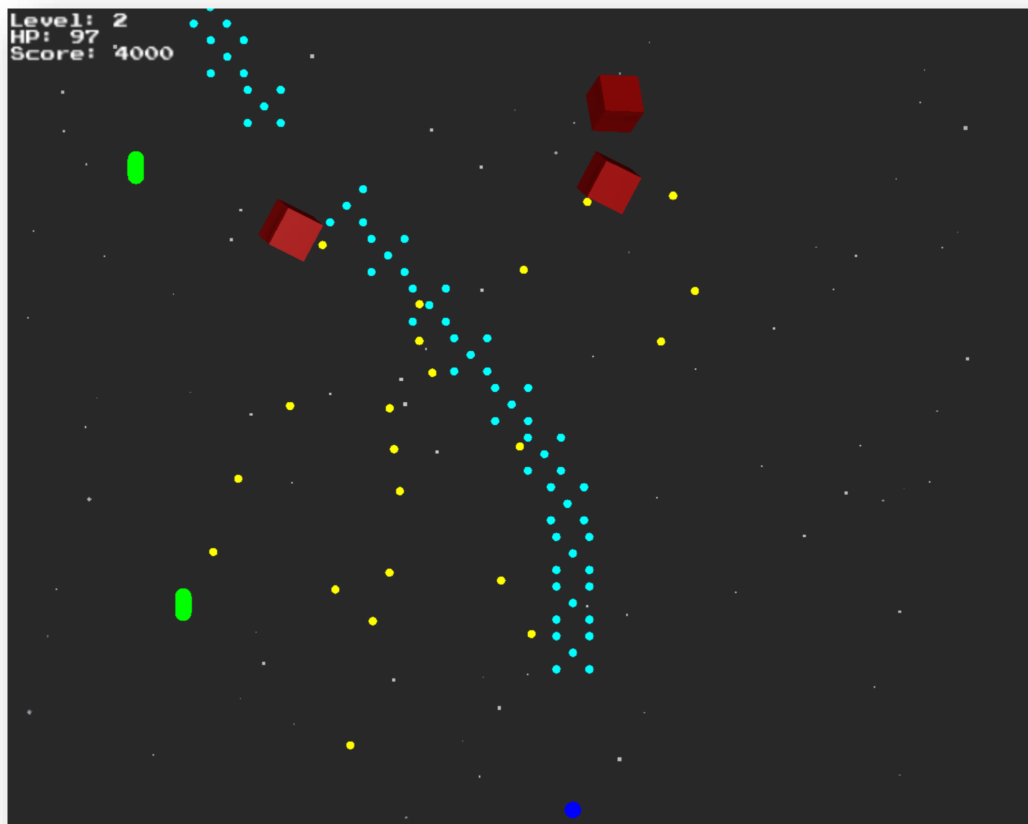
Silver Belt Ninja Guide
Activity 08: Shape Jam

Activity 8

Shape Jam

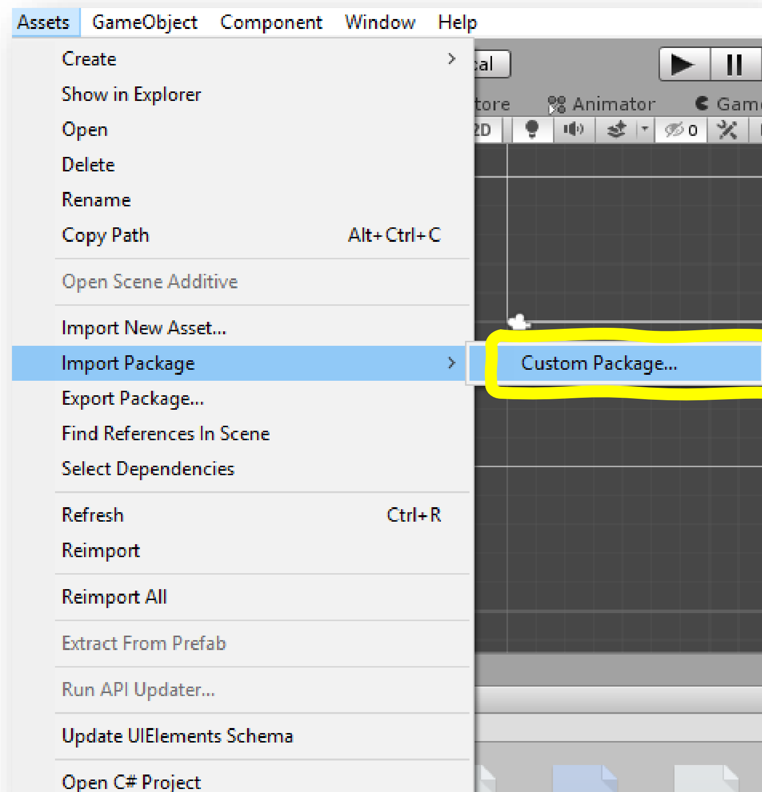
Your mission: A nearly complete game is given to you, but the most essential functions are missing - those that create new objects in the scene. You will add the code that creates these new objects and elements within the game.

In the end, the player will be creating projectiles. Enemies and powerups will be created on timers, and these enemies will constantly fire hazardous objects at the player. You will learn a few ways to use code to create these new objects using the `Instantiate()` function.

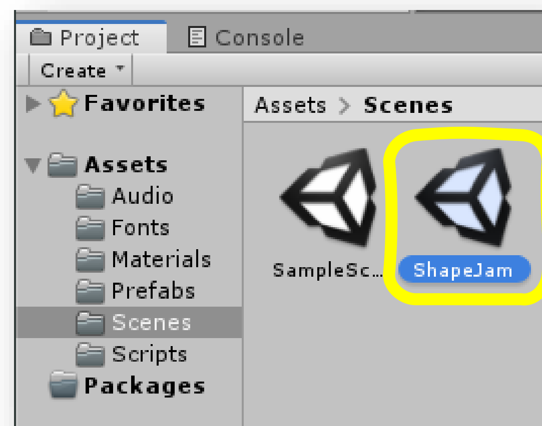


1 Start a new Unity Project and name it *YOUR INITIALS – Shape Jam*. Select **3D core**.

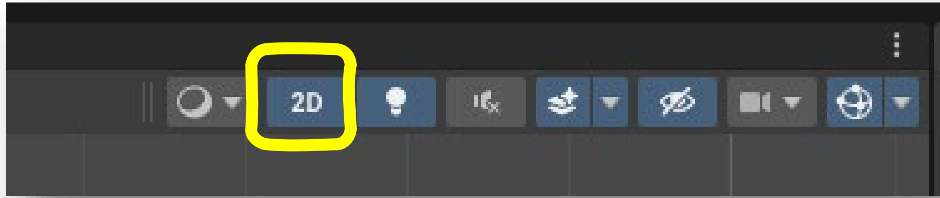
2 We've created a starter pack to give you a head start! To use it, import the **Activity 08 – Shape Jam.unitypackage** by going to **Assets > Import Package > Custom Package> All > Import**.



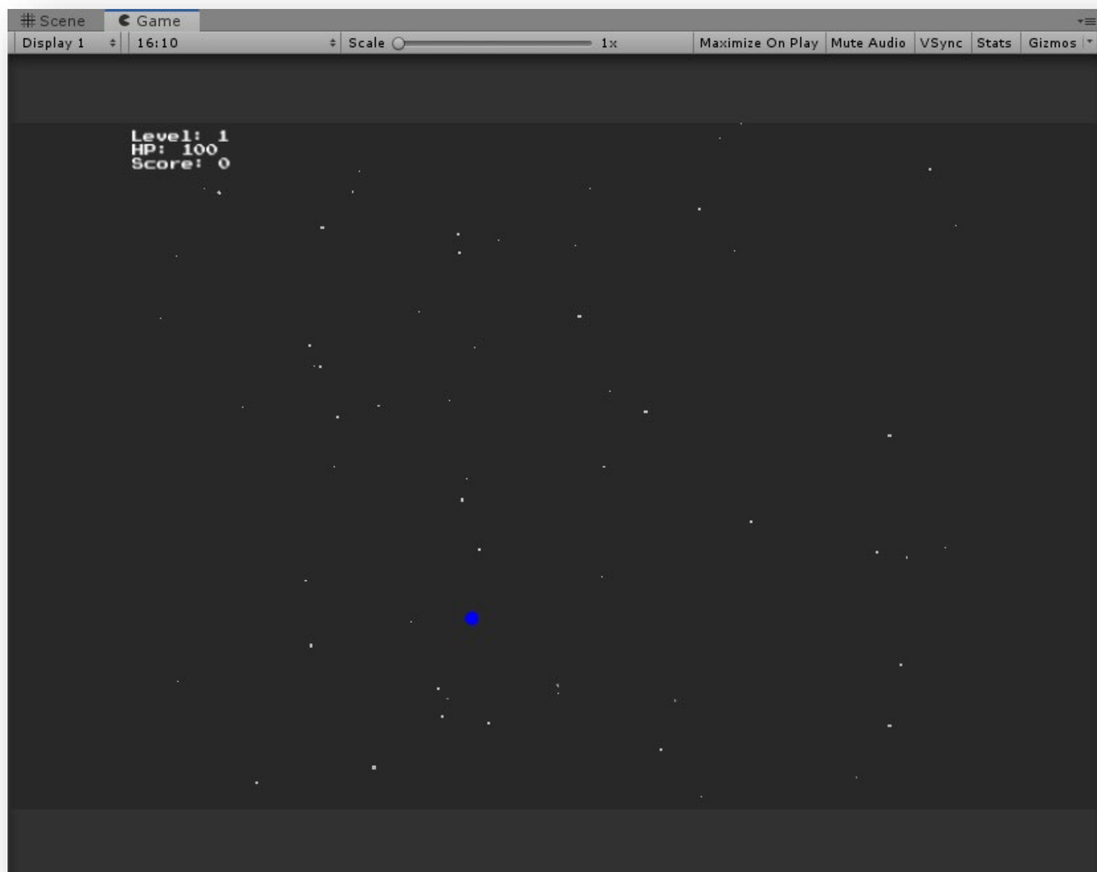
3 To open the starter package, double click on the "**ShapeJam**" Scene. You can find this in the **Project** tab under **Assets > Scenes.**



