Unity3D Basics

CSE165: 3D User Interaction

Robin Xu

Edited by Connor and Robin

Understanding These Slides

C

- Slides are broken into "**C**onceptual" and "**T**echnical" slides
- Denoted by a "C" and a "T", respectively, and color coded
- Conceptual slides are best understood through live workshops/lecture
- Technical slides are best reviewed on your own & **practiced**
- Why do we use this method?
  - Concepts are easy to remember and recall. Lines of code & methods aren't
  - Conceptual talks are good for live anecdotes, examples, and explanation
  - Concepts introduce what's possible, rather than technical ideas
  - Live implementation is hard unless everyone has equipment with them
  - Everybody works at different speeds, so self-paced implementation is best

Agenda

C

- Introduction to Scripting
- MonoBehaviours & Debugging
- Variables and Serialization
- Components
- Raycasting
- Instantiation
- Colliders and Triggers
- Coroutines
- Linear Interpolation (Lerp)

What We’ll Be Covering

Unity Scripting: C#

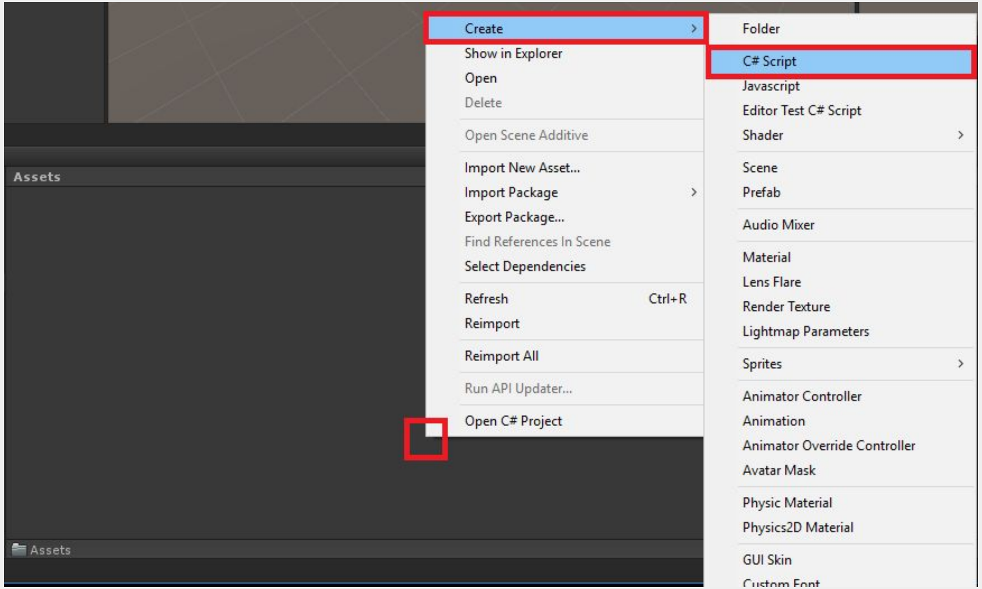
- “Scripting” in Unity is the programming side of game development
- Unity primarily uses the **C#** language (C Sharp).
  - C# is *very* similar to Java, another programming language.
- C# is ideal for game development because it’s very *object-oriented*!
  - After all, everything we want to interact with is a GameObject!
  - Much easier to write code if we can think in terms of objects.
- Unity Scripting is primarily interacting with GameObject components.
  - GameObjects are just collections of components.
  - Modifying components at runtime gives us dynamic control over the game.
  - I.e. How can we change things at ***runtime?***

Scripts As Components

- … but what is a script in Unity?
- Scripts are really just **custom components**!
- When you create a Script, you're creating your very own component
  - You can give that component behaviour, and even create your own fields!
- You add scripts to GameObjects just like any other component
- Once it's added, your script will appear in the Inspector as well
  - With all the other components!
  - We'll go over how to add your own editable fields in later slides

