02 - Introduction to Unity

CS 3160 - Game Programming

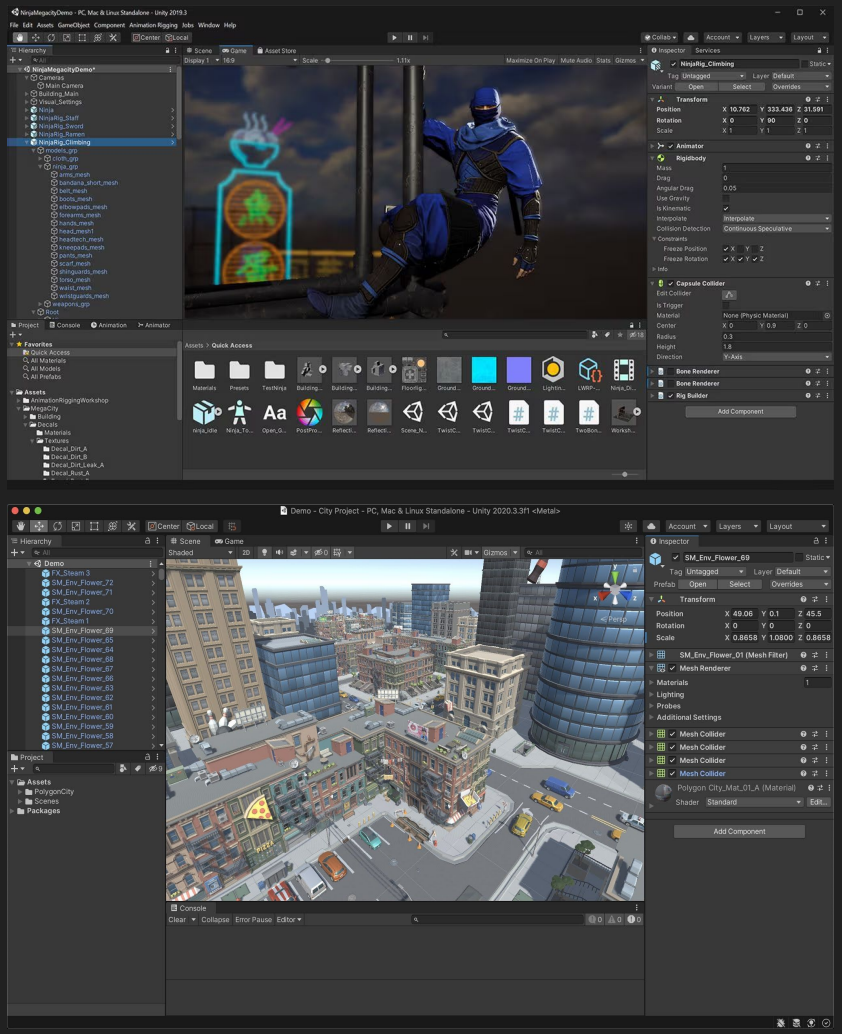
Max Gilson

Game Engines

- Many game engines are available to freely play around with:
  - Unity (indie games, mobile games)
  - Unreal (high end games, PC games, console games)
  - Godot (indie games, open source)
  - GameMaker (easy to learn, 2D games)
  - Source engine (hard to get license for)
  - Create your own (hard, rewarding)

Why Use Unity?

- We will use Unity in this course
  - Unity is easy to learn
  - Unity is an industry standard
  - Unity is affordable for small teams
  - Unity is featureful enough for large teams
  - We need a common foundation for our assignments and knowledge
- Unity is not always the best solution for everything but it’s good enough for our course



Still Not Convinced?

- Unity is used by many popular games that you might recognize

Unity Game Engine

- Unity comes packed with many tools that make it easy to create games:
  - Editor (visualizing changes as you edit)
  - 3D/2D renderer
  - Physics system
  - AI navigation
  - Profiler (for checking performance)
  - And much much more!

Unity Game Engine (cont.)

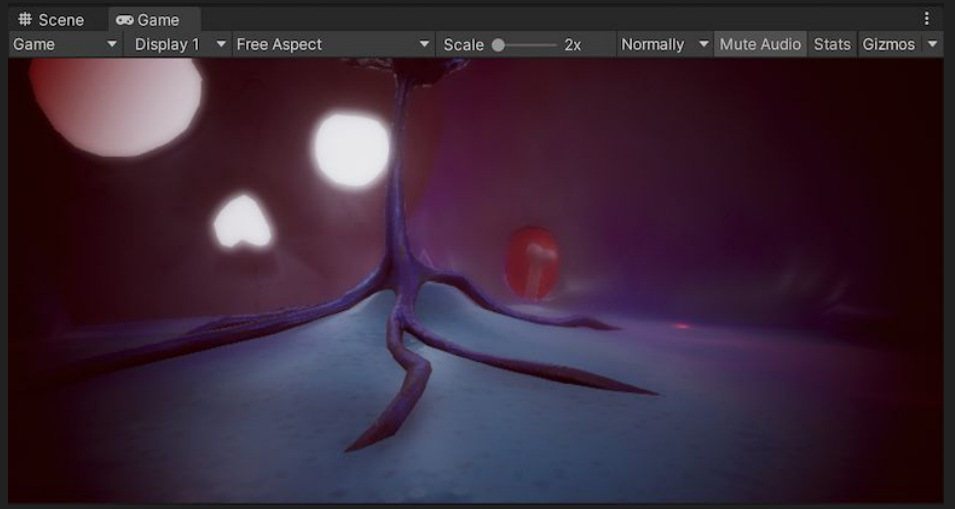
- Download and Install Unity Hub: https://unity.com/download
- Create a Unity ID (account)
- Install Unity Editor version 2022.3.61f1 LTS
  - Select “WebGL Build Support” only
  - Takes a long long time
  - Slow computers may struggle running the Unity Editor
- Create a new Unity project with the “3D (Built-In Render Pipeline)” template
  - Use the name “YourLastName-CS3160”
- It is recommended that you use Visual Studio Community (free) for your IDE and code editor
- The following set of slides are heavily inspired by this guide: https://www.youtube.com/watch?v=pwZpJzpE2lQ

Unity Editor

- The Unity Editor allows us to add/modify things in our game
- In Unity, we use “scenes” which are like “levels”
  - Each “scene” contains a bunch of objects that exist in your game
  - You can have multiple scenes in your game think: “Startup Menu”, “Level 1”, etc.
- Most important user interface elements:
  - Game view
  - Scene view
  - Hierarchy window
  - Inspector window
  - Project window
  - Console window
- When in doubt, go to the documentation: https://docs.unity3d.com/2022.3/Documentation/Manual/index.html

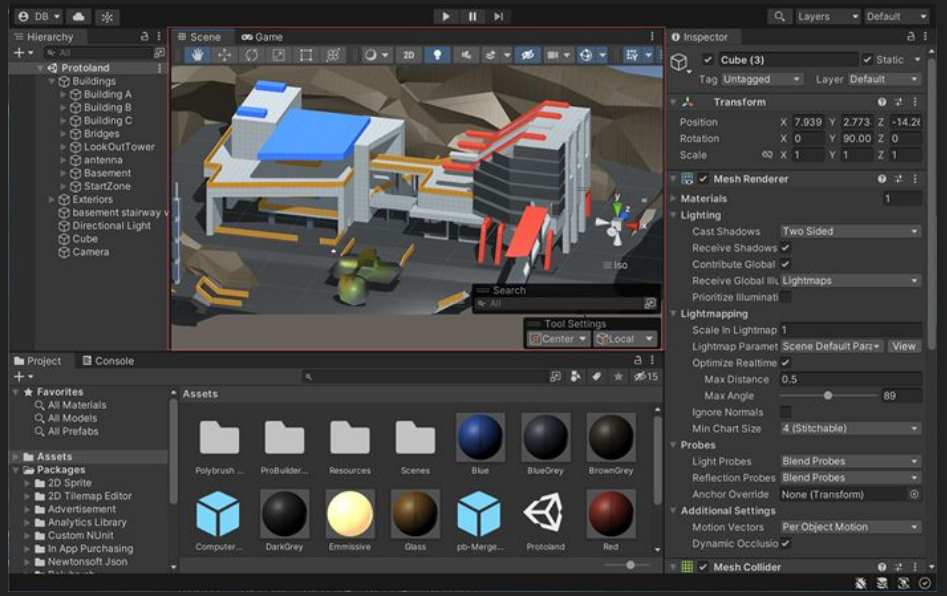
Unity Editor - Game View

- The Game View shows you what your player will see when they are playing your game
  - This is like how a viewer watches a movie
- The Game View is controlled by the main camera and its position and rotation



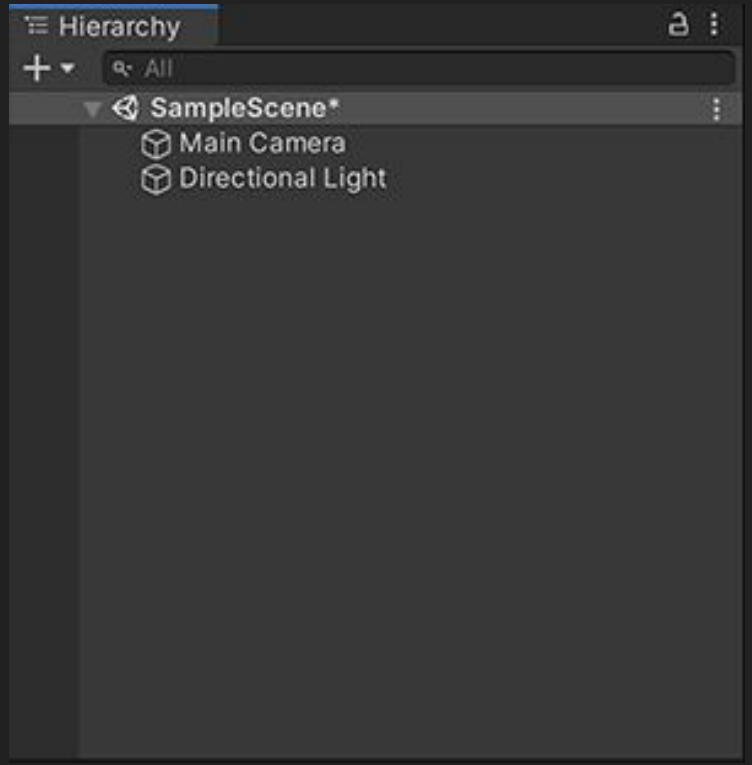
Unity Editor - Scene View

- The Scene View shows you a “behind the scenes” view of your game
  - This is like how a director watches a movie being made
- The Scene View is not accessible when you send your game to players



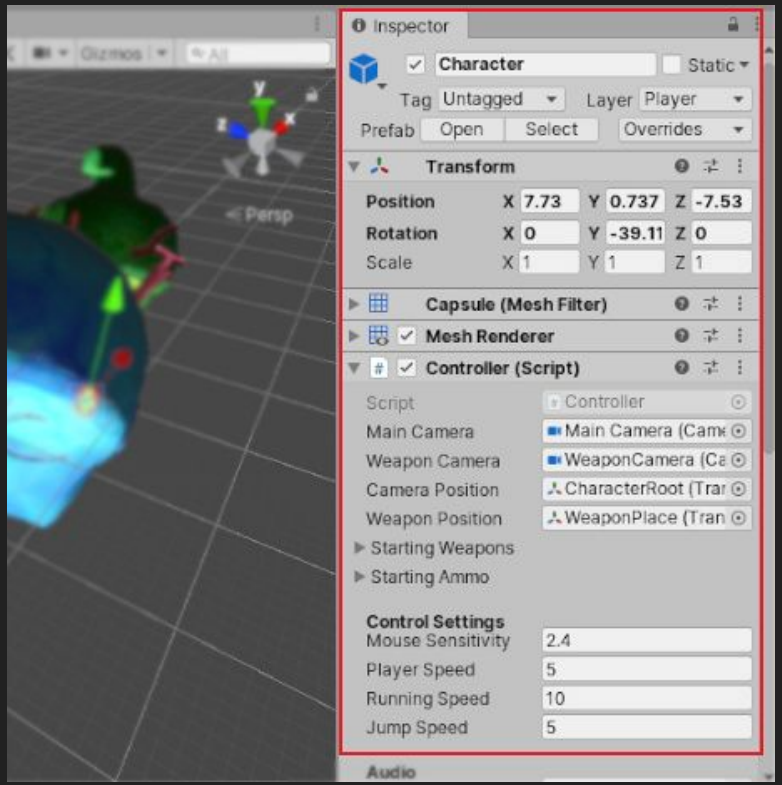
Unity Editor - Hierarchy Window

- The Hierarchy shows all of the “GameObjects” in the scene
  - This is a list of everything inside of your scene
- The Hierarchy allows for parenting objects together
  - This is useful when you have objects that are attached to each other
  - Example: the player is the parent of a sword, the sword is the child of the player
- Usually the order of the GameObjects in the hierarchy does not matter, except for UI



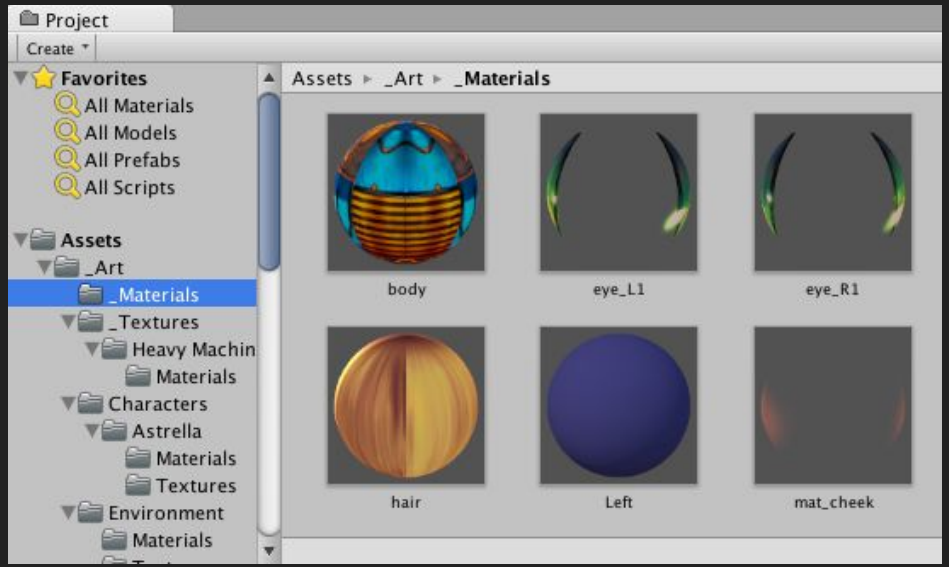
Unity Editor - Inspector Window

- The Inspector shows all of the properties and settings of a selected GameObject
  - This allows you to modify a specific GameObject
- The Inspector allows you to change the position, rotation, and scale of a GameObject
- Add custom “Components” to add more functionality