4 Click the **2D** button at the top of the scene to align the camera properly.

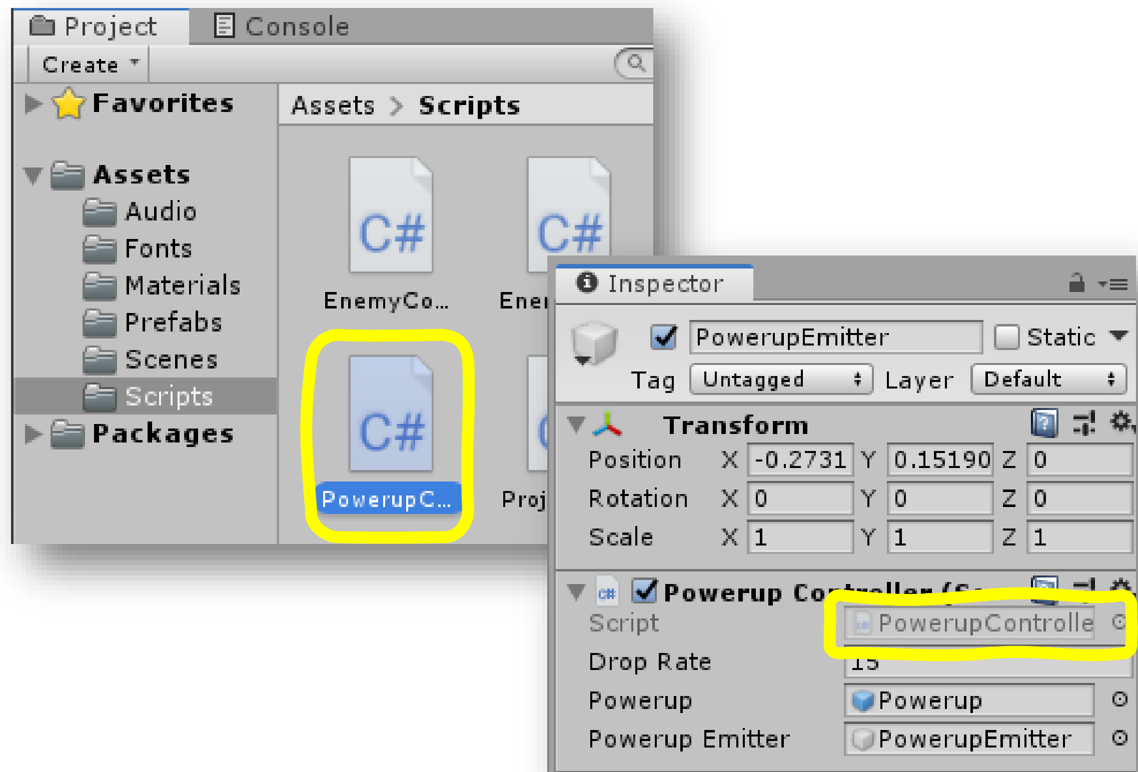


5 Before getting to work, **play** your game and see what happens!



What is already coded for you? You can control the blue dot, but you cannot fire your blaster. Right now, there are no enemies!

6 The first thing we want to do is add a powerup for the player to collect. We need to edit the **PowerupController** script. We can find this in the scripts folder or attached to the **PowerupEmitter** object in the **Hierarchy**.



Open the **PowerupController** script in Visual Studio by double clicking on it.

7 At the top of the class, there are different private and public variables that help the script run.

Some of those variables are initialized in the `Start` function. We need to write the code that will **instantiate,** or **create**, the power up objects for the scene.

Since we need to react to the game as it runs, we will write this code in the `Update` function.

8 Find the `Update` function and look at the code that is already present. It might seem like a lot, but there are a lot of lines of comments!

```
void Update()
{
    // Null check for gameOver state
    if (playerController.gameOver)
    {
        return;
    }

    // At the interval decided by dropRate,
    //    powerupPoosition is set to a random x from -6 to 6
    // above the game the powerupEmitter itself is moved to that position,
    //   and the nextDrop is set
    // a powerup is instantiated at the emitter's position and rotation

    if (Time.time > nextDrop)
    {
        float randomX = Random.Range(-6.0f, 6.0f);
        Vector3 powerupPosition = new Vector3(randomX, 6.0f, 0.0f);
        nextDrop = Time.time + dropRate;

        /****************************\
        |**** Add your code below ****|
```

Look at the first `if` statement. What do you think that statement is checking for? What happens if the statement evaluates to true? What happens if the statement evaluates to false? What does `return;` do?

This `if` statement checks to see if the game is over.
If the game is over, then `playerController.gameOver` will have the value of true and the code inside the if statement will be executed.

The code inside the statement is just `return;` which will tell Unity to stop running the function and to ignore the rest of the code in the function.

If the game is not over, then `playerController.gameOver` will have the value of false and the code inside the statement, `return;`, will not be run. Then Unity will continue executing the rest of the `Update` function.

9 Look at the second `if` statement. `Time.time` gets the number of seconds since the game started. The statement here is checking to see if it is time to drop a powerup to the player by seeing if the current time of `Time.time` is past the `nextDrop` time.

```
if (Time.time > nextDrop)
{
    float randomX = Random.Range(-6.0f, 6.0f);
    Vector3 powerupPosition = new Vector3(randomX, 6.0f, 0.0f);
    nextDrop = Time.time + dropRate;

    /*****************************\
    |**** Add your code below ****|
    \*****************************/


    /*****************************\
    |**** Add your code above ****|
    \*****************************/
}
```

10 If the condition is met, we first get a random value between -6 and 6 to make sure the powerup spawns inside the view of the camera.

```
float randomX = Random.Range(-6.0f, 6.0f);
```

11 We then use that `randomX` value to create a `new Vector3` to describe the powerup's position.

```
Vector3 powerupPosition = new Vector3(randomX, 6.0f, 0.0f);
```

12 By changing the value of `nextDrop` we can control when the next powerup spawns, so we set it equal to the current time and add the `dropRate`.

```
nextDrop = Time.time + dropRate;
```

13 You might have noticed that even though we are setting a position and deciding when the next powerup should spawn, no powerups are spawning! This is because we never **create** a powerup!

We create objects in Unity by using the `Instantiate` function. The `Instantiate` function takes three parameters.

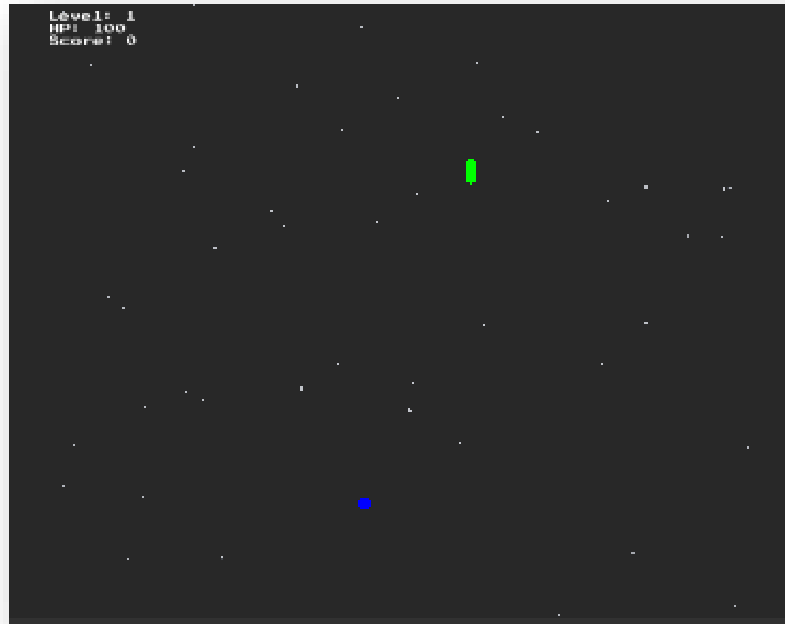
- The first parameter is the **game object** we want to create inside the scene.
- The second parameter is the **position** of where we want the object to be created.
- The third and final parameter is the **rotation** of what direction we want our object to be facing when it is created.

