# Introduction to Unity for Web Developers

Nick Pettit

# Hi, I'm Nick Pettit!

I'm a former teacher at Treehouse, an online technology school.

I've been making video games for the last 8 years.

Now I make games for a living at BUCK, a global creative company that brings brands, stories, and experiences to life through art, design, and technology.

# NEPTUNE FLUX

A DEEP SEA ADVENTURE GAME

3 0 3

5

$68,222


N
0
0
3
$68,222


3
0
1
N
$36,823

# Agenda: We're Going to Make a Game!

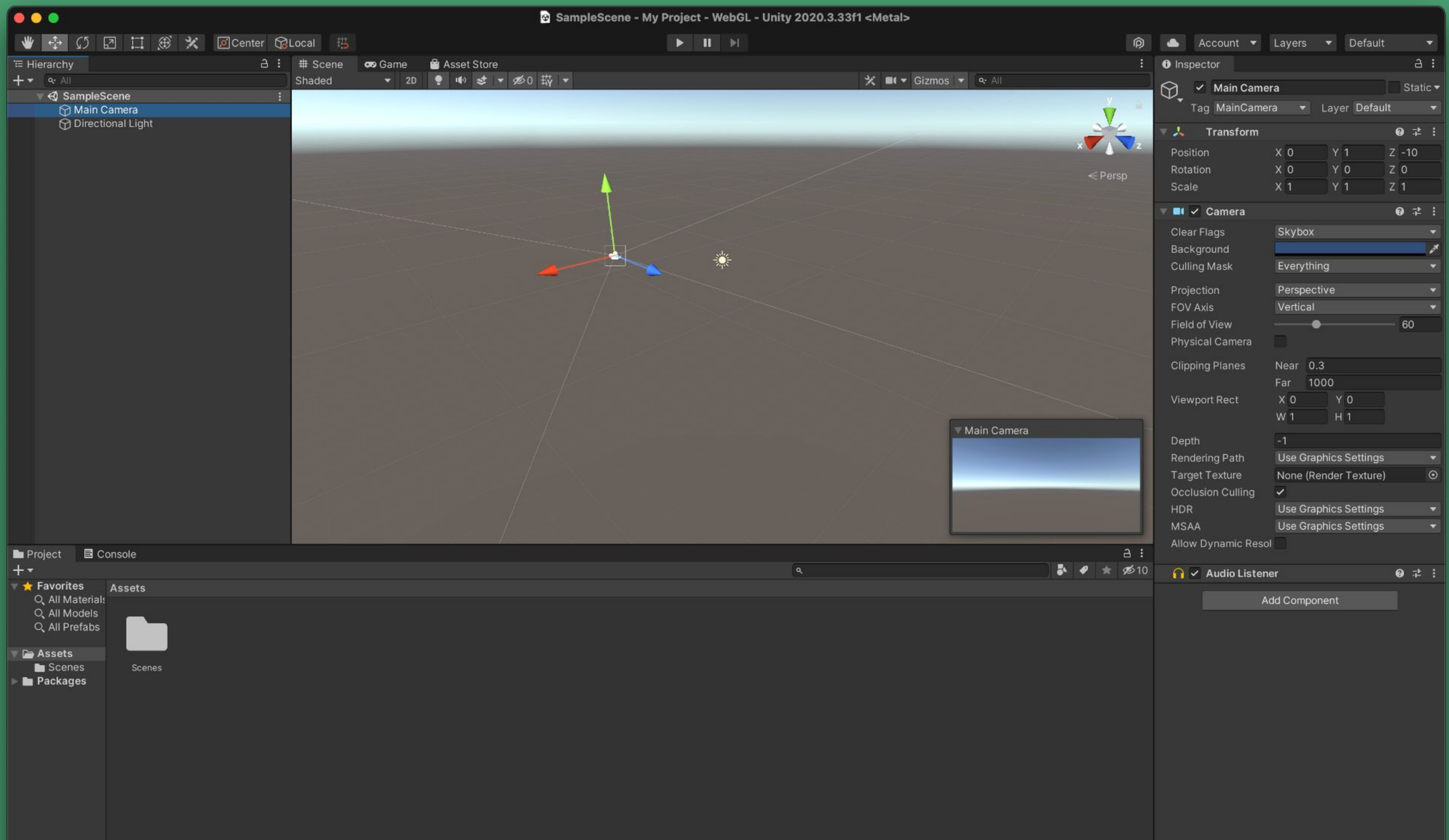
- Install Unity and tour the interface
- Learn about scenes, GameObjects, and components
- Use C# to make a physics sandbox and add player input
- Create a game manager script with a simple game loop
- Add UI to display a score and a timer
- Introduce more complex behavior with composition
- Add materials, lighting, and post-processing
- Deploy the game to a web browser using WebGL

# Game Demo

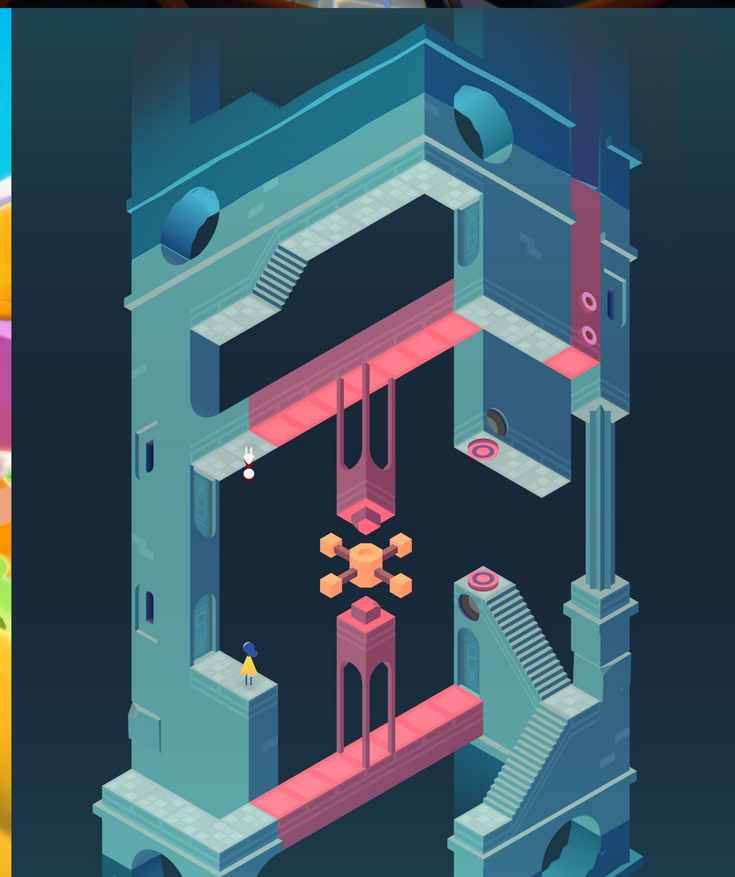
bit.ly/fm-unity

# What is Unity?

# Unity is a game engine!




SUBNAUTICA

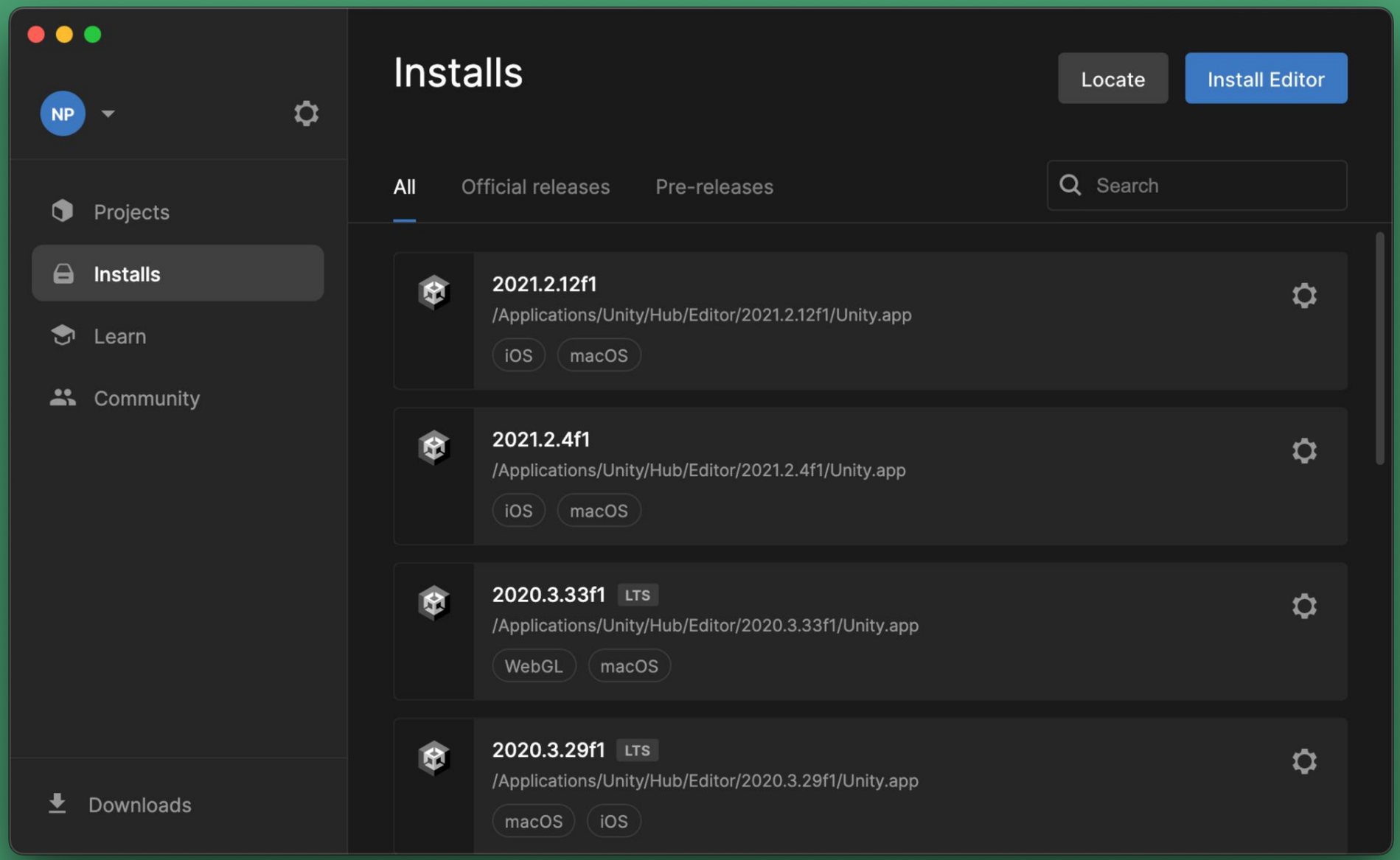


# Additional Applications


- AR/VR Product Demos
- Film and Animation
- Automotives
- Manufacturing
- Architectural Visualization
- Simulations and Training

