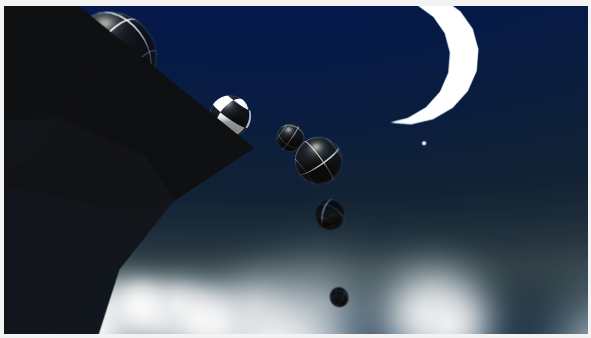
Create with Code

Unit 4 Lesson Plans



© Unity 2021

unity

4.1 Watch Where You're Going

Steps:

Step 1: Create project and open scene

Step 2: Set up the player and add a texture

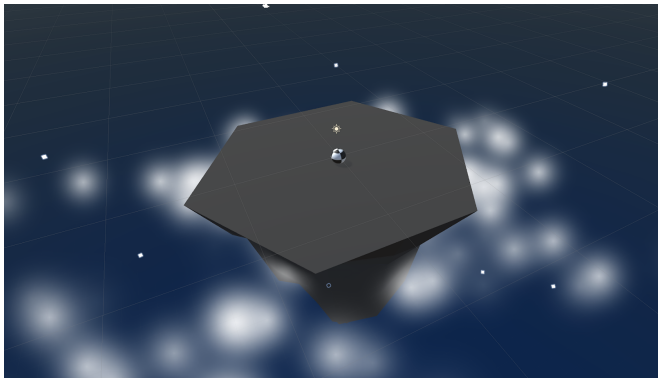
Step 3: Create a focal point for the camera

Step 4: Rotate the focal point by user input

Step 5: Add forward force to the player

Step 6: Move in direction of focal point

Example of project by end of lesson



Length: 60 minutes

Overview: First thing's first, we will create a new prototype and download the starter files! You'll notice a beautiful island, sky, and particle effect... all of which can be customized! Next you will allow the player to rotate the camera around the island in a perfect radius, providing a glorious view of the scene. The player will be represented by a sphere, wrapped in a detailed texture of your choice. Finally you will add force to the player, allowing them to move forwards or backwards in the direction of the camera.

Project Outcome: The camera will evenly rotate around a focal point in the center of the island, provided a horizontal input from the player. The player will control a textured sphere, and move them forwards or backwards in the direction of the camera's focal point.

Learning Objectives: By the end of this lesson, you will be able to:
- Apply Texture wraps to objects
- Attach a camera to its focal point using parent-child relationships
- Transform objects based on local XYZ values

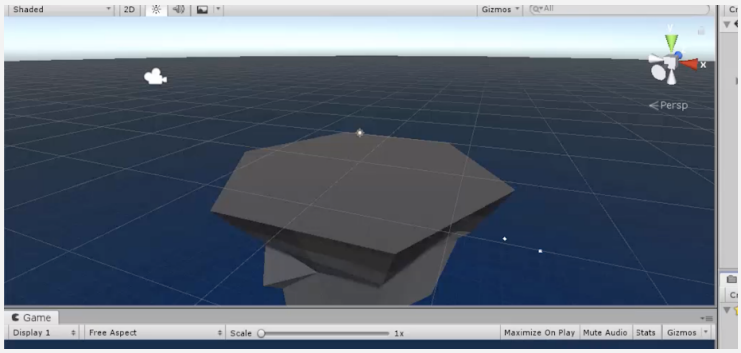
© Unity 2021

Step 1: Create project and open scene

You've done it before, and it's time to do it again... we must start a new project and import the starter files.

1. Open **Unity Hub** and create an empty "Prototype 4" project in your course directory on the correct Unity version.
   If you forget how to do this, refer to the instructions in Lesson 1.1 - Step 1
2. Click to download the Prototype 4 Starter Files, **extract** the compressed folder, and then **import** the .unitypackage into your project.
   If you forget how to do this, refer to the instructions in Lesson 1.1 - Step 2
3. Open the **Prototype 4 scene** and delete the **Sample Scene** without saving
4. Click **Run** to see the **particle effects**

- **Don't worry:** You can change texture of floating island and the color of the sky later
- **Don't worry:** We're in isometric/orthographic view for a reason: It just looks nicer when we rotate around the island