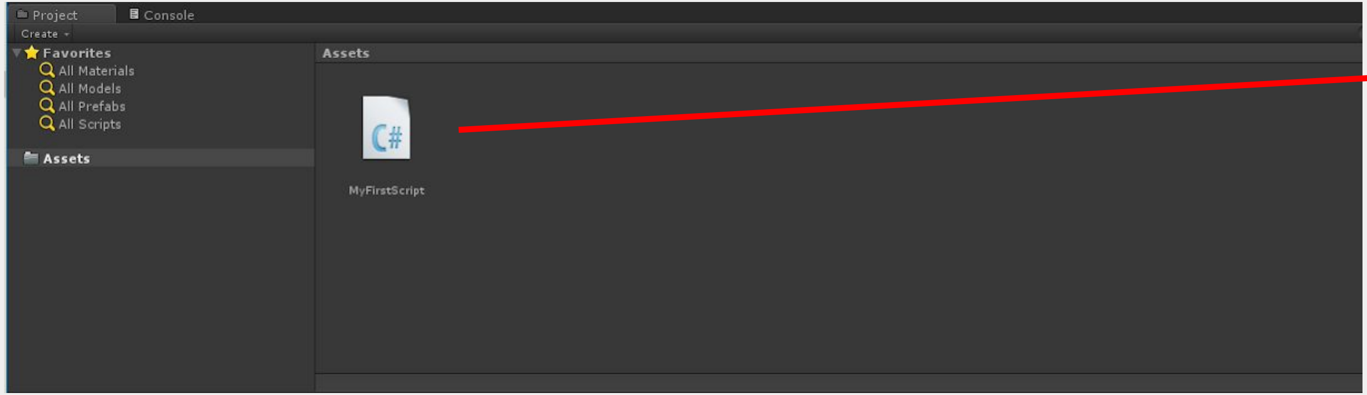
Scripts As Components

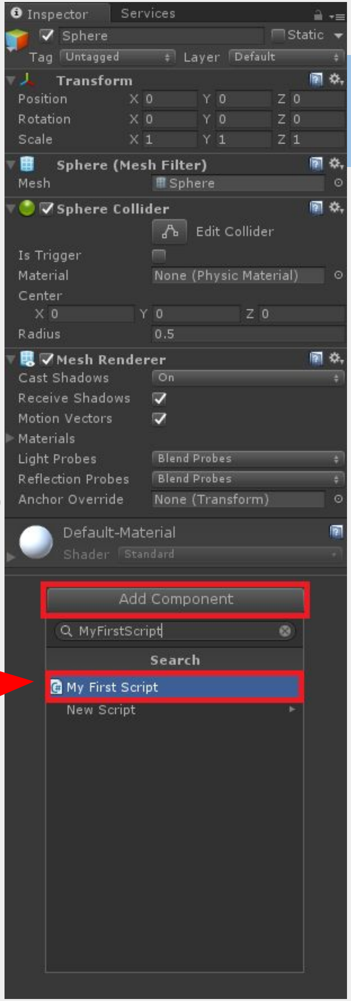
- You can create a new C# script from inside Unity!
  - Right click in your Assets folder -> “Create” -> “C# Script”
  - Give it a name!



Scripts As Components

- First, select an object to add your script to
  - Remember, all GameObjects have components!
- Click “Add Component” in the Inspector
  - Type in the name of your script and add it!
  - You can also just drag your script into the Inspecto





☑ Sphere ☐ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Sphere (Mesh Filter)
Mesh Sphere

☑ Sphere Collider
Is Trigger ☐
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0.5

☑ Mesh Renderer
Cast Shadows On
Receive Shadows ☑
Motion Vectors ☑
Materials
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)

Default-Material
Shader Standard

MyFirstScript

Scripts

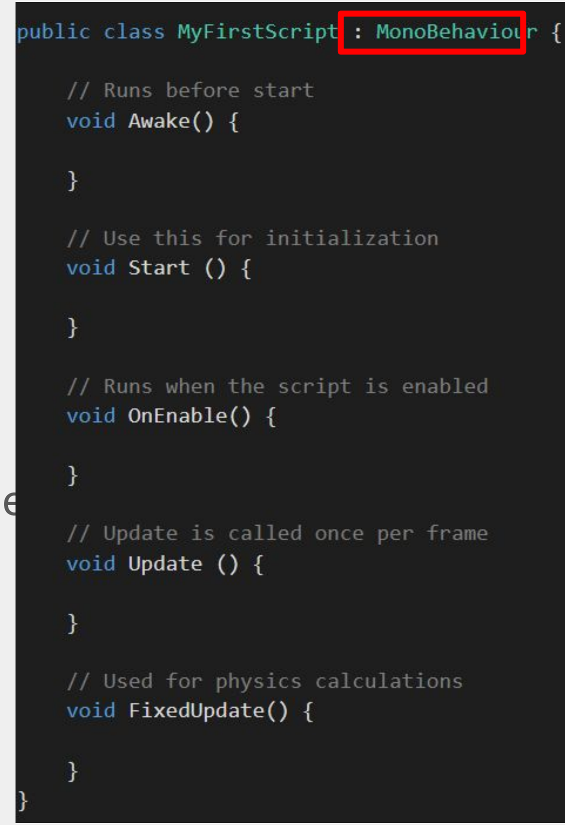
- We’re now ready to dive into our new script!
- Go ahead and open your C# script.
  - If you’re on Windows, this should open in Visual Studio.
  - If you’re on Mac, it will open in MonoDevelop
  - Both of these are fine, they’re just different development environments
- You’ll first notice a few things…
  - “MonoBehaviour”
  - “Start()”
  - “Update()”

MonoBehaviour

- All scripts in Unity are children of a class called **MonoBehaviour**
- Most importantly, MonoBehaviour provides us with our **core game loop**
- This comes in the form of a function called **Update()**
    - Update runs once every single frame, automatically
    - This means it could run ~90 times/second in VR!
- You also get access to other MonoBehaviour functions
    - Awake() - runs before the first frame of the game
    - Start() - runs on the first frame of the game
    - FixedUpdate() - runs at a fixed interval, independent of framerate
    - Execution order: https://docs.unity3d.com/Manual/ExecutionOrder.html

MonoBehaviour

- Notice the class “extends” MonoBehaviour
- “Awake()” runs first, before the game starts
- “Start()” runs first frame, use for initialization
- “OnEnable()” runs when the script is enabled
- “Update()” is called every single frame
- “FixedUpdate” is called at a fixed interval
  - Similar to Update()
  - Doesn’t depend on the framerate of your game
  - Best for physics calculations!



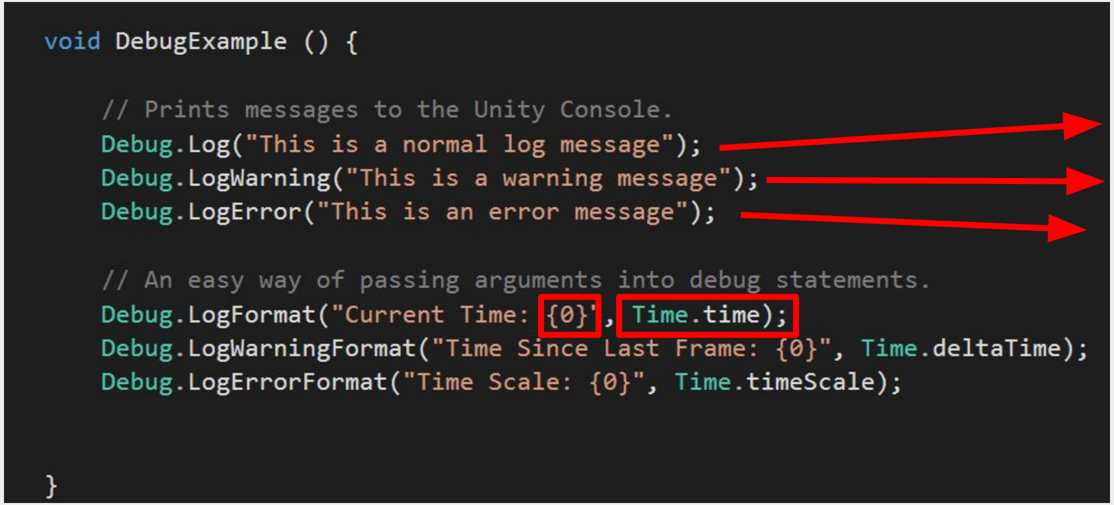
```
public class MyFirstScript : MonoBehaviour {

    // Runs before start
    void Awake() {

    }

    // Use this for initialization
    void Start () {

    }

    // Runs when the script is enabled
    void OnEnable() {

    }

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

    }

    // Used for physics calculations
    void FixedUpdate() {

    }
}
```