# Install Unity and Create a New Project

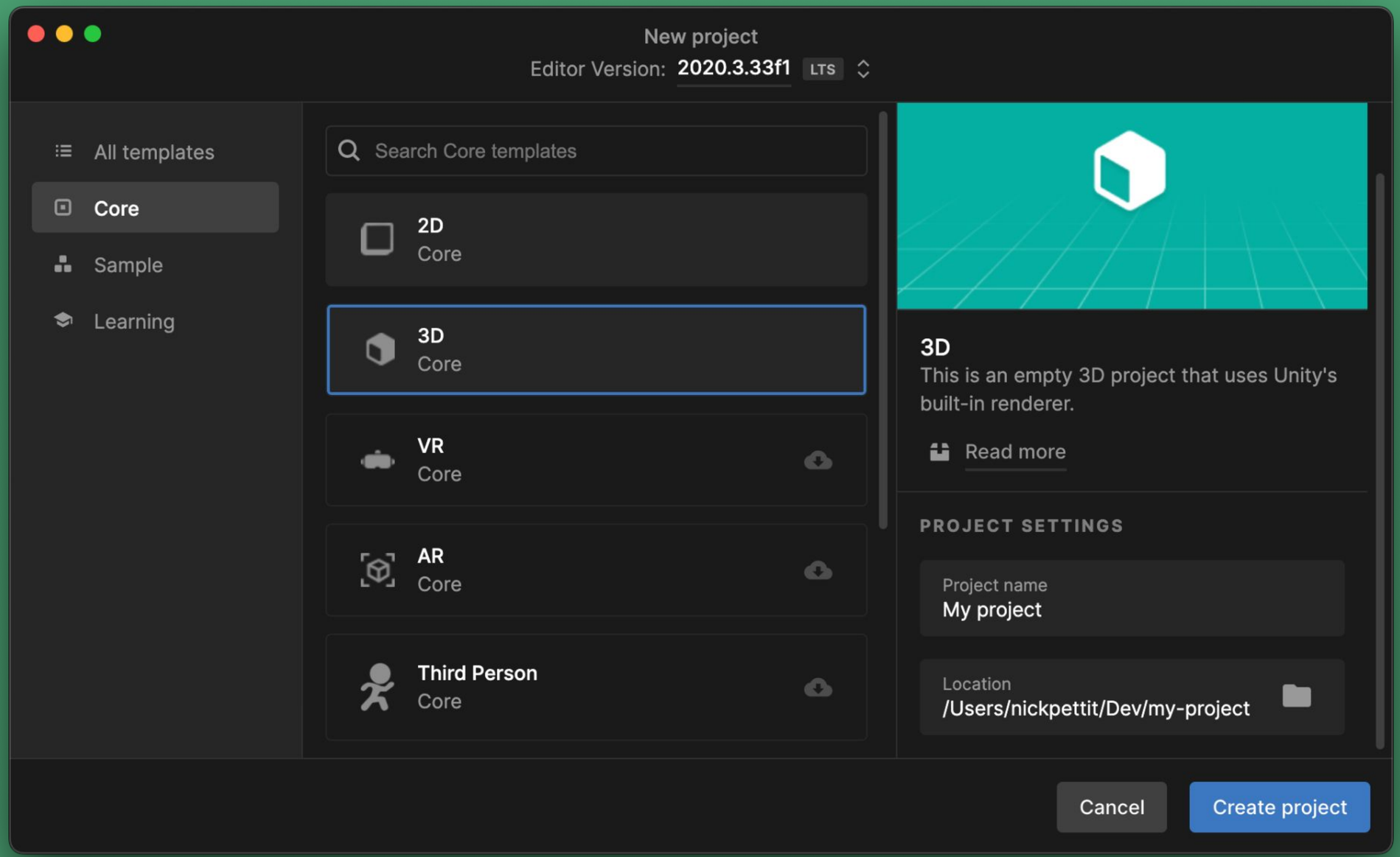
# The Unity Hub



# Installing Unity

- Go to Unity.com and download the Unity Hub
- Personal should be fine, but if you're using this for business, pay attention to the licensing agreements
- Follow the setup instructions for the Hub
- Install the latest "LTS" version of Unity from the Hub
    - LTS stands for Long Term Support and is generally the most stable version available
- When installing Unity, include WebGL support

# Creating a New Project



# Creating a New Project

We're going to open a starter project, but when creating your own, here's generally how the process works (it changes frequently!)

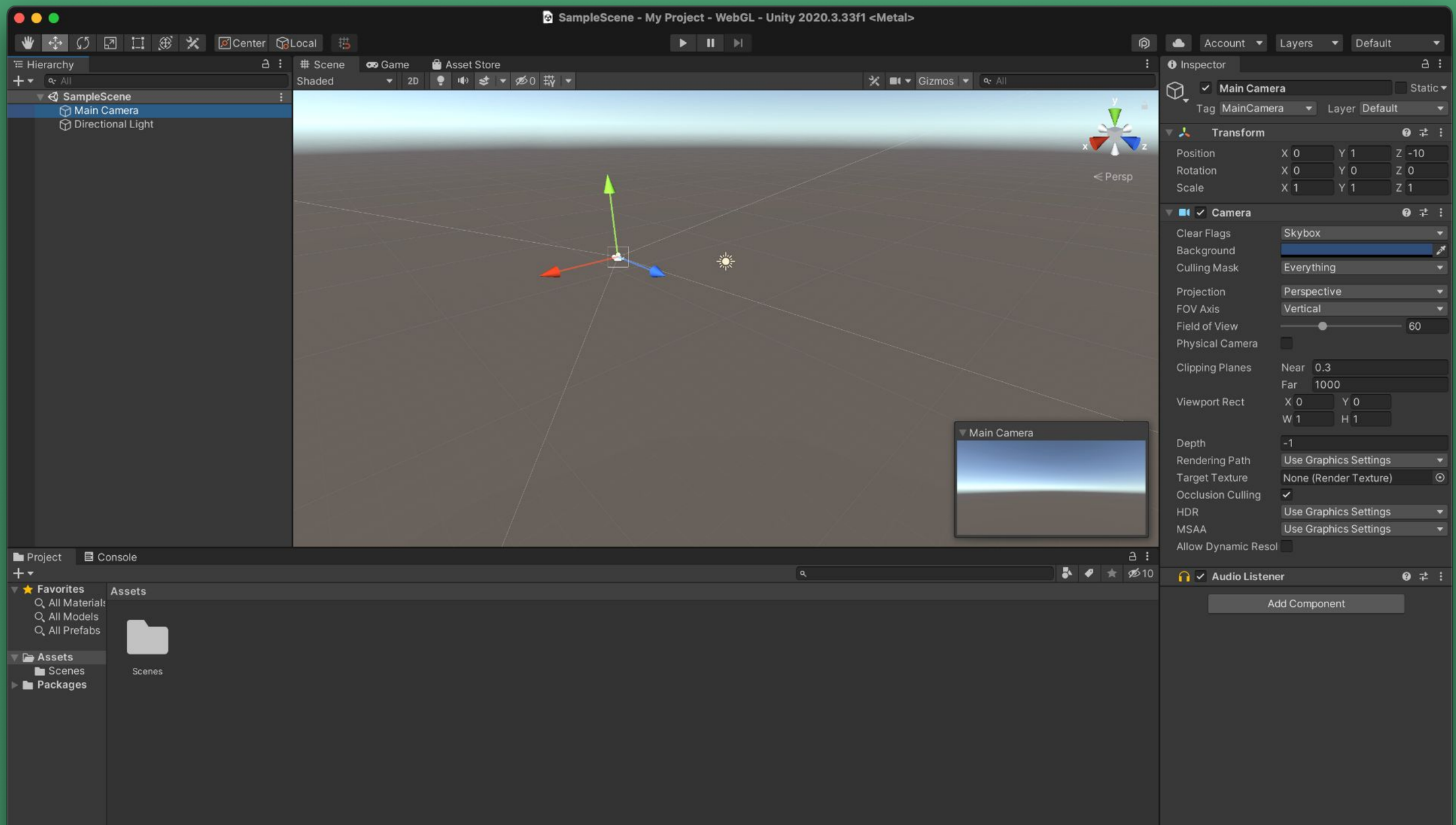
- In the Unity Hub, choose the "New Project" button
- Choose one of the templates, in this case "3D Core"
- Name the project (you can change this later)
- Choose a location (wherever you like to do your development)

# Opening the Starter Project

- Head over to github.com/nickpettit/unity-for-web-developers
- In the Unity Hub's Projects tab, hit the Open button
- Navigate to the "Start" project directory and open it in Unity