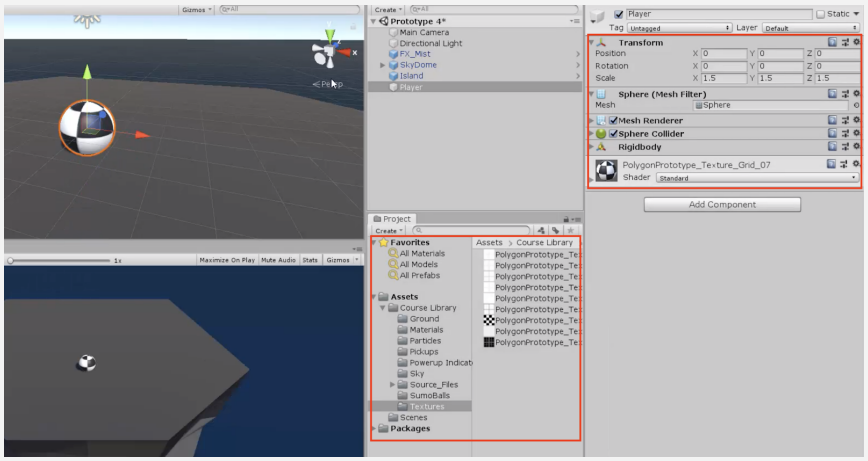


Step 2: Set up the player and add a texture

We've got an island for the game to take place on, and now we need a sphere for the player to control and roll around.

1. In the **Hierarchy**, *create 3D Object* > ***Sphere***
2. Rename it "Player", reset its **position** and increase its XYZ **scale** to 1.5
3. Add a **RigidBody** component to the **Player**
4. From the *Library* > *Textures*, drag a **texture** onto the **sphere**

- **New Concept:** Texture wraps

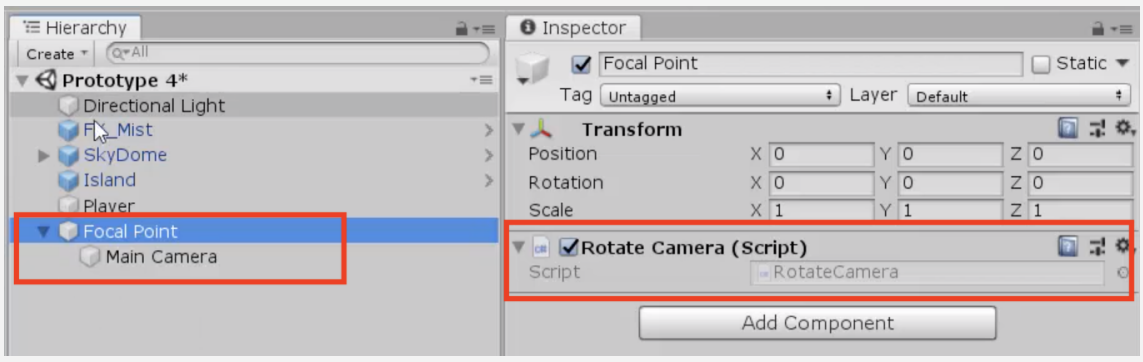


Step 3: Create a focal point for the camera

If we want the camera to rotate around the game in a smooth and cinematic fashion, we need to pin it to the center of the island with a focal point.

1. Create a new **Empty Object** and rename it "Focal Point",
2. Reset its position to the origin (0, 0, 0), and make the Camera a **child object** of it
3. Create a new "Scripts" folder, and a new "RotateCamera" script inside it
4. **Attach** the "RotateCamera" script to the **Focal Point**

- **Don't worry:** This whole "focal point" business may be confusing at first, but it will make sense once you see it in action
- **Tip:** Try rotating the Focal point around the Y axis and see the camera rotate in scene view



Step 4: Rotate the focal point by user input

Now that the camera is attached to the focal point, the player must be able to rotate it - and the camera child object - around the island with horizontal input.

1. Create the code to rotate the camera based on ***rotationSpeed*** and ***horizontalInput***
2. Tweak the ***rotation speed*** value to get the speed you want

- **Tip:** Horizontal input should be familiar, we used it all the way back in Unit 1! Feel free to reference your old code for guidance.

```
public float rotationSpeed;

void Update()
{
  float horizontalInput = Input.GetAxis("Horizontal");
  transform.Rotate(Vector3.up, horizontalInput * rotationSpeed * Time.deltaTime);
}
```

Step 5: Add forward force to the player

The camera is rotating perfectly around the island, but now we need to move the player.

1. Create a new "PlayerController" script, apply it to the **Player**, and open it
2. Declare a new ***public float speed*** variable and initialize it
3. Declare a new ***private Rigidbody playerRb*** and initialize it in ***Start()***
4. In ***Update()***, declare a new ***forwardInput*** variable based on **"Vertical"** input
5. Call the **AddForce()** method to move the player forward based ***forwardInput***

- **Tip:** Moving objects with RigidBody and Addforce should be familiar, we did it back in Unit 3! Feel free to reference old code.
- **Don't worry:** We don't have control over its direction yet - we'll get to that next

```
private Rigidbody playerRb;
public float speed = 5.0f;

void Start() {
  playerRb = GetComponent<Rigidbody>(); }

void Update() {
  float forwardInput = Input.GetAxis("Vertical");
  playerRb.AddForce(Vector3.forward * speed * forwardInput); }
```

Step 6: Move in direction of focal point

We've got the ball rolling, but it only goes forwards and backwards in a single direction! It should instead move in the direction the camera (and focal point) are facing.