The Console and Debugging

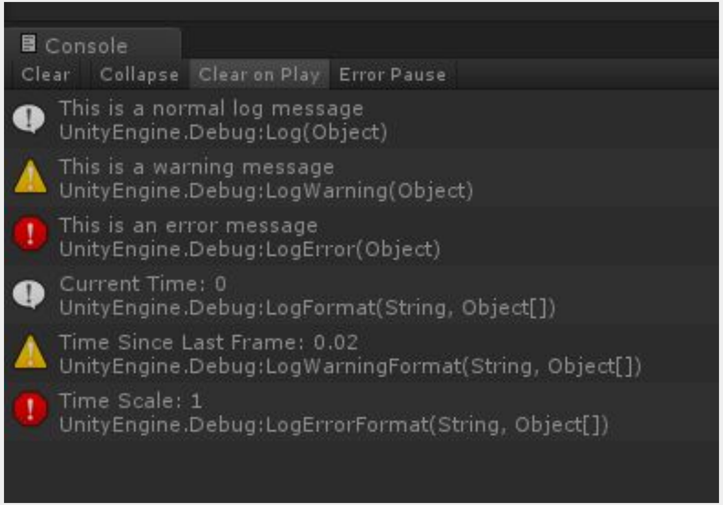
- Debugging in Unity is easy through the Unity **console**
- You should’ve already seen the “Console” tab in your Unity window
- When trying to Debug, any messages are printed to that console
- You can filter by regular messages, warnings, and errors
- You can even pause the editor on a certain line of code!

The Console and Debugging

- To print debug messages in Unity, use Debug.Log(string message)
  - You can also use LogWarning and LogError to filter your messages
  - Use LogFormat to add parameters to your debug messages
- Debug.Break() will pause the editor as soon as it’s reached



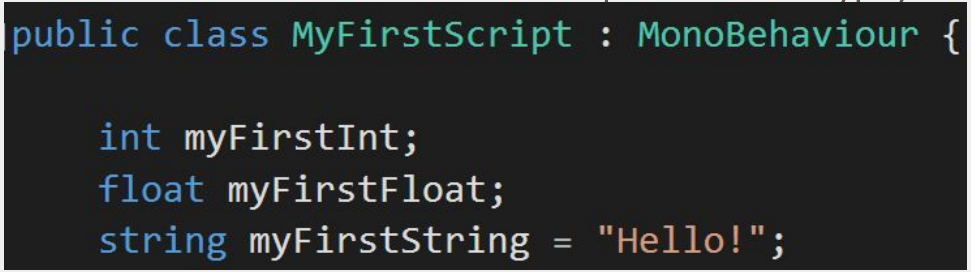
```
void DebugExample () {

    // Prints messages to the Unity Console.
    Debug.Log("This is a normal log message");
    Debug.LogWarning("This is a warning message");
    Debug.LogError("This is an error message");

    // An easy way of passing arguments into debug statements.
    Debug.LogFormat("Current Time: {0}", Time.time);
    Debug.LogWarningFormat("Time Since Last Frame: {0}", Time.deltaTime);
    Debug.LogErrorFormat("Time Scale: {0}", Time.timeScale);


}
```



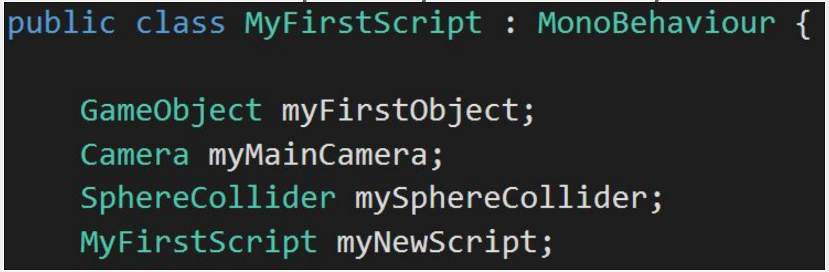
```
This is a normal log message
UnityEngine.Debug:Log(Object)
This is a warning message
UnityEngine.Debug:LogWarning(Object)
This is an error message
UnityEngine.Debug:LogError(Object)
Current Time: 0
UnityEngine.Debug:LogFormat(String, Object[])
Time Since Last Frame: 0.02
UnityEngine.Debug:LogWarningFormat(String, Object[])
Time Scale: 1
UnityEngine.Debug:LogErrorFormat(String, Object[])
```

Variables and Types

- In C#, you get access to all the regular primitive types for variables
  - Int, float, string, bool, etc.
  - Float is most common when using non-integer numbers
  - Vector3 is an extremely important variable that has an x, y, and z value
- However, you also can use **components** and other scripts as types!
  - Thanks MonoBehaviour!
  - Things like Collider, Rigidbody, Material, and etc. are all considered types
  - Your scripts are types too!
  - GameObject is also a type, that references an object in your hierarchy



```
public class MyFirstScript : MonoBehaviour {

    int myFirstInt;
    float myFirstFloat;
    string myFirstString = "Hello!";
```



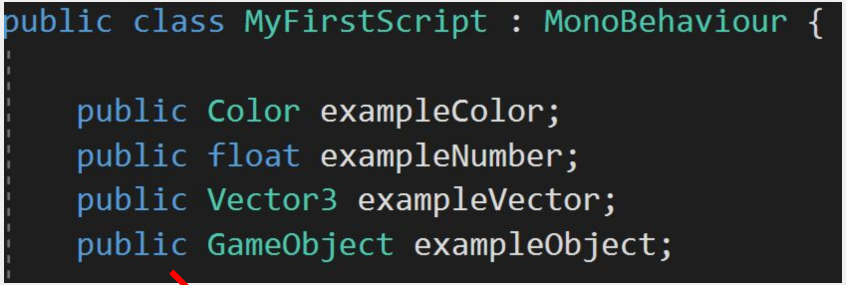
```
public class MyFirstScript : MonoBehaviour {

    GameObject myFirstObject;
    Camera myMainCamera;
    SphereCollider mySphereCollider;
    MyFirstScript myNewScript;
```