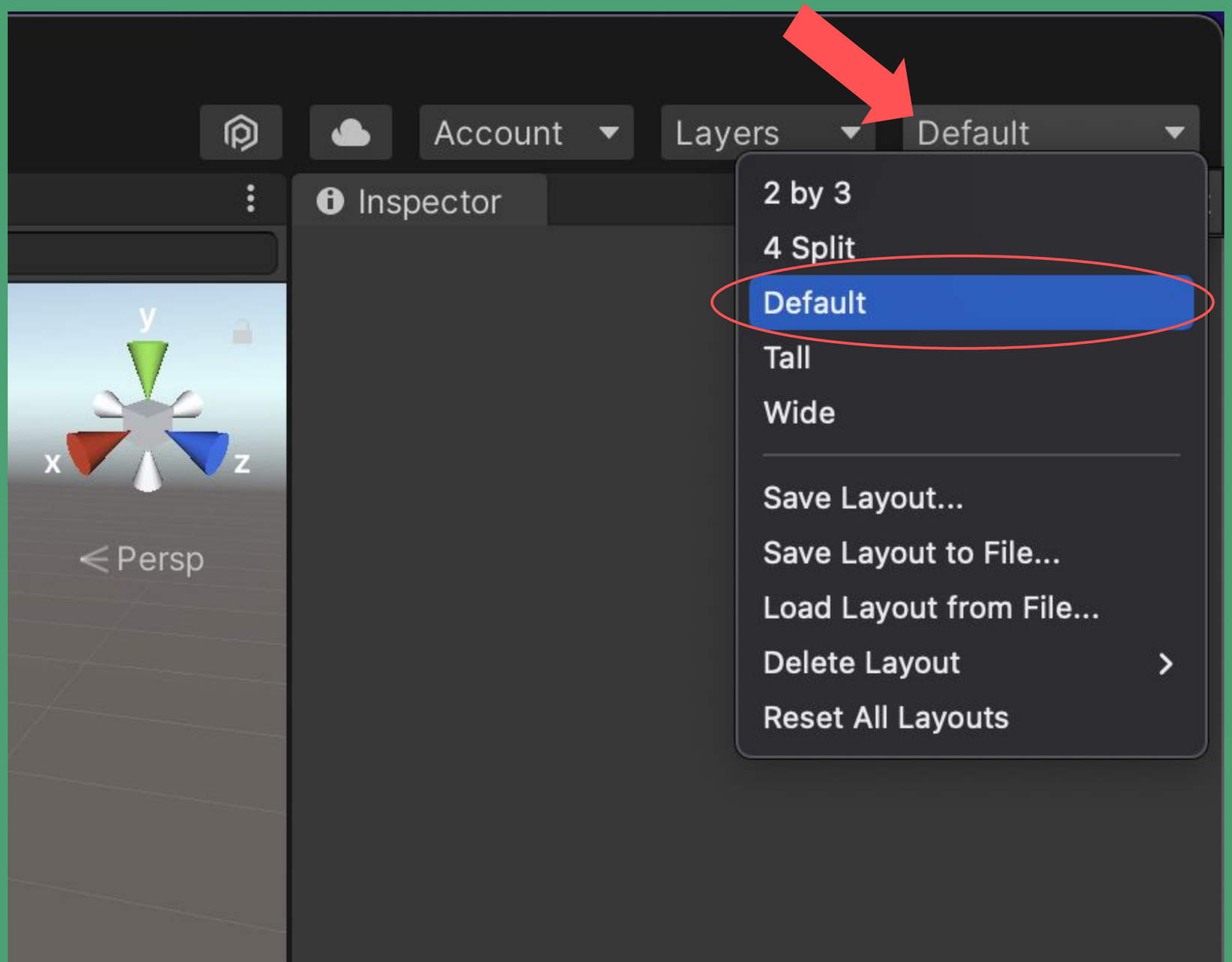
# The Unity Interface

# The Unity Interface



# Interface Tour

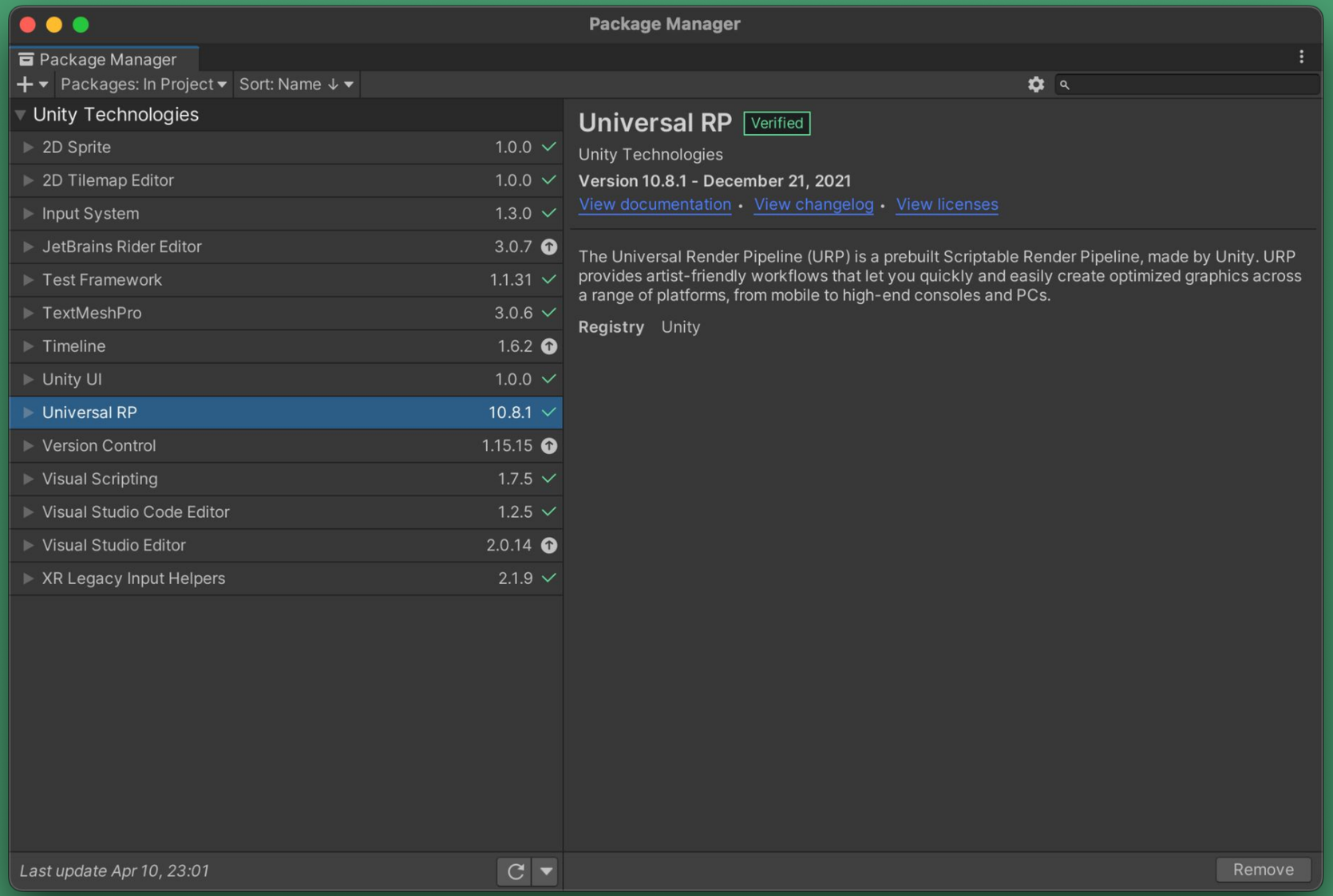
# Tip: Set the Default Layout



# Recap of The Unity Interface

- **Scene View**: 3D view of GameObjects in the current Scene(s)
- **Game View**: Preview of the game camera
- **Hierarchy**: List view of GameObjects in the current Scene(s)
- **Inspector**: Detail-view of currently selected GameObject
- **Project**: File browser for the currently open project
- **Console**: Output from scripts such as debug logging and errors
- **Play Button**: Runs your game in the Unity Editor.

# The Package Manager



# Non-Default Packages in this Project

- **Universal RP**: The basic 3D rendering pipeline suitable for most games.
- **Input System**: The more modern approach to Unity input handling.
- **TextMesh Pro**: The better alternative to Unity's default UI text.

# The Unity Ecosystem

# The Unity Ecosystem

- Unity Manual
  - docs.unity3d.com/Manual
- Scripting API
  - docs.unity3d.com/ScriptReference
- The Asset Store
  - assetstore.unity.com
- Community
  - Blog: blog.unity.com
  - Answers: answers.unity.com
  - Forums: forum.unity.com
- Unity Services (Analytics, Multiplayer, Monetization, etc)
  - Much more at unity.com/solutions/gaming-services

# Scenes, GameObjects, and Components

# Scenes, GameObjects, and Components

## Scenes

Scenes are assets that contain all or part of your game's content. Multiple scenes can be loaded at a time to help split up content (such as in the case of streaming assets). Scenes can also be used for creating multiple "levels" in a game.

## GameObjects

GameObjects are the building blocks for scenes in Unity and a container for components. They always have a Transform component with a position, rotation, and scale that places them within the space of the scene.

## Components

Unity leans toward a component-based architecture. Scripts, that when attached to a GameObject, define the behavior of that GameObject. Components are created via C# scripts and can expose values that can be modified in the Inspector.

