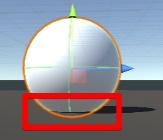
# Setup Project

* Create a new Unity project using the 3D template
* In **Assets** menu 🡪 **Import Package 🡪 Custom Package**
  + Choose the Prototype-4-Starter-Files
* Open the Scene labeled **Prototype 4**
* Delete the **SampleScene**

# Add & Setup Player Character

* In the Hierarchy, click **Create 🡪 3D Object 🡪 Sphere**
* Rename the object as "Player”
* In the Transform component:
  + Click the "gear" icon  and click **Reset**, to reset the sphere's position the sphere   
    back to (0, 0, 0)
  + Make the sphere a little larger by setting the XYZ **Scale** all to **1.5**
* If you position the Scene view so that you are fairly flat with the surface of the platform, you will see that the sphere is slightly below the surface. Drag the sphere on the **Y-axis** up so that the sphere is above the platform.  
  (Or just set it's Y position to **0.1**)
* Add a **Rigidbody** component so we can apply physics
* Look in the **Assets🡪Textures** folder, find a texture and drag it onto the sphere (either directly onto the sphere in the Scene view, or onto the Player object in the Hierarchy view). For this demo, we will use **PolygonPrototype\_Texture\_Grid\_Glass\_01**

# Create a Focal Point for the Camera

*For this game, the camera is supposed to rotate around the platform. To accomplish this, we will create an empty game object in the center of the platform that will be used to rotate the camera around. The camera should rotate when the user presses left/right arrow keys.*

* In the Hierarchy, create an **Empty Game Object**, and name it "Focal Point”
  + Use the gear icon in the transform component to reset it's position to (0, 0, 0)
* In the Hierarchy drag the **Main Camera** onto the Focal Point, so that the camera becomes a child of the focal point. This "attaches" the camera to the focal point so that when the focal point rotates, the camera moves with it. *Go ahead and try changing the Y-axis rotation and see the camera move with it – but change it back to 0 when you get finished.*
* Create a **Scripts** folder under the **Assets** folder
* Create a new C# script called **RotateCamera** 
  + Drag the script onto the **Focal Point** object so add the script as a component

*Now, update the script so that the user can rotate the camera around the platform by using the left/right arrow keys.*

* Add a class variable in the **RotateCamera** script to control speed:

public float rotationSpeed = 50;

* Add the **FixedUpdate** method: void FixedUpdate()
* In the **FixedUpdate** method:
  + Get the player's input:

float horizontalInput = Input.GetAxis("Horizontal");

* + Then rotate the game object on the Y-axis *(in this case, the focal point)*:

transform.Rotate(Vector3.up, horizontalInput \* rotationSpeed \* Time.deltaTime);

# Create a Script to Control the Player

*For this game, the sphere should move forward/backward when the up/down arrow keys are pressed.*

* Create a new C# script called **PlayerController** 
  + Drag the script onto the **Player** object so add the script as a component
* Add class variables for the movement speed and reference to the player's Rigidbody component:

private Rigidbody playerRb;

public float speed = 5;

* In the **Start** method, initialize the Rigidbody reference:

playerRb = GetComponent<Rigidbody>();

* Add the **FixedUpdate** method: void FixedUpdate()
* In the **FixedUpdate** method::
  + Get the player's input:

float forwardInput = Input.GetAxis("Vertical");

* + Then add force to move the sphere in the forward direction:

playerRb.AddForce(Vector3.forward \* forwardInput \* speed);

* Test the game to see the player movement and the camera movement

# Update the PlayerController

*Instead of the sphere moving in a global forward/backward motion, we want it to move forward in relation to the camera position (i.e. when UP is pressed, the sphere should move in the same direction that the camera is facing)*

* Add a class variable to hold a reference to the focal point:

private GameObject focalPoint;

* In the **Start** method, initialize the focal point reference:

focalPoint = GameObject.Find("Focal Point");

* Modify the **Update** method to change the direction in which the force is added, so that instead of always adding force in the global forward direction (Vector3.forward), it will be added in the same direction as the game object is facing:

playerRb.AddForce(**focalPoint.transform.forward** \* forwardInput \* speed);

# Create an Enemy

* In the Hierarchy, click **Create 🡪 3D Object 🡪 Sphere**
* Rename the object as "Enemy”
* In the Transform component:
  + Set the enemy's **position** to **(0, 0.1, 6)**
  + Set the XYZ **Scale** all to **1.5**
* Look in the **Assets🡪Textures** folder, find a texture and drag it onto the sphere. For this demo, we will use **PolygonPrototype\_Texture\_Grid\_03**
* Create and assign a **Tag** to the enemy called "Enemy"
* Add a **Rigidbody** component so we can apply physics
* Test the game and try to push the white sphere off of the platform. Pay attention to how the objects interact and move.