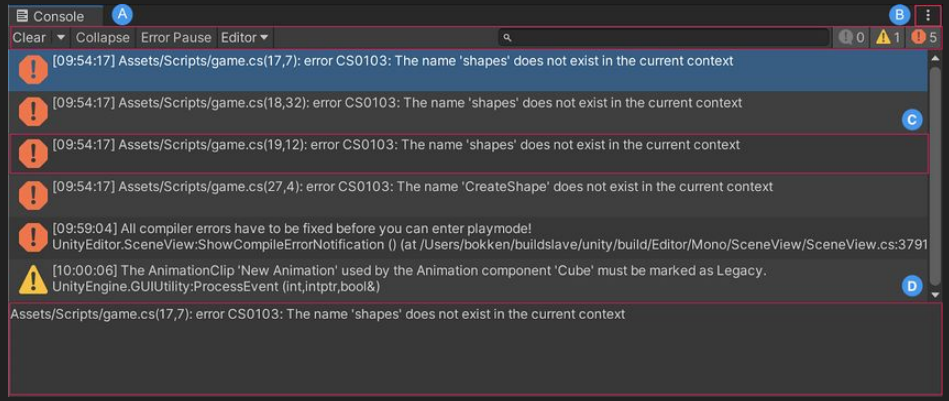
Unity Editor - Project Window

- The Project Window shows all of the “Assets” your game has access to
  - Add new models, sounds, music, scenes, etc. here
- Right click inside the Project Window to create new assets



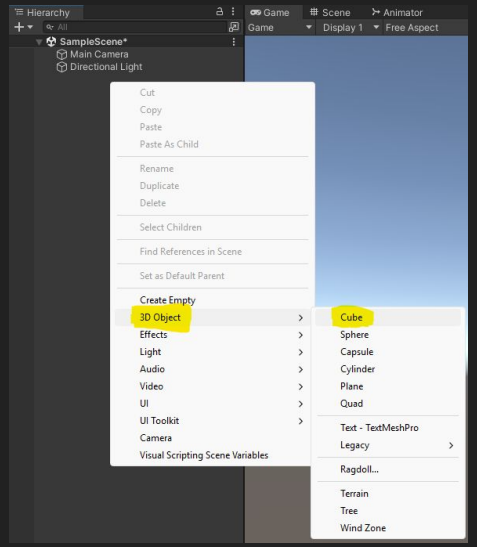
Unity Editor - Console Window

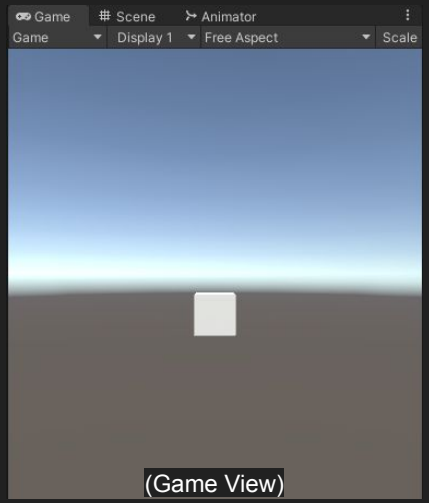
- The Console Window shows you text based feedback of your game
  - Errors, warnings, and logs will appear here



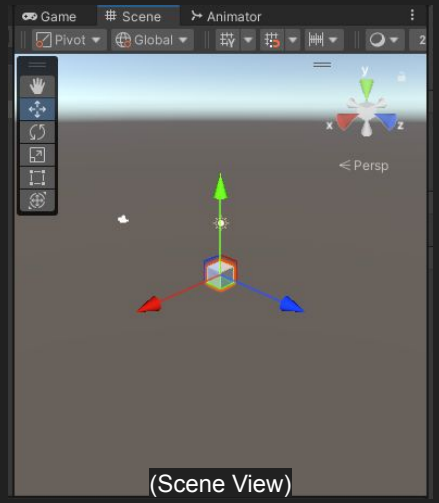
Unity Editor - Hello World

- The game equivalent of “Hello World” is simply creating a cube
- Create a cube:
  - Right click the hierarchy
  - Click 3D Object > Cube
  - Click on the “Cube” in the hierarchy to select it
    - It will now be highlighted in the Scene View
- Click “Play” button at the top of your screen
  - Notice how the cube does nothing
  - Welcome to game development
  - Press “Play” again to stop the game





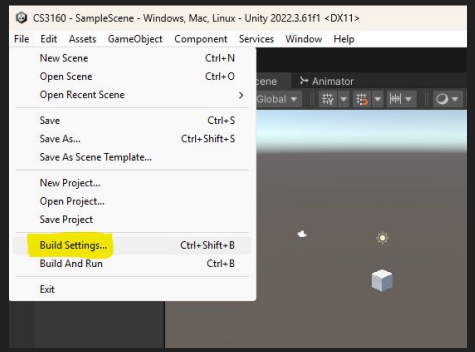
(Game View)

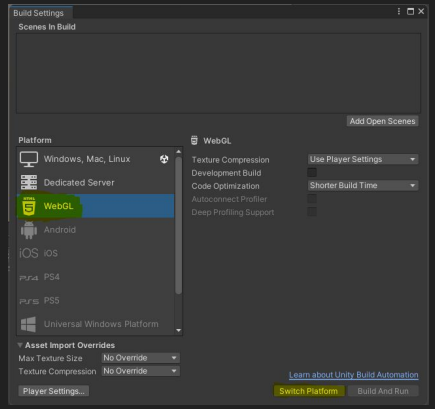


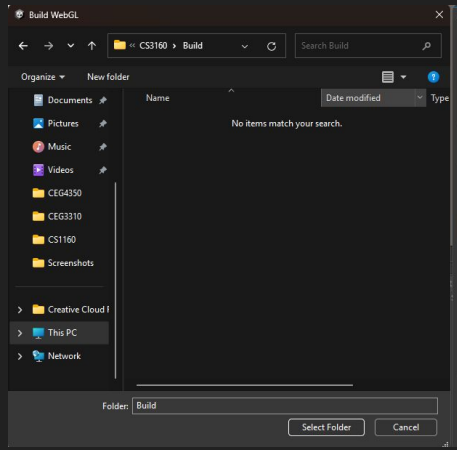
(Scene View)

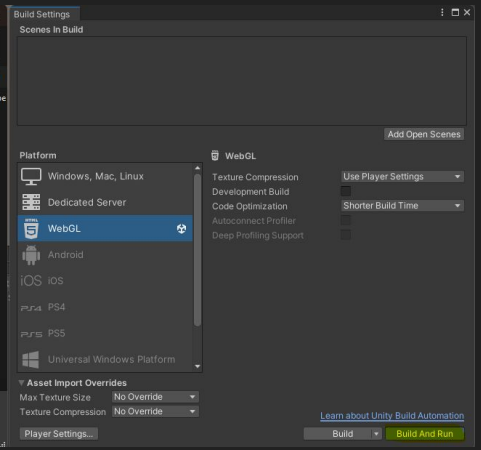
Unity Editor - Building for WebGL

- We will develop our games to run in the web browser
  - These games use the WebGL platform
  - Makes it easy to publish games for everyone to play
- Build for Web:
  - Open the “Build Settings” in File > Build Settings
  - Click “WebGL” then “Switch Platform”
  - Click “Build And Run” and create and select a new folder called “Build”

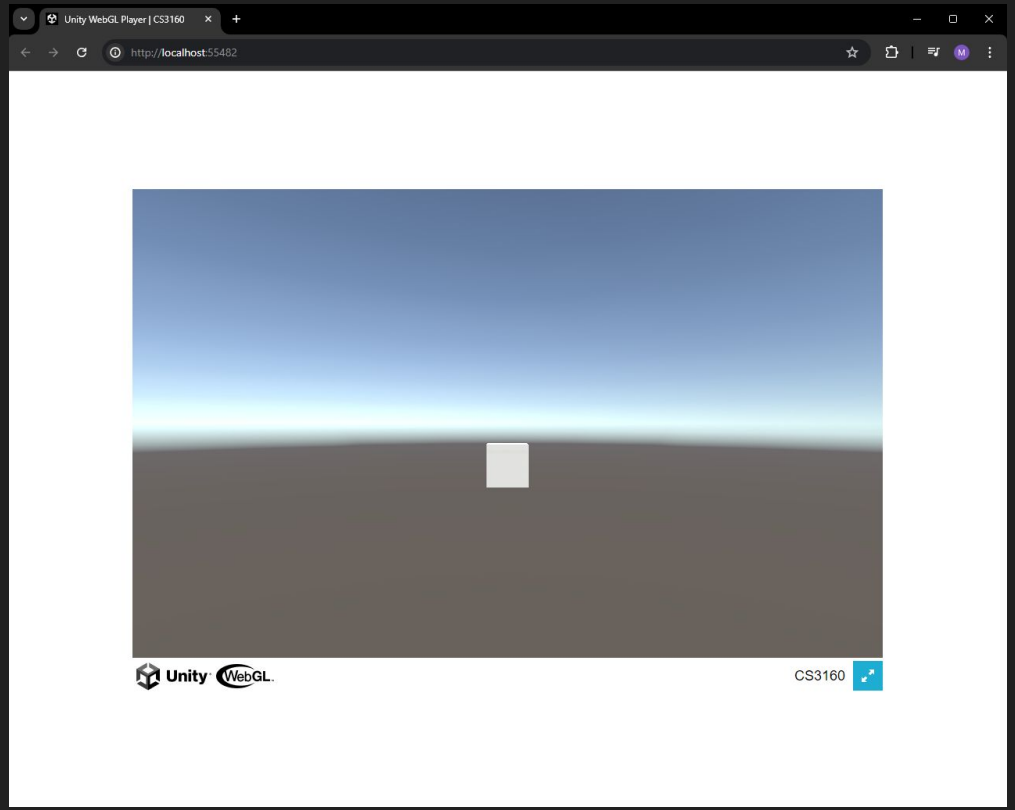






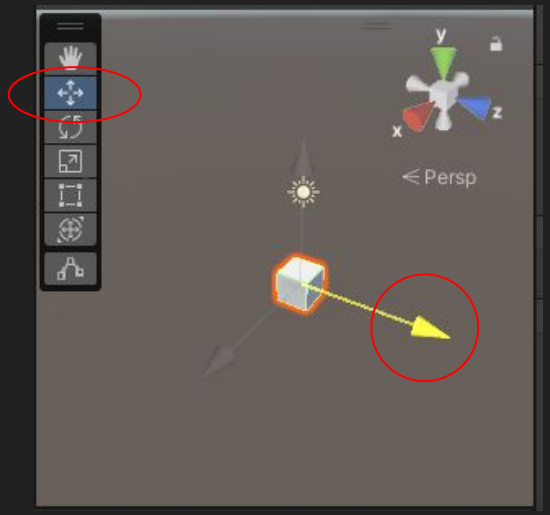


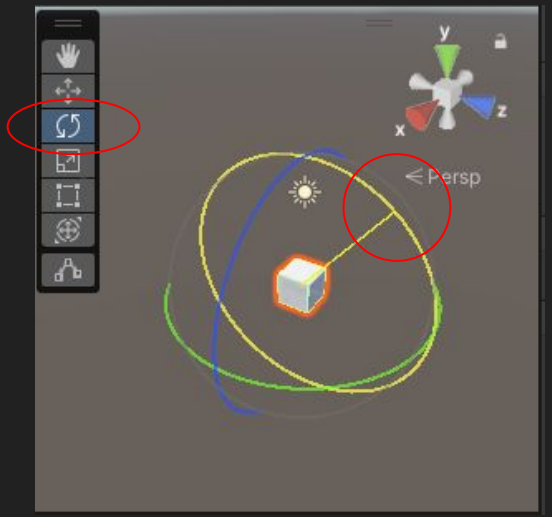
Unity Editor - Building for WebGL (cont.)



Unity Editor - Scene View Controls

- Change the position and rotation of Scene View:
  - Right Click + Drag to rotate
  - Right Click + W, A, S, or D to move
  - Middle Click + Drag to move
  - Shift + F to focus on selected GameObject
  - Scroll Wheel to zoom in and zoom out
  - This does not affect the game
- Change the position and rotation of GameObjects:
  - Click Move Tool + Drag Arrows on Selected GameObject to move
    - Use Ctrl to snap in place
  - Click Rotate Tool + Drag Axis on Selected GameObject to rotate



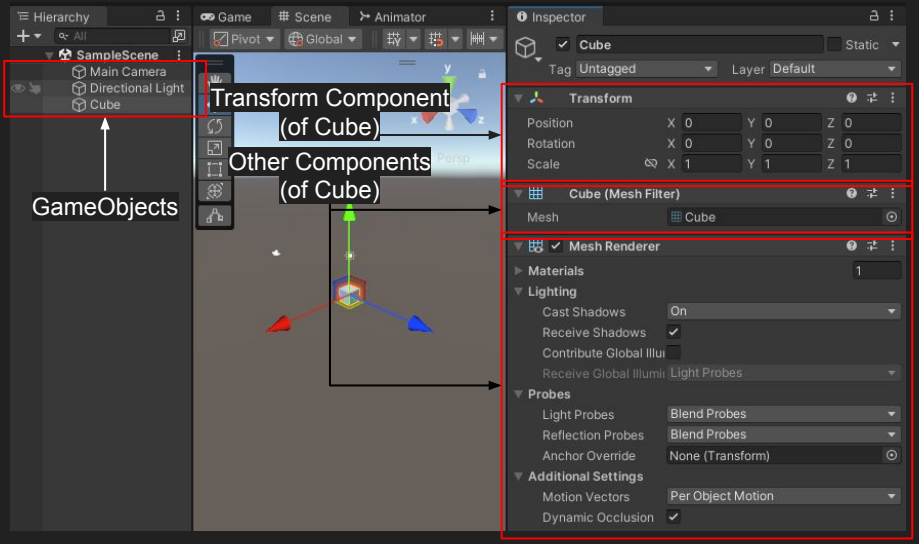


Unity Engine - Objects

- In Unity, we have 2 very specific classes that are used that everything else inherits from:
  - GameObject (all of the objects in your Scene)
  - Component (all the extra functionality attached to a specific GameObject)
    - “Transform” is a type of Component

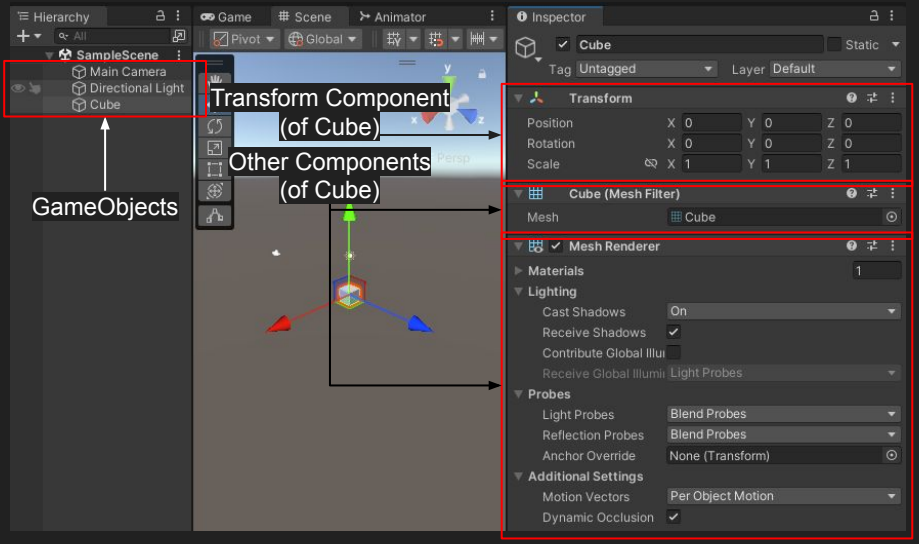
Unity Engine - GameObjects and Components