Scene

GameObject

Component

Component

GameObject

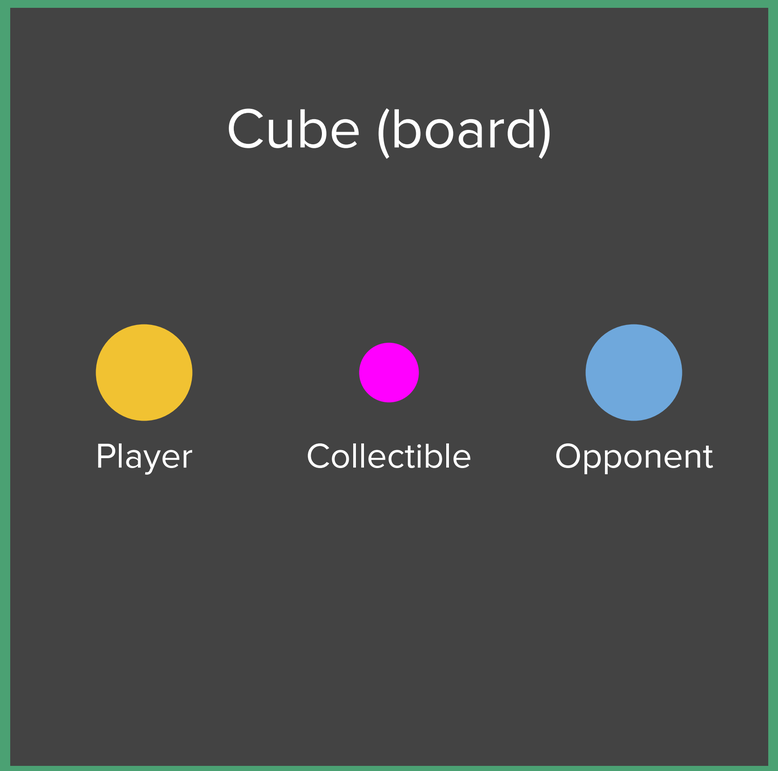
Component

Component

Component

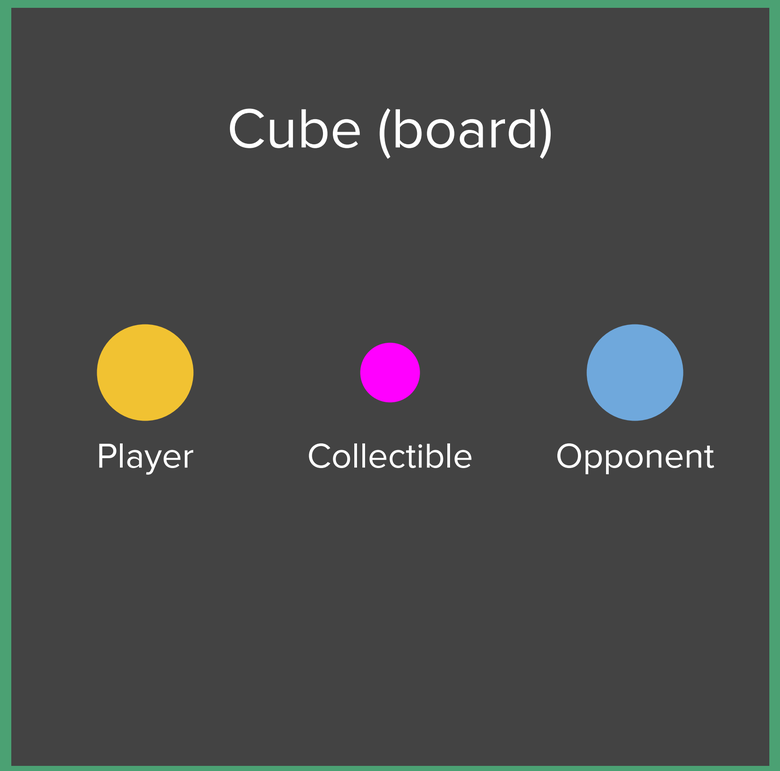
# Scene Setup

(top down view, not to scale)



# Scene Setup

(top down view, not to scale)

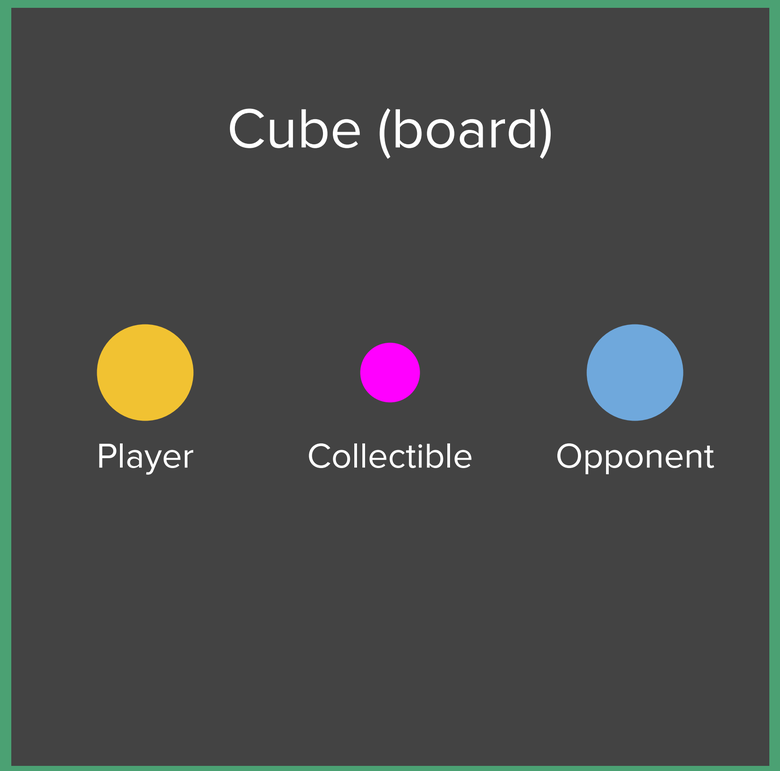



**Values**

- **Board (Cube)**
  - Position: 0, 0.5, 0
- **Player (Sphere)**
  - Position: -0.3, 1.75, 0
  - Scale: 0.1
- **Collectible (Sphere)**
  - Position: 0, 1.05, 0
  - Scale: 0.025
  - Make this a "Trigger" collider

# More Scene Setup

(top down view, not to scale)




**Values**

- **Directional Light**
  - Rotation: 42, -138, -108
  - Intensity: 0.75
  - Near Plane: 0.2

- **Camera**
  - Position: 0, 1.6, -0.6
  - Rotation: 55, 0, 0
  - Post Processing: True
  - Anti-aliasing: SMAA
  - Dithering: True
  - Environment Type: Solid Color (Black)

(we'll make the Opponent and other GameObjects later on)

# Player Input and C# Components

# Adding Player Input

Add these components to the **Player** GameObject:

- **Rigidbody**
    - **Mass**: 0.5
    - **Drag**: 2
    - **Interpolate**: Interpolate
- **Player Input**
    - **Behavior**: Invoke Unity Events
        - (This will make it easier to invoke other scripts.)
- **RollingMovement** (Custom Script Component)
- **PlayerMovement** (Custom Script Component)

# C# Components

## RollingMovement Component

Components modify the behavior of GameObjects in the Scene. This `RollingMovement` component takes a 3D vector, `m_movementDirection`, and applies it as a force to its physics component, the `Rigidbody`, to roll the ball in the vector's direction. The vector has a magnitude of 1, so it is multiplied by `m_speed` to move it faster.

```
using UnityEngine;

public class RollingMovement : MonoBehaviour
{
    [SerializeField] float m_speed = 1f;
    Rigidbody m_rigidbody;
    [HideInInspector] public Vector3 m_movementDirection;
    Vector3 m_startPosition;

    void Start()
    {
        // Gather components and variables that will be needed later
        m_rigidbody = GetComponent<Rigidbody>();
        m_startPosition = transform.position;
    }

    void FixedUpdate()
    {
        // Every physics tick, add a force equal to the movement vector multiplied by speed
        m_rigidbody.AddForce(m_movementDirection * m_speed);
    }

    public void ResetPosition()
    {
        // Reset the velocity to zero and set the position back to start
        m_rigidbody.velocity = Vector3.zero;
        m_rigidbody.angularVelocity = Vector3.zero;
        transform.position = m_startPosition;
    }
}
```

## RollingMovement Component

**Components modify the behavior of GameObjects in the Scene. This `RollingMovement` component takes a 3D vector, `m_movementDirection`, and applies it as a force to its physics component, the `Rigidbody`, to roll the ball in the vector's direction. The vector has a magnitude of 1, so it is multiplied by `m_speed` to move it faster.**

```
using UnityEngine;

public class RollingMovement : MonoBehaviour
{
    [SerializeField]        m_speed = 1f;
    Rigidbody m_rigid
    [HideInInspector] pub       ctor3 m_movementDirection;
    Vector3 m_st

    void Start()
    {
        // Gather components and variables that will be needed later
        m_rigidbody = GetComponent<Rigidbody>();
        m_startPosition = transform.position;
    }

    void FixedUpdate()
    {
        // Every physics tick, add a force equal to the movement vector multiplied by speed
        m_rigidbody.AddForce(m_movementDirection * m_speed);
    }

    public void ResetPosition()
    {
        // Reset the velocity to zero and set the position back to start
        m_rigidbody.velocity = Vector3.zero;
        m_rigidbody.angularVelocity = Vector3.zero;
        transform.position = m_startPosition;
    }
}
```

**The name of the class, `RollingMovement` is also the name of the component.**

# RollingMovement Component

Components modify the behavior of GameObjects in the Scene. This `RollingMovement` component takes a 3D vector, `m_movementDirection`, and applies it as a force to its physics component, the `Rigidbody`, to roll the ball in the vector's direction. The vector has a magnitude of 1, so it is multiplied by `m_speed` to move it faster.

```
using UnityEngine;

public class RollingMovement : MonoBehaviour
{
    [SerializeField] float m_speed
    Rigidbody m_rigidbody;
    [HideInInspector] public Vector3      vementDirection;
    Vector3 m_startPosition;

    void Start()
    {
        // Gather components ar
        m_rigidbody = GetComponent<Rigidbody>();
        m_startPosition = transform.position;
    }

    void FixedUpdate()
    {
        // Every physics tick, add a force equal to the movement vector multiplied by speed
        m_rigidbody.AddForce(m_movementDirection * m_speed);
    }

    public void ResetPosition()
    {
        // Reset the velocity to zero and set the position back to start
        m_rigidbody.velocity = Vector3.zero;
        m_rigidbody.angularVelocity = Vector3.zero;
        transform.position = m_startPosition;
    }
}
```