Serialized Variables

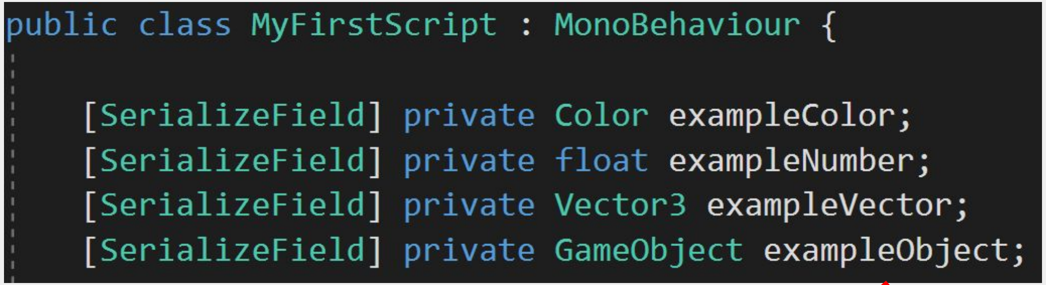
- So, how do you create fields in the Unity Inspector from variables?
  - Components can have values edited from the Unity interface
  - Since our script is a component, we can do the same thing!
- There are two ways to make components appear in the Unity inspector
  - Method 1: Make the variable **public**
  - Method 2: Add a **[SerializeField]** attribute before the variable
- For primitive type variables, you can edit the value from the Inspector
- For non-primitive types (objects/components), you drag in a reference
  - This gives your script immediate access to another object or component!
  - Very handy sometimes for connecting objects in your hierarchy

Serialized Variables

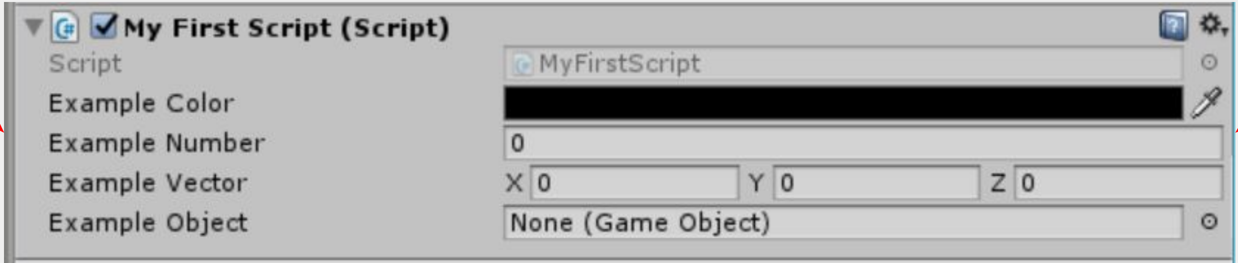
- Add “public” before you’re variable… then look at it in the inspector!
- This can also be done by adding [SerializeField] before the variable.
  - Works for any variable! Try it out!
  - Primitive types can be entered directly. Objects need to be dragged



```
public class MyFirstScript : MonoBehaviour {

    public Color exampleColor;
    public float exampleNumber;
    public Vector3 exampleVector;
    public GameObject exampleObject;
```



```
public class MyFirstScript : MonoBehaviour {

    [SerializeField] private Color exampleColor;
    [SerializeField] private float exampleNumber;
    [SerializeField] private Vector3 exampleVector;
    [SerializeField] private GameObject exampleObject;
```

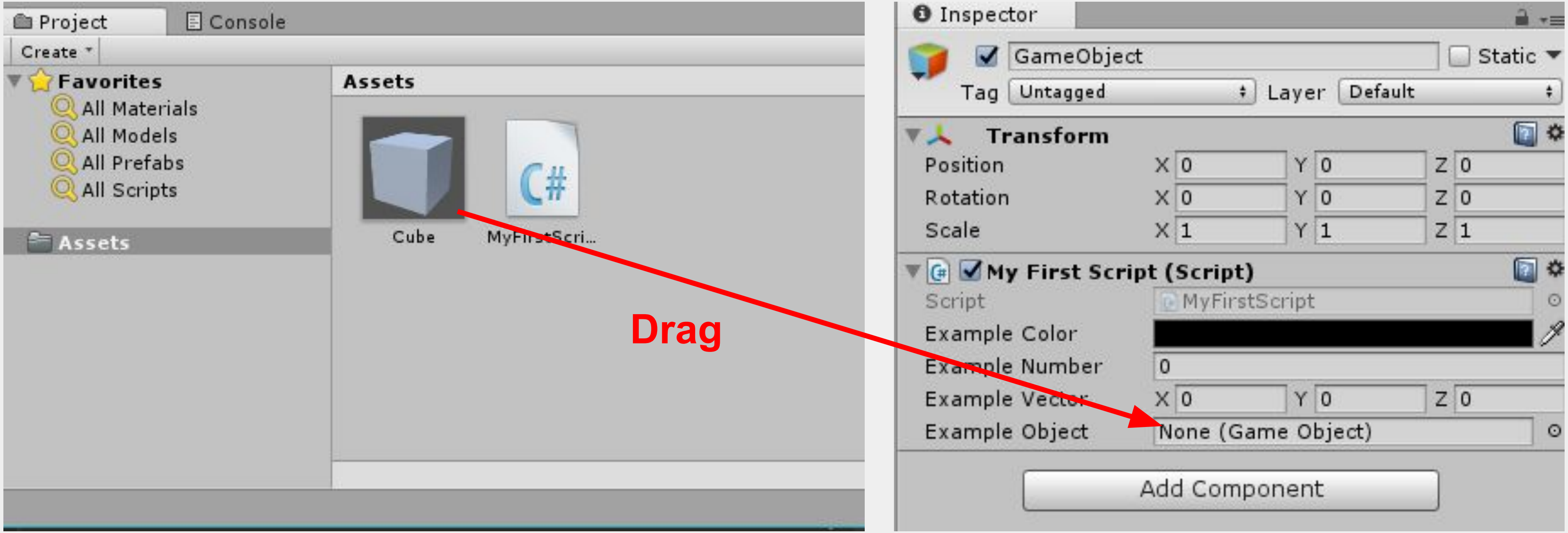


☑ My First Script (Script)