1. Declare a new ***private GameObject focalPoint;*** and initialize it in ***Start()***: *focalPoint = GameObject.Find("Focal Point");*
2. In the ***AddForce*** call, Replace ***Vector3.forward*** with ***focalPoint.transform.forward***

- **New Concept:** Global vs Local XYZ
- **Tip:** Global XYZ directions relate to the entire scene, whereas local XYZ directions relate to the object in question

```
private GameObject focalPoint;

void Start() {
  playerRb = GetComponent<Rigidbody>();
  focalPoint = GameObject.Find("Focal Point"); }

void Update() {
  float forwardInput = Input.GetAxis("Vertical");
  playerRb.AddForce(~~Vector3.forward~~ focalPoint.transform.forward
  * speed * forwardInput); }
```

Lesson Recap

New Functionality	• Camera rotates around the island based on horizontal input • Player rolls in direction of camera based on vertical input
New Concepts and Skills	• Texture Wraps • Camera as child object • Global vs Local coordinates • Get direction of other object
Next Lesson	• In the next lesson, we'll add more challenge to the player, by creating enemies that chase them in the game.

unity

4.2 Follow the Player

Steps:

Step 1: Add an enemy and a physics material

Step 2: Create enemy script to follow player

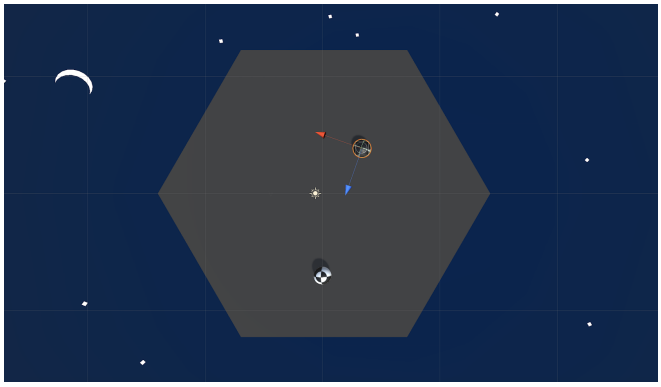
Step 3: Create a lookDirection variable

Step 4: Create a Spawn Manager for the enemy

Step 5: Randomly generate spawn position

Step 6: Make a method return a spawn point

Example of project by end of lesson



Length: 60 minutes

Overview: The player can roll around to its heart's content… but it has no purpose. In this lesson, we fill that purpose by creating an enemy to challenge the player! First we will give the enemy a texture of your choice, then give it the ability to bounce the player away... potentially knocking them off the cliff. Lastly, we will let the enemy chase the player around the island and spawn in random positions.

Project Outcome: A textured and spherical enemy will spawn on the island at start, in a random location determined by a custom function. It will chase the player around the island, bouncing them off the edge if they get too close.

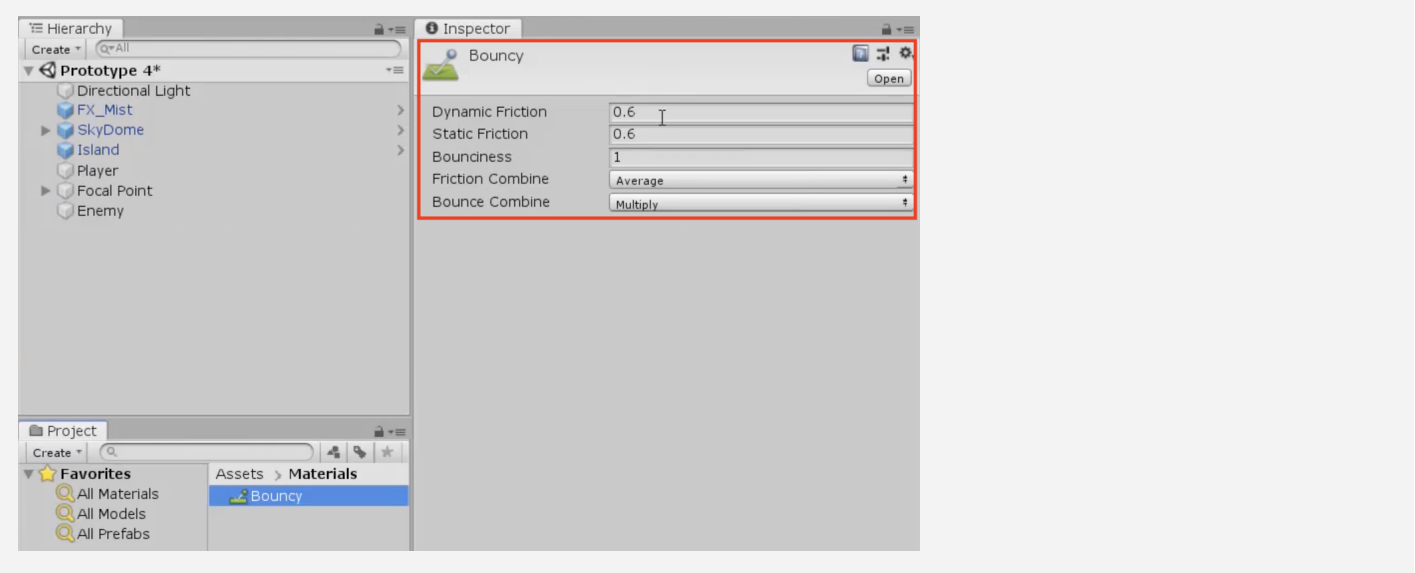
Learning Objectives: By the end of this lesson, you will be able to:
- Apply Physics Materials to make game objects bouncy
- Normalize vectors to point the enemy in the direction of the player
- Randomly spawn with Random.Range on two axes
- Write more advanced custom functions and variables to make your code clean and professional