14 On a line in-between the two big comments telling you where to write your code, **instantiate** a new powerup object at the **PowerupEmitter's** location and direction by typing `Instantiate(powerup, powerupPosition, transform.rotation);`.

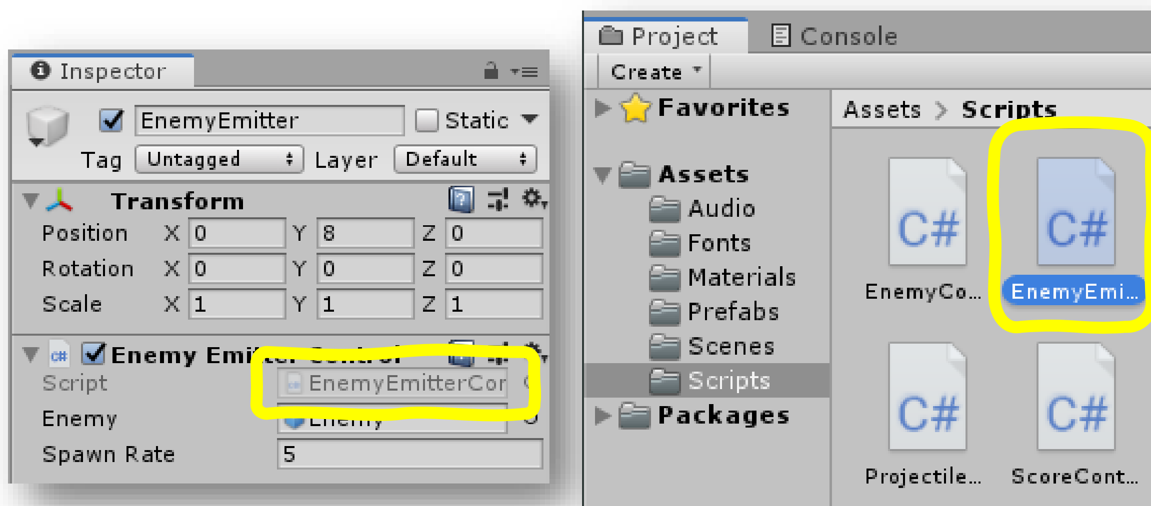
```
/******************************\
|**** Add your code below ****|
\******************************/
Instantiate(powerup, powerupPosition, transform.rotation);

/******************************\
|**** Add your code above ****|
\******************************/
```

Remember that Unity understands that the `transform` will refer to the **PowerupEmitter** because the script is attached to it.

15 **Play** your game and wait to see what happens. After 15 seconds, a green powerup will fall from above the screen! What happens when you try to catch it?



16 Now that we have powerups working with the `Instantiate` function, we can try something a little more complex when we create enemies. We need to edit the **EnemyEmitterController** script. We can find this in the scripts folder or attached to the **EnemyEmitter** object in the **Hierarchy**.



17 Open the **EnemyEmitterController** script in Visual Studio by double clicking on it.

18 At the top of the `class`, there are different `private` and `public` **variables** that help the **script** run. Some of those **variables** are initialized in the `Start` function.

We need to write the code that will "instantiate" (create) the enemy objects in the scene.

Since we need to react to the game as it runs, we will write this code in the `Update` function.

19 Find the `Update` function and look at the code that is already present. There's less code than the last `Update` function that we looked at!

```
void Update()
{
    // Null check
    if (playerController.gameOver)
    {
        return;
    }
    // Usual time delay code
    if (Time.time > nextSpawn)
    {
        nextSpawn = Time.time + spawnRate;
        // Spawns a number of Enemy objects equivalent to the currentLevel
        // These spawn at a random x-value above the stage

        /******************************\
        |**** Add your code below ****|
        \******************************/


        /******************************\
        |**** Add your code above ****|
        \******************************/
    }
}
```

20 Look at the first `if` statement. Have we seen this code before? Yes! We saw it in the **PowerupController's** `Update` function. Recall that it checks to see if the game is over.

If it is over, it ignores the code below the `return;` statement.

If the game is not over, it will run all the code in the function.

21 Look at the next `if` statement. This is very similar to the second `if` statement in the **PowerupController's** `Update` function.

We want to check to see if it is time to spawn an enemy.

If it is, then we want to execute our spawn code that lives inside of the if statement.

```
if (Time.time > nextSpawn)
{
    nextSpawn = Time.time + spawnRate;
    // Spawns a number of Enemy objects equivalent to the currentLevel
    // These spawn at a random x-value above the stage

    /*****************************\
    |**** Add your code below ****|
```

22 There is only one line of provided code in this if statement.

This line of code sets the value of `nextSpawn` to be equal to the current time plus the `spawnRate` of enemies.

This controls when the next enemy will spawn!

23 To make sure the player has a challenge, we want to spawn enemies based on the current level the player is on.

For example, the game will start spawning one enemy at a time.

Once the player picks up a green powerup and gets to level 2, we want the game to spawn two enemies at the same time.

We can do this by using a `for` loop that starts at `0` and loops up to the current level.

Type `for (int i = 0; i < playerController.currentLevel; i++) {   }` making sure to leave an empty space between the brackets.

```
/*****************************\
|**** Add your code below ****|
\*****************************/

for (int i = 0; i < playerController.currentLevel; i++)
{

}

/*****************************\
|**** Add your code above ****|
\*****************************/
```

24 Inside the loop, we need to select a random x position for the enemy and then use that to create a position vector.

Have you seen code like that already? Yes, in the **PowerupController**! We can use that exact same code with a few name changes.

25 Inside the loop, create a `float` variable named `randomX` and set it equal to a **random** number between `-6` and `6` by typing `float randomX = Random.Range(-6.0f, 6.0f);`.

We are putting an “f” at the end of the two numbers to tell Unity that we want our random values to not be just whole numbers or integers like -2 or 4 and instead numbers that are **floats** like -2.41571623 and 4.000245. This will give a lot more variation to where the enemies spawn.

```
for (int i = 0; i < playerController.currentLevel; i++)
{
    float randomX = Random.Range(-6.0f, 6.0f);
}
```

26 Next, we want to create a new vector called `enemyPosition`.

We want the x position to be the random number we just created, the y position to be 6 so it starts above the screen, and the z position to be 0 because we want our game to behave like a 2D game.

Type `Vector3 enemyPosition = new Vector3(randomX, 6, 0);` after the `float randomX` line.

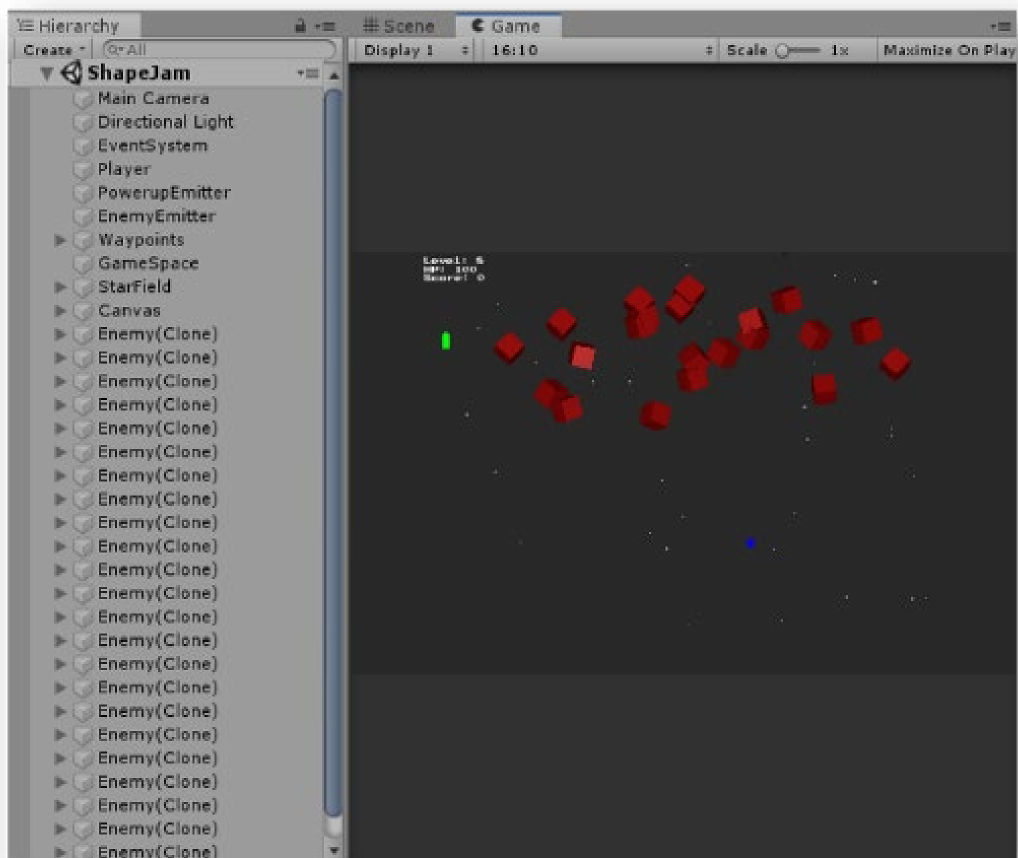
```
for (int i = 0; i < playerController.currentLevel; i++)
{
    float randomX = Random.Range(-6.0f, 6.0f);
    Vector3 enemyPosition = new Vector3(randomX, 6, 0);
}
```

27 Now that we have the enemy emitter in the new position, we can create our enemy. This is just like how we created the powerup.