# Modify Physics Properties

* Create a new folder called **Physics Materials**
* Right-click on the folder, and **create a new Physics Material**. Name it "Bouncy"
* Drag the Bouncy material onto both the player and enemy objects *(look for it under the Sphere Collider if you want to make sure it was added properly)*
* In the Inspector for the **Bouncy material**, change the following properties:
  + **Bounciness: 1** *to add some bounciness when they collide*
  + **Bounce Combine: Multiply** *to amplify/multiply the bounciness of the two colliding objects*

# Create a Script for the Enemy to Follow the Player

* Create a new C# script called **Enemy** 
  + Drag the script onto the **Enemy** object so add the script as a component
* Add class variables for the movement speed, reference to the enemy's Rigidbody component, and a reference to the player:

private Rigidbody enemyRb;  
private GameObject player;

public float speed = 3;

* In the **Start** method, initialize the Rigidbody and player references:

enemyRb = GetComponent<Rigidbody>();  
player = GameObject.Find("Player");

* Add the **FixedUpdate** method: void FixedUpdate()
* In the **FixedUpdate** method, add code that move the enemy toward the player:

Vector3 lookDirection = (player.transform.position - transform.position);  
enemyRb.AddForce(lookDirection \* speed);

*Note that by subtracting the enemy's position from the player's position, we end up with a vector that indicates the direction the enemy must travel in to reach the player, and also indicates the distance that must be traveled.*

* Play the game and notice how aggressively the enemy pursues the player – even if both objects fall off the platform!  *As the distance between player and enemy increase, the force is increased.*
* To understand the calculations a little better, you can add the following debugging lines. Pause the game and then press play so that you can see the initial values printed to the console.

Debug.Log(player.transform.position);

Debug.Log(transform.position);

Debug.Log(lookDirection);

* Modify the calculation so that the vector only represents the direction, but NOT the distance to travel. In this way, the enemy will not increase in force/speed as distance increases:

Vector3 lookDirection = (player.transform.position -   
 transform.position)**.normalized;**

* Play the game again and notice that the enemy movement is more reasonable – and also notice the difference in the debugging statements.

# Create an Enemy Spawn Manager

* Create a **Prefabs** folder
  + Drag the **Enemy** object into it to create an enemy prefab
  + Delete the Enemy from the Hierarchy
* Create an empty object in the Hierarchy called **Spawn Manager**
* Create a new C# script called **SpawnManager** and drag it onto the Spawn Manager object
* Add a class variable for the enemy prefab reference:

public GameObject enemyPrefab;

* To make the enemy spawn at a random position, in the Scene view, position the view direcly overhead of the platform, and move the Spawn Manager object around on the X and Z axes to find the range of space where an enemy should appear. For this demo, it will be a range of **-9 to +9.**
* Add a class variable for the enemy prefab reference:

private float spawnRange = 9;

* Create a new method to generate and return a random spawn position:

private Vector3 GenerateSpawnPosition()

{

float spawnPosX = Random.Range(-spawnRange, spawnRange);

float spawnPosZ = Random.Range(-spawnRange, spawnRange);

return new Vector3(spawnPosX, 0, spawnPosZ);

}

* In the **Start** method, add code that creates an enemy at a random position:

Instantiate(enemyPrefab, GenerateSpawnPosition(), enemyPrefab.transform.rotation);

# Powerups

*The player should be able to pickup a powerup that will give them super strength (to push the enemy far away from the player). The powerup is only good for 5 seconds. At the end of the 4 seconds, the powerup should disappear.*

* In the **Pickups** folder, look for the **Gem\_01** object and drag it into the Hierarchy.
  + Rename it as "Powerup"
  + Change it's position temporarily to (0, 0, 4)
  + Change it's scale to (2, 2, 2)
* Add a **Box Collider** component to the powerup
  + Check the **Is Trigger** box
* Create and assign a **Tag** to the powerup called "Powerup"
* Drag the Powerup into the **Prefabs** folder, and choose **Original Prefab** when asked

# Add Code to Support the Powerup

* In the **PlayerController** script add class variables to record when the player has a powerup, and the strength of the powerup, and how long the powerup should last:

public bool hasPowerup = false;  
private float powerUpStrength = 15;  
private float powerUpTime = 5;

* Add the **OnTriggerEnter** method. *Remember that you can press Ctrl+Shift+M to bring up the list of methods that can be added.*

private void OnTriggerEnter(Collider other)

* Write code for the OnTriggerEnter that will remove the powerup from the scene when the player runs into it, and record that the player has it

if (other.CompareTag("Powerup"))

{

Destroy(other.gameObject);

hasPowerup = true;

}

* Add the **OnCollisionEnter** method.

private void OnCollisionEnter(Collision collision)

* Add code to check if the player has collided with an enemy while holding the powerup:

if (collision.gameObject.CompareTag("Enemy") && hasPowerup)

* Inside the IF statement, add code to:
  + Get the Rigidbody component for the enemy that the player collided with:

Rigidbody enemyRb = collision.gameObject.GetComponent<Rigidbody>();

* + Calculate the direction to send the enemy away from the player, by subtracting the player's position from the enemy's position:

Vector3 awayFromPlayer = collision.gameObject.transform.position - transform.position;

* + Apply a force to the enemy to send it away. This line sends the enemy in the calculated direction multipled by a value that will give more strength to the force, and uses the *Impulse* ForceMode so that the force is applied instantly.

enemyRb.AddForce(awayFromPlayer \* powerUpStrength, ForceMode.Impulse);

# Add a Timer to Remove the Powerup

* In the **PlayerController** script, create a *coroutine method* to wait for the amount of time that the powerup should last, and then remove it from the player. Recall that a coroutine is used when a block of code will take more than one frame to complete.