Step 1: Add an enemy and a physics material

Our camera rotation and player movement are working like a charm. Next we're going to set up an enemy and give them them some special new physics to bounce the player away!

1. Create a new **Sphere**, rename it "Enemy" reposition it, and drag a **texture** onto it
2. Add a new **RigidBody** component and adjust its XYZ **scale**, then test
3. In a new "Physics Materials**"** folder, *Create > Physics Material,* then name it "Bouncy"
4. Increase the **Bounciness** to "1", change **Bounce Combine** to "Multiply", apply it to your player and enemy, then **test**

- **Don't worry:** If your game is lagging, uncheck the "Active" checkbox for your clouds
- **New Concept:** Physics Materials
- **New Concept:** Bounciness property and Bounce Combine



Step 2: Create enemy script to follow player

The enemy has the power to bounce the player away, but only if the player approaches it. We must tell the enemy to follow the player's position, chasing them around the island.

1. Make a new "Enemy" script and attach it to the **Enemy**
2. Declare 3 new variables for ***Rigidbody enemyRb;***, ***GameObject player;***, and ***public float speed;***
3. Initialize ***enemyRb = GetComponent Rigidbody>();*** and ***player = GameObject.Find("Player");***
4. In ***Update()***, AddForce towards in the direction between the Player and the Enemy

- **Tip:** Imagine we're generating this new vector by drawing an arrow from the enemy to the player.
- **Tip:** We should start thinking ahead and writing our variables in advance. Think… what are you going to need?
- **Tip:** When normalized, a vector keeps the same direction but its length is 1.0, forcing the enemy to try and keep up

```
public float speed = 3.0f;
private Rigidbody enemyRb;
private GameObject player;

void Update() {
  enemyRb.AddForce((player.transform.position
  - transform.position).normalized * speed); }
```

Step 3: Create a lookDirection variable

The enemy is now rolling towards the player, but our code is a bit messy. Let's clean up by adding a variable for the new vector.

1. In ***Update()***, declare a new ***Vector3 lookDirection*** variable
2. Set ***Vector3 lookDirection = (player.transform.position - transform.position).normalized;***
3. Implement the ***lookDirection*** variable in the ***AddForce*** call

- **Tip:** As always, adding variables makes the code more readable

```
void Update() {
  Vector3 lookDirection = (player.transform.position
  - transform.position).normalized;

  enemyRb.AddForce(lookDirection ~~(player.transform.position
  - transform.position).normalized~~ * speed); }
```

Step 4: Create a Spawn Manager for the enemy

Now that the enemy is acting exactly how we want, we're going to turn it into a prefab so it can be instantiated by a Spawn Manager.

1. Drag **Enemy** into the Prefabs folder to create a new **Prefab**, then delete **Enemy** from scene
2. Create a new "Spawn Manager" **object**, attach a new "SpawnManager" **script**, and open it
3. Declare a new ***public GameObject enemyPrefab*** variable then assign the prefab in the **inspector**
4. In ***Start()***, instantiate a new ***enemyPrefab*** at a predetermined location

```
public GameObject enemyPrefab;

void Start()
{
  Instantiate(enemyPrefab, new Vector3(0, 0, 6),
enemyPrefab.transform.rotation); }
```

Step 5: Randomly generate spawn position

The enemy spawns at start, but it always appears in the same spot. Using the familiar Random class, we can spawn the enemy in a random position.