**All C# scripts that are components are `public` and they inherit from the class `MonoBehaviour`**

## Namespaces

## Variables

## Methods

```
using UnityEngine;

public class RollingMovement : MonoBehaviour
{
    [SerializeField] float m_speed = 1f;
    Rigidbody m_rigidbody;
    [HideInInspector] public Vector3 m_movementDirection;
    Vector3 m_startPosition;

    void Start()
    {
        // Gather components and variables that will be needed later
        m_rigidbody = GetComponent<Rigidbody>();
        m_startPosition = transform.position;
    }

    void FixedUpdate()
    {
        // Every physics tick, add a force equal to the movement vector multiplied by speed
        m_rigidbody.AddForce(m_movementDirection * m_speed);
    }

    public void ResetPosition()
    {
        // Reset the velocity to zero and set the position back to start
        m_rigidbody.velocity = Vector3.zero;
        m_rigidbody.angularVelocity = Vector3.zero;
        transform.position = m_startPosition;
    }
}
```

Let's break down a variable.

```
using UnityEngine;

public class RollingMovement : MonoBehaviour
{
    [SerializeField] float m_speed = 1f;
    Rigidbody m_rigidbody;
    [HideInInspector] public Vector3 m_movementDirection;
    Vector3 m_startPosition;

    void Start()
    {
        // Gather components and variables that will be needed later
        m_rigidbody = GetComponent<Rigidbody>();
        m_startPosition = transform.position;
    }

    void FixedUpdate()
    {
        // Every physics tick, add a force equal to the movement vector multiplied by speed
        m_rigidbody.AddForce(m_movementDirection * m_speed);
    }

    public void ResetPosition()
    {
        // Reset the velocity to zero and set the position back to start
        m_rigidbody.velocity = Vector3.zero;
        m_rigidbody.angularVelocity = Vector3.zero;
        transform.position = m_startPosition;
    }
}
```

| Attribute(s) | Type | Name | Value |
| --- | --- | --- | --- |
| `[SerializeField]` | `float` | `m_speed` | `= 1f;` |

If there is no access modifier,
then the variable is `private`

**Attributes are markers that can be placed above a class, property or function in a script to indicate special behavior. They are placed in square brackets.**

`[SerializeField]` **forces Unity to "serialize" a private variable and show it in the Inspector for modification. All public variables are serialized by default. Serialization is the process of transforming data structures or GameObject states into a format that Unity can store and reconstruct later.**

`[HideInInspector]` **will hide a public variable from showing in the Inspector, since public variables are serialized by default.**

# MonoBehaviour Methods

`Start()` and `FixedUpdate()` are two of the methods on the `MonoBehaviour` class, among several others.

`Start()` is called on the frame when a script is enabled (such as when you hit the Play button in the editor).

`FixedUpdate()` is a framerate independent method that is called every physics update (typically 50 times per second by default). Generally speaking, any physics calculations that need to be updated as the game is playing should be done in `FixedUpdate()` so that they don't fall out of sync with one another. This is contrast to `Update()` which is called *every frame* and is therefore NOT framerate independent.

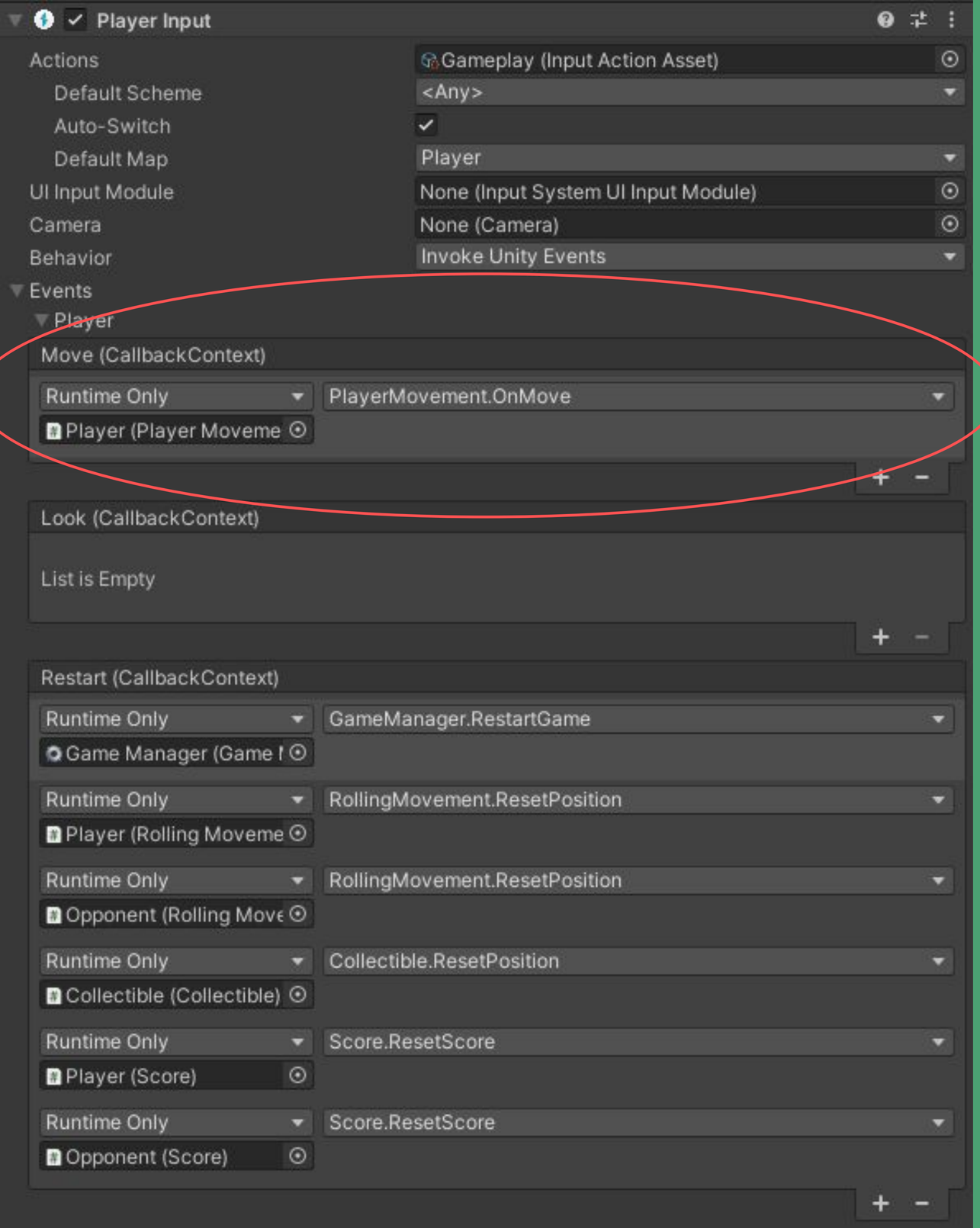
```
using UnityEngine;

public class RollingMovement : MonoBehaviour
{
    [SerializeField] float m_speed = 1f;
    Rigidbody m_rigidbody;
    [HideInInspector] public Vector3 m_movementDirection;
    Vector3 m_startPosition;

    void Start()
    {
        // Gather components and variables that will be needed later
        m_rigidbody = GetComponent<Rigidbody>();
        m_startPosition = transform.position;
    }

    void FixedUpdate()
    {
        // Every physics tick, add a force equal to the movement vector multiplied by speed
        m_rigidbody.AddForce(m_movementDirection * m_speed);
    }

    public void ResetPosition()
    {
        // Reset the velocity to zero and set the position back to start
        m_rigidbody.velocity = Vector3.zero;
        m_rigidbody.angularVelocity = Vector3.zero;
        transform.position = m_startPosition;
    }
}
```

# Let's Code

## PlayerMovement

`PlayerMovement` reads directional input from the player in `OnMove()` and applies it to a `RollingMovement` component on the same GameObject.

```
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
    RollingMovement m_rollingMovement;

    void Start()
    {
        // Gather components
        m_rollingMovement = GetComponent<RollingMovement>();
    }

    public void OnMove(InputAction.CallbackContext context)
    {
        // Get the movement vector and apply it to the RollingMovement component
        Vector2 movement = context.ReadValue<Vector2>();
        m_rollingMovement.m_movementDirection = new Vector3(movement.x, 0f, movement.y);
    }
}
```




## Player Input Events

In order to accept input, we need to invoke the callback method for the Move event on the Player Input component.

- **Move (Callback Context)**
  - PlayerMovement.OnMove

(we'll make the rest of the events later on)

# Exercise

# Exercise: Score

`Score` is a simple component that will keep track of each player's individual score. Later we'll update it to interact with UI.

We'll use a C# property to store the score as an integer type. That way the value can be accessed publicly but only modified privately by the script.

Write a method to reset this value, and another method to increment the value.

```
using UnityEngine;

public class Score : MonoBehaviour
{
    public int ScoreValue { get; private set; }

    public void ResetScore()
    {
        // ...
    }

    public void IncrementScore()
    {
        // ...
    }
}
```

# Solution

# Solution: Score

**(And yes, it will get more challenging.)**

```
using UnityEngine;

public class Score : MonoBehaviour
{
    public int ScoreValue { get; private set; }

    public void ResetScore()
    {
        ScoreValue = 0;
    }

    public void IncrementScore()
    {
        ScoreValue++;
    }
}
```

# Collectibles

## Collectible

**When another GameObject with a `Score` component physically intersects with a `Collectible`, it calls the `IncrementScore()` method on the intersecting `Score` object and then moves its position to a new location within the game board.**

**Admittedly, spawning the collectible to a new random position on the game board will sometimes result in spawns that are directly next to one of the players, which isn't great! One approach to improve this might be to have several predefined spawn points and always choose a spawn point that's furthest away from all of the players.**

```
using UnityEngine;

public class Collectible : MonoBehaviour
{
    float range = 0.4f;
    Vector3 m_startPosition;

    void Start()
    {
        m_startPosition = transform.position;
    }

    void OnTriggerStay(Collider other)
    {
        // If the other GameObject has a Score component, increment it
        other.gameObject.GetComponent<Score>().IncrementScore();

        // Move the position to a new random position within the board bounds
        transform.position = new Vector3(Random.Range(-range, range),
                                         transform.position.y,
                                         Random.Range(-range, range));
    }

    public void ResetPosition()
    {
        transform.position = m_startPosition;
    }
}
```

# Game Manager

# What is a Game Manager?

Generally speaking, a "game manager" is a de facto architectural approach to making games in Unity where a script component, called GameManager, is used to control high level aspects of the game's loop.

