7. Save the script and return to Unity. Set bullet prefab on the player script component.



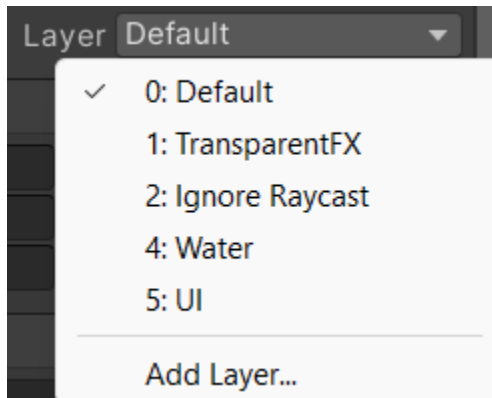
Once the game is running you should be able to spawn a bullet at the player's position. The bullets will travel upwards and off of the screen.



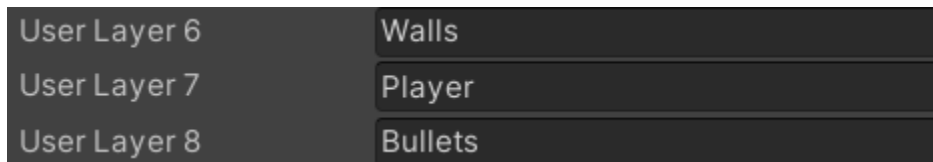
However, you should notice that the player is being pushed by the bullets once they are spawned. This is bad because we do not want the player to be able to collide with their own bullets. To fix this, we need to set up the collision layers for each element of our game.

Collision Layers

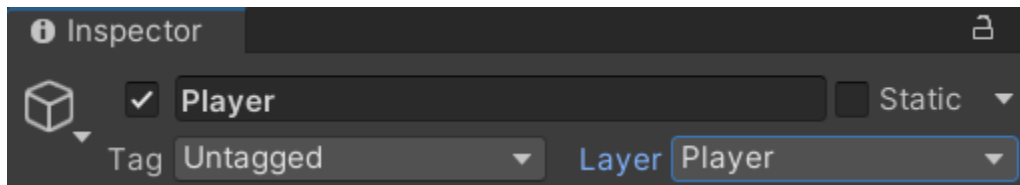
1. Select the player and then find the **Layer** drop down in the inspector. Click on add layer.



2. You should then be taken to a menu that will allow you to create custom layers. Add the layers **Walls**, **Player** and **Bullets**.



3. Go back to the player's inspector and then set the player's layer to the **Player** layer.



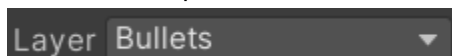
Then find the rigidbody and change the **Layer Overrides** to:



This will make it so the player can only collide with the Walls and will never be able to collide with the bullets.

4. We then need to put the other objects on the correct Layers.

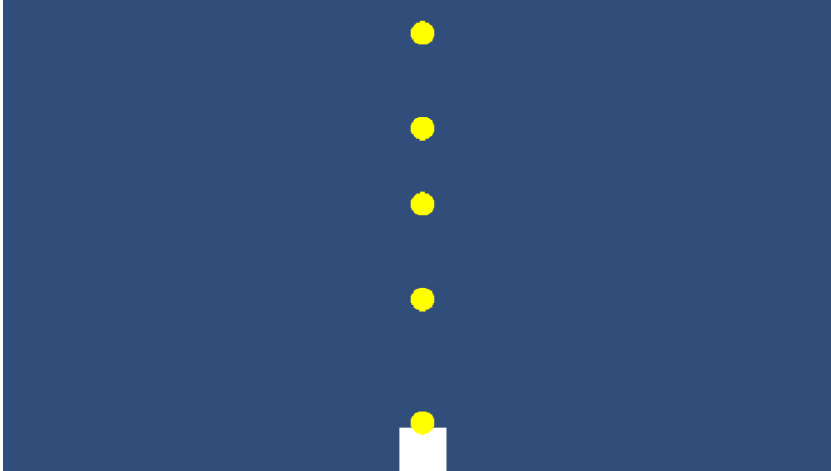
Put the Bullet prefab on the Bullets layer.



Select each of the walls and put them on the Walls layer.