1. In SpawnManager.cs, in ***Start()***, create new **randomly generated** X and Z
2. Create a new ***Vector3 randomPos*** variable with those random X and Z positions
3. Incorporate the new ***randomPos*** variable into the ***Instantiate*** call
4. Replace the hard-coded **values** with a ***spawnRange*** variable
5. **Start** and **Restart** your project to make sure it's working

- **Tip:** Remember, we used Random.Range all the way back in Unit 2! Feel free to reference old code.

```
public GameObject enemyPrefab;
private float spawnRange = 9;

void Start() {
  float spawnPosX = Random.Range(-9, 9-spawnRange, spawnRange);
  float spawnPosZ = Random.Range(-9, 9-spawnRange, spawnRange);
  Vector3 randomPos = new Vector3(spawnPosX, 0, spawnPosZ);
  Instantiate(enemyPrefab, randomPos, enemyPrefab.transform.rotation); }
```

Step 6: Make a method return a spawn point

The code we use to generate a random spawn position is perfect, and we're going to be using it a lot. If we want to clean the script and use this code later down the road, we should store it in a custom function.

1. Create a new function ***Vector3 GenerateSpawnPosition() { }***
2. Copy and Paste the ***spawnPosX*** and ***spawnPosZ*** variables into the new method
3. Add the line to ***return randomPos;*** in your new method
4. Replace the code in your **Instantiate call** with your new function name: ***GenerateSpawnPosition()***

- **Tip:** This function will come in handy later, once we randomize a spawn position for the powerup
- **New Concept:** Functions that return a value
- **Tip:** This function is different from "void" calls, which do not return a value. Look at "GetAxis" in PlayerController for example - it returns a float

```
void Start() {
  Instantiate(enemyPrefab, GenerateSpawnPosition()
  new Vector3(spawnPosX, 0, spawnPosZ), enemyPrefab.transform.rotation);
  float spawnPosX = Random.Range(-spawnRange, spawnRange);
  float spawnPosZ = Random.Range(-spawnRange, spawnRange); }

private Vector3 GenerateSpawnPosition () {
  float spawnPosX = Random.Range(-spawnRange, spawnRange);
  float spawnPosZ = Random.Range(-spawnRange, spawnRange);
  Vector3 randomPos = new Vector3(spawnPosX, 0, spawnPosZ);
  return randomPos; }
```

Lesson Recap

New Functionality	• Enemy spawns at random location on the island • Enemy follows the player around • Spheres bounce off of each other
New Concepts and Skills	• Physics Materials • Defining vectors in 3D space • Normalizing values • Methods with return values
Next Lesson	• In our next lesson, we'll create ways to fight back against these enemies using Powerups!

unity

4.3 PowerUp and CountDown

Steps:

Step 1: Choose and prepare a powerup

Step 2: Destroy powerup on collision

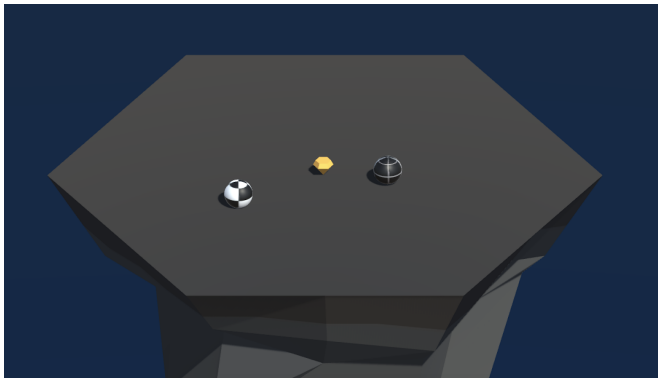
Step 3: Test for collision with a powerup

Step 4: Apply extra knockback with powerup

Step 5: Create Countdown Routine for powerup

Step 6: Add a powerup indicator

Example of project by end of lesson



Length: 60 minutes

Overview: The enemy chases the player around the island, but the player needs a better way to defend themselves... especially if we add more enemies. In this lesson, we're going to create a powerup that gives the player a temporary strength boost, shoving away enemies that come into contact! The powerup will spawn in a random position on the island, and highlight the player with an indicator when it is picked up. The powerup indicator and the powerup itself will be represented by stylish game assets of your choice.

Project Outcome: A powerup will spawn in a random position on the map. Once the player collides with this powerup, the powerup will disappear and the player will be highlighted by an indicator. The powerup will last for a certain number of seconds after pickup, granting the player super strength that blasts away enemies.