This might include game and level initialization, along with other state changes such as a "game over" state or restarting the game. A game manager might also track important game-wide numbers like scores and round timers.

## GameManager

**First, let's set up the variables we'll need to keep track of and set their values.**

**This includes:**

- **Player's score**
- **Opponent's score**
- **Amount of time for each round**
- **Remaining time in each round**
- **A boolean to track game state**
- **Global values for use by other scripts**

**At the start, we'll set the time scale to zero to pause the game, set the game clock, and then update the timer for the first time.**

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    [SerializeField] Score m_playerScore;
    [SerializeField] Score m_opponentScore;
    [SerializeField] float m_gameTimeInSeconds = 60f;
    float m_timer;
    bool m_gameIsPlaying = false;
    public static float RespawnHeight = 0.25f;

    void Start()
    {
        // Set the time scale to zero to pause the game at the start
        Time.timeScale = 0f;

        // Set the game timer to number of seconds in a round
        m_timer = m_gameTimeInSeconds;

        // Update the timer text
        UpdateTimer();
    }

   // More methods here...
}
```

# GameManager

**Next, every physics update we need to update the timer. And if the timer runs out, we should end the game.**

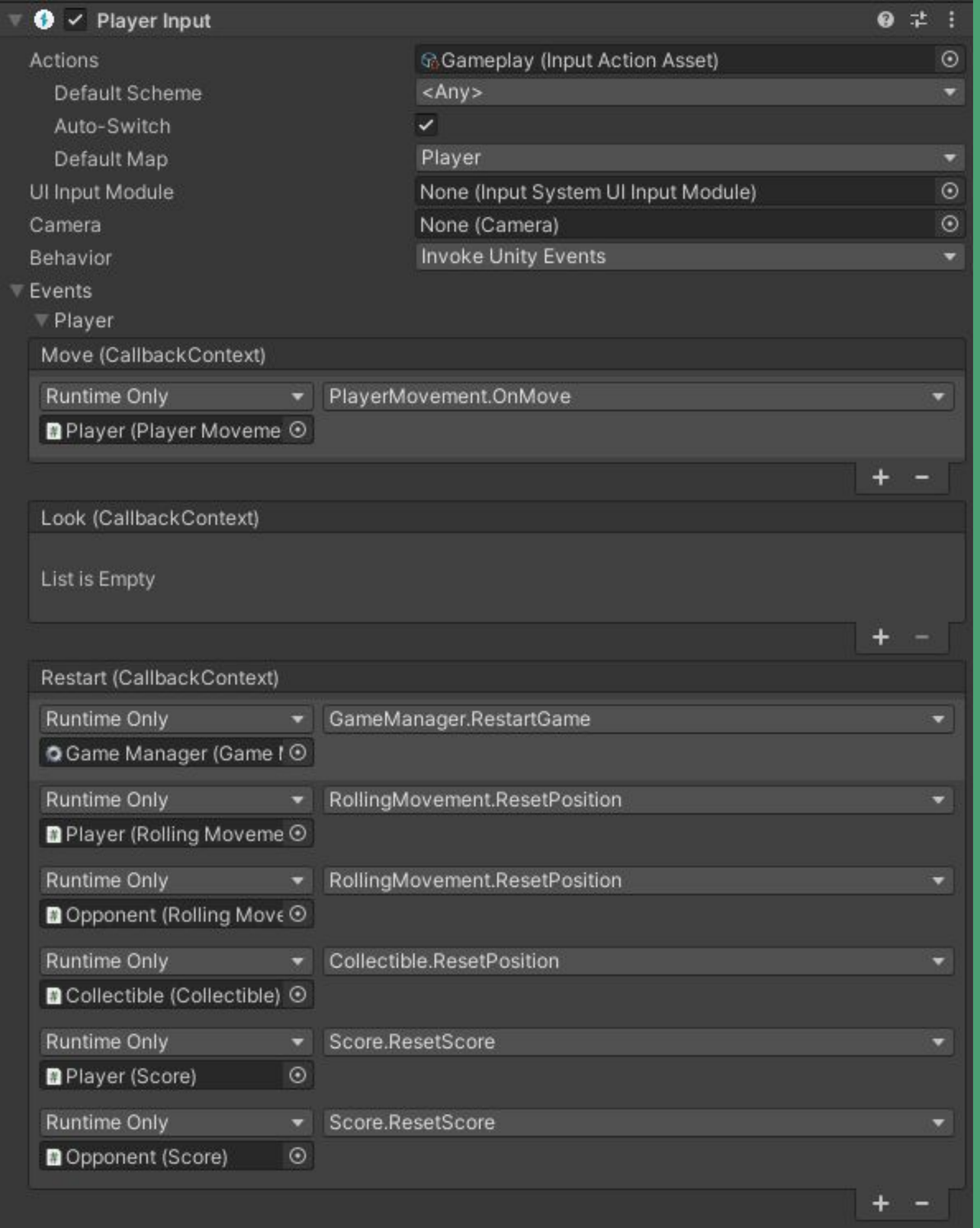
```
void FixedUpdate()
{
    // If the game isn't playing, stop here
    if (!m_gameIsPlaying) return;

    // Subtract the fixed delta time from the timer
    m_timer -= Time.fixedDeltaTime;

    // If the timer hits zero, enter the game over state
    if (m_timer <= 0)
    {
        GameOver();
        m_timer = 0;
    }
}
```

```
void GameOver()
{
    m_gameIsPlaying = false;

    // Pause the game by setting the time scale to zero
    Time.timeScale = 0f;
}
```

## GameManager

**Finally, we need a method to restart the game once it's over.**

```
public void RestartGame()
{
    if (!m_gameIsPlaying)
    {
        // Unpause the game by setting the timescale to 1
        Time.timeScale = 1f;

        // Reset the game timer
        m_timer = m_gameTimeInSeconds;
        m_gameIsPlaying = true;
    }
}
```




## Player Input Events

Now that we have all our callback methods in place, we should connect them to the remaining input events.

- **Move (Callback Context)**
  - PlayerMovement.OnMove
- **Restart (CallbackContext)**
  - GameManager.RestartGame
  - RollingMovement.ResetPosition (Player)
  - RollingMovement.ResetPosition (Opponent)
  - Collectible.ResetPosition
  - Score.ResetScore (Player)
  - Score.ResetScore (Opponent)

# Exercise

# Exercise: Add Y-height check to RollingMovement

**If the spheres fall off the side of the board, we should reset the position.**

**In the GameManager, we created a globally accessible static variable to hold what that `RespawnHeight` should be. Use it to reset the position.**

```
// ...

void FixedUpdate()
{
    // Every physics tick, add a force equal to the movement vector multiplied by speed
    m_rigidbody.AddForce(m_movementDirection * m_speed);

    // If the Y position of the sphere is below the respawn height, reset its position
}

// ...
```

# Solution

## Solution: Add Y-height check to RollingMovement

```
// ...

void FixedUpdate()
{
    // Every physics tick, add a force equal to the movement vector multiplied by speed
    m_rigidbody.AddForce(m_movementDirection * m_speed);

    // If the Y position of the sphere is below the respawn height, reset its position
    if (transform.position.y <= GameManager.RespawnHeight)
        ResetPosition();
}