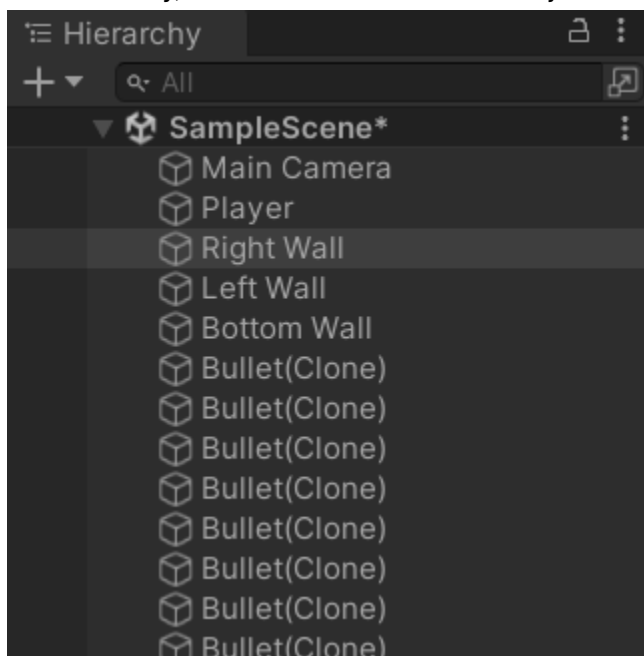


5. If we now run the game, the player should only be able to collide with walls and if a bullet is shot it will pass through the player without moving them.



Destroying Bullets

1. Something that we need to address is that whenever we fire a bullet it will still exist in our game even though it has left the screen. We know that they still exist because if we look at the Hierarchy, we can see that the bullet objects are still there.

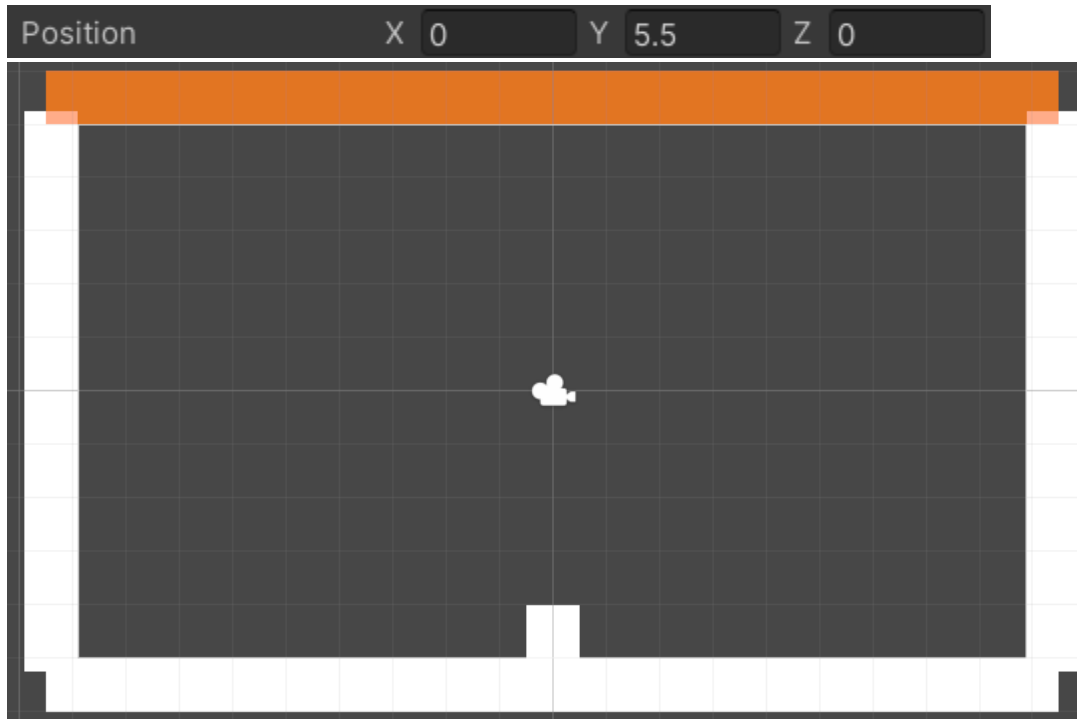


Even though we do not see them, the bullet objects are still traveling in our game world.

We want to make sure to destroy these objects since they are no longer useful to the player. If we didn't the amount of game objects would increase to a point in which it would start to slow down our game.

In order to make sure the bullets get destroyed, we will make a trigger that will destroy them.