Learning Objectives: By the end of this lesson, you will be able to:
- Write informative debug messages with Concatenation and variables
- Repeat functions with the power of IEnumerator and Coroutines
- Use SetActive to make game objects appear and disappear from the scene

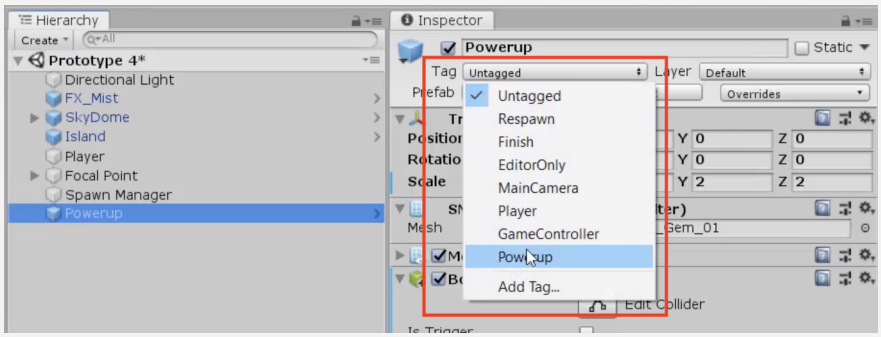
© Unity 2021

Step 1: Choose and prepare a powerup

In order to add a completely new gameplay mechanic to this project, we will introduce a new powerup object that will give the player temporary superpowers.

1. From the *Library*, drag a **Powerup** object into the scene, rename it "Powerup" and edit its **scale** & **position**
2. Add a **Box Collider** to the powerup, click **Edit Collider** to make sure it fits, then check the "**Is Trigger**" checkbox
3. Create a new "Powerup" **tag** and apply it to the **powerup**
4. Drag the **Powerup** into the **Prefabs folder** to create a new "Original Prefab"

- **Warning:** Remember, you still have to apply the tag after it has been created.



Step 2: Destroy powerup on collision

As a first step to getting the powerup working, we'll make it disappear when the player hits it and set up a new boolean variable to track that the player got it.

1. In **PlayerController.cs**, add a **new *OnTriggerEnter()*** method
2. Add an **if-statement** that destroys ***other.CompareTag("Powerup")*** **powerup** on collision
3. Create a new ***public bool hasPowerup;*** and set ***hasPowerup = true;*** when you **collide** with the Powerup

- **Don't worry:** If this doesn't work, make sure that the Powerup's collider "Is trigger" and player's collider is NOT
- **Tip:** Make sure hasPowerup = true in the inspector when you collide

```
public bool hasPowerup

private void OnTriggerEnter(Collider other) {
  if (other.CompareTag("Powerup")) {
    hasPowerup = true;
    Destroy(other.gameObject); } }
```

Step 3: Test for enemy and powerup

The powerup will only come into play in a very particular circumstance: when the player has a powerup AND they collide with an enemy - so we'll first test for that very specific condition.

1. Create a new "Enemy" tag and apply it to the **Enemy Prefab**
2. In **PlayerController.cs**, add the ***OnCollisionEnter()*** function
3. Create the **if-statement** with the **double-condition** testing for enemy tag and hasPowerup boolean
4. Create a **Debug.Log** to make sure it's working

- **Tip:** OnTriggerEnter is good for stuff like picking up powerups, but you should use OnCollisionEnter when you want something to do with physics
- **New Concept:** Concatenation in Debug messages
- **Tip:** When you concatenate a variable in a debug message, it will returns its VALUE not its name

```
private void OnCollisionEnter(Collision collision) {
  if (collision.gameObject.CompareTag("Enemy") && hasPowerup) {
    Debug.Log("Collided with " + collision.gameObject.name
      + " with powerup set to " + hasPowerup);
  }
}
```

Step 4: Apply extra knockback with powerup

With the condition for the powerup set up perfectly, we are now ready to program the actual powerup ability: when the player collides with an enemy, the enemy should go flying!

1. In ***OnCollisionEnter()*** declare a new **local variable** to get the Enemy's **Rigidbody** component
2. Declare a new variable to get the **direction** away from the **player**
3. Add an **impulse force** to the **enemy**, using a new **powerupStrength** variable

- **Tip:** Reference the code in Enemy.cs that makes the enemy follow the player. In a way, we're reversing that code in order to push the enemy away.
- **Don't worry:** No need to use .Normalize, since they're colliding

```
private float powerupStrength = 15.0f;

private void OnCollisionEnter(Collision collision) {
  if (collision.gameObject.CompareTag("Enemy") && hasPowerup) {

    Rigidbody enemyRigidbody = collision.gameObject.GetComponent<Rigidbody>();
    Vector3 awayFromPlayer = (collision.gameObject.transform.position
    - transform.position);

    Debug.Log("Player collided with " + collision.gameObject
    + " with powerup set to " + hasPowerup);
    enemyRigidbody.AddForce(awayFromPlayer * powerupStrength,
    ForceMode.Impulse); } }
```

Step 5: Create Countdown Routine for powerup

It wouldn't be fair to the enemies if the powerup lasted forever - so we'll program a countdown timer that starts when the player collects the powerup, removing the powerup ability when the timer is finished.