Script	MyFirstScript
Example Color	
Example Number	0
Example Vector	X 0 Y 0 Z 0
Example Object	None (Game Object)

Serialized Variables

- Just drag objects into any object fields to assign references!
- These variables will be set in the script (and override default values)



GameObject ☐ Static

Tag Untagged Layer Default

Transform

Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

☑ My First Script (Script)

Script MyFirstScript

Example Color

Example Number 0

Example Vector X 0 Y 0 Z 0

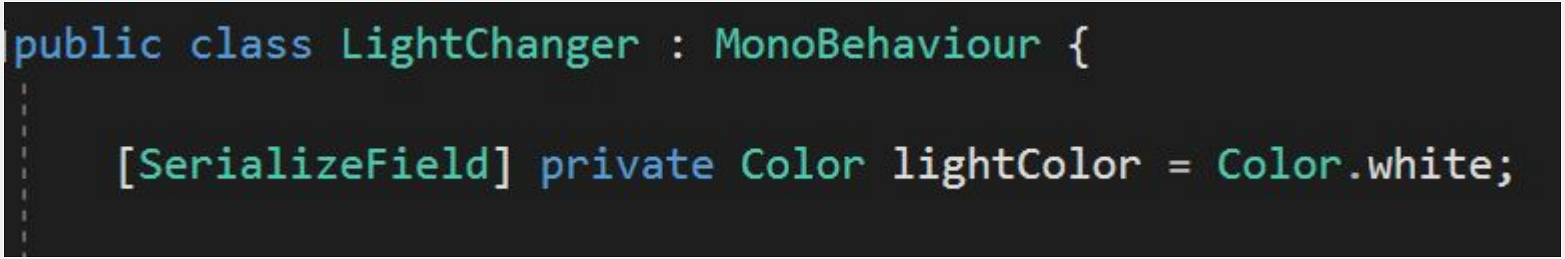
Example Object None (Game Object)

Getting & Modifying Components

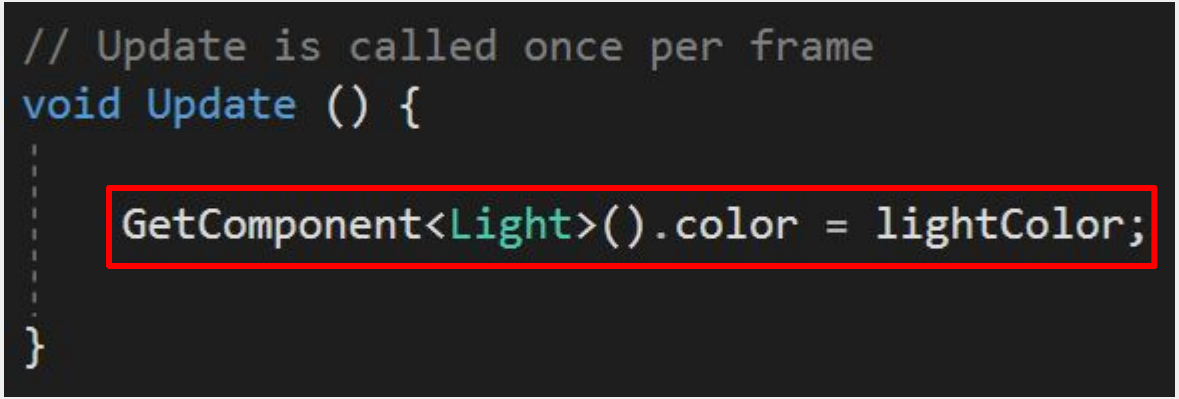
- Getting and modifying components at runtime is **critical** for scripting
  - The behaviour of a GameObject is entirely defined by its components
  - Changing these components at runtime is a **majority of scripting**
- To get a component on an object: **GetComponent<Type>()**
  - “Type” can be any other component type, or even another script name
  - Ex: GetComponent<Collider>(), GetComponent<MyFirstScript>()
- GetComponent, by itself, checks the object **the script is on**
- To check another object, use objectreference.GetComponent<...>()
  - You can get the object reference through SerializedVariables if needed!
  - You could also get it through raycasting and other methods (future slides)

Getting & Modifying Components

- First, create a serialized variable for the light color that you can change
- Then, let’s try using GetComponent on just a simple Light object



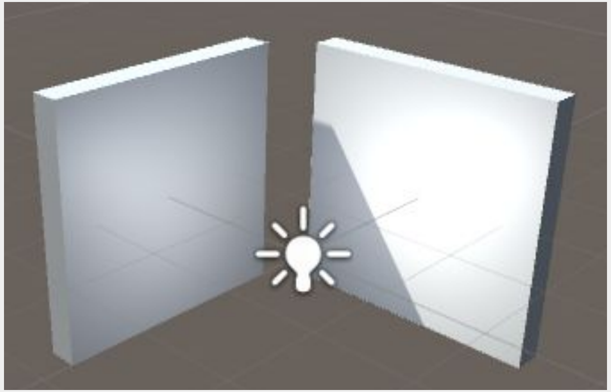
```
public class LightChanger : MonoBehaviour {

    [SerializeField] private Color lightColor = Color.white;