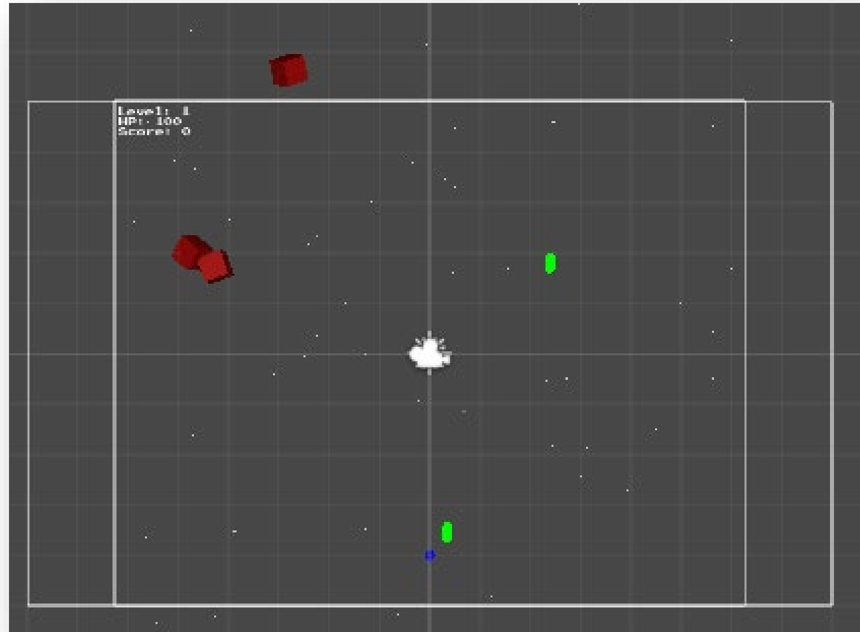
We want to `Instantiate` the **Enemy** game object and set its **position** and **rotation** based on the enemy emitter.

Type `Instantiate(Enemy, enemyPosition, transform.rotation);` after the `enemyPosition` line.

28 **Save** your code and **play** your game. Pay attention to the **current level** and **how many enemies** are spawned. Every time you get a green powerup, the enemy emitter will start to create more enemies based on the player's level! Without a way to fight back, things can get crazy really fast! What happens if the player collides with an enemy?



29 While the game is running, click on the # Scene tab. You'll be able to see outside of the player's view and watch the enemies and powerups spawn!



30 During your playtest, you should have noticed that nothing happens when the player touches an enemy.

That means that there is no challenge because the player can never lose! We can fix this by editing the **HazardFiring** script.

We can find this in the **scripts** folder. Double click it to open it in Visual Studio.

31 At the top of the class, there are different private and public variables that help the script run.

Some of those variables are initialized in the Start function. We need to write the code that will instantiate or create enemy hazards in the scene.

Since we need to react to the game as it runs, we will write this code in the Update function.

32 Find the Update function and look at the code that is already present. It might seem like a lot, but there are a lot of lines of comments!

```
void Update()
{
    // Null check for GameOver
    if (playerController.gameOver)
    {
        return;
    }

    // Target is set to the the Player's position,
    // and a vector is made from the current position to the target
    // At the rate decided by nextFire, and the rotation dictated by the hazardSpawn,
    // a clone is instantiated
    // which will start with normalized velocity in the direction of the target (player)
    target = player.transform.position;
    HazardMoveDirection = target - transform.position;

    if (Time.time > nextFire)
    {
        nextFire = Time.time + fireRate;

        /*****************************\
        |**** Add your code below ****|
        \*****************************/


        /*****************************\
        |**** Add your code above ****|
        \*****************************/
    }
}
```

33 Look at the first if statement. This is the third time we have seen this exact code! Before moving on, explain what it does to your Code Sensei.

```
if (playerController.gameOver)
{
    return;
}
```

34 There are two lines of code between the two if statements. The first statement sets up the target position for the hazard by finding out the current position of the player.

```
target = player.transform.position;
```

35 The second statement calculates the direction that the hazard needs to move in by taking the difference of the target and the enemy's position.

```
HazardMoveDirection = target - transform.position;
```

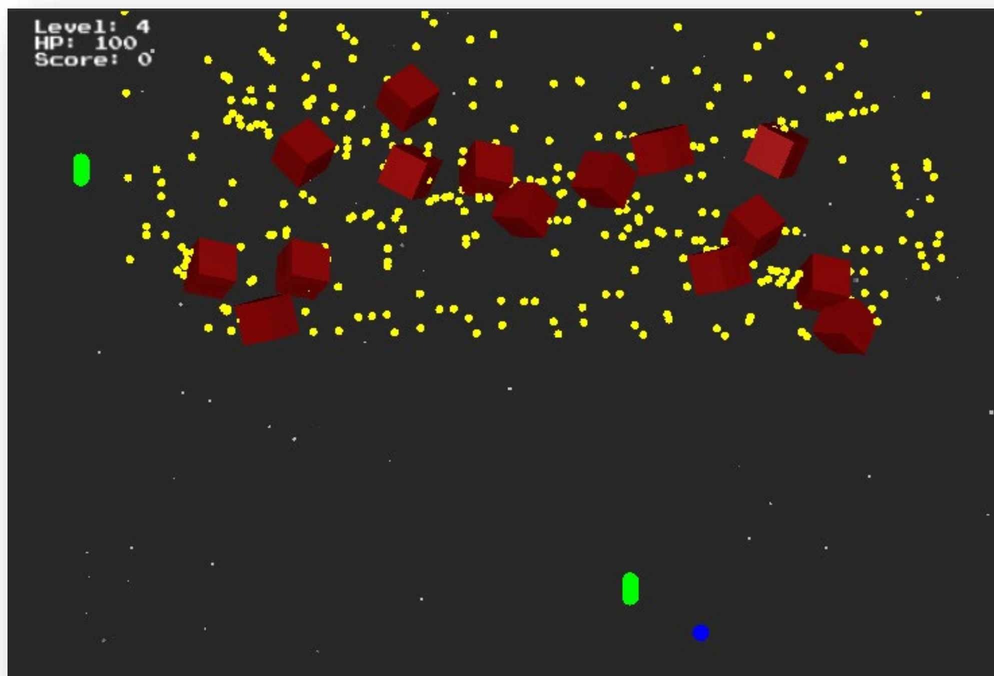
36 Look at the second if statement. This is very similar to the if statements in the `PowerupController` and `EnemyEmitterController` Update functions that we looked at before. Before moving on, explain what it does to your Code Sensei.

```
if (Time.time > nextFire)
{
    nextFire = Time.time + fireRate;

    /*******************************\
    |**** Add your code below ****|
    \*******************************/


    /*******************************\
    |**** Add your code above ****|
    \*******************************/
}
```

37 The first thing we want to do is create a clone of the hazard object. We do this by using the Instantiate function and providing it the original game object, the position where we want our new cloned object, and the direction we want our object to be facing. Type `GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);` to create a clone of the hazard.

```
if (Time.time > nextFire)
{
    nextFire = Time.time + fireRate;

    /******************************\
    |**** Add your code below ****|
    \******************************/
    GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);

    /******************************\
    |**** Add your code above ****|
    \******************************/
}
```

38 Play your game! Why are the hazards frozen in the air? We instantiated them, but did we ever tell them what to do after they were created?



39 We need to tell the hazards to move! We can do that by getting our new hazardClone's rigid body and giving it a velocity.