- GameObjects are the objects in our game inside of our Scene:
  - “Cube”
  - “Main Camera”
  - “Directional Light”
  - Everything is a GameObject
- If you click on a GameObject, the Inspector will show you different properties and settings and all of its Components
  - You can also click on it inside the Scene View
- Components give functionality to the GameObject
  - Without Components, the GameObject will sit there and do nothing
  - You can create an “Empty GameObject” just like how we created the Cube



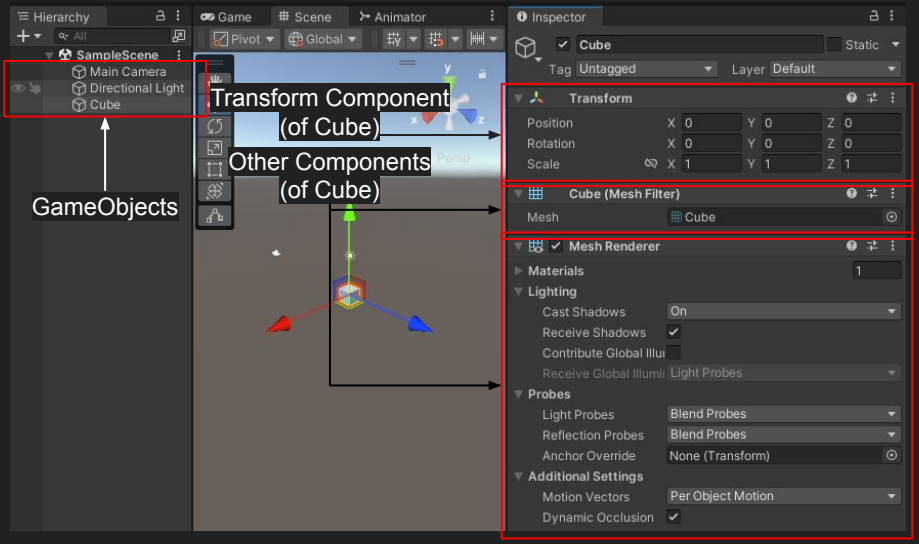
Unity Engine - Transforms

- Every GameObject has a “Transform” Component
- This controls the position, rotation, and scale of the GameObject *relative* to the GameObject’s parent
  - If the GameObject does not have a parent, it is relative to the world
  - Also contains information on its parent and children
- Try dragging the arrows just like before and watch the Transform values
- Try changing values in the Transform manually too



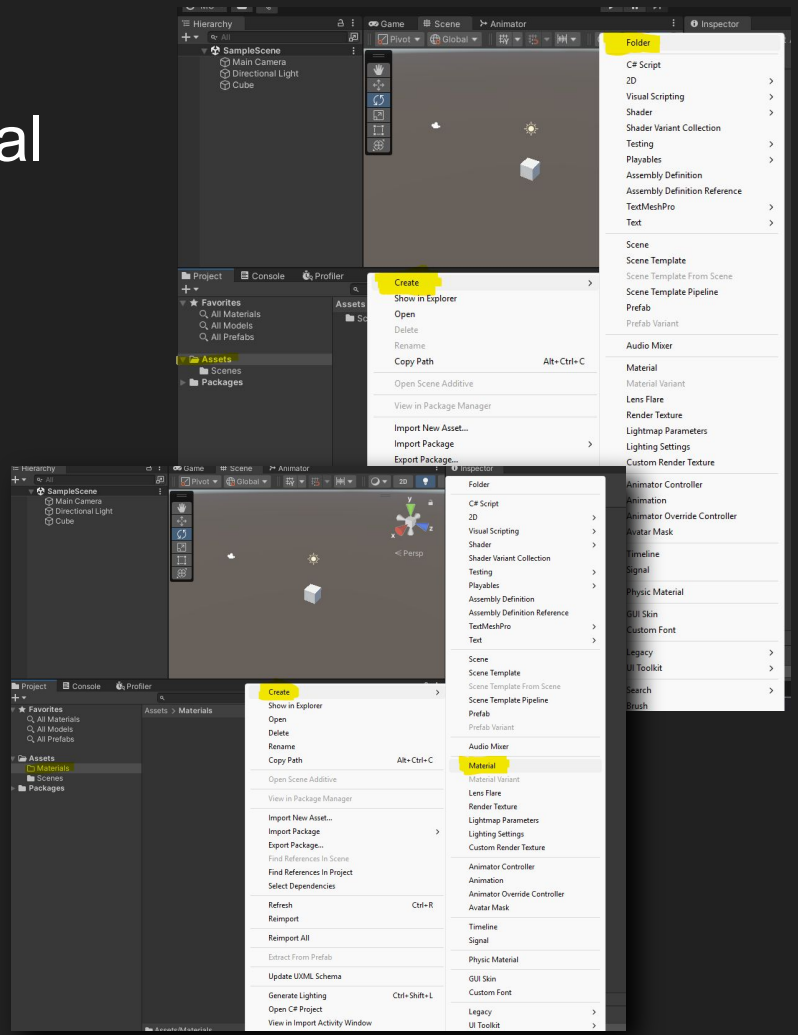
Unity Engine - Other Components

- Notice our Cube has other Components:
  - Cube (Mesh Filter)
    - Provides the 3D model of a cube to be displayed
  - Mesh Renderer
    - Displays the cube 3D model using a Material
    - A Material is texture/image/color that is displayed on top of our cube model
- We will also learn how to make our own components



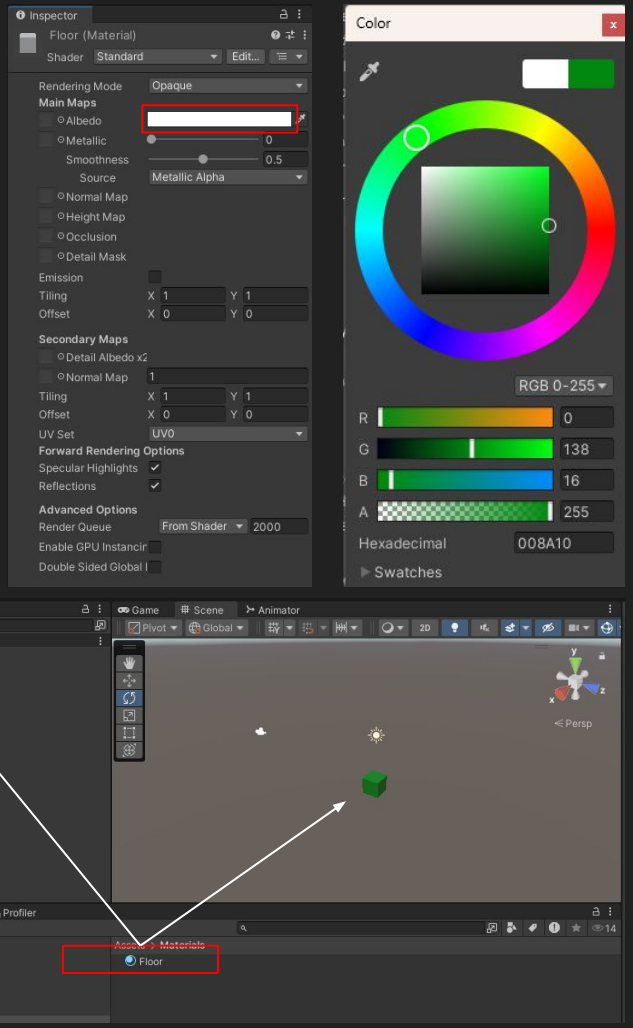
Unity Editor - Creating a new Material

- Materials allow us to place images and colors on our 3D models
- Click “Assets” in Project Window
- Right Click Project Window > Create > Folder
  - Name the Folder “Materials”
- Open “Materials” folder
- Right Click Project Window > Create > Material
  - Name the Material “Floor”



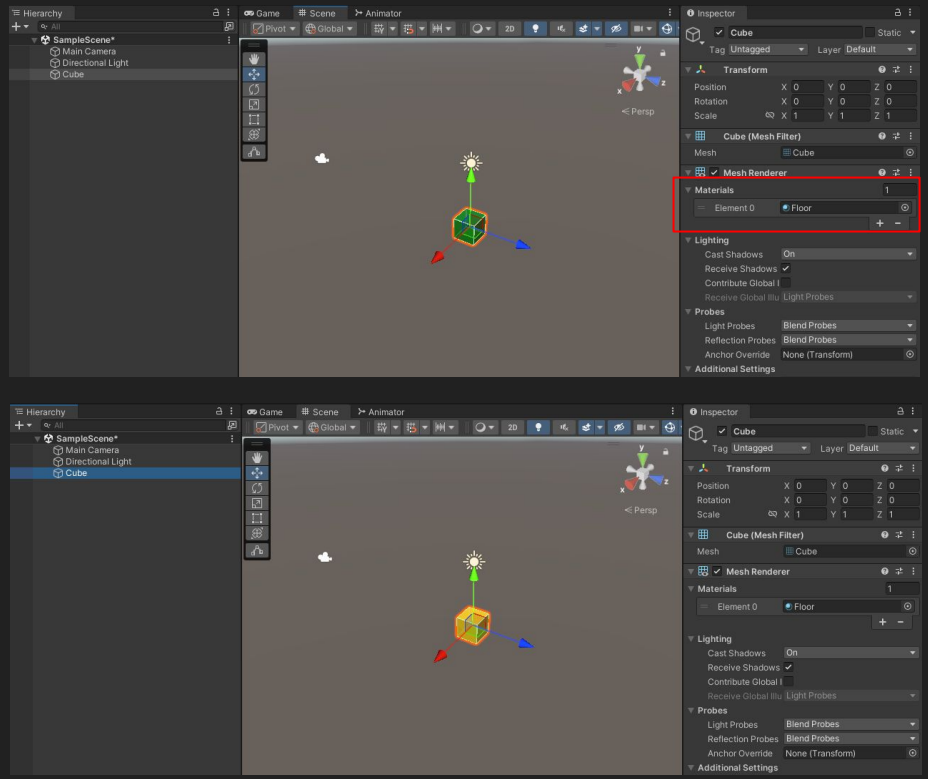
Unity Editor - Applying a new Material

- Choose a random color for the floor
  - Click the white box next to “Albedo”
  - Select any color (other than grey or white)
- Click, Drag, and Drop the Material from the Project Window onto the Cube

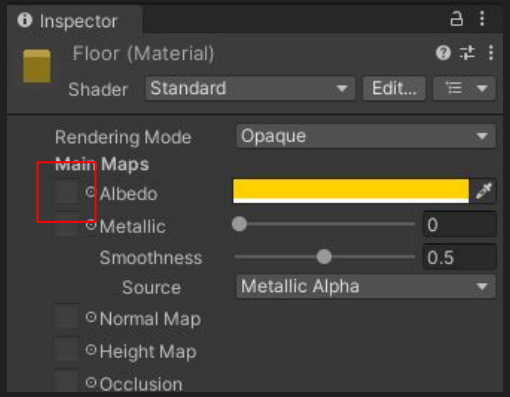


Unity Editor - Changing a Material

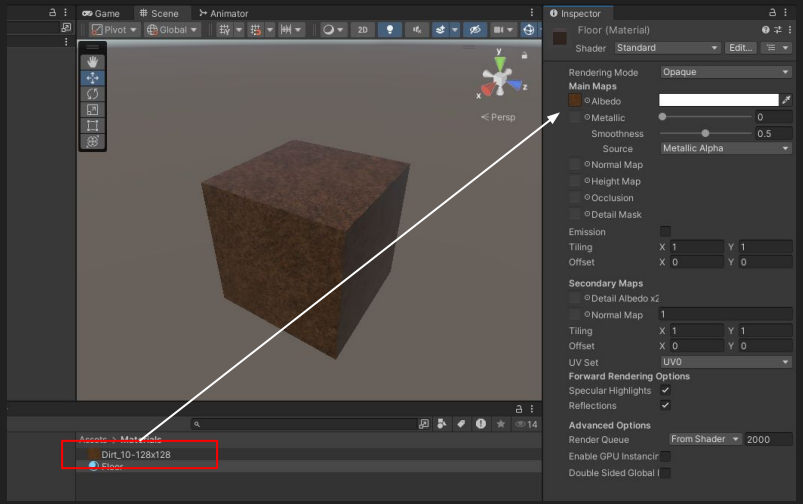
- Notice the Material in the Mesh Renderer Component is now updated to show the Floor Material
- The Floor Material's color can be changed at any time, even while the game is running
- If multiple Components use this Material, they will all receive these color changes



Unity Editor - Adding Textures to Materials

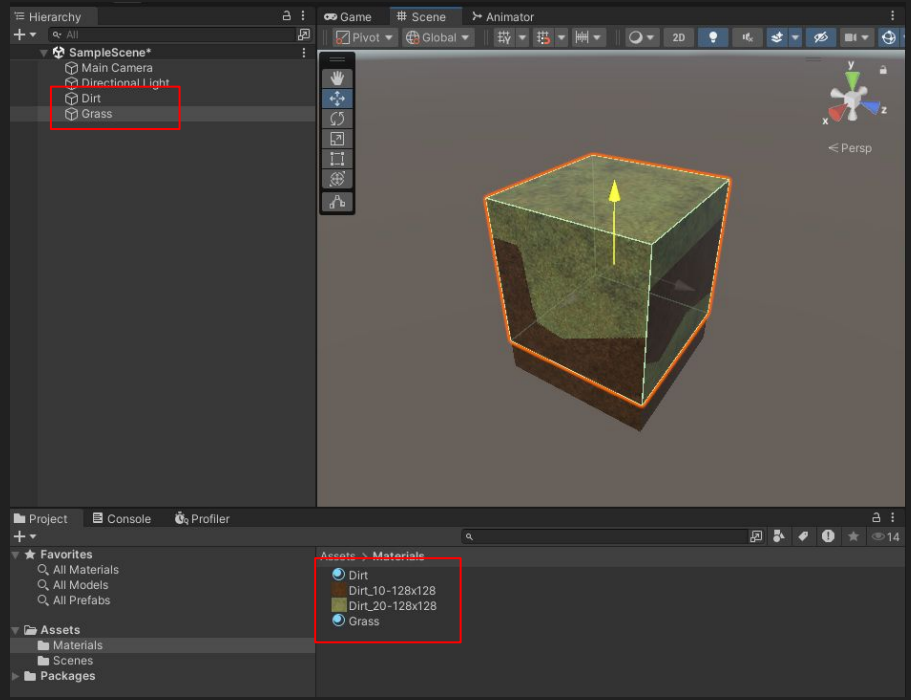


- First, download an image from a reputable source
  - https://itch.io/game-assets/free/tag-textures
  - Check the license to see how you are allowed to use it
  - It would be good practice to keep a list of licenses in the root of your project
- Next, drag the image into your Project Window into a folder
- Lastly, drag the image from the Project Window to the box next to Albedo
- If the Texture appears to have a strange color, reset the Albedo color to white
- You can also lock the Inspector if clicking on the Texture changes the Inspector