// ...
```

# Unity UI

# Unity UI

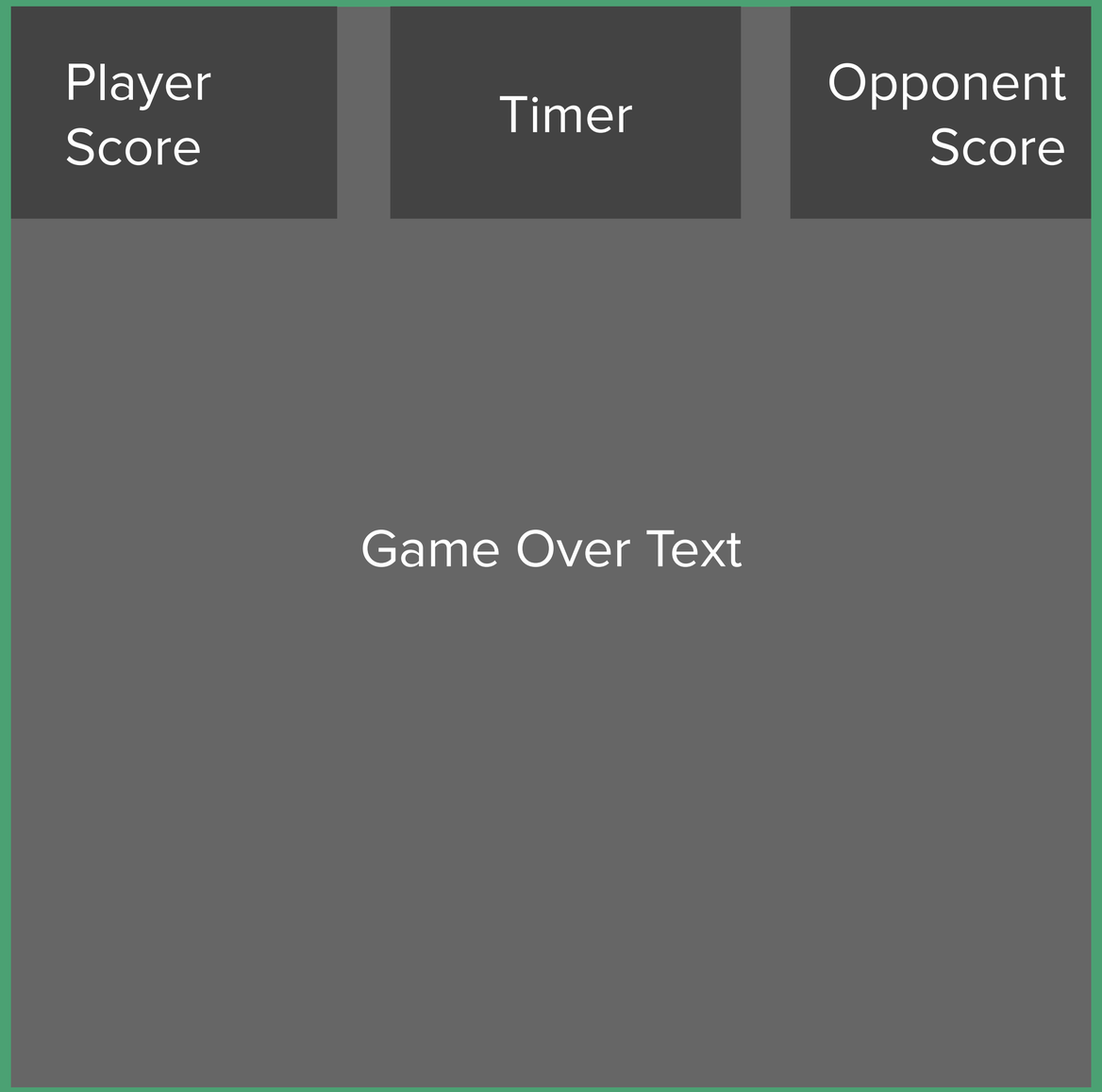
In Unity, UI exists as 2D sprite-base graphics with a Canvas component as the parent GameObject. A canvas can have one of three render modes:

- **Screen Space - Overlay:** Renders UI elements on top of the scene.
- **Screen Space - Camera:** Renders UI elements at a given distance (in 3D space) in front of the camera, which can be useful for adding depth and perspective effects.
- **World Space:** Renders UI elements as if they were 2D "billboards" in the scene. This is useful for things like health bars or speech bubbles above 3D characters.

For this game, we'll use "Screen Space - Overlay" but it's possible to achieve a wide variety of effects using the other render modes.

# UI Setup

(front view, not to scale)

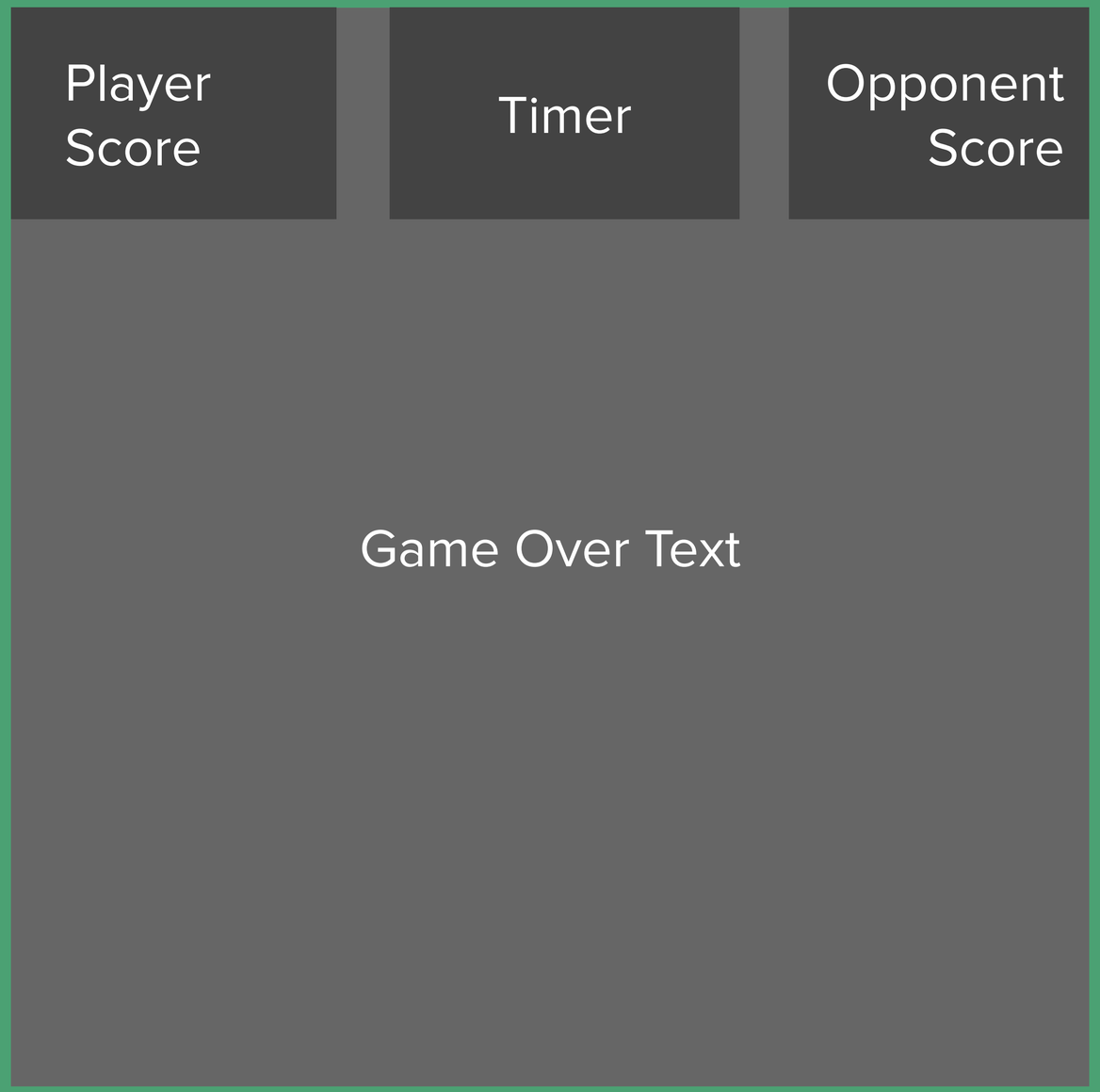



**Values**

- **All Top Row Items**
  - Anchored Top Center
  - Pos Y: -50
  - Width: 200
  - Height: 100
  - Font: Oswald Bold
  - Auto Size: True
- **Timer**
  - Pos X: 0
  - Color: #FFFFFF
- **Player Score**
  - Pos X: -200
  - Color: #FFCE00
- **Opponent Score**
  - Pos X: 200
  - Color: #45A3FF

# UI Setup

(front view, not to scale)




**Values**

- **Game Over Text**
  - Centered and Stretched
  - Text: "Collect more cubes than your opponent! Press SPACE to Start"

## Score

We need to display the score to the player. To do this, we'll add a serialized field for a `TextMeshProUGUI` object from the `TMPro` namespace, which we'll then hook up in the editor to the TextMeshPro UI objects we created. Then we'll update the string every `FixedUpdate()`

```
using UnityEngine;
using TMPro;

public class Score : MonoBehaviour
{
    [SerializeField] TextMeshProUGUI m_scoreText;
    public int ScoreValue { get; private set; }

    void FixedUpdate()
    {
        // Every fixed update, update the score string
        m_scoreText.text = ScoreValue.ToString();
    }

    public void ResetScore()
    {
        ScoreValue = 0;
    }

    public void IncrementScore()
    {
        ScoreValue++;
    }
}
```

## GameManager Updates

**Now that we have a timer and game over text, we should update those values from the `GameManager`.**

```
using UnityEngine;
using TMPro;

public class GameManager : MonoBehaviour
{
    [SerializeField] TextMeshProUGUI m_gameOverText;
    [SerializeField] TextMeshProUGUI m_timerText;
    [SerializeField] Score m_playerScore;
    [SerializeField] Score m_opponentScore;
    [SerializeField] float m_gameTimeInSeconds = 60f;
    float m_timer;
    bool m_gameIsPlaying = false;
    public static float RespawnHeight = 0.25f;

    void Start()
    {
        // Set the time scale to zero to pause the game at the start
        Time.timeScale = 0f;

        // Set the game timer to number of seconds in a round
        m_timer = m_gameTimeInSeconds;

        // Update the timer text
        UpdateTimer();
    }

    // More methods here...
}
```

## GameManager Updates

**The `UpdateTimer()` method formats the float value of the timer into a more typical human readable timer format with minutes and seconds.**

```
void UpdateTimer()
{
    // {0:0}:{1:00}
    // first number (0 and 1) is the index
    // : separates the index and the formatting
    // the 0 are leading zeros
    m_timerText.text = string.Format("{0:0}:{1:00}",
                                     Mathf.Floor(m_timer / 60),
                                     Mathf.Floor(m_timer % 60));
}
```

## GameManager Updates

**In the `GameOver()` method we need to update the game over text based on the final scores.**

**Additionally, we should set the GameObject that contains the text to be active, because restarting the game should deactivate it.**

```
void GameOver()
{
    m_gameIsPlaying = false;

    // Pause the game by setting the time scale to zero
    Time.timeScale = 0f;

    // Display the end game text
    string gameovertext;
    if (m_playerScore.ScoreValue > m_opponentScore.ScoreValue)
        gameovertext = "YOU WIN!";
    else if (m_playerScore.ScoreValue < m_opponentScore.ScoreValue)
        gameovertext = "YOU LOSE!";
    else
        gameovertext = "TIE!";

    m_gameOverText.text = gameovertext + "\n\nPress SPACE to Restart";

    // Show the game over text once it has been set
    m_gameOverText.gameObject.SetActive(true);
}
```

# GameManager Updates

**Finally, if the game is restarting, we should hide the game over game text.**

```
public void RestartGame()
    {
        if (!m_gameIsPlaying)
        {
            // Unpause the game by setting the timescale to 1
            Time.timeScale = 1f;

            // Hide the end game text
            m_gameOverText.gameObject.SetActive(false);

            // Reset the game timer
            m_timer = m_gameTimeInSeconds;
            m_gameIsPlaying = true;
        }
    }
```

# Component Composition

# FollowTarget

**Unity's component-based architecture leans into "composition" rather than other approaches to diversification, such as object-oriented inheritance. While C# allows for inheritance and it can be useful in Unity projects, thinking in terms of GameObjects that use several components tends to lead toward more variety and emergent behavior.**

**We've actually already done this. Similar to the `PlayerMovement` script, we can create a `FollowTarget` script that automatically drives the existing `RollingMovement` script, creating the appearance of a simple opponent AI.**