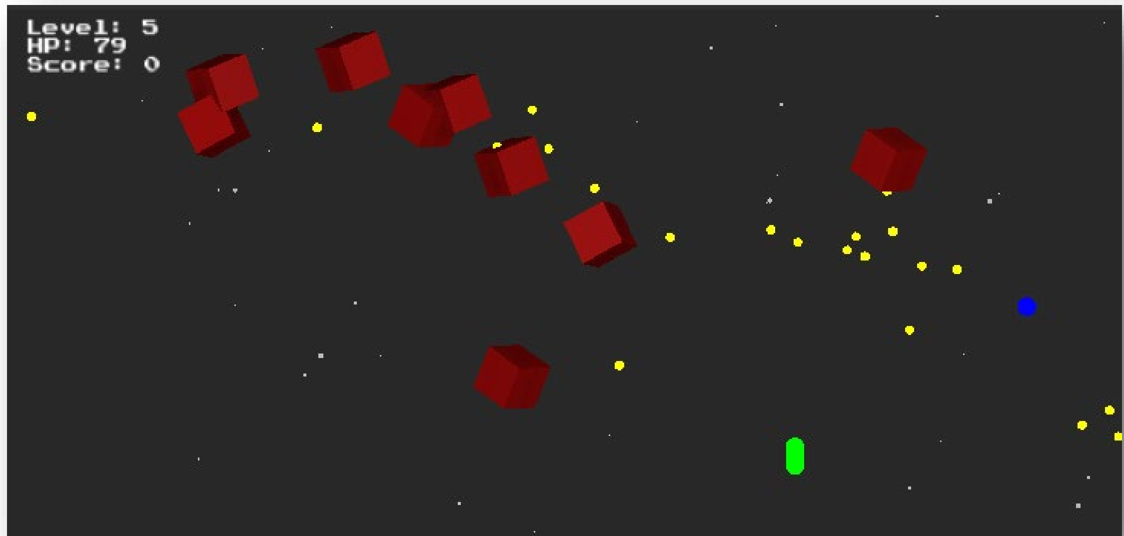
First, we need get the Rigidbody component that is attached to the hazardClone by using Unity's `GetComponent<Rigidbody>()` function on our hazardClone and storing it in a variable named hazardRigidbody.

```
/*****************************\
|**** Add your code below ****|
\*****************************/
GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);

Rigidbody hazardrigidBody = hazardClone.GetComponent<Rigidbody>();

/*****************************\
|**** Add your code above ****|
\*****************************/
```

40 Next, we need to apply a velocity to that rigid body. On the next line, type `hazardrigidBody.velocity = HazardMoveDirection;` to set the hazard's velocity to the vector that represents the direction between the enemy and the player that we talked about in the previous step.

```
/*****************************\
|**** Add your code below ****|
\*****************************/
GameObject hazardClone = Instantiate(hazard, transform.position, tran

Rigidbody hazardrigidBody = hazardClone.GetComponent<Rigidbody>();
hazardrigidBody.velocity = HazardMoveDirection;

/*****************************\
|**** Add your code above ****|
\*****************************/
```

41 Play your game! The hazards are all moving now! What can you tell about the behavior of the hazards? Where do they start and what's their destination? Do they all travel at the same speed? What is the difference between a hazard that starts far away from the player and a hazard that starts really close to the player?

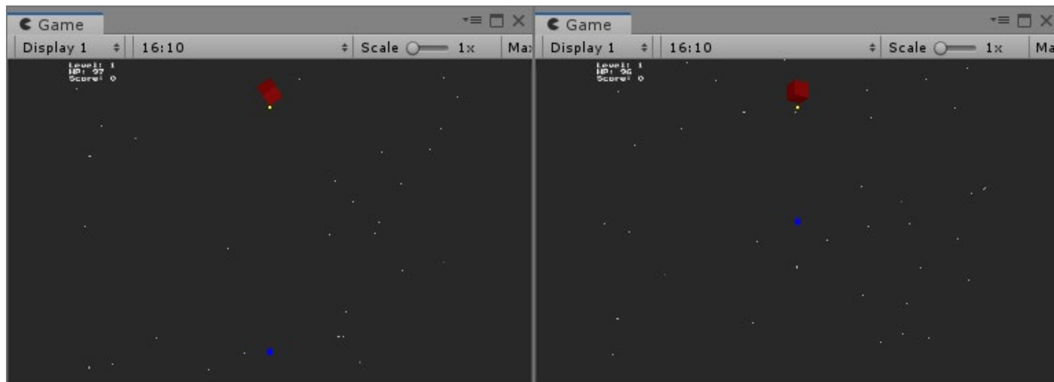


42 Why are the hazards traveling at different speeds based on their distance from the player?

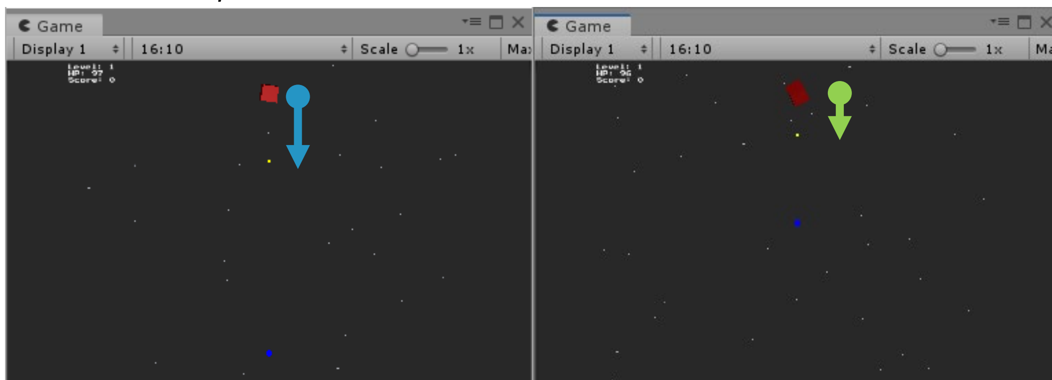
We are setting the velocity equal to a vector that describes the distance and direction between the enemy and the player.

The greater the distance the greater the velocity.