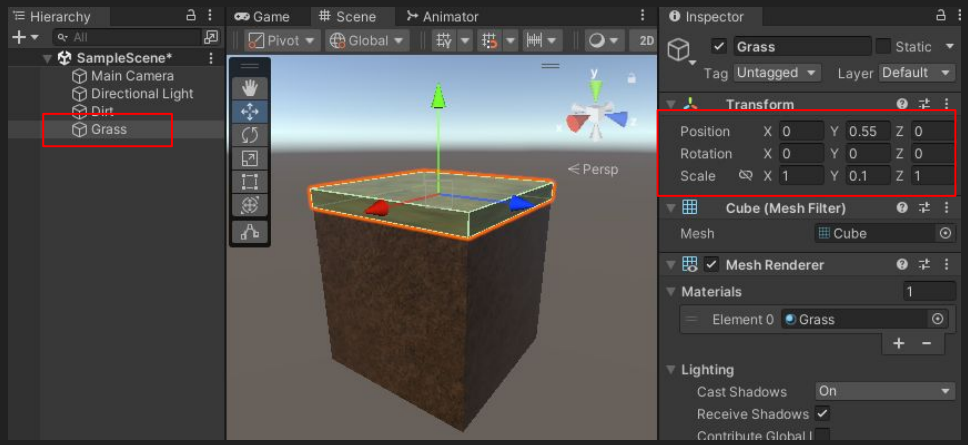
Unity Engine - Parenting GameObjects

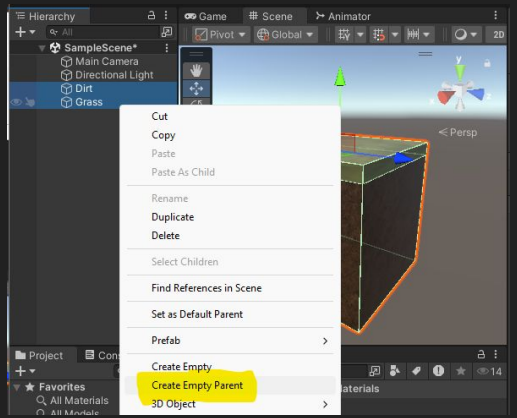
- We can create grass to go on top of our dirt floor
- First, rename “Cube” in the Hierarchy to “Dirt”
- Next, used Ctrl + D to duplicate the Dirt in the Hierarchy
- Rename the new Dirt GameObject to “Grass”
- Create a new Material called Grass and apply it to the “Grass” GameObject
- Notice the z-fighting when moving the Grass GameObject

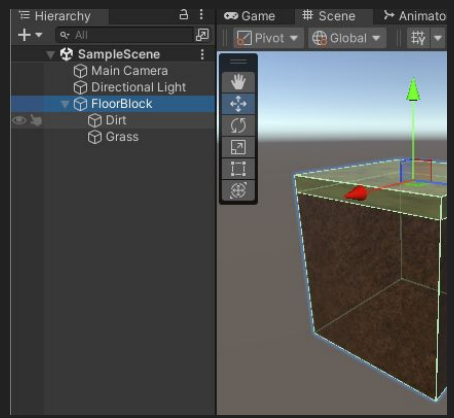


Unity Editor - Parenting GameObjects (cont.)

- Change the position and scale of the Grass using the Transform Component
- Notice, changing the position of the Grass does not affect the Dirt and vice versa
- Select both Grass and Dirt, Right Click > Create Empty Parent
- Name the new parent “FloorBlock”
    - FloorBlock does not contain any components other than Transform
- Now, whenever you need to move both the Grass and the Dirt, just use the FloorBlock Parent

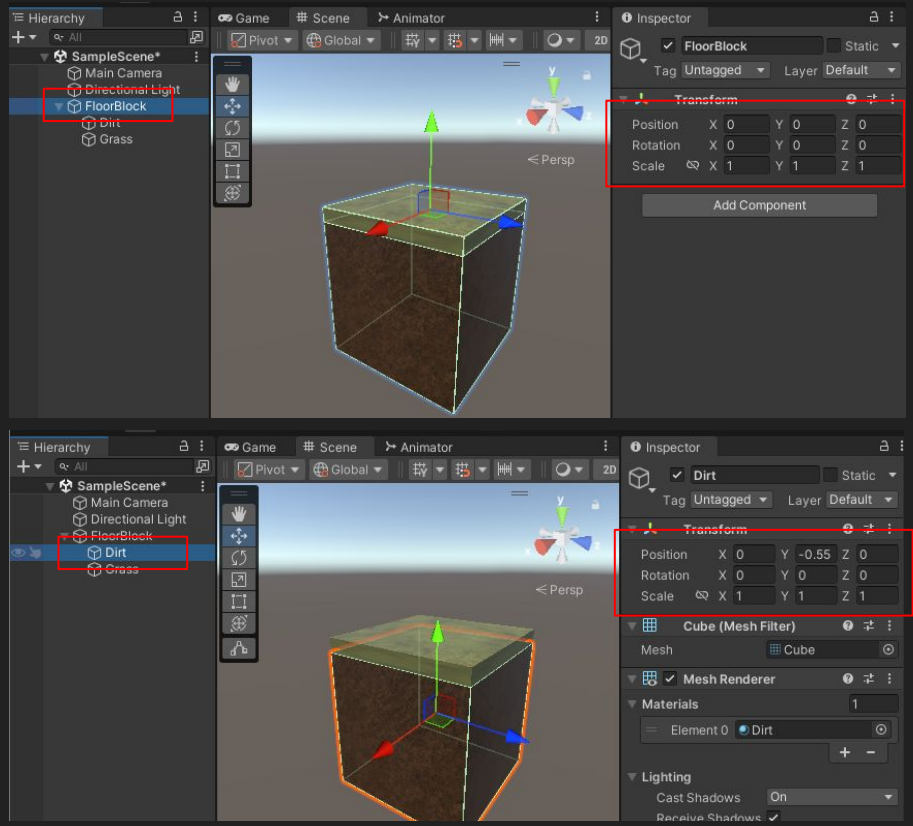






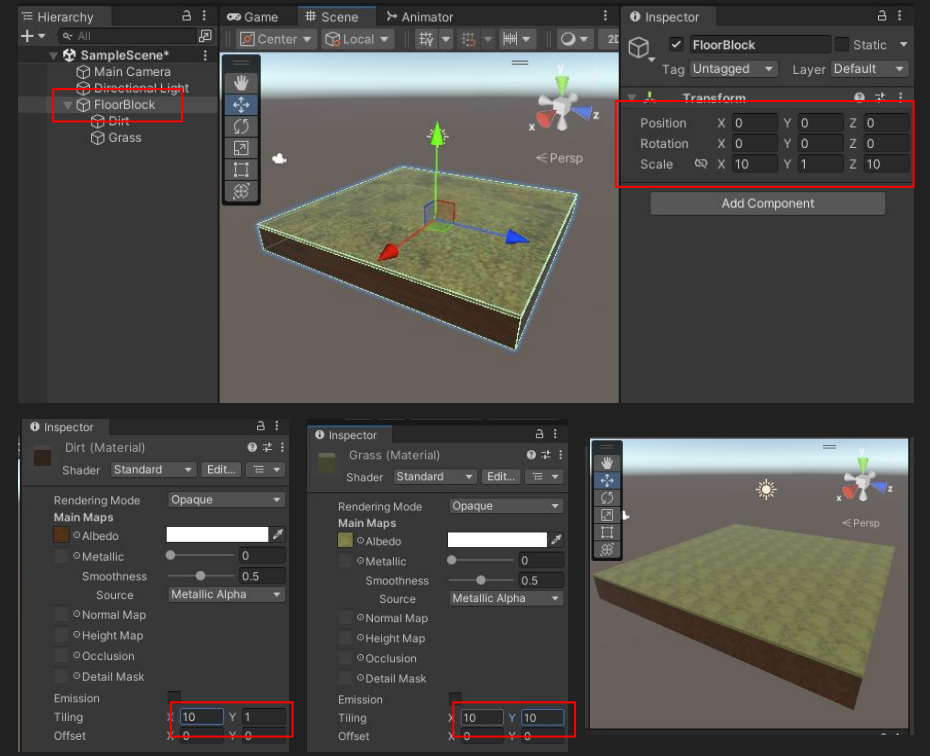
Unity Editor - Parenting GameObjects (cont.)

- The Transform position of the Dirt and Grass is relative to FloorBlock
  - This is called *local position*
- The Transform position of the FloorBlock is relative to the world
  - This is called *world position*



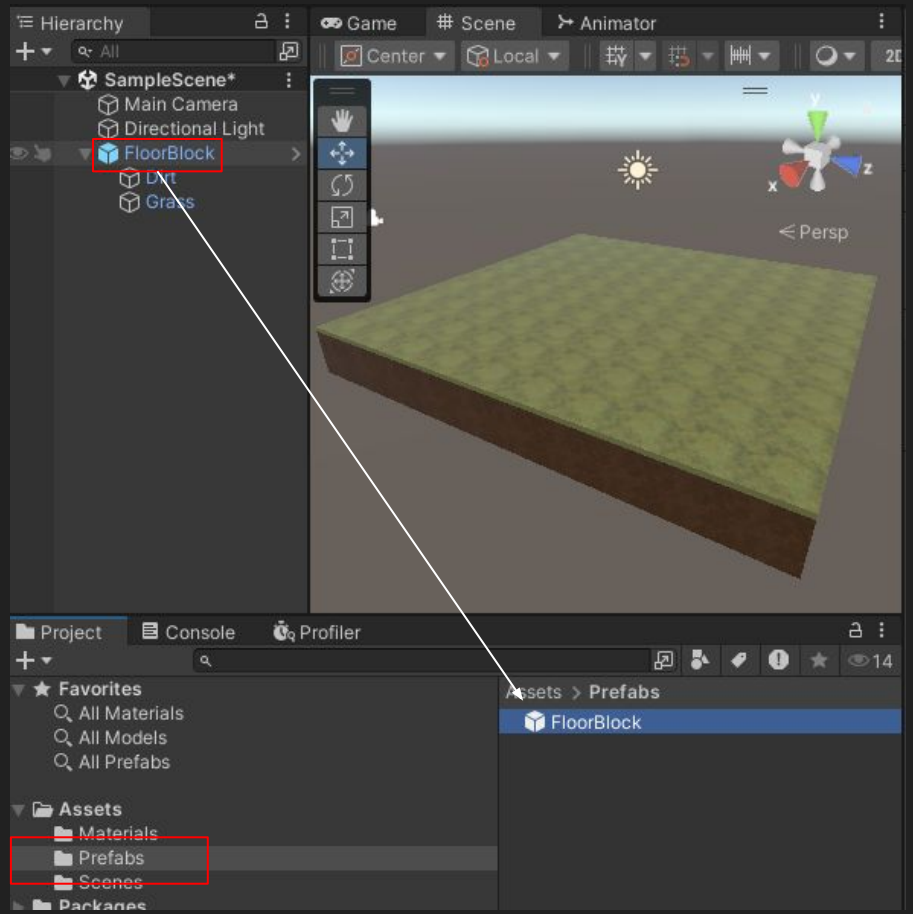
Unity Editor - Scaling Parents

- When you scale a parent, the children will scale also
  - This can cause some strange issues if not careful
  - If we assume the children will not move or rotate locally we can scale the parent without issue
- Change the tiling setting on the Materials to improve the Texture's appearance when scaled



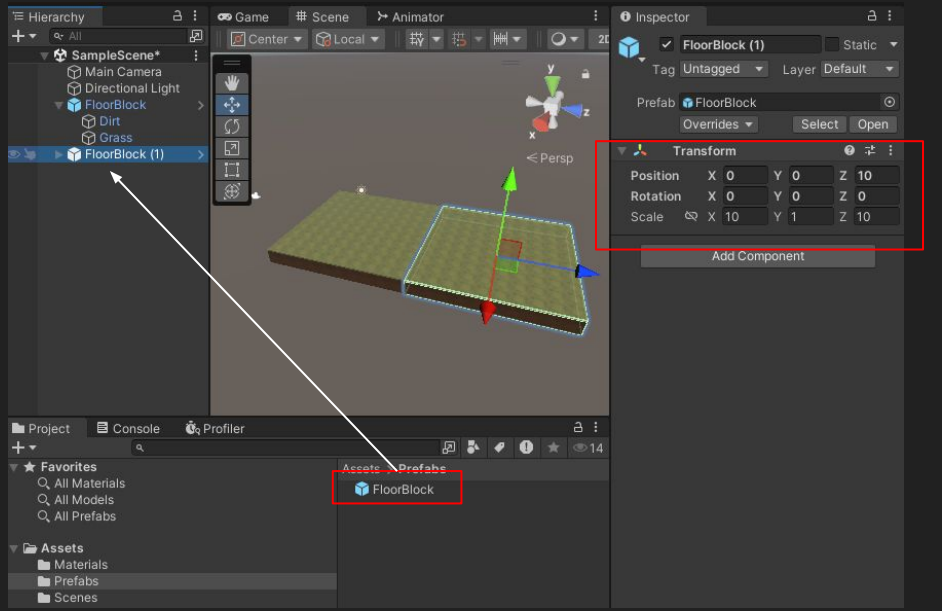
Unity Editor - Creating Prefabs

- You should use Prefabs to create copies of GameObjects
- Prefabs are a template that all GameObjects created from it can pull from
- Create a new folder in the Project Window called “Prefabs”
- Drag FloorBlock from the Hierarchy into the “Prefabs” folder
- A Prefab is now created!



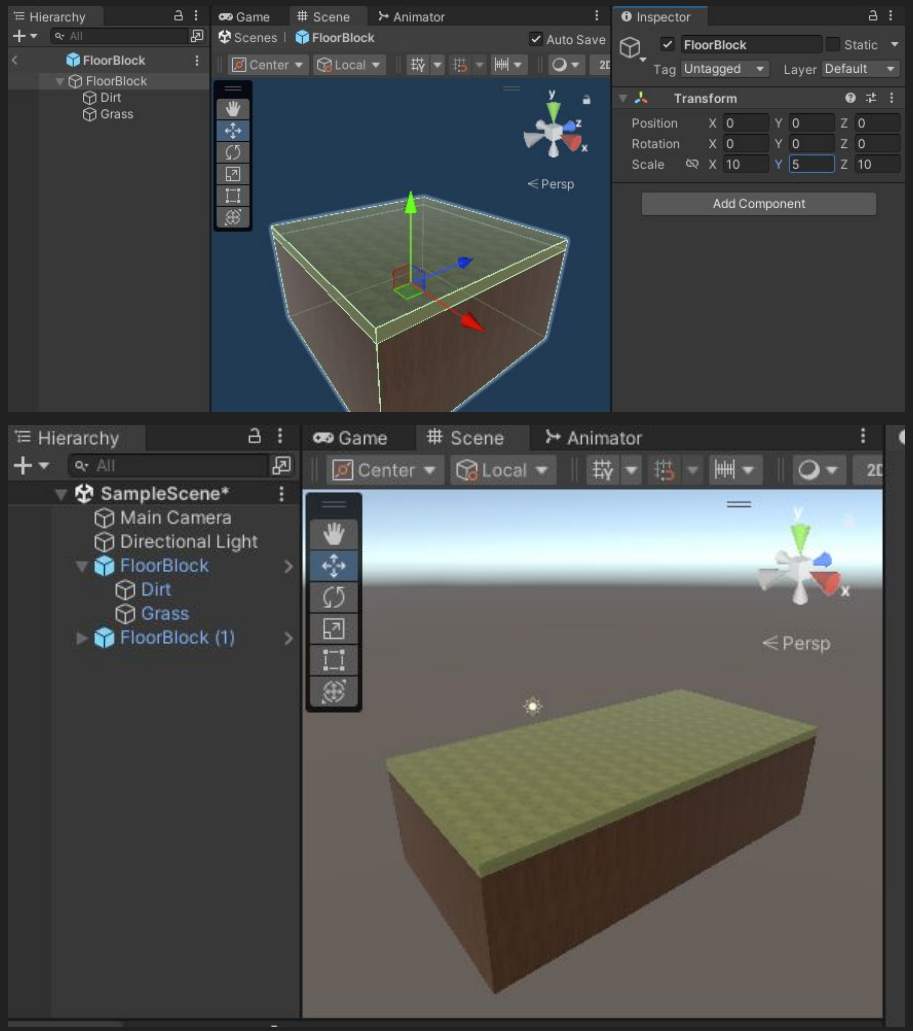
Unity Editor - Using Prefabs

- If you ever want to create a new instance of a Prefab in your Scene, just drag and drop or use Ctrl + D
  - You can change the position too