IEnumerator PowerupCountdownRoutine()

{

yield return new WaitForSeconds(powerUpTime);

hasPowerup = false;

}

* In the **OnTriggerEnter** method, when the player picks up the powerup, start the count down timer:

StartCoroutine(PowerupCountdownRoutine());

# Add a Powerup Indicator

* In the **Powerup Indicators** folder, find a powerup indicator and drag it into the Hierarchy. For this demo, we will use **SelectionRing\_02**
  + Rename it as "Powerup Indicator"
  + Make it larger by changing its **scale** to **(3, 1, 3)**
  + Move it to the bottom of the sphere to **position (0, -0.5, 0)**
* In the Inspector, beside the name of the object uncheck the box to make the object inactive
* In the **PlayerController** script, add a class variable that will be a reference to the powerup indicator:

public GameObject powerupIndicator;

* In the Unity editor, drag the powerup indicator object onto the **Powerup Indicator** reference in the Player's Player Controller script.
* In the **PlayerController** script, **OnTriggerEnter**, when the player picks up the powerup, the indicator object should be enabled:

powerupIndicator.gameObject.SetActive(true);

* And in the **PowerupCountdownRoutine**, when the powerup is lost, the indicator should be disabled

powerupIndicator.gameObject.SetActive(false);

# Make the Powerup Follow the Player

* In the **PlayerController** script, add a class variable to record the starting position of the powerup indicator, as it was set in the Unity editor:

private Vector3 powerUpOffset;

* In the **Start** method, record the powerup indicator starting position:

powerUpOffset = powerupIndicator.transform.position;

* In the **FixedUpdate** method, make the powerup indicator follow the player's position, but offset it by the same amount that it was at the start of the game:

powerupIndicator.transform.position = transform.position + powerUpOffset;

# Create Waves Of Enemies

*When all enemies are pushed off of the board, a new wave of enemies should appear, with one more enemy than we had before.*

* In the **SpawnManager** script, remove the code from the **Start** method, and move it into a new method that will be used to spawn a wave of enemies:

void SpawnEnemyWave(int enemiesToSpawn)

{

for(int i = 0; i < enemiesToSpawn; i++)

{

Instantiate(enemyPrefab, GenerateSpawnPosition(),   
 enemyPrefab.transform.rotation);

}

}

* To test, in the **Start** method, you can simply call the method with the number of desired enemies:

SpawnEnemyWave(3);

# Destroy Enemies When They Fall Off

* In the **Enemy** script, in the **Update** method, write code to check if an enemy is below the platform, and if so, destroy it:

if (transform.position.y < -10)

{

Destroy(gameObject);

}

* In the **SpawnManager** script, we need to keep track of how many enemies we currently have. Add a class variable to track this:

public int enemyCount;

* In the **Update** method, get the count of enemies. We will do this by finding all objects of a certain type (in this case the type == *Enemy*, which is the enemy script attached to each enemy). This will return an array, which we can then check the length of.

enemyCount = GameObject.FindGameObjectsWithTag("Enemy").Length;

* Then, whenever the enemyCount drops to zero, spawn a new wave of enemies:

if (enemyCount == 0)

{

SpawnEnemyWave(3);

}

* Test the game now to see all of this in action

# Increase Enemy Count with Waves

* In the **SpawnManager** script, add a class variable to keep track of which wave the player is on:

public int waveNumber = 1;

* In the **Start** method, modify the starting enemy code:

SpawnEnemyWave(waveNumber);

* And in the **Update** method, increase the wave each time a group is wiped out:

if (enemyCount == 0)

{  
 waveNumber++;

SpawnEnemyWave(waveNumber);

}

* Test the game now to see all of this in action

# Spawn a Powerup with Each Wave

* In the **SpawnManager** script, add a method to spawn a powerup at a random position:

void SpawnPowerup()

{

Instantiate(powerUpPrefab, GenerateSpawnPosition(),  
 powerUpPrefab.transform.rotation);

}

* In the **Start** method, and in the **Update** methods, spawn the powerup

SpawnPowerup();

* In the Unity editor, remove the powerup that's in the Hierarchy

# Stop Spawning When Player Falls Off

* In the **SpawnManager** script, add a class variable for game over:

public bool gameOver = false;

* In the **Update** method, set the gameOver variable by trying to find the player, and then change the IF statement to run only if the game isn't over:

gameOver = !GameObject.Find("Player");  
  
if (enemyCount == 0 && !gameOver)