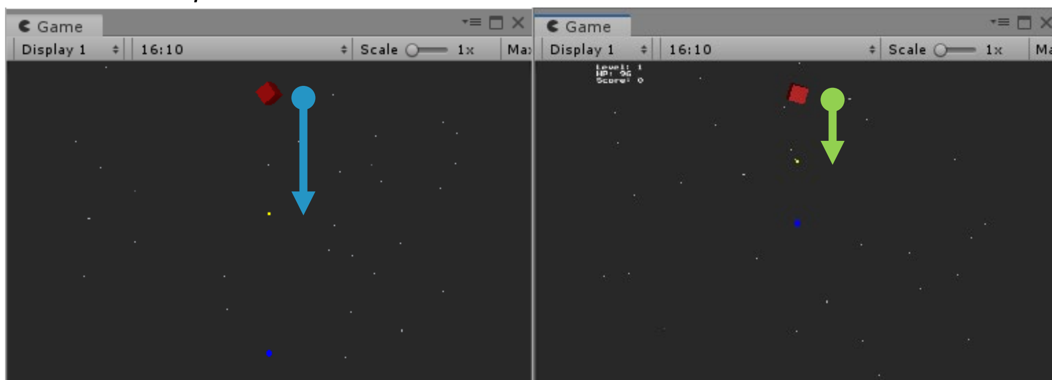
0 Seconds, frame 0



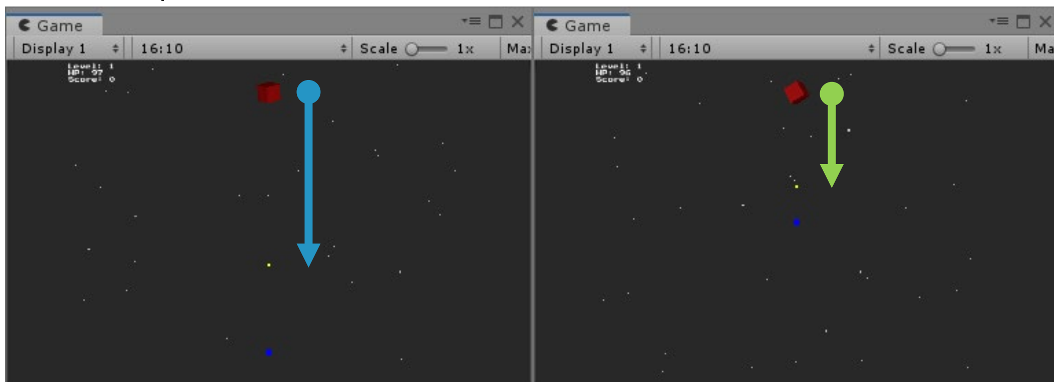
0.3 Seconds, frame 10



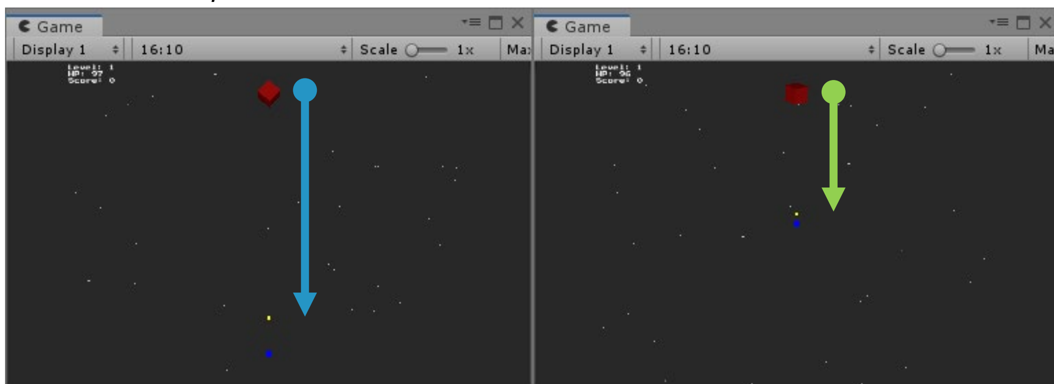
.6 seconds, frame 20



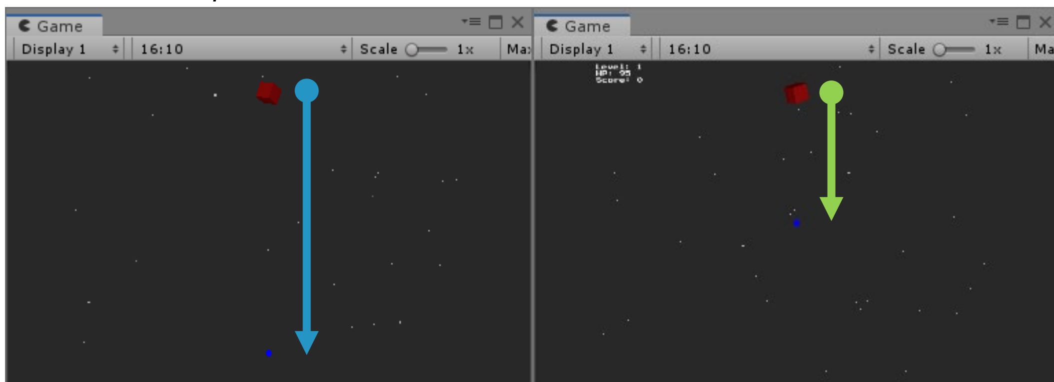
1 second, frame 30



1.3 seconds, frame 40



1.6 seconds, frame 50



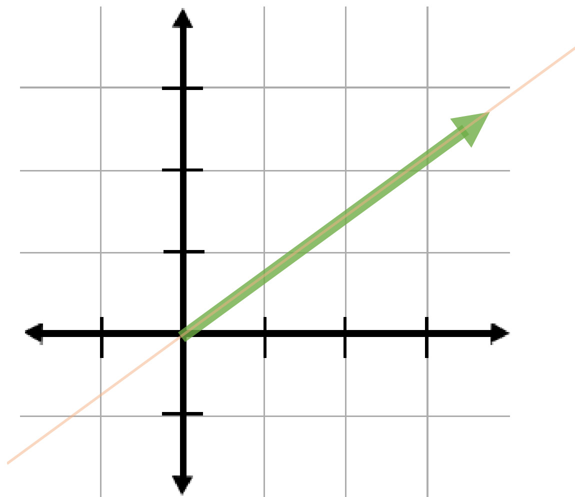
43 We want to make sure that our hazards travel towards the player, but we do not want the speed to change based on how close or far the player is.

We can accomplish this by normalizing our HazardMoveDirection variable to make sure we only use the direction in our calculations and not the distance.

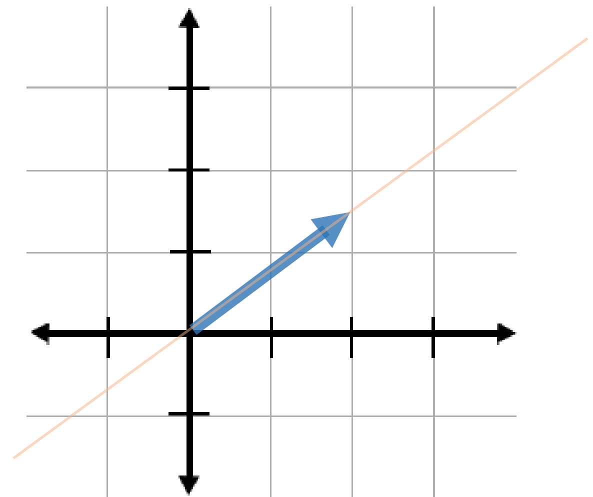
44 Every vector has **magnitude** (or size) and a **direction**. When we normalize a vector, we are keeping the direction, but we are shrinking it or growing it to make its magnitude exactly equal to 1.

For this first example, we have a green vector that is really big! When we normalize it, we shrink the size down to exactly 1 but keep it pointing in the same direction along the orange line.

Original

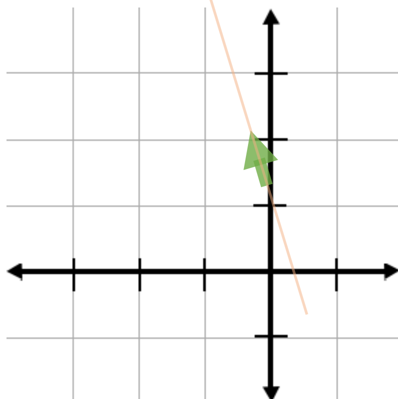


Normalized

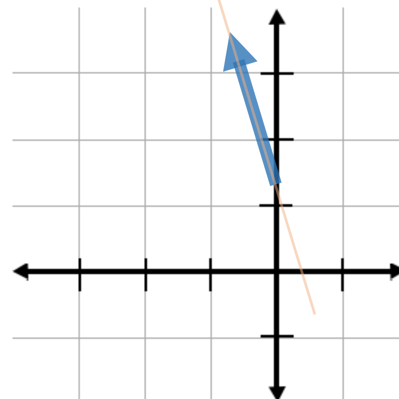


For this next example, we have a green vector that is really small! When we normalize it, we increase the size up to exactly 1 but keep it pointing in the same direction along the orange line.

Original



Normalized



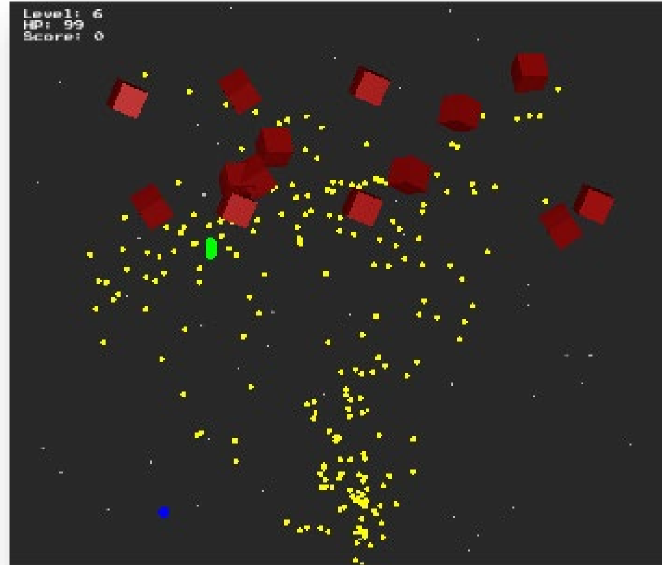
The blue arrow in the first example and the blue arrow in the second example are facing different directions, but they are both the exact same length! They each have a magnitude of 1.

Before moving on, explain what happens when we normalize a vector to your **Code Sensei** and then to one other **Ninja** in your Dojo!

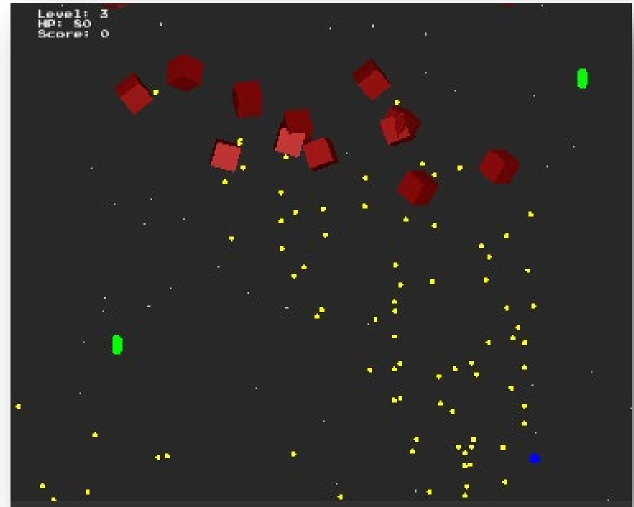
45 The math it takes to normalize a vector might seem complicated, but Unity already knows how to do it! Change the last line we typed from `hazardrigidBody.velocity = HazardMoveDirection;` to `hazardrigidBody.velocity = HazardMoveDirection.normalized;` to have Unity normalize the `HazardMoveDirection` before setting the hazard's velocity.

```
/******************************\
|**** Add your code below ****|
\******************************/
GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);

Rigidbody hazardrigidBody = hazardClone.GetComponent<Rigidbody>();
hazardrigidBody.velocity = HazardMoveDirection.normalized;

/******************************\
|**** Add your code above ****|
\******************************/
```

46 **Save** your script and **play** your game! What has changed since before you normalized the direction vector? All the hazards are now moving at the same exact speed no matter where the player is, but now they are moving very slowly!