```

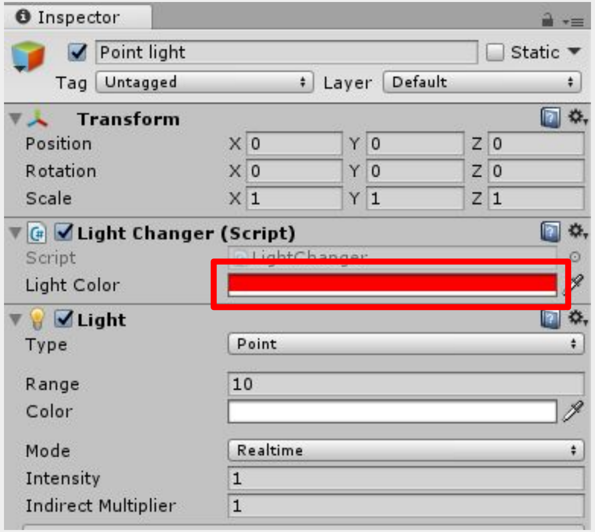


```
// Update is called once per frame
void Update () {

    GetComponent<Light>().color = lightColor;

}
```

Getting & Modifying Components

- Next, make sure to add your LightChanger script to a Light object!
    - What happens if you add it to an object without a Light component?
    - **NullReferenceException!** Can't find the light component, can't change it
- Try changing this color during runtime! It should change live





Inspector

☑ Point light □ Static

Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

☑ **Light Changer (Script)**

Script LightChanger

Light Color 🟥

☑ **Light**

Type Point

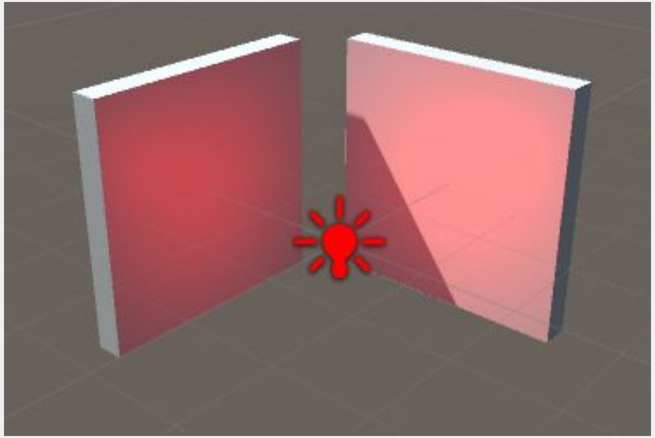
Range 10

Color ⬜

Mode Realtime

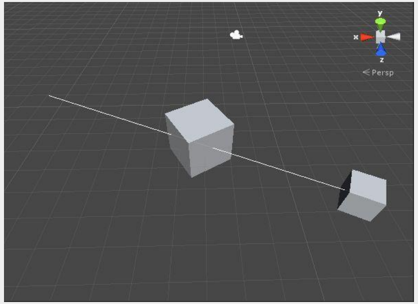
Intensity 1

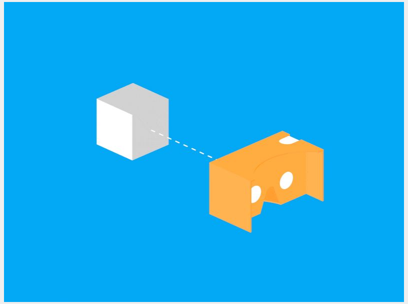
Indirect Multiplier 1

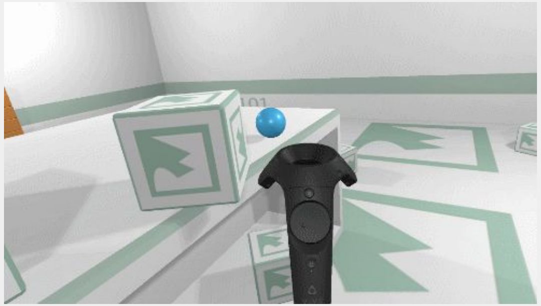


Raycasting

- In 3D space (and in VR), we often need to interact with distant objects
- We can do this through a process called **Raycasting**
- Raycasting involves projecting a 3D ray from a point in a direction
- Once the ray hits something, it returns information about what it hits
- In VR, this is often used with either the HMD or the controller objects
  - Raycasting is the base of **gaze interaction**: Looking at objects to interact
  - With controllers, allows the user to select using rays, or “lasers”