1. Add a new ***IEnumerator PowerupCountdownRoutine () {}***
2. Inside the ***PowerupCountdownRoutine***, wait 7 seconds, then **disable** the powerup
3. When player **collides** with powerup, start the **coroutine**

- **New Concept:** IEnumerator
- **New Concept:** Coroutines
- **Tip:** WaitForSeconds()

```
private void OnTriggerEnter(Collider other) {
  if (other.CompareTag("Powerup")) {
    hasPowerup = true;
    Destroy(other.gameObject);
    StartCoroutine(PowerupCountdownRoutine()); } }

IEnumerator PowerupCountdownRoutine() {
  yield return new WaitForSeconds(7); hasPowerup = false; }
```

Step 6: Add a powerup indicator

To make this game a lot more playable, it should be clear when the player does or does not have the powerup, so we'll program a visual indicator to display this to the user.

1. From the *Library*, drag a **Powerup object** into the scene, rename it "Powerup Indicator", and edit its **scale**
2. **Uncheck** the "**Active**" checkbox in the inspector
3. In **PlayerController.cs**, declare a new ***public GameObject powerupIndicator*** variable, then assign the **Powerup Indicator** variable in the inspector
4. When the player collides with the powerup, set the indicator object to **Active**, then set to **Inactive** when the powerup expires
5. In ***Update()***, set the Indicator position to the player's position + an **offset value**

- **New Function:** SetActive
- **Tip:** Make sure the indicator is turning on and off before making it follow the player

```
public GameObject powerupIndicator

void Update() {
  ... powerupIndicator.transform.position = transform.position
  + new Vector3(0, -0.5f, 0); }

private void OnTriggerEnter(Collider other) {
  if (other.CompareTag("Powerup")) {
    ... powerupIndicator.gameObject.SetActive(true); } }

IEnumerator PowerupCountdownRoutine() {
  ... powerupIndicator.gameObject.SetActive(false); }
```

Lesson Recap

New Functionality	• When the player collects a powerup, a visual indicator appears • When the player collides with an enemy while they have the powerup, the enemy goes flying • After a certain amount of time, the powerup ability and indicator disappear
New Concepts and Skills	• *Debug concatenation* • *Local component variables* • *IEnumerators and WaitForSeconds()* • *Coroutines* • SetActive(true/false)
Next Lesson	• We'll start generating waves of enemies for our player to fend off!

unity

4.4 For-Loops For Waves

Steps:

Step 1: Write a for-loop to spawn 3 enemies

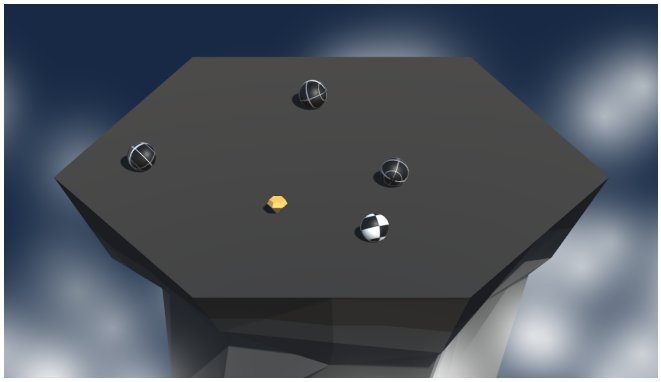
Step 2: Give the for-loop a parameter

Step 3: Destroy enemies if they fall off

Step 4: Increase enemyCount with waves

Step 5: Spawn Powerups with new waves

Example of project by end of lesson



Length: 60 minutes

Overview: We have all the makings of a great game; A player that rolls around and rotates the camera, a powerup that grants super strength, and an enemy that chases the player until the bitter end. In this lesson we will wrap things up by putting these pieces together!
First we will enhance the enemy spawn manager, allowing it to spawn multiple enemies and increase their number every time a wave is defeated. Lastly we will spawn the powerup with every wave, giving the player a chance to fight back against the ever-increasing horde of enemies.

Project Outcome: The Spawn Manager will operate in waves, spawning multiple enemies and a new powerup with each iteration. Every time the enemies drop to zero, a new wave is spawned and the enemy count increases.

Learning Objectives: By the end of this lesson, you will be able to:
- Repeat functions with For-loops
- Increment integer values in a loop with the ++ operator
- Target objects in a scene with FindObjectsOfType
- Return the length of an array as an integer with .Length

© Unity 2021

Step 1: Write a for-loop to spawn 3 enemies

We should challenge the player by spawning more than one enemy. In order to do so, we will repeat enemy instantiation with a loop.