```
using UnityEngine;

public class FollowTarget : MonoBehaviour
{
    [SerializeField] Transform m_target;
    [SerializeField] float m_retargetingSpeed = 2f;

    RollingMovement m_rollingMovement;

    void Start()
    {
        // Gather components
        m_rollingMovement = GetComponent<RollingMovement>();
    }

    void FixedUpdate()
    {
        // Point the target direction vector toward the target
        Vector3 m_targetDirection = m_target.position - transform.position;

        // If the sphere is close, set the retarget speed to be high
        // Otherwise, use the typical retargeting speed
        float retargetSpeed = Vector3.SqrMagnitude(m_targetDirection) < 0.1f ? 1000f : m_retargetingSpeed;

        // Linearly interpolate the movement vector from the current direction to the target
        // where T is fixed delta time multiplied by the retargeting speed
        m_rollingMovement.m_movementDirection = Vector3.Lerp(m_rollingMovement.m_movementDirection,
                                                             m_targetDirection.normalized,
                                                             Time.fixedDeltaTime * retargetSpeed);
    }
}
```

# Exercise

## Exercise: CameraMovement

**Using what we've learned so far, try to stretch and create a script component called `CameraMovement` that attaches to the camera GameObject and moves the camera gently to frame the action.**

**Hints:**

- **You should use `Vector3.Lerp()` inside of `Update()` to move the camera's position every frame.**
- **You'll probably want to calculate the camera's position by offsetting it by some distance from the board.**

```
using UnityEngine;

public class CameraMovement : MonoBehaviour
{
    [SerializeField] Transform m_target;
    [SerializeField] Vector3 m_boardCenter;
    Vector3 m_offset;

    void Start()
    {
        // Calculate the offset at the start
    }

    void Update()
    {
        // Set the movement target to the target's position plus an X and Z offset
        // Linearly interpolate the position from the current position to the target
    }
}
```

# Solution

# Solution: CameraMovement

```
using UnityEngine;

public class CameraMovement : MonoBehaviour
{
    [SerializeField] Transform m_target;
    [SerializeField] Vector3 m_boardCenter;
    Vector3 m_offset;

    void Start()
    {
        // Calculate the offset at the start
        m_offset = transform.position - m_boardCenter;
    }

    void Update()
    {
        // Set the movement target to the target's position plus an X and Z offset
        Vector3 target = new Vector3(m_target.position.x + m_offset.x,
                                     transform.position.y,
                                     m_target.position.z + m_offset.z);

        // Linearly interpolate the position from the current position to the target
        transform.position = Vector3.Lerp(transform.position,
                                          target,
                                          Time.deltaTime);
    }
}
```

# Materials

# What is a Material?

A Material is an asset that combines both a shader and a set of textures and other values to partially determine how a 3D mesh should be rendered.

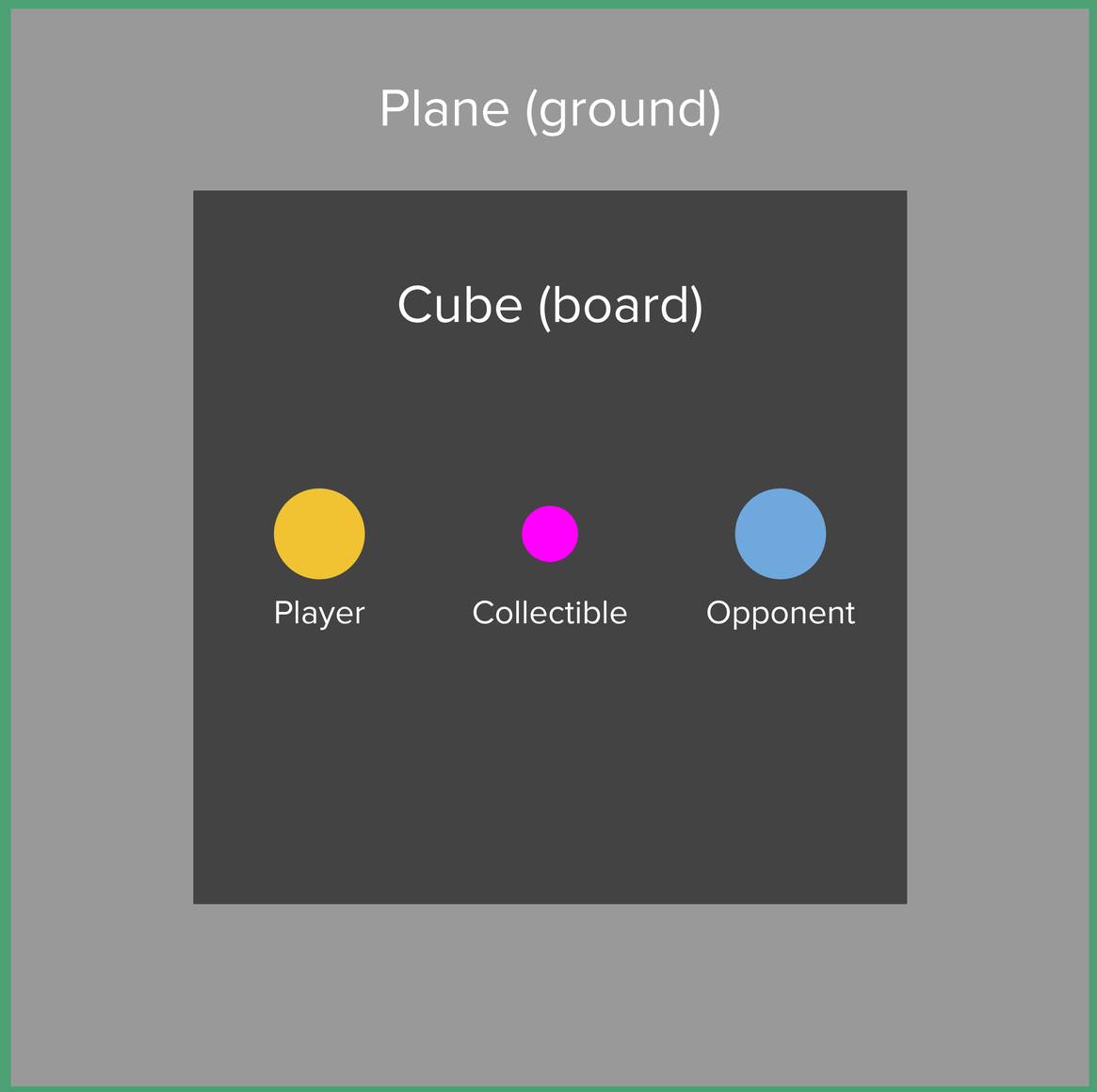
A shader is a small program that runs on the GPU and generally shades each pixel, although it can also modify the vertices of 3D meshes.

Textures are 2D images that get applied to the 3D surface of a model, and it's the shader that helps determine *how* those textures are applied.

A material combines all of these elements into a more easily understandable asset, such as a "Gold" material for anything in the game that we want to look like metallic gold surface.

# More Scene Setup

(top down view, not to scale)




**Values**

- **Ground (Plane)**
  - Position: 0, 0, 0

# Lighting

# What is a Light?

Lights are a bit more intuitive. Just like in the real world, lights in Unity contribute to how surfaces are rendered. The shaders in each material are able to access lighting data and make parts of a surface brighter while shading other parts darker, depending on the parameters of each light source in the scene, such as its position, intensity, light type, color, and so forth.

# Reflection Probes

Reflections in computer graphics are costly, so real-time environments like games tend to rely on precomputed data rather than calculating reflections every frame.

A reflection probe captures a spherical view of its surroundings and then any shaders in its area of influence can use the image data to generate convincing reflections.

# Reflection Probe Setup

(top view, not to scale)


Area of Influence

**Values**

- **Reflection Probe**
  - Position: 0, 1.5, 0
  - Box Projection: True
  - Resolution: 512
  - Shadow Distance: 1000

Then hit the "Bake" button!

# Post Processing

# Post Processing

Post Processing can improve the look of a game by applying full-screen effects after a frame has been rendered. Think of these like Photoshop adjustments or Instagram filters. Many effects are intended to simulate the properties of a real world camera. Some post processing effects include:

- **Bloom**: Applies a soft blur to some of the brighter parts of the image.
- **Tonemapping**: Remaps HDR images into LDR space for your screen.
- **Chromatic Aberration**: Adds subtle color shifting toward the edges of the camera lens to simulate the breakdown of light into component parts.
- **Vignette**: Darkens the edges of the image to help highlight the center.
- **Panini Projection**: Warps the view similar to a camera lens.

# Deploy to WebGL

# PreBuildProcessing

**Creating WebGL builds in Unity requires Python. If you're on Windows or an older version of macOS, you should be able to skip this step.**

**However, if you're on macOS Monterey (and possibly later), you'll need to add this build preprocessor that points to the location of Python3, since macOS Monterey doesn't include Python2 anymore.**

```
#if UNITY_EDITOR_OSX
using UnityEditor.Build;
using UnityEditor.Build.Reporting;

public class PreBuildProcessing : IPreprocessBuildWithReport
{
    public int callbackOrder => 1;

    public void OnPreprocessBuild(BuildReport report)
    {
        // This should be the path to your Python installation. Python 2 or 3 should work.
        // This should only be necessary on macOS Monterey, since it doesn't include Python 2 anymore.
        System.Environment.SetEnvironmentVariable("EMSDK_PYTHON", "/usr/bin/python3");
    }
}
#endif
```

# Let's Build!

# Key Ideas

- Unity is great for making games, but can also be used for simulations, product demos, AR and VR applications, and much more.

- Deploy your projects to a wide variety of platforms! Unity allows you to write your game once and then easily "port" it everywhere.

- Think in terms of Unity's component-based architecture and try to favor composition over other approaches to code diversification, like inheritance.

- Leverage the community, asset store, and documentation. Presently, this is the true advantage of Unity over other game engines.

- Game development is a massive and highly interdisciplinary topic. Furthermore, Unity is a massive application and ecosystem. This was a "golden path" for creating a very simple game, but as you can imagine, there's lots more to learn and explore!

# Questions?

# Thanks!