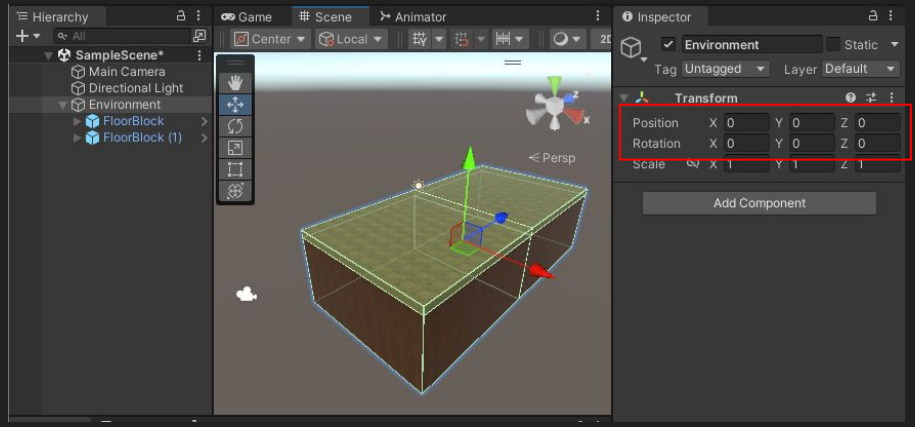
Unity Editor - Modifying Prefabs

- To modify a Prefab to change all existing Prefabs, double click on the Prefab inside the Project Window
- Change the Prefab as desired
  - As long as the settings you change are not modified by the Prefab instances, the instances will be updated
- You can also “Apply To Prefab” if you are manipulating instances of Prefabs in your Scene



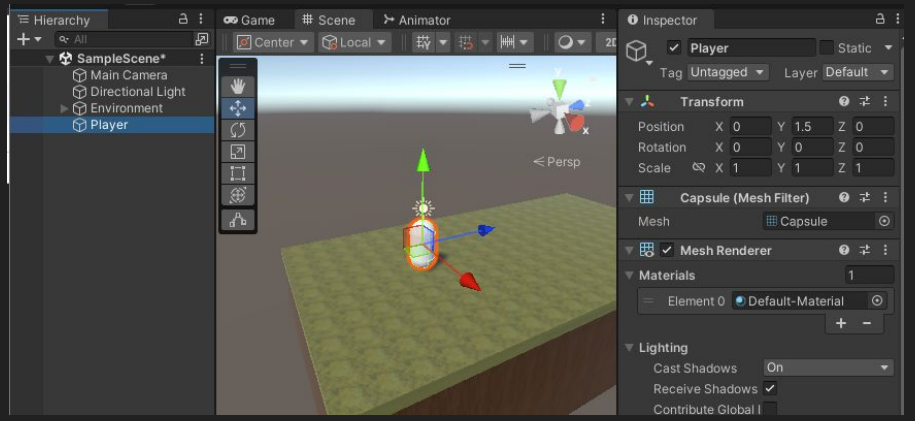
Unity Editor - Keep Your Hierarchy Clean

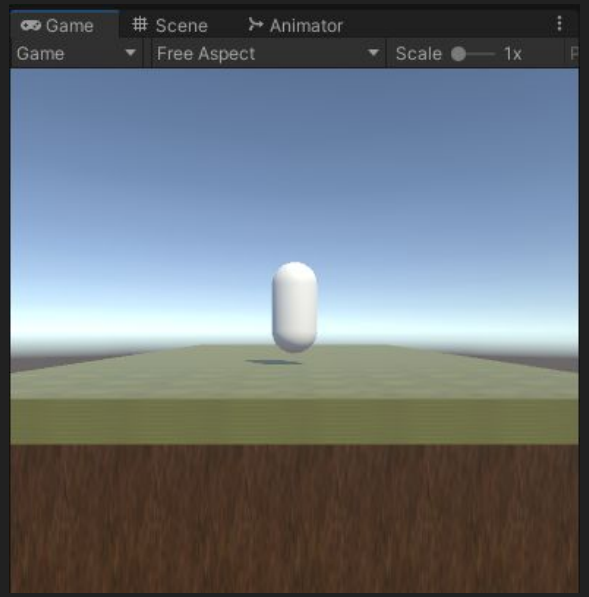
- If you have a lot of objects in your Scene, drag them into a parent GameObject
  - Before dragging, set the position and rotation of the parent to 0
- Helps group similar GameObjects together and reduce clutter



Unity Editor - Adding a Player

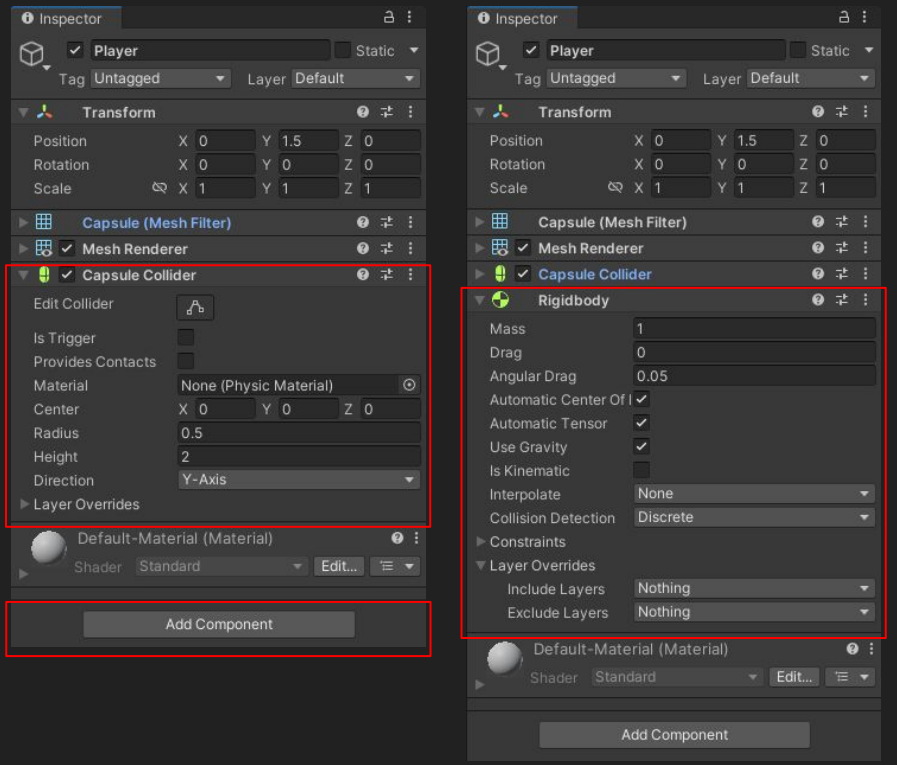
- Right now, the player cannot control anything
- Create a “Capsule” in your Scene, just like how we created the Cube
- Rename it “Player”
- Set its position somewhere above the floor
- Notice nothing happens when we press play





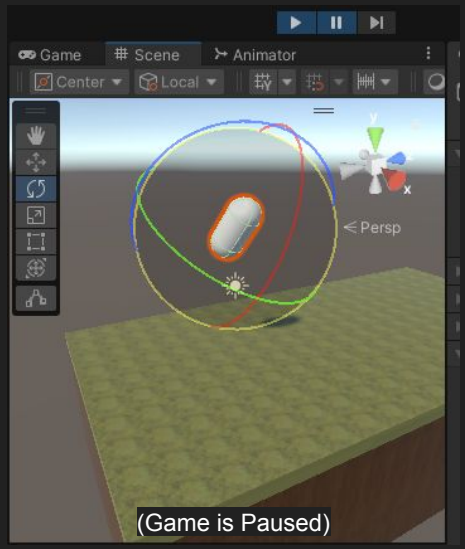
Unity Editor - Adding a Player (cont.)

- To have the Capsule interact with Unity's physics engine it requires two Components:
  - A collider (already provided)
  - A rigidbody
- The collider tells Unity what parts of our Capsule can interact with other colliders
- The rigidbody tells Unity how our Capsule should interact with gravity, friction, and other forces
  - Add a rigidbody using "Add Component" at the bottom of the Inspector

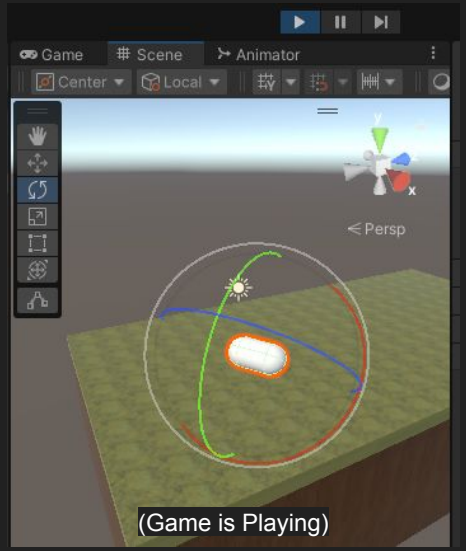


Unity Editor - Modifying Scene View When Playing

- You can modify GameObjects when in Scene view while the game is running
- Changes made while the game is running are not saved
- You can also pause and modify



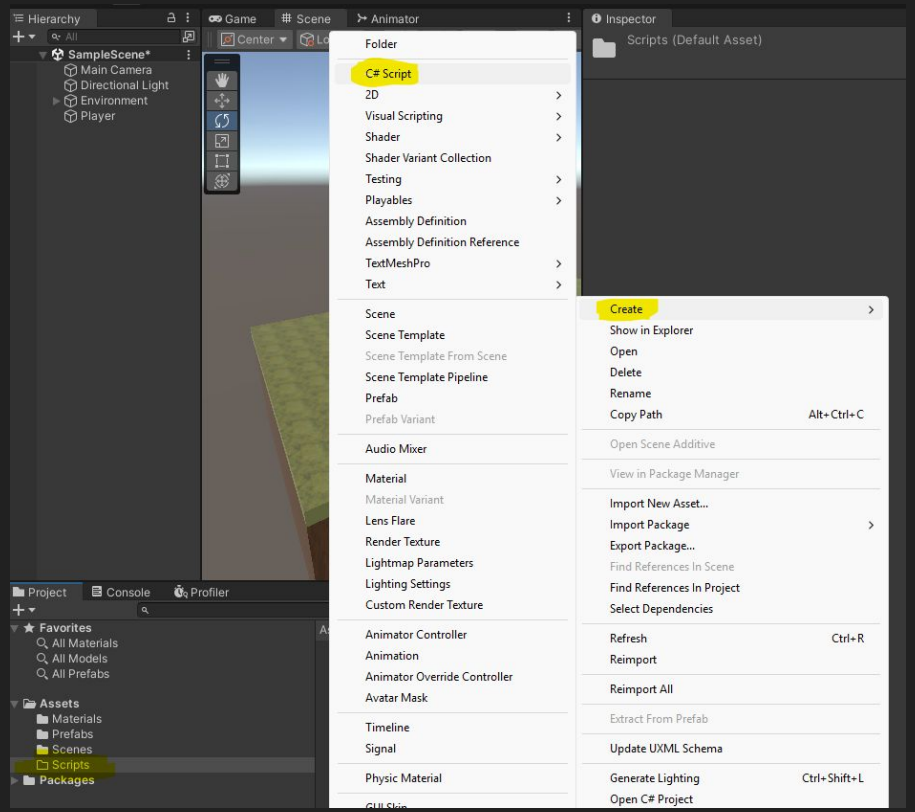
(Game is Paused)



(Game is Playing)

Unity Editor - Custom Scripts

- Unity does not know how we want the player to be controller
- We have to write custom code to do this ourselves
- First, create a “Scripts” folder in the Project Window
- Next, inside the “Scripts” folder create a new C# script
- Name the new script “Player” and double click it to open



Unity Editor - Empty Script

- Every time you create a new script, it comes with some empty methods
  - Every new script you make must inherit from MonoBehavior to be attached to a GameObject
  - Start() runs when the script is created in your scene
  - Update() runs every frame
    - The speed that Update() runs at is not guaranteed!
- The name of the class must be the same as the name of the file
  - Otherwise you will get errors

Player.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Player : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

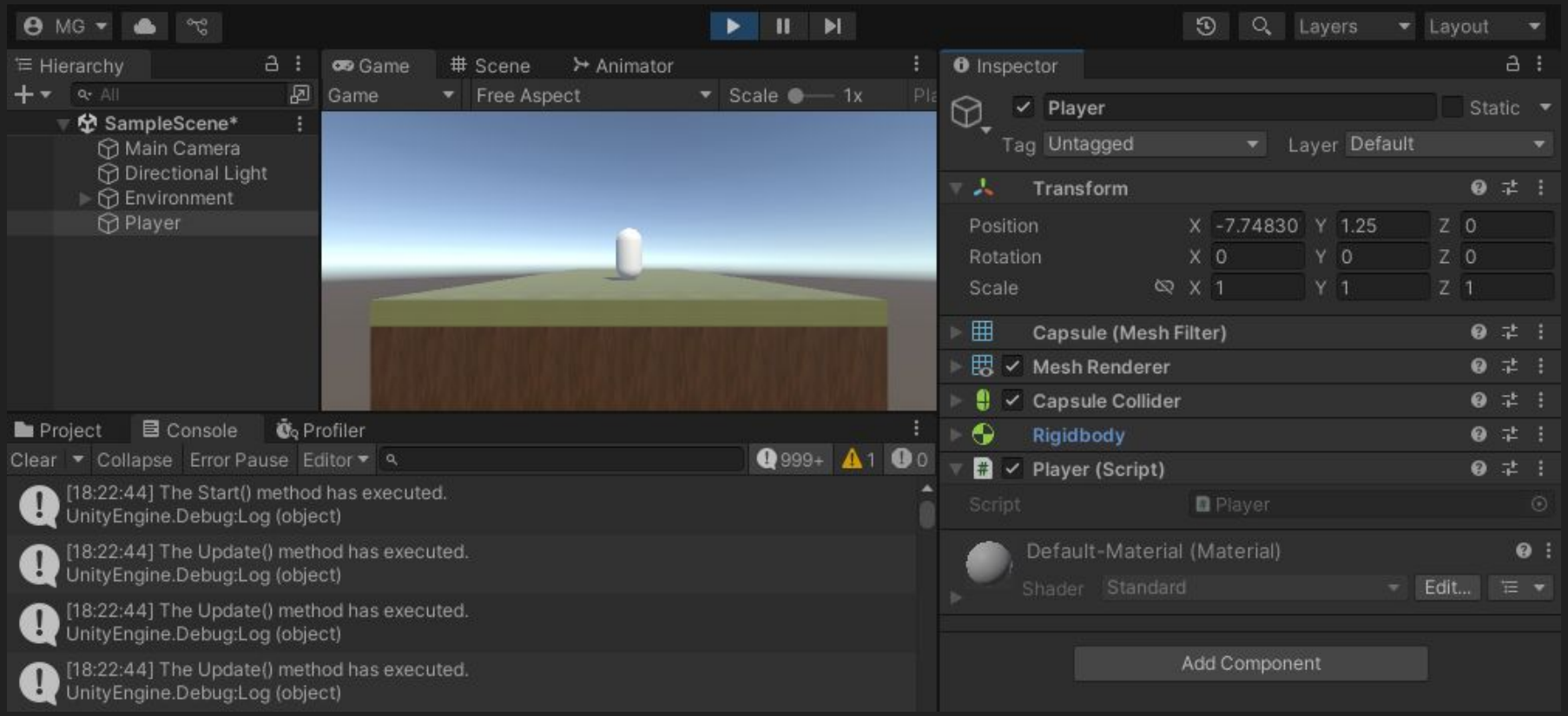
Unity Editor - Debug Log

- We can add Debug.Log() to print messages to the Console
- To add this script to the Player, drag and drop the script from the Project Window to the Player GameObject