2. Duplicate the bottom wall and rename it to **Bullet Destroyer Trigger**. Color it orange, make it slightly transparent and then move it to the top of the game window.

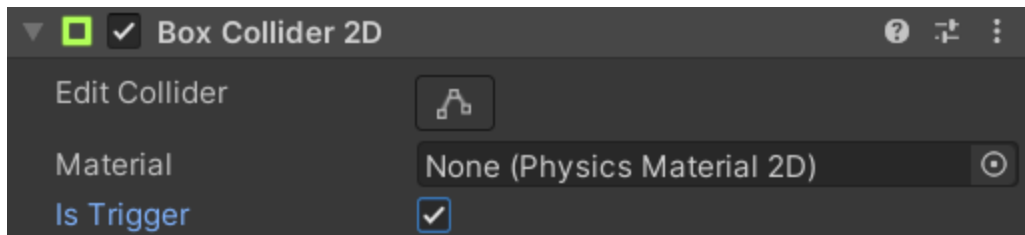


Whenever I create a trigger in a game, I like to color code them transparent orange. This signifies to me that the object is a trigger. This is also the color that was used the Hammer map editor of Valve games.

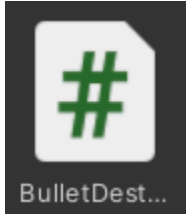
3. The object we just created is going to be a **Trigger**. A trigger in a game is another way for us to detect collisions. A trigger allows an object to pass through it as if nothing were there, however since it is a trigger it will realize that something has hit it and will try to do something as a response to this collision.

In our case this trigger is going to check if a bullet object collided with it. If so, it will respond by destroying the bullet that it collided with.

To make the object into a trigger find the **BoxCollider2D** and check the **Is Trigger** box.



- Next let's create a script for the Bullet Destroyer Trigger game object called **BulletDestroyer**. Once made **attach** it to the trigger.



- Open the script and remove the Start() and Update() function. Then just write the code:

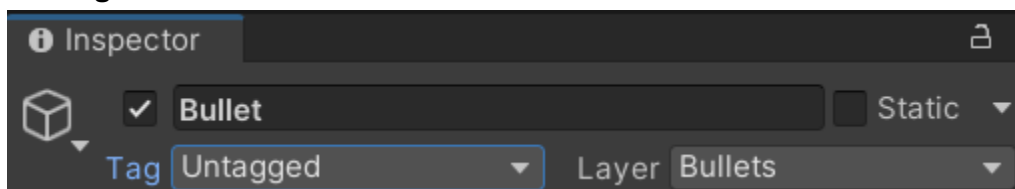
```
private void OnTriggerEnter2D(Collider2D collision)
{
    if(collision.gameObject.tag == "Bullet")
    {
        Destroy(collision.gameObject);
    }
}
```

The function **OnTriggerEnter2D** will only run if it detects a collision with a **Collider2D**. If the trigger has collided with something we check the objects tag to see if it is "Bullet". If so, we destroy the game object that the collider is attached to.

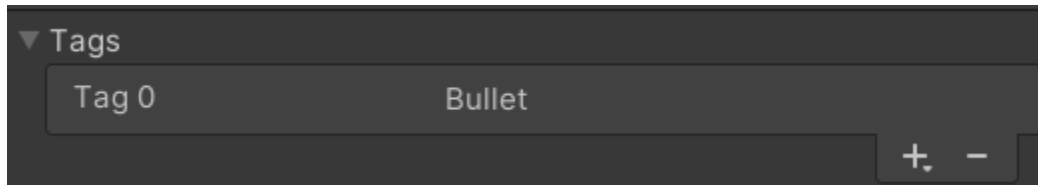
- To keep things simple, we are going to use **Tags** to see if a game object is a certain thing. A tag is a nice way of putting our game objects into groups.

Since we are checking to see if we are colliding with something that is tagged as a Bullet. We should add the bullet tag.

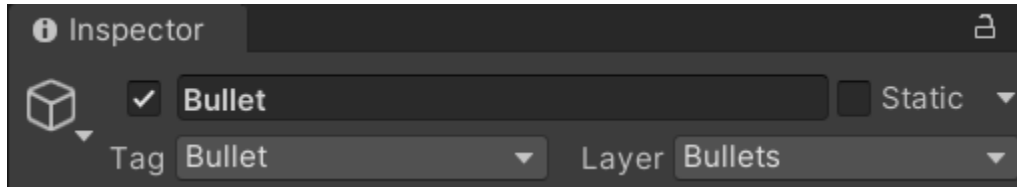
Go into our bullet prefab scene and find the **Tag** drop down. At the bottom of the list select **Add Tag**.