1. In SpawnManager.cs, in ***Start()***, replace single **Instantiation** with a **for-loop** that spawns 3 enemies
2. Move the for-loop to a new ***void SpawnEnemyWave()*** function, then call that function from ***Start()***

- **New Concept:** For-loops
- **Don't worry:** Loops are a bit confusing at first, but they make sense eventually. Loops are powerful tools that programmers use often
- **New Concept:** ++ Increment Operator

```
void Start() {
  SpawnEnemyWave();
  for (int i = 0; i < 3; i++) {
    Instantiate(enemyPrefab, GenerateSpawnPosition(),
    enemyPrefab.transform.rotation); } }

void SpawnEnemyWave() {
  for (int i = 0; i < 3; i++) {
    Instantiate(enemyPrefab, GenerateSpawnPosition(),
    enemyPrefab.transform.rotation); } }
```

Step 2: Give the for-loop a parameter

Right now, SpawnEnemyWave spawns exactly 3 enemies, but if we're going to dynamically increase the number of enemies that spawn during gameplay, we need to be able to pass information to that method.

1. Add a parameter ***int enemiesToSpawn*** to the ***SpawnEnemyWave*** function
2. Replace ***i < __*** with ***i < enemiesToSpawn***
3. Add this new variable to the function call in ***Start()***: ***SpawnEnemyWave(___);***

- **New Concept:** Custom methods with parameters
- **Tip:** GenerateSpawnPosition returns a value, SpawnEnemyWave does not. SpawnEnemyWave takes a parameter, GenerateSpawnPosition does not.

```
void Start() {
  SpawnEnemyWave(3); }

void SpawnEnemyWave(int enemiesToSpawn) {
  for (int i = 0; i < 3 enemiesToSpawn; i++) {
    Instantiate(enemyPrefab, GenerateSpawnPosition(),
    enemyPrefab.transform.rotation); } }
```

Step 3: Destroy enemies if they fall off

Once the player gets rid of all the enemies, they're left feeling a bit lonely. We need to destroy enemies that fall, and spawn a new enemy wave once the last one is vanquished!

1. In Enemy.cs, **destroy** the enemies if their position is less than a **-Y value**
2. In SpawnManager.cs, declare a new ***public int enemyCount*** variable
3. In ***Update()***, set ***enemyCount = FindObjectsOfType<Enemy>().Length;***
4. Write the **if-statement** that if ***enemyCount == 0*** then ***SpawnEnemyWave***

- **New Function:** FindObjectsOfType

```
void Update() {
  ... if (transform.position.y < -10) { Destroy(gameObject); } }

<------>
public int enemyCount

void Update() {
  enemyCount = FindObjectsOfType<Enemy>().Length;
  if (enemyCount == 0) { SpawnEnemyWave(1); } }
```

Step 4: Increase enemyCount with waves

Now that we control the amount of enemies that spawn, we should increase their number in waves. Every time the player defeats a wave of enemies, more should rise to take their place.

1. Declare a new ***public int waveNumber = 1;***, then implement it in ***SpawnEnemyWave(waveNumber);***
2. In the if-statement that tests if there are 0 enemies left, **increment waveNumber** by 1

- **Tip:** Incrementing with the ++ operator is very handy, you may find yourself using it in the future

```
public int waveNumber = 1;

void Start() {
  SpawnEnemyWave(3 waveNumber); }

void Update() {
  enemyCount = FindObjectsOfType<Enemy>().Length;
  if (enemyCount == 0) { waveNumber++;  SpawnEnemyWave(1 waveNumber); } }
```

Step 5: Spawn Powerups with new waves

Our game is almost complete, but we're missing something. Enemies continue to spawn with every wave, but the powerup gets used once and disappears forever, leaving the player vulnerable. We need to spawn the powerup in a random position with every wave, so the player has a chance to fight back.

1. In SpawnManager.cs, declare a new ***public GameObject powerupPrefab*** variable, assign the **prefab** in the inspector and **delete** it from the scene
2. In **Start()**, **Instantiate** a new Powerup
3. Before the ***SpawnEnemyWave()*** call, **Instantiate** a new Powerup

- **Tip:** Now that we have a very playable game, let's test and tweak values

```
public GameObject powerupPrefab;

void Start() {
  ... Instantiate(powerupPrefab, GenerateSpawnPosition(),
  powerupPrefab.transform.rotation); }

void Update() {
  ... if (enemyCount == 0) { ... Instantiate(powerupPrefab,
    GenerateSpawnPosition(), powerupPrefab.transform.rotation); } }
```

Lesson Recap

New Functionality	• Enemies spawn in waves • The number of enemies spawned increases after every wave is defeated • A new power up spawns with every wave
New Concepts and Skills	• For-loops • Increment (++) operator • Custom methods with parameters • FindObjectsOfType