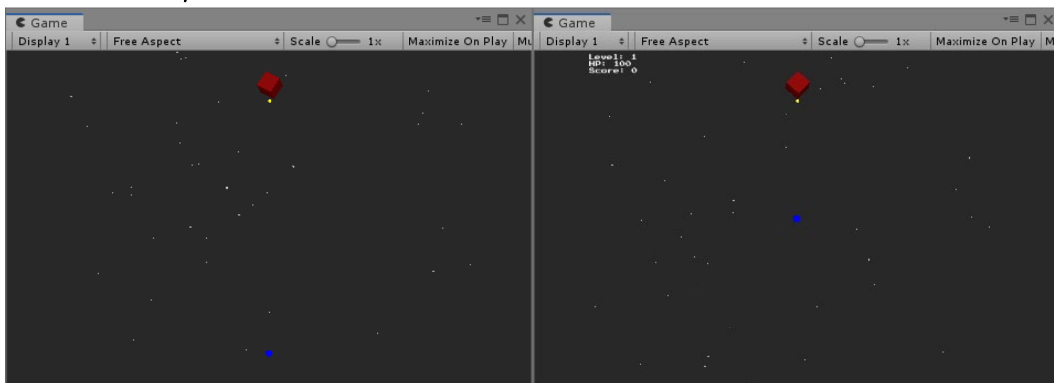


47 The **HazardFiring** script has a `public` variable named `speed`. Change the last line of code we typed to include the `speed` variable by changing it from `hazardrigidBody.velocity = HazardMoveDirection.normalized;` to `hazardrigidBody.velocity = HazardMoveDirection.normalized * speed;` to multiply the normalized direction vector by the speed variable. **Play** your game and see that the hazards are traveling faster!

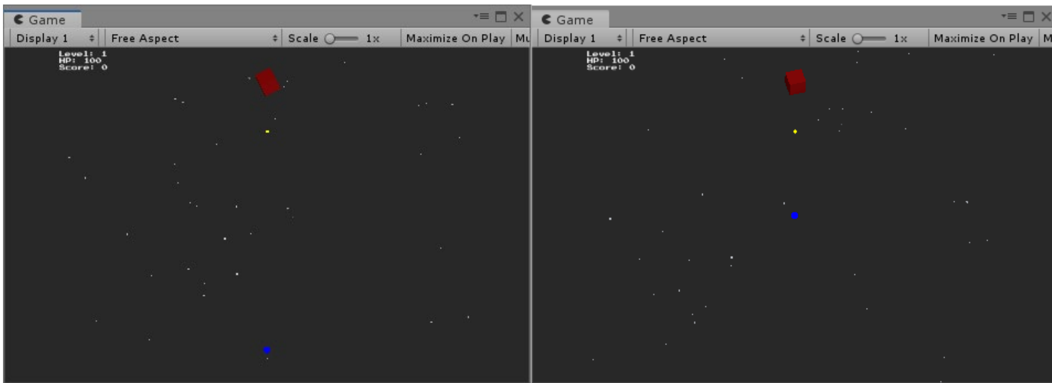


48 Let's double check to see if the player's **position** has an impact on the **speed** of the hazard now that we have **normalized** the **HazardMoveDirection** vector.

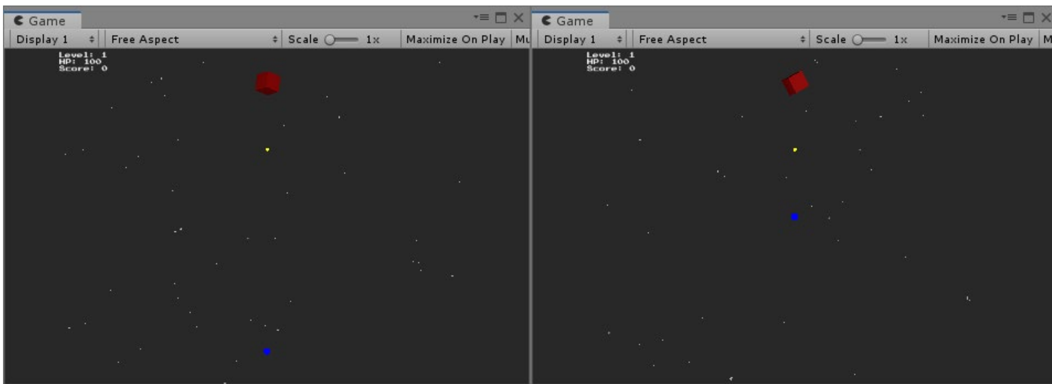
0 seconds, frame 0



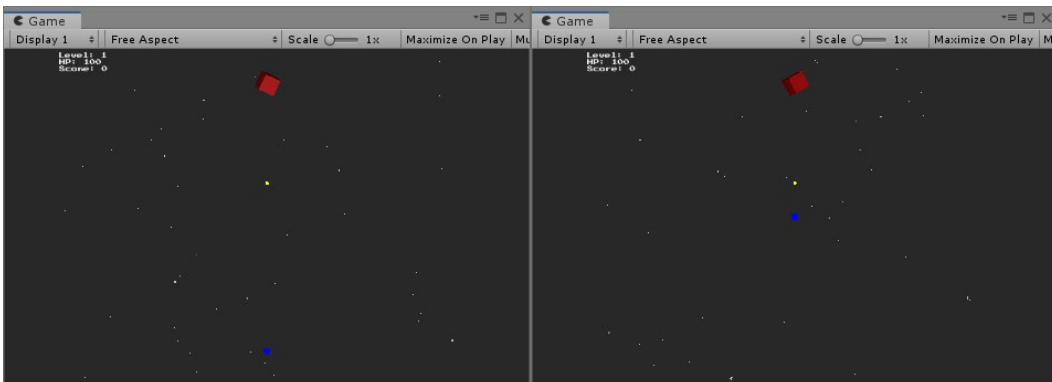
.6 seconds, frame 20



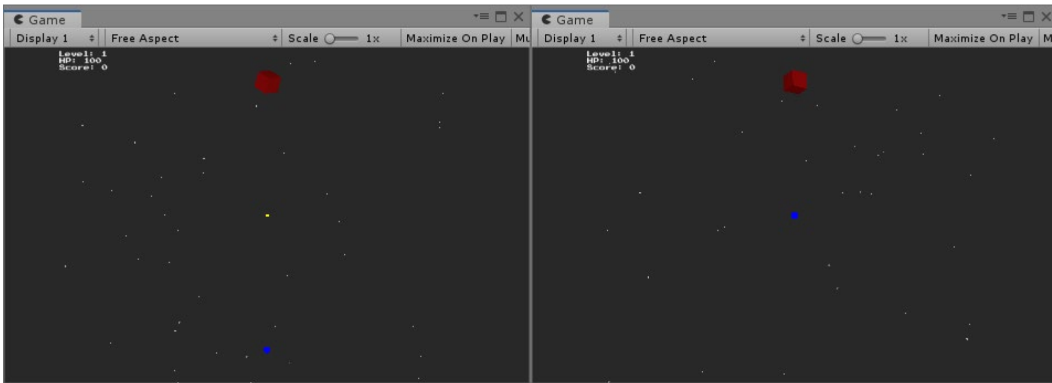
1.3 seconds, frame 40



2 seconds, frame 60



2.6 seconds, frame 80



Compare the hazard's location in both images (left and right). Although the player's position is closer in the images on the right, what do you notice about the hazard's location on the screen?

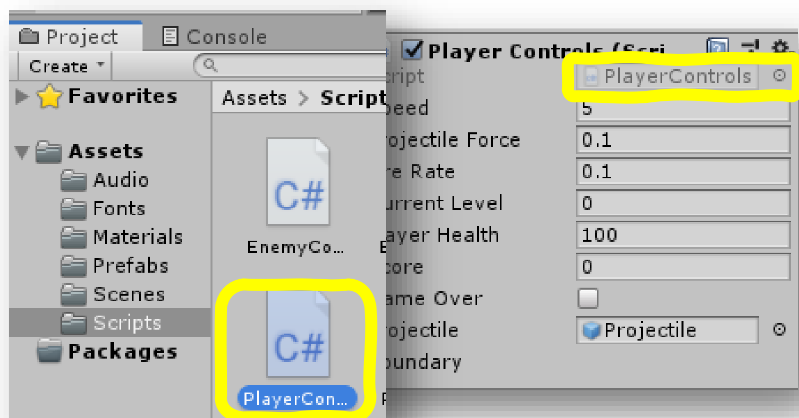
The hazard is traveling at the same speed no matter where the player is because we normalized the HazardMoveDirection vector!

49 **Play** your game and think about what is still missing.

You can play it to see how long you last, but we can still add in the ability for the player to fire hazards back at the enemies!

Next, let's add player bullets!

50 We can give the player the ability to fire hazards by editing the **PlayerControls** script. We can find this in the scripts folder or attached to the **Player** object in the **Inspector**. Double click it to open it in Visual Studio.