Once in the new menu, hit the plus sign to add a new tag. Name the new tag **Bullet**.



Head back to the bullet prefab and in its inspector set its tag to **Bullet**.



7. Save all the scripts and run the game and you should now see that the bullets collide with the trigger. You know they are destroyed since they disappear from the hierarchy.

Rate of Fire and Number of Bullets

1. The last thing we need to do is make it so the player cannot spam the space bar to fire bullets. As it is now, the faster the player mashes the space bar the more bullets they will fire.

To make the game more balanced we are going to reduce the rate of fire for shooting as well as limit the number of bullets that can be in the game at one time.

2. Let's first write code for controlling the player's rate of fire. Open the player script and add the variables:

```
private float rate_of_fire = 0.05f; //In seconds
private float shoot_timer = 0.0f;
```

Rate_Of_Fire will be the amount of time that needs to pass before the player can shoot again.

Shoot_Timer will be the timer that counts down for when the player can fire again.

Next in the Update() function, let's change the shooting if statement to:

```
//Shooting
if (Input.GetKeyDown(KeyCode.Space) && shoot_timer <= 0)
{
    shoot_timer = rate_of_fire;

    SpawnBullet();
}
```

We add another condition to our if statement. We want to make sure that the shoot timer has fully counted down before the player can shoot again.

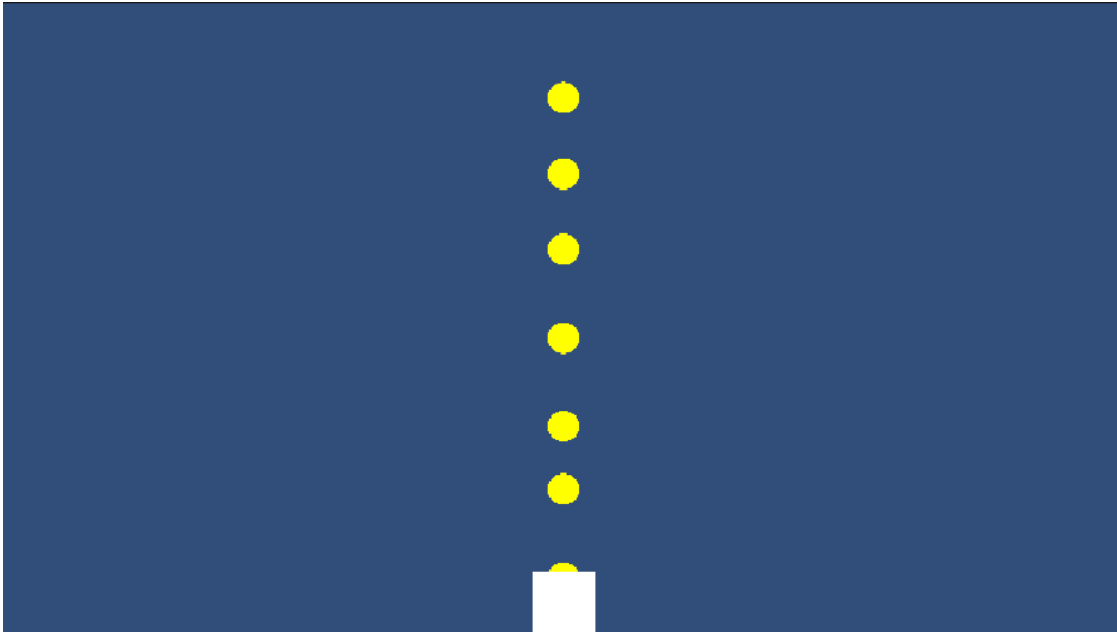
We also make sure to set the shoot_timer to the rate of fire, this way we add a delay between our shots.

Then below the if statement we add the code:

```
//Count down timer for shooting.
if(shoot_timer > 0)
{
    shoot_timer -= Time.deltaTime;
}
```

As with our other timers, we are just checking to see if the timer is above zero and if it is we subtract delta time to make the timer count down.

3. If we run the game, the player will not be able to shoot as fast. However, we are still able to shoot more than five bullets at a time.