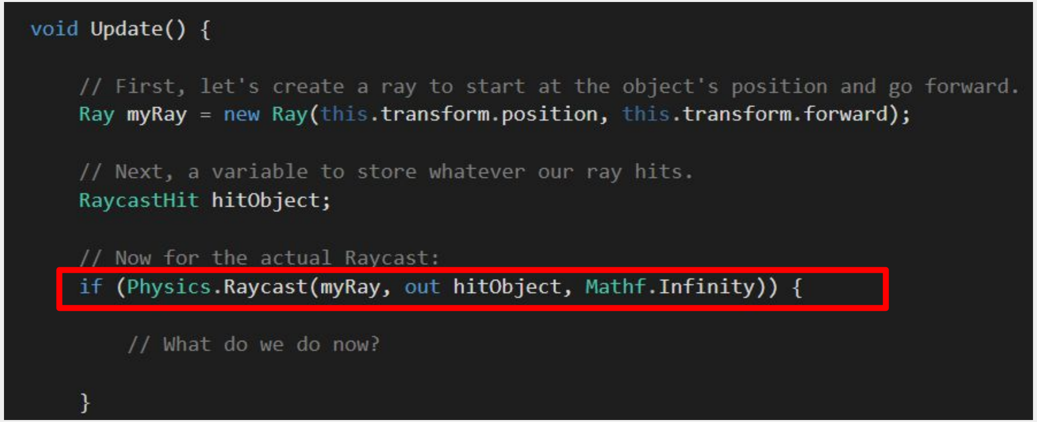






Raycasting

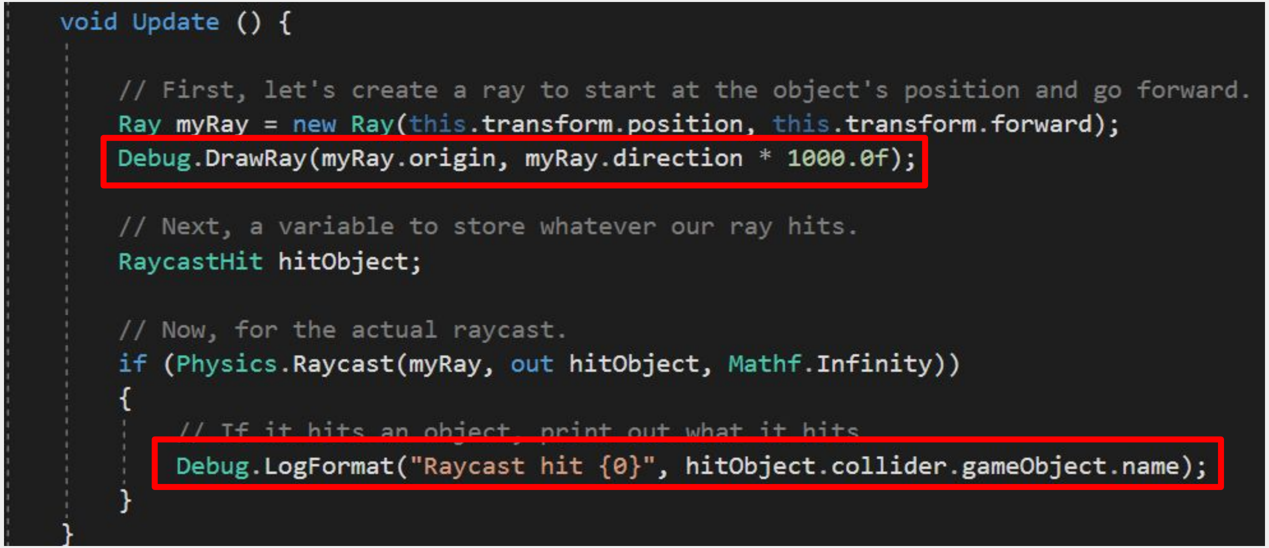
- The function for raycasting in Unity is Physics.Raycast
  - https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
  - **The object you want to hit must have a collider**
- Give an initial position and direction, checks if the ray hits anything
  - Returns “true” if so, “false” otherwise
  - Stores the hit result in “out RaycastHit hitInfo”



```
void Update() {

    // First, let's create a ray to start at the object's position and go forward.
    Ray myRay = new Ray(this.transform.position, this.transform.forward);

    // Next, a variable to store whatever our ray hits.
    RaycastHit hitObject;

    // Now for the actual Raycast:
    if (Physics.Raycast(myRay, out hitObject, Mathf.Infinity)) {

        // What do we do now?

    }
```

Raycasting

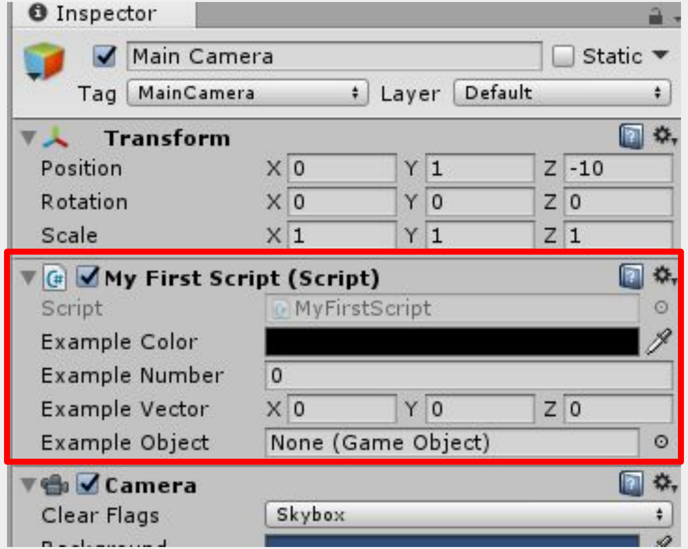
- Let’s try just a really basic raycast from the user’s camera! (Gaze)
  - “Debug.DrawRay” will show a visible ray in the Scene view
- For this one, just print out whatever the ray hits as an example
  - Note we access the object through hitObject’s **collider**



```
void Update () {

    // First, let's create a ray to start at the object's position and go forward.
    Ray myRay = new Ray(this.transform.position, this.transform.forward);
    Debug.DrawRay(myRay.origin, myRay.direction * 1000.0f);

    // Next, a variable to store whatever our ray hits.
    RaycastHit hitObject;

    // Now, for the actual raycast.
    if (Physics.Raycast(myRay, out hitObject, Mathf.Infinity))
    {
        // If it hits an object, print out what it hits.
        Debug.LogFormat("Raycast hit {0}", hitObject.collider.gameObject.name);
    }
}
```

Raycasting

- Now just add this script to the MainCamera and test!



Inspector

☑ Main Camera □ Static

Tag MainCamera Layer Default

Transform

Position X 0 Y 1 Z -10

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

☑ My First Script (Script)