Player.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Player : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Debug.Log("The Start() method has executed.");
    }

    // Update is called once per frame
    void Update()
    {
        Debug.Log("The Update() method has executed.");
    }
}
```

Unity Editor - Debug Log (cont.)



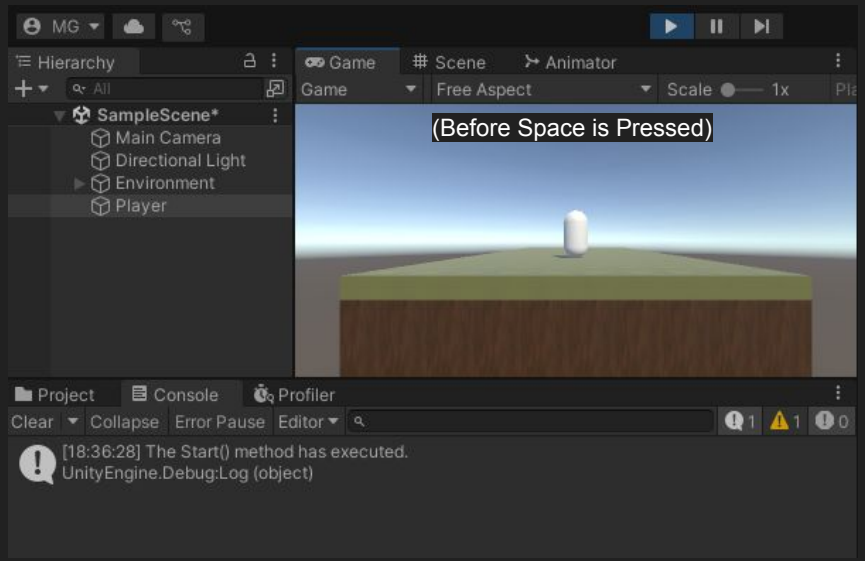
Unity Editor - Player Input

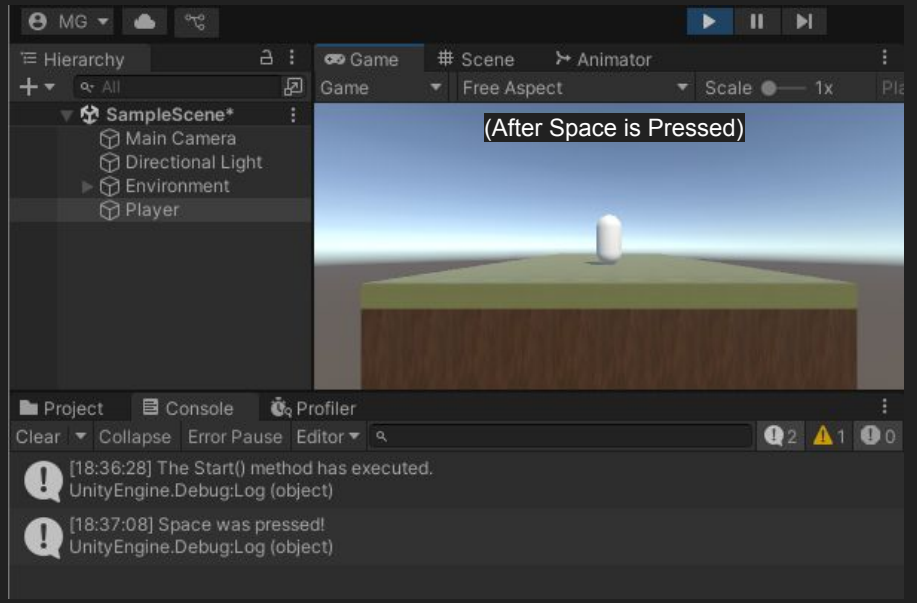
- We can get a player's input by using the Input class
- Use the "Input.GetKeyDown()" method for checking if the player pressed a button
  - Returns "true" only on the frame where the key is pressed
- When in doubt, go to the documentation: https://docs.unity3d.com/2022.3/Documentation/ScriptReference/Input.html

Player.cs:

```
...
public class Player : MonoBehaviour
{
    ...
    // Update is called once per frame
    void Update()
    {
        // Check if the user pressed Space this frame
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Debug.Log("Space was pressed!");
        }
    }
}
```

Unity Editor - Player Input (cont.)





Unity Editor - Affecting The Rigidbody

- Printing to the log is not fun
- We can affect a Rigidbody by applying a force
- Use GetComponent() method to get a Component from the GameObject
- After getting the Rigidbody, apply a force using a Vector3
  - A Vector3 is 3 floats, one for X, Y, and Z
  - In the code, a Vector3 is created with 0 in X, 500 in Y, and 0 in Z

Player.cs:

```
...
// Update is called once per frame
void Update()
{
    // Check if the user pressed Space this frame
    if (Input.GetKeyDown(KeyCode.Space))
    {
        // Get the Rigidbody attached to
        // this GameObject (the Player)
        var rigidBody = GetComponent<Rigidbody>();
        // If the Rigidbody was found (not null)
        if (rigidBody != null)
        {
            // Add force of 10 in the Y direction
            var force = new Vector3(0f, 10f, 0f);
            rigidBody.AddForce(force, ForceMode.VelocityChange);
        }
    }
}
```

Unity Editor - Affecting The Rigidbody (cont.)

- Remember, Update() does not run at a consistent speed
- Therefore, we should put anything relating to movement or something that occurs over time in FixedUpdate() method
  - FixedUpdate() is guaranteed to run at a specific rate (50 Hz default)

Player.cs:

```
...
    private bool jumpWasPressed;
...
    // Update is called once per frame
    void Update()
    {
        // Check if the user pressed Space this frame
        if (Input.GetKeyDown(KeyCode.Space))
        {
            // The user pressed the jump button
            jumpWasPressed = true;
        }
    }

    // FixedUpdate is called once per fixed frame
    private void FixedUpdate()
    {
        // If the user pressed the jump button
        if (jumpWasPressed)
        {
            // Get the Rigidbody and apply a force
            var rigidBody = GetComponent<Rigidbody>();
            if (rigidBody != null)
            {
                var force = new Vector3(0f, 10f, 0f);
                rigidBody.AddForce(force, ForceMode.VelocityChange);
            }

            // Reset jumpWasPressed
            jumpWasPressed = false;
        }
    }
```

Unity Editor - Affecting The Rigidbody (cont.)

- We can also get an Axis which will allow us to get “Joystick” like values
  - Use Input.GetAxis() to return a float value (-1.0 to +1.0)
  - Input.GetAxis("Horizontal") is controlled by A and D keys
  - Input.GetAxis("Vertical") is controlled by W and S keys

Unity Editor - Affecting The Rigidbody (cont.)

Player.cs:

```
...
    private bool jumpWasPressed;
    public Vector2 desiredMovement;
...

    // Update is called once per frame
    void Update()
    {
        // Check if the user pressed Space this frame
        if (Input.GetKeyDown(KeyCode.Space))
        {
            jumpWasPressed = true; // The user pressed the jump button
        }

        // desiredMovement has 2 directions, X and Y, for horizontal and vertical
        desiredMovement.x = Input.GetAxis("Horizontal");
        desiredMovement.y = Input.GetAxis("Vertical");
    }
...
```

Player.cs:

```
...
    // FixedUpdate is called once per fixed frame
    private void FixedUpdate()
    {
        // Get the Rigidbody
        var rigidBody = GetComponent<Rigidbody>();

        // If the user pressed the jump button
        if (jumpWasPressed)
        {
            // Apply the force
            if (rigidBody != null)
            {
                var force = new Vector3(0f, 10f, 0f);
                rigidBody.AddForce(force, ForceMode.VelocityChange);
            }

            // Reset jumpWasPressed
            jumpWasPressed = false;
        }

        // If the Rigidbody was found
        if (rigidBody != null)
        {
            // Get the current velocity
            var currentVelocity = rigidBody.velocity;

            // Apply a new velocity in the x and z direction based on the movement
            var newVelocity = 5f * new Vector3(desiredMovement.x, 0f, desiredMovement.y);

            // Let's not lose the y velocity we gained from jumping!
            newVelocity.y = currentVelocity.y;

            rigidBody.velocity = newVelocity;
        }
    }
```

Unity Editor - Cleaning Things Up

- It is not a good idea to call the GetComponent() method every single frame
  - This is bad for performance if done often
- Let’s get the Rigidbody inside of the Start() method instead
- Then, if the Rigidbody is not found, we can just return inside the FixedUpdate() method
  - Added bonus: we don’t ever have to check if Rigidbody is null afterwards! If it was null, we returned early!

Unity Editor - Cleaning Things Upd (cont.)

Player.cs:

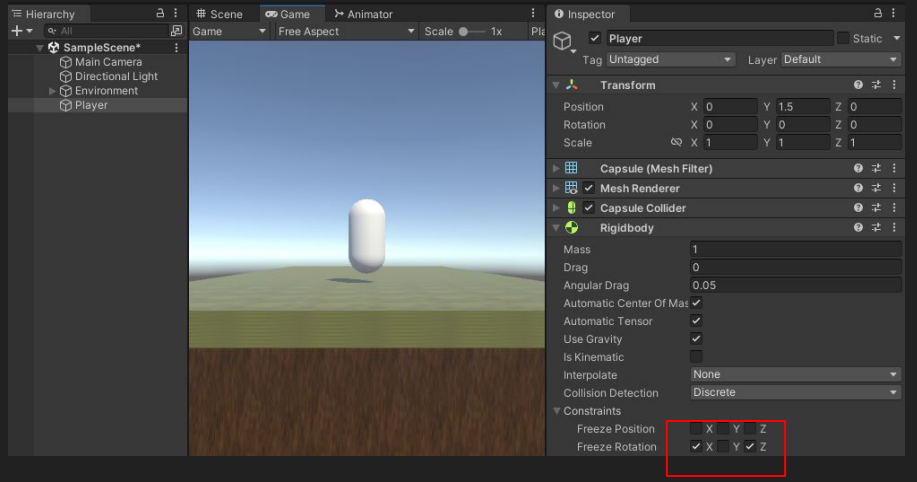
```
...
    private Rigidbody rigidBody;
    // Start is called before the first frame update
    void Start()
    {
        // Get the Rigidbody when this script first starts running
        rigidBody = GetComponent<Rigidbody>();
    }
...
```

Player.cs:

```
    // FixedUpdate is called once per fixed frame
    private void FixedUpdate()
    {
        // Check for the Rigidbody
        // If it's not found, return, we have nothing left to do!
        if (rigidBody == null) return;

        // If the user pressed the jump button
        if (jumpWasPressed)
        {
            // Apply the force
            var force = new Vector3(0f, 10f, 0f);
            rigidBody.AddForce(force, ForceMode.VelocityChange);

            // Reset jumpWasPressed
            jumpWasPressed = false;
        }

        // Get the current velocity
        var currentVelocity = rigidBody.velocity;

        // Apply a new velocity in the x and z direction based on the movement
        var newVelocity = 5f * new Vector3(desiredMovement.x, 0f, desiredMovement.y);

        // Let's not lose the y velocity we gained from jumping!
        newVelocity.y = currentVelocity.y;

        rigidBody.velocity = newVelocity;
    }
```

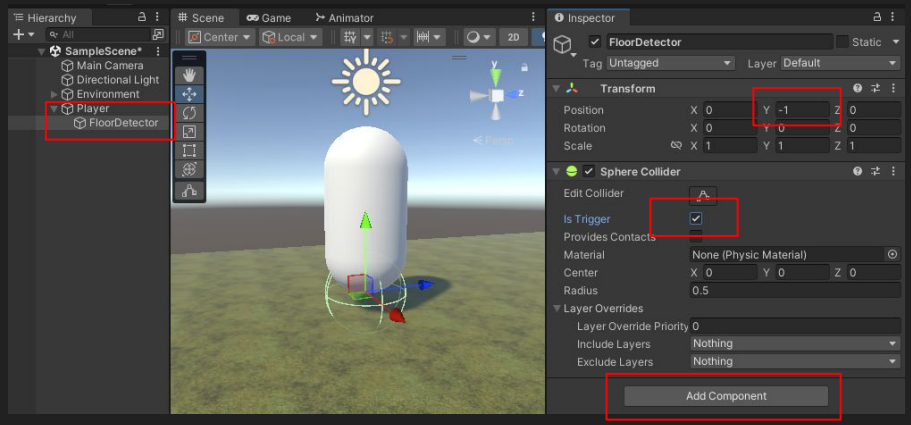
Unity Editor - Deactivating Rotation on Rigidbodies

- When the player moves, it topples over
- Easiest fix: lock rotation on the Rigidbody



Unity Editor - Checking If Player is Touching Floor

- The player can jump mid-air
  - We'll call this a bug
- To fix this, we have to check if the player is touching the ground or not
- Create an empty GameObject and make it a child of Player
- Rename the GameObject to "FloorDetector"
- Add a Sphere Collider Component to the FloorDetector
- Reposition the FloorDetector to the bottom of Player
- Set "Is Trigger" to true (check the box)



Unity Editor - Checking If Player is Touching Floor (cont.)

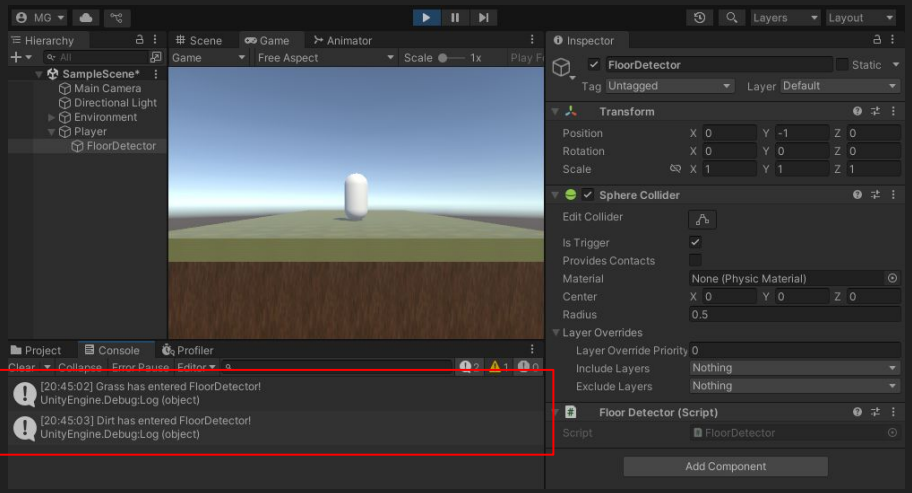
- Create a new script called “FloorDetector”
- Drag it onto the FloorDetector GameObject
- Delete the Start() and Update() methods inside of FloorDetector, they won’t be used
- Create two new methods called “OnTriggerEnter” and “OnTriggerExit”
  - Intellisense should autocomplete
  - These are special methods provided by Unity that execute when a trigger collider is entered or exited

FloorDetector.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FloorDetector : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        Debug.Log($"{other.name} has entered {name}!");
    }

    private void OnTriggerExit(Collider other)
    {
        Debug.Log($"{other.name} has exited {name}!");
    }
}
```