4. To limit the amount of bullets, still in the Player script, lets add the variable:

```
[HideInInspector] public int bullet_count = 0;
```

Bullet_Count will keep track of how many bullets the player has currently fired.

Then in the Update() function we add a new condition to if statement.

```
//Shooting  
if (Input.GetKeyDown(KeyCode.Space) && shoot_timer <= 0 && bullet_count < 5)
```

So now we also need to check to see if the bullet count is less than five before the player will be able to shoot.

Lastly in the **SpawnBullet()** function we add the line:

```
//Increase the bullet count.  
bullet_count += 1;
```

We need to increase the bullet count whenever the player is able to successfully shoot a bullet. This way we can track how many bullets the player has shot so far.

5. If we save the script and run the game, the player can only shoot five bullets. We now need a way for the game to tell the player that when a bullet is destroyed the Player's bullet_count variable should decrease.

The best way we can do this is by using the **OnDestroy()** function. This function is one that is provided to each class when it extends from MonoBehaviour. We can simply give our bullet a

reference to the player and then when the bullet is destroyed for any reason it will subtracted one from the bullet count.

6. In the Bullet script let's add the variable:

```
[HideInInspector] public Player player;
```

Player will just hold a reference to the player object like stated above. We want to hide this object from the inspector since we do not need to set it at the start of the game.

Then under the Update() function let's write the **OnDestroy()** function and put the code for lowering the player's bullet count.

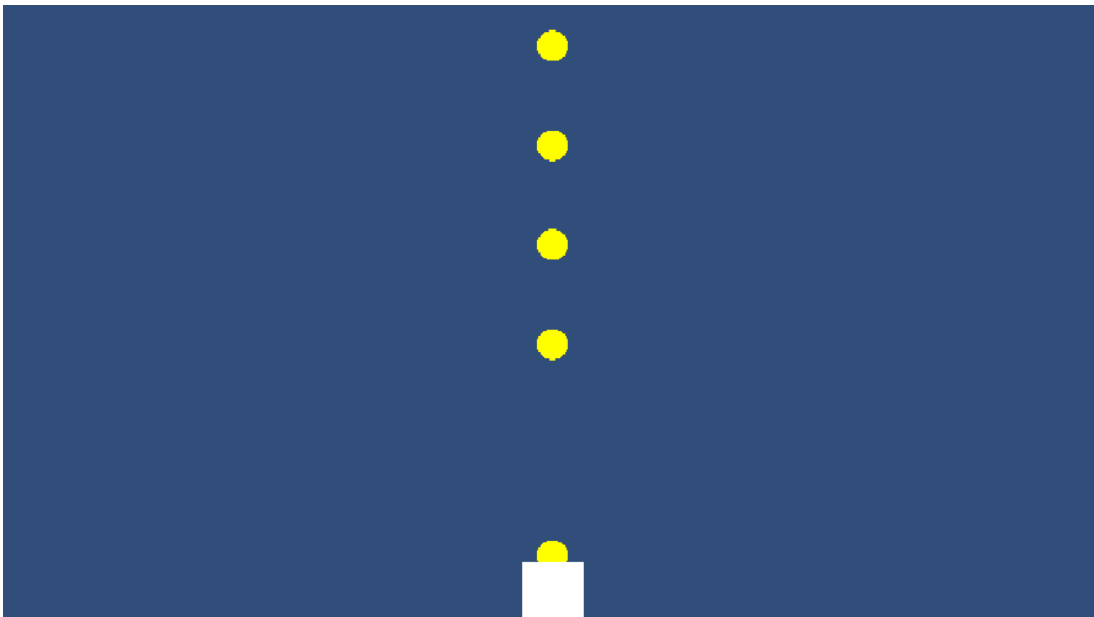
```
private void OnDestroy()  
{  
    player.bullet_count -= 1;  
}
```

Again, anytime the bullet is destroyed the player's bullet count variable will be decreased by one.

Lastly we just need to give the bullet a reference to the player. So in the **Player()** script, at the bottom of the **SpawnBullet()** function lets write the line of code:

```
//Give the bullet a reference to the player.  
new_bullet.GetComponent<Bullet>().player = this;
```

7. Save the script and run the game. We should now see that the player can only shoot five bullets at a time. If the bullets are destroyed, then the player can will be able to shoot again.



GitHub

1. Push the project to the GitHub repository.