51 At the top of the `PlayerControls` class, there are different `private` and `public` variables that help the script run. Some of those variables are initialized in the Start function. We need to write the code that will instantiate the player hazards in the scene. Since we need to react to the game as it runs, we will write this code in the `Update` function.

```
void Update()
{
    if (gameOver)
    {
        return;
    }
    // While the game is running, if the player presses space or z, and you can fire
    // Potentially 5 clones of the projectile object will be made
    // If the currentLevel is 1 (or greater) a projectile is created directly at the player
    // If the currentLevel is 3 (or greater) additional projectiles are added to the left and right
    // If the currentLevel is 5 (or greater) additional projectiles are added again to the left and right
    if ((Input.GetKey("space") || Input.GetKey("z")) && (Time.time > nextFire))
    {
        nextFire = Time.time + fireRate;

        /*****************************\
        |**** Add your code below ****|
        \*****************************/


        /*****************************\
        |**** Add your code above ****|
        \*****************************/
    }


    // If the playerHealth is reduced to 0, the gameOver bool is set to true, having various effects across multiple scripts
    if (playerHealth <= 0) {
        gameOver = true;
    }
}
```

52 Find the Update function and look at the code that is already present. It might seem like a lot, but there are a lot of lines of comments!

53 Look at the first if statement. This isn't the exact same code that we looked at in the other three scripts.

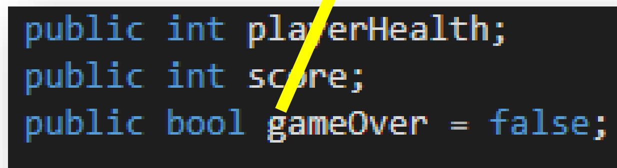
In the other scripts it says if (playerController.gameOver) while in this script it just says if (gameOver).

Why do you think we do not need to use playerController in this script?

It's because this script is the playerController variable we used previously!

```
if (playerController.gameOver)
{
    return;
}
```

```
if (gameOver)
{
    return;
}
```



The gameOver variable belongs to the script we are working on!

54 Look at the last if statement. This checks to see if the player's health is less than or equal to zero and sets the gameOver variable to true. Setting gameOver to true will stop the enemy and powerup emitters, the enemies will not fire, and the player will not be able to move.

```
if (playerHealth <= 0) {
    gameOver = true;
}
```

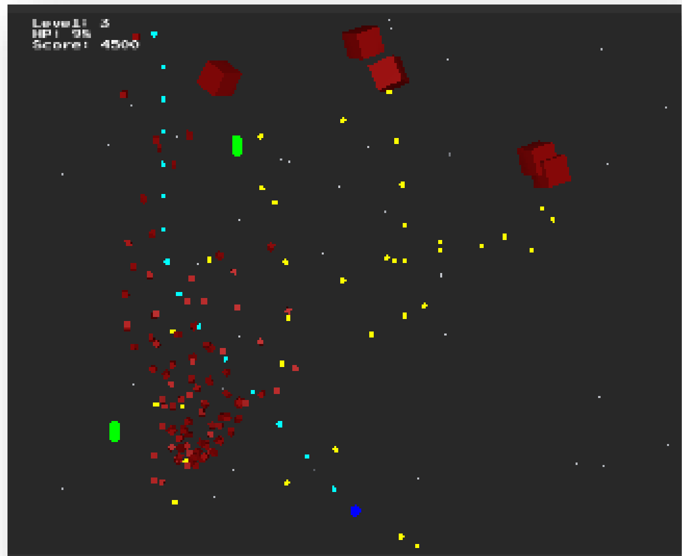
55 Finally, look at the middle if statement. The conditional is checking two things. First, it checks to see if the user has pressed the space or z keys. If one of those keys is pressed, then it checks to see if the player is allowed to fire by comparing the current time to the next time the player is allowed to fire. The one line of provided code inside the statement sets up when the player can fire next.

```
if ((Input.GetKey("space") || Input.GetKey("z")) && (Time.time > nextFire))
{
    nextFire = Time.time + fireRate;

    /******************************\
    |**** Add your code below ****|
    \******************************/


    /******************************\
    |**** Add your code above ****|
    \******************************/
}
```

56 If the player is pressing the correct key and can fire, we want to create a new projectile object at the player's location with the player's direction. Type Instantiate(projectile, transform.position, transform.rotation);.

```
if ((Input.GetKey("space") || Input.GetKey("z")) && (Time.time > nextFire))
{
    nextFire = Time.time + fireRate;

    /******************************\
    |**** Add your code below ****|
    \******************************/

    Instantiate(projectile, transform.position, transform.rotation);

    /******************************\
    |**** Add your code above ****|
    \******************************/
}
```

57 Play your game and try to fire! Everything works! We did not set a velocity for the new projectile because the projectile understands that it needs to go up no matter what. You can open the ProjectileScript file to see exactly how that is coded!



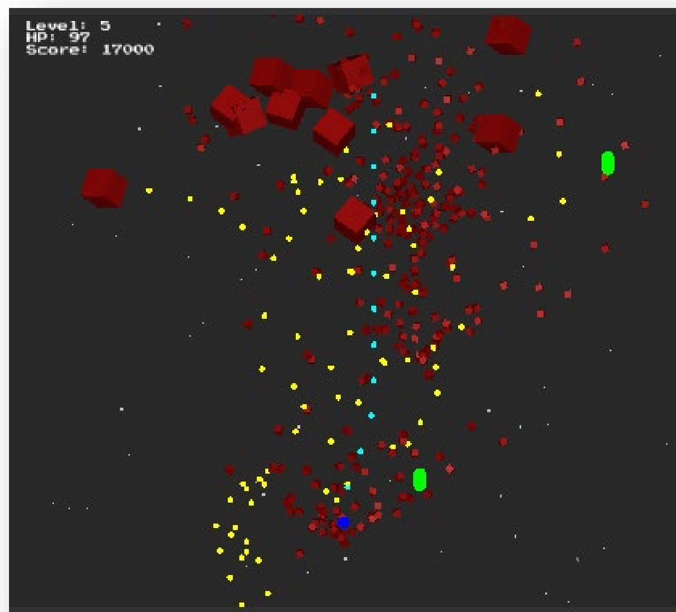
58 As the player levels up, more and more enemies appear, but the player doesn't get more powerful. We should reward the player for leveling up. After the Instantiate line you just wrote, create an if statement that checks to see if the currentLevel is greater than or equal to 3 by typing if (currentLevel >= 3) { } making sure to leave a blank line between the curly brackets.

```
/*******************************\
|**** Add your code below ****|
\*******************************/

Instantiate(projectile, transform.position, transform.rotation);
if (currentLevel >= 3)
{
    
}

/*******************************\
|**** Add your code above ****|
\*******************************/
```

59 We can reward the player by adding additional projectiles! If they get to level three, they should have three projectiles. Add two more Instantiate statements inside of the if statement.

```
/*****************************\
|**** Add your code below ****|
\*****************************/

Instantiate(projectile, transform.position, transform.rotation);
if (currentLevel >= 3)
{
    Instantiate(projectile, transform.position, transform.rotation);
    Instantiate(projectile, transform.position, transform.rotation);
}

/*****************************\
|**** Add your code above ****|
\*****************************/
```

60 Play your game, get to level 3, and see what happens! It's hard to tell, but even though it looks like the player is only firing one projectile, there are actually three that have the same exact position in the game scene.



61 We can fix this by changing the x positions of our two new projectiles we are spawning.

Above the two Instantiate lines, create a new Vector3 variable named rightOffset and set it equal to a new Vector3 with an x value of 0.2f, a y value of 0, and a z value of 0.

```
if (currentLevel >= 3)
{
    Vector3 rightOffset = new Vector3(0.2f, 0, 0);
    Instantiate(projectile, transform.position, transform.rotation);
    Instantiate(projectile, transform.position, transform.rotation);
}
```

62 After our rightOffset variable, create a second new Vector3 variable named leftOffset and set it equal to a new Vector3 with an x value of -0.2f, a y value of 0, and a z value of 0.

```
if (currentLevel >= 3)
{
    Vector3 rightOffset = new Vector3(0.2f, 0, 0);
    Vector3 leftOffset = new Vector3(-0.2f, 0, 0);
    Instantiate(projectile, transform.position, transform.rotation);
    Instantiate(projectile, transform.position, transform.rotation);
}
```

63

We now need to add the right and left offsets to the position of our new projectiles. When adding vectors, each number gets added individually. So a Vector3(1, 2, 3) plus a Vector3(4, 5, 6) would equal Vector3(1 + 4, 2 + 5, 3 + 6) or Vector3(5, 7, 9). In the first Instantiate, change the second argument to be transform.position + rightOffset.

```
if (currentLevel >= 3)
{
    Vector3 rightOffset = new Vector3(0.2f, 0, 0);
    Vector3 leftOffset = new Vector3(-0.2f, 0, 0);
    Instantiate(projectile, transform.position + rightOffset, transform.rotation);
    Instantiate(projectile, transform.position, transform.rotation);
}
```

64

In the second Instantiate, change the second argument to be transform.position + leftOffset.

```
if (currentLevel >= 3)
{
    Vector3 rightOffset = new Vector3(0.2f, 0, 0);
    Vector3 leftOffset = new Vector3(-0.2f, 0, 0);
    Instantiate(projectile, transform.position + rightOffset, transform.rotation);
    Instantiate(projectile, transform.position + leftOffset, transform.rotation);
}
```

65

Play your game and see how adding offsets changed the extra projectiles in level 3 and above.