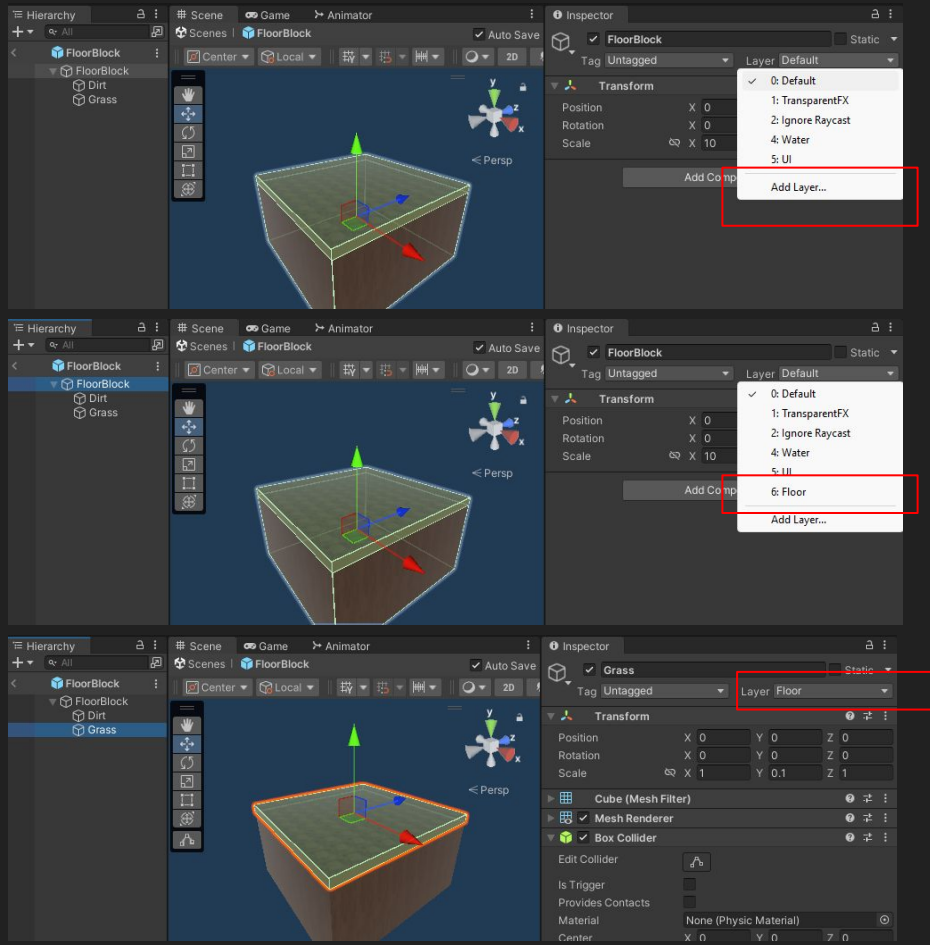
Unity Editor - Checking If Player is Touching Floor (cont.)

- Notice, the trigger has been entered twice:
  - Grass
  - Dirt
  - Both of these have colliders
- Also, it might be possible for the Player to touch their own FloorDetector
  - Or worse: anything will cause this to trigger!



Unity Editor - Layers

- To only check for certain collisions in Unity, you use Layers
- Open the FloorBlock prefab and create a new Layer called Floor
- Set the FloorBlock's Layer to Floor
  - If prompted to set children, select "Yes, change children"
  - If selected "No" accidentally, just set Dirt and Grass' Layer to "Floor"



Unity Editor - Checking Layers

- In the FloorDetector script, we should check only for “Floor”
- Use an integer to keep track of how many floors we are touching
  - If this number is 0 we are in the air!

FloorDetector.cs:

```
...
public class FloorDetector : MonoBehaviour
{
    public int touchingFloorsCount = 0;
    private void OnTriggerEnter(Collider other)
    {
        // If the other collider that collided with the trigger
        // has its layer set to "Floor"
        if (LayerMask.LayerToName(other.gameObject.layer) == "Floor")
        {
            // Increase the number of floors we're touching
            touchingFloorsCount++;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        // If the other collider that stopped colliding with the trigger
        // has its layer set to "Floor"
        if (LayerMask.LayerToName(other.gameObject.layer) == "Floor")
        {
            // Decrease the number of floors we're touching
            touchingFloorsCount--;
        }
    }

    // Create our own method for telling other scripts if we're touching the floor
    public bool IsTouchingFloor()
    {
        return touchingFloorsCount > 0;
    }
}
```

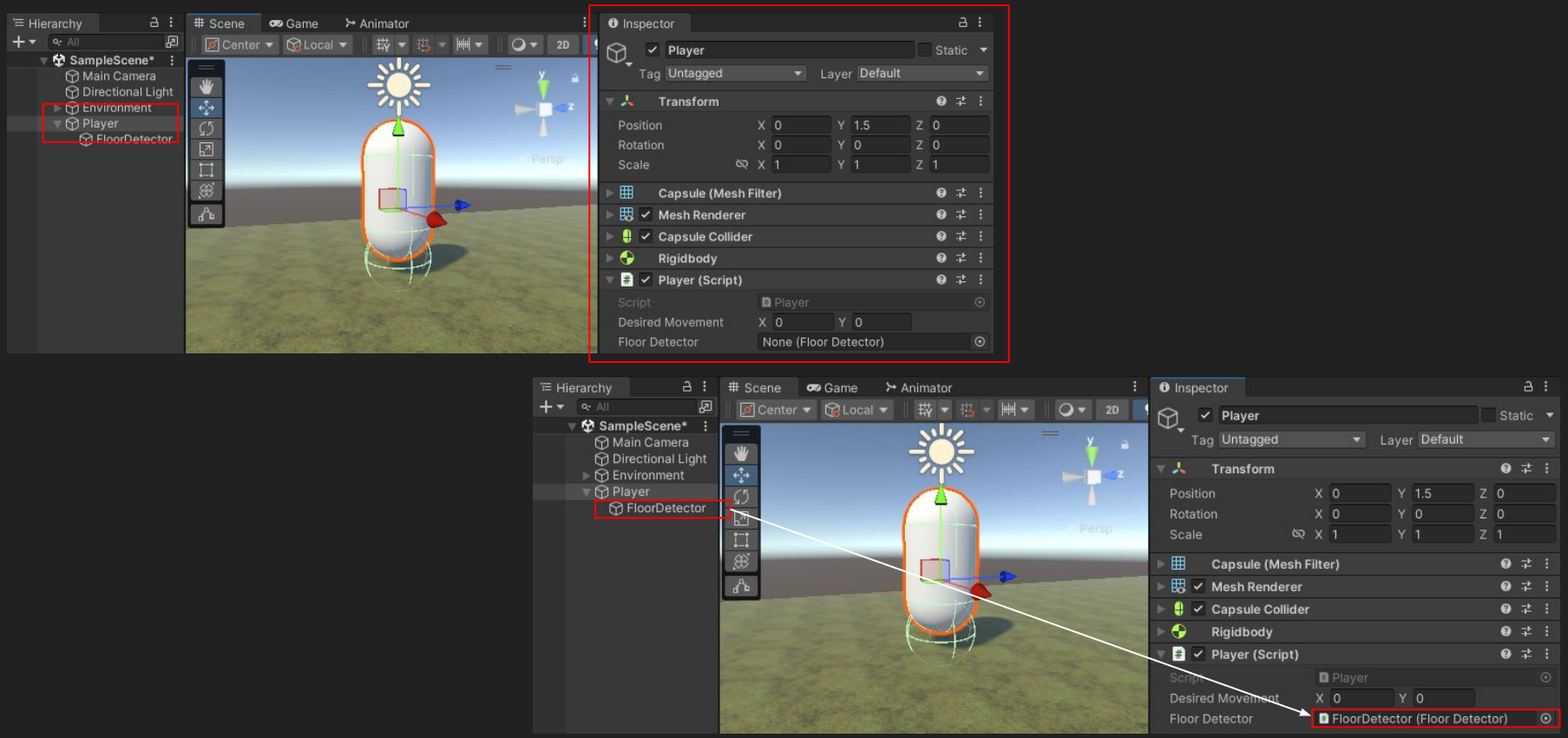
Unity Editor - Referencing Other Scripts

- The Player script and the FloorDetector script are not connected
- To connect them:
  - Add a FloorDetector object in the Player script
  - Save the script!
  - Drag and drop the FloorDetector GameObject from the Hierarchy into this exposed value in the Inspector

Player.cs:

```
...
public class Player : MonoBehaviour
{
    private bool jumpWasPressed;
    public Vector2 desiredMovement;
    private Rigidbody rigidBody;
    public FloorDetector floorDetector;
    ...
}
```

Unity Editor - Referencing Other Scripts (cont.)



Unity Editor - Referencing Other Scripts (cont.)

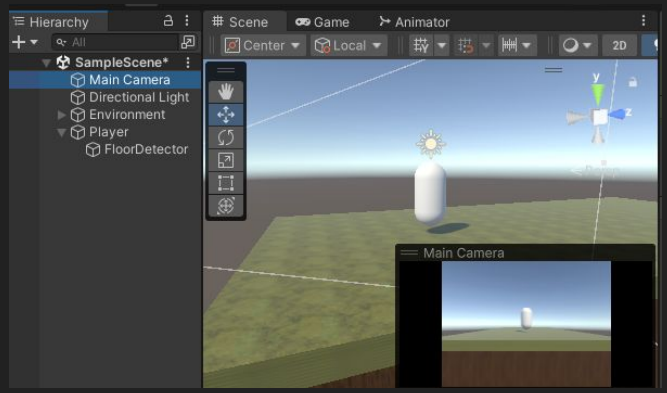
- In the Player script, we should check if floorDetector says we’re touching the floor, before we jump
- Save and rerun!

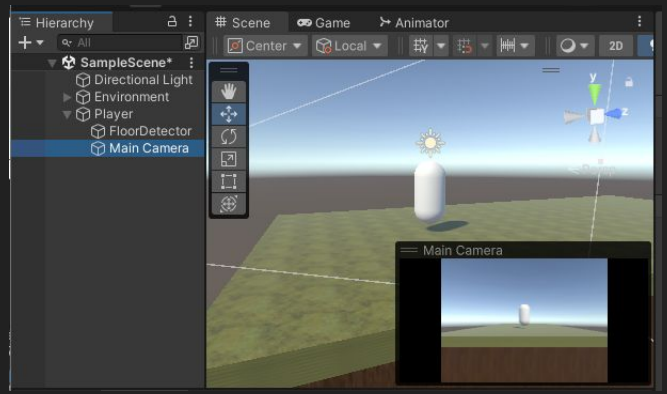
Player.cs:

```
...
    // FixedUpdate is called once per fixed frame
    private void FixedUpdate()
    {
        // Check for the Rigidbody
        // If it's not found, return, we have nothing left to do!
        if (rigidBody == null) return;

        // Ask the floor detector if we're touching the floor
        var isTouchingFloor = floorDetector.IsTouchingFloor();

        // If the user pressed the jump button
        // AND the player is touching the floor
        if (jumpWasPressed && isTouchingFloor)
        {
            // Apply the force
            var force = new Vector3(0f, 10f, 0f);
            rigidBody.AddForce(force, ForceMode.VelocityChange);

            // Reset jumpWasPressed
            jumpWasPressed = false;
        }

        // Get the current velocity
        var currentVelocity = rigidBody.velocity;

        // Apply a new velocity in the x and z direction based on the movement
        var newVelocity = 5f * new Vector3(desiredMovement.x, 0f, desiredMovement.y);

        // Let's not lose the y velocity we gained from jumping!
        newVelocity.y = currentVelocity.y;

        rigidBody.velocity = newVelocity;
    }
...
```

Unity Editor - Camera Following Player

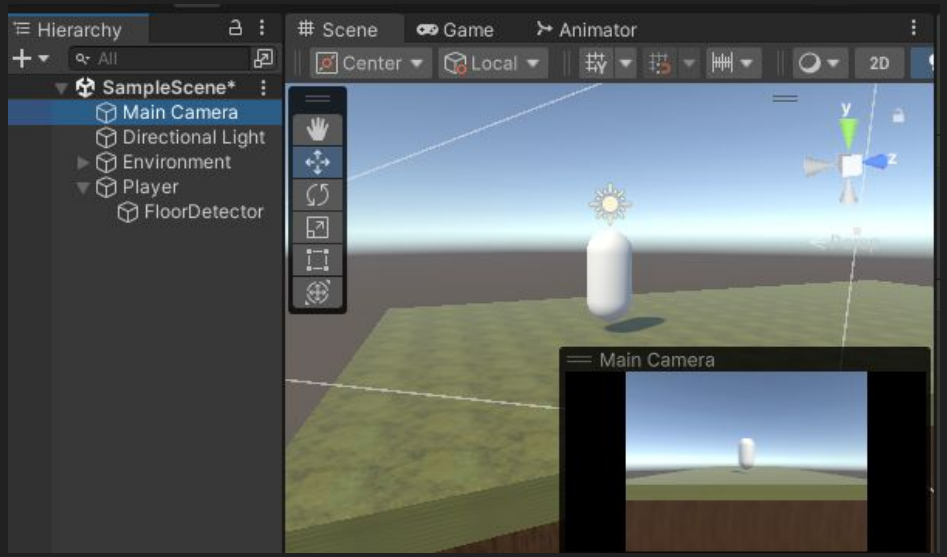
- The Main Camera should follow the player for this game
- Easy way: make the Main Camera a child of Player (drag and drop)
- Hard way (better way): update the Main Camera's position and rotation based on the Player





Unity Editor - Camera Following Player (cont.)

- Move the Main Camera out of the Player
- Create a new script called “CameraFollow”
- We need to use Input to get the mouse’s movement
- Then, we update the Camera’s transform based on this



Unity Editor - Referencing Other Scripts (cont.)

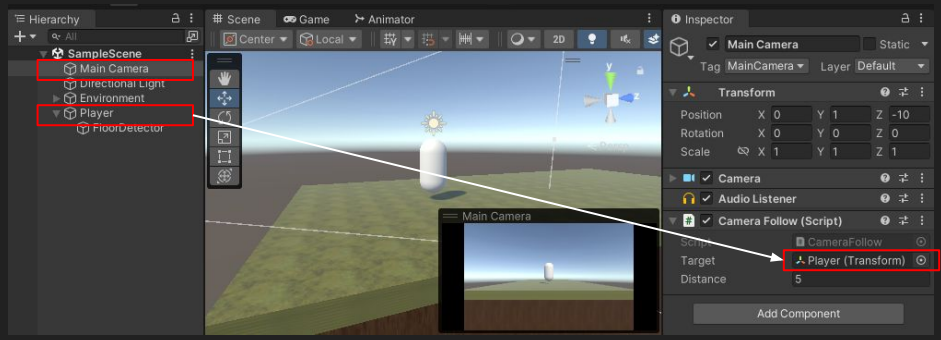
- Use another Transform called “Target”
  - This will be our player
- Get the mouse’s input
- Add that to a yaw/pitch rotation
- Given this rotation, multiply by some distance offset
- Apply the distance offset to the camera Transform’s position
- Force the camera Transform to look at the target

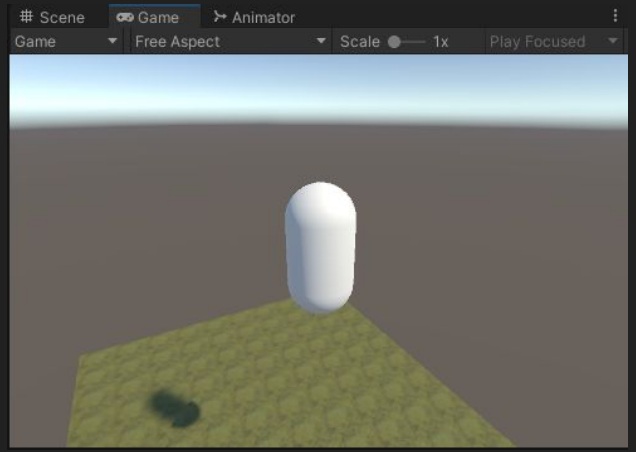
CameraFollow.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    public Transform target;
    public float distance = 5f;

    private float yaw = 0f;
    private float pitch = 0f;

    void Update()
    {
        // Lock the cursor on the screen
        Cursor.lockState = CursorLockMode.Locked;

        // Mouse input
        float mouseX = Input.GetAxis("Mouse X");
        float mouseY = Input.GetAxis("Mouse Y");

        // Update angles
        yaw += mouseX;
        pitch -= mouseY;

        // Calculate rotation and offset
        Quaternion rotation = Quaternion.Euler(pitch, yaw, 0);
        Vector3 offset = rotation * new Vector3(0, 0, -distance);

        // Apply position and rotation
        transform.position = target.position + offset;
        transform.LookAt(target.position);
    }
}
```

Unity Editor - Camera Following Player (cont.)

- Drag and drop Player into the Target for the CameraFollow script attached to the Main Camera GameObject
- Now the Camera follows the Player and is controlled by the mouse
  - Movement doesn't feel quite right



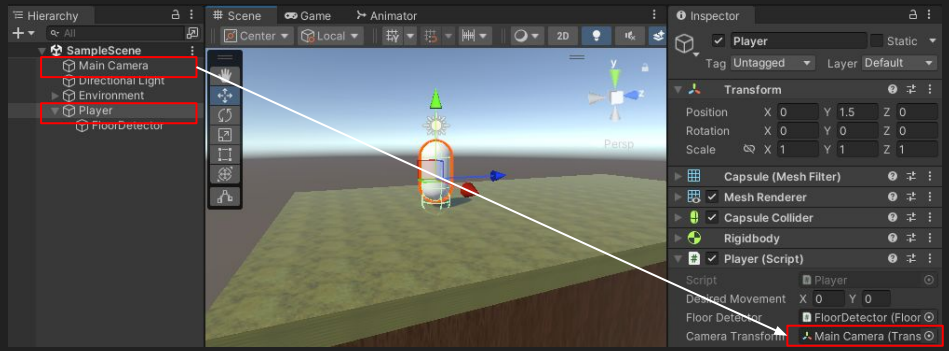


Unity Editor - Player Moves in the Direction of the Camera

- We want to move in the direction of the camera
- Get the camera's transform
- Multiply the camera's rotation by our movement vector
  - This makes the resulting movement vector to point in the direction of our camera's rotation

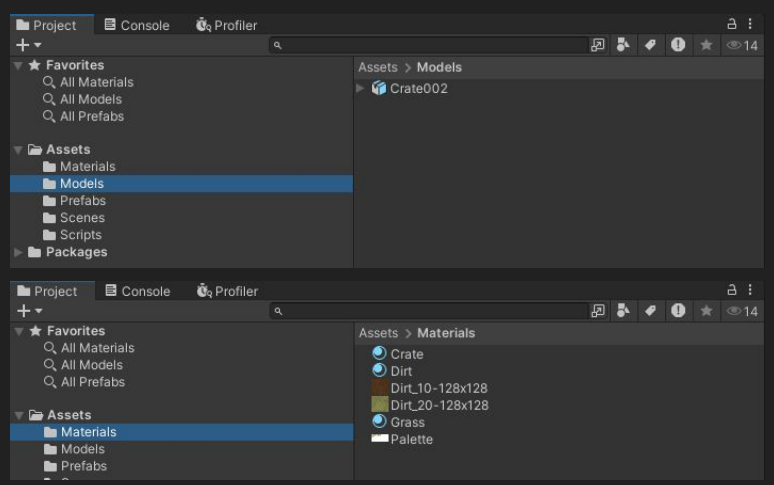
Player.cs:

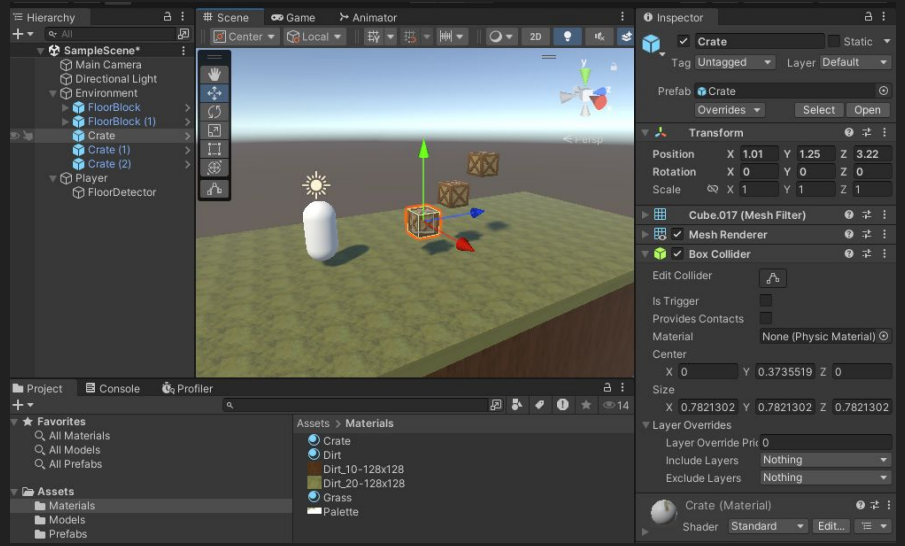
```
...
    public Transform cameraTransform;
...
    private void FixedUpdate()
    {
...
        // Apply movement in the x and z direction based on the camera's rotation
        var localMovement = new Vector3(desiredMovement.x, 0f, desiredMovement.y);
        var newVelocity = 5f * (cameraTransform.rotation * localMovement);
...
    }
}
```



Unity Editor - Adding Custom Models

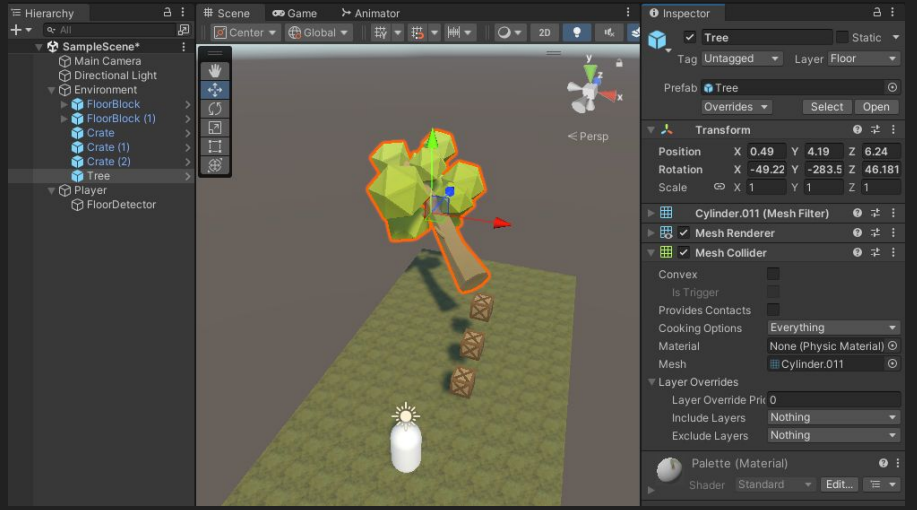
- First, download a model from a reputable source
  - https://itch.io/game-assets/free/tag-models
  - https://braintide.itch.io/low-poly-3d-starter-pack
  - Check the license to see how you are allowed to use it
  - It would be good practice to keep a list of licenses in the root of your project
  - It should be an FBX file for use in Unity
- Create a new folder called “Models”
- Drag and drop an FBX model into this folder
- Drag and drop the FBX file into the Scene
  - Add colliders to the new GameObject
  - Set the Layer to “Floor” if the Player should be able to jump off it
  - Don’t forget to create a Prefab afterwards!
- Create new Materials as necessary





Unity Editor - Adding Custom Models

- For models with non-primitive shapes (trees, cars, etc.) use a Mesh Collider
  - Mesh Colliders are bad for performance, but are ok if only a few are used



Unity Editor - Creating and Destroying GameObjects

- GameObjects can be created and destroyed at runtime
- We might want to give the Player a projectile
- Create a Sphere with a Rigidbody
- Create a script called “Projectile” and attach it to the Sphere
- In the Projectile script, add a force to the Rigidbody in the Start() method
- Multiply this force by the projectile’s starting rotation

Projectile.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Projectile : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        var forceInDirection = transform.rotation * new Vector3(0f, 0f, 20f);
        GetComponent<Rigidbody>().AddForce(forceInDirection, ForceMode.Impulse);
    }
}
```

Unity Editor - Creating and Destroying GameObjects (cont.)

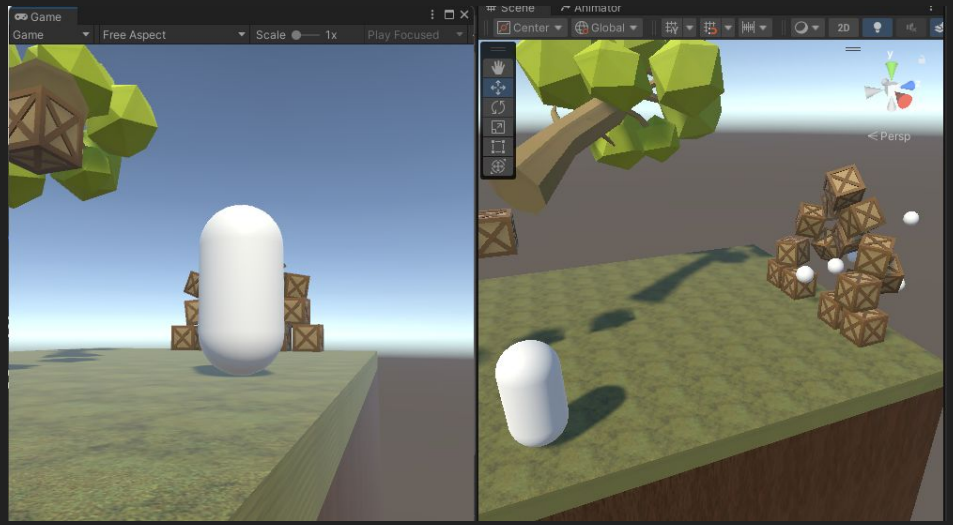
- Write a function in Player that instantiates a new Prefab in the Scene
- Set the projectile’s rotation to the camera’s rotation
- Set the projectile’s position to the Player’s position plus a small offset
- Destroy the projectile after 5 seconds

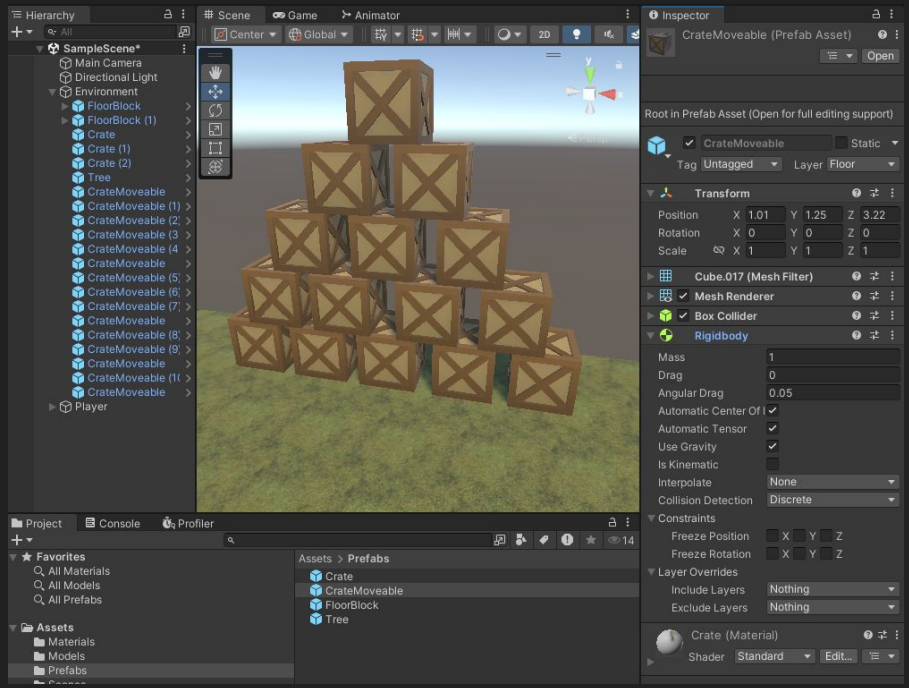
Player.cs:

```
...
   void Update()
   {
...
        // If the user hits the mouse button, fire the projectile
        if (Input.GetMouseButtonDown(0))
        {
            FireProjectile();
        }

        // desiredMovement has 2 directions, X and Y, for horizontal and vertical
        desiredMovement.x = Input.GetAxis("Horizontal");
        desiredMovement.y = Input.GetAxis("Vertical");
    }
...
    private void FireProjectile()
    {
        // Instantiate a new projectile without a parent
        var newProjectile = Instantiate(projectile, null);

        // Calculate the direction the camera is facing in
        var direction = (transform.position - cameraTransform.position).normalized;

        // Set the projectile’s position and rotation
        newProjectile.transform.rotation = cameraTransform.rotation;
        newProjectile.transform.position = 0.5f * direction + transform.position;

        // Destroy the projectile after 5 seconds
        Destroy(newProjectile, 5f);
    }
...
```

Unity Editor - Creating and Destroying GameObjects (cont.)

- To play with projectiles, create a bunch of stacked boxes with Rigidbodies





A Foundation To Build Upon

- It would be impossible to cover everything Unity has to offer in these slides
- Now you know the basics:
  - Transforms
  - Colliders/Rigidbodies
  - GetComponent
  - Instantiating Prefabs
  - Referencing Other GameObjects
- When in doubt, read the documentation:
  - https://docs.unity3d.com/2022.3/Documentation/Manual/UnityManual.html
- If you want to figure out how to do something in Unity, chances are someone has done it before:
  - “How can I make enemies with health bars that the player can battle?”
  - https://www.youtube.com/watch?v=oLT4k-Irnwg
  - “How can I make an open world to explore?”
  - https://www.youtube.com/watch?v=WbZpj8WcjN0

How to Build Bigger and Better Games

- Your first game will be bad
  - Make another one!
  - Learn from mistakes and struggles
- Start extremely simple
  - Remember: GTA V took 1,000+ people and $250 million dollars to make!
- Keep making more until you accumulate so many skills that you can finally make something competent
- Try to find joy in making small gradual progress