Script MyFirstScript

Example Color

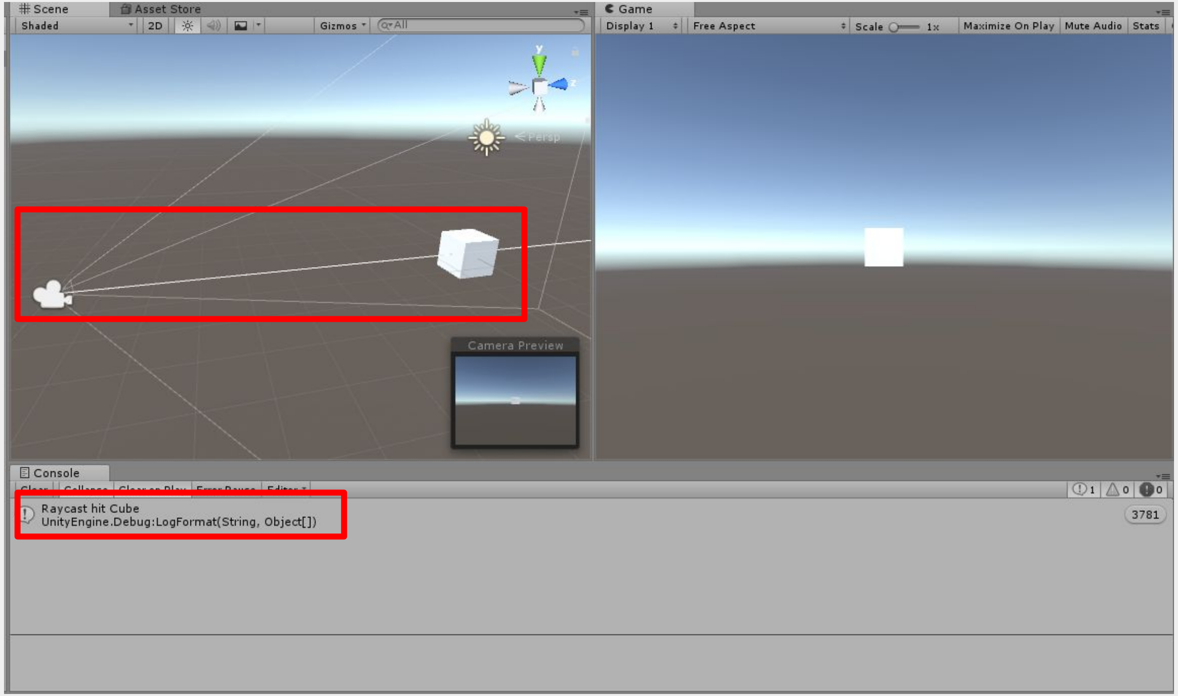
Example Number 0

Example Vector X 0 Y 0 Z 0

Example Object None (Game Object)

☑ Camera

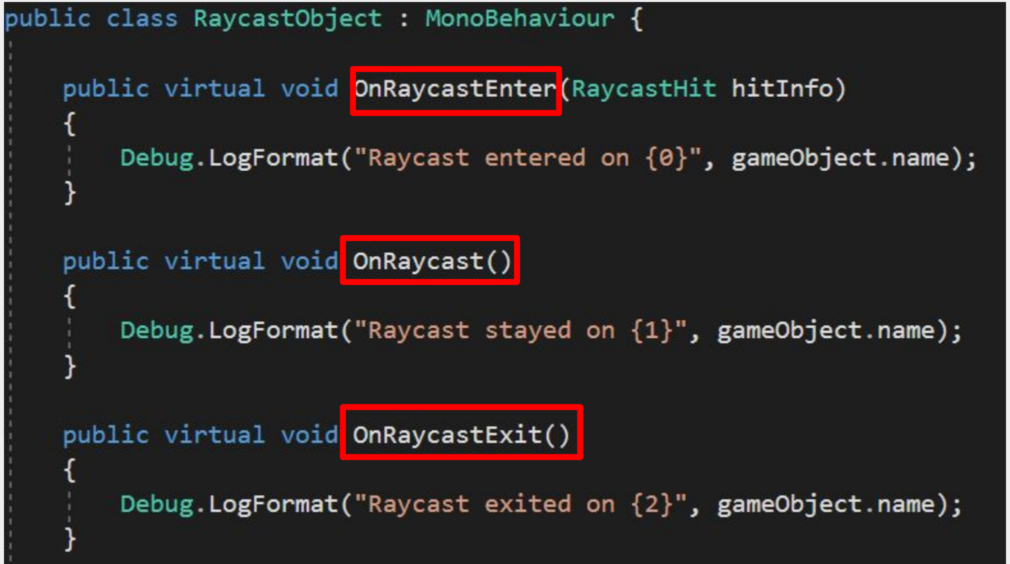
Clear Flags Skybox



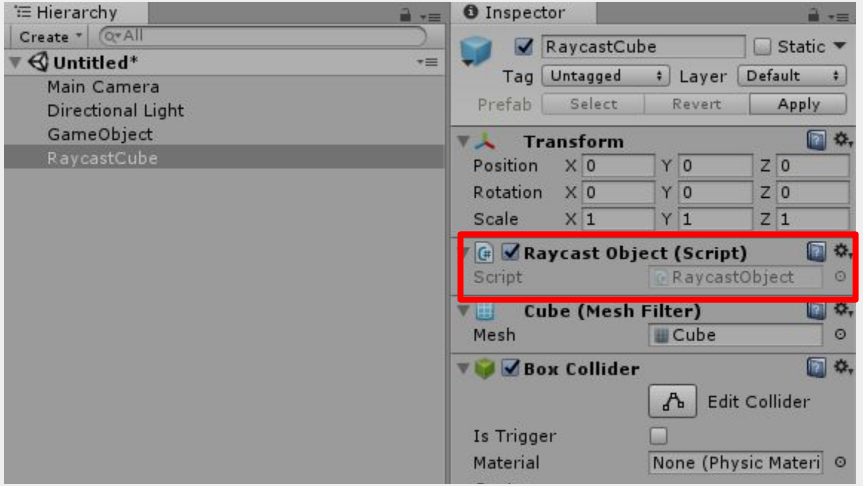
```
Raycast hit Cube
UnityEngine.Debug:LogFormat(String, Object[])
```

Raycasting

- You may not want every single object to respond to raycasting, though
- It may help to make an **InteractableObject** or **RaycastObject** script
  - Then any object that should respond to your raycast can extend it!
  - Make sure to add this script to the object you want to respond



```
public class RaycastObject : MonoBehaviour {

    public virtual void OnRaycastEnter(RaycastHit hitInfo)
    {
        Debug.LogFormat("Raycast entered on {0}", gameObject.name);
    }

    public virtual void OnRaycast()
    {
        Debug.LogFormat("Raycast stayed on {1}", gameObject.name);
    }

    public virtual void OnRaycastExit()
    {
        Debug.LogFormat("Raycast exited on {2}", gameObject.name);
    }
```



☑ RaycastCube □ Static

Tag Untagged Layer Default

Prefab Select Revert Apply

Transform

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

☑ Raycast Object (Script)

Script RaycastObject

Cube (Mesh Filter)

Mesh Cube

☑ Box Collider

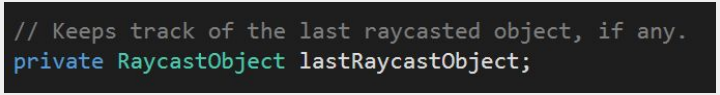
Edit Collider

Is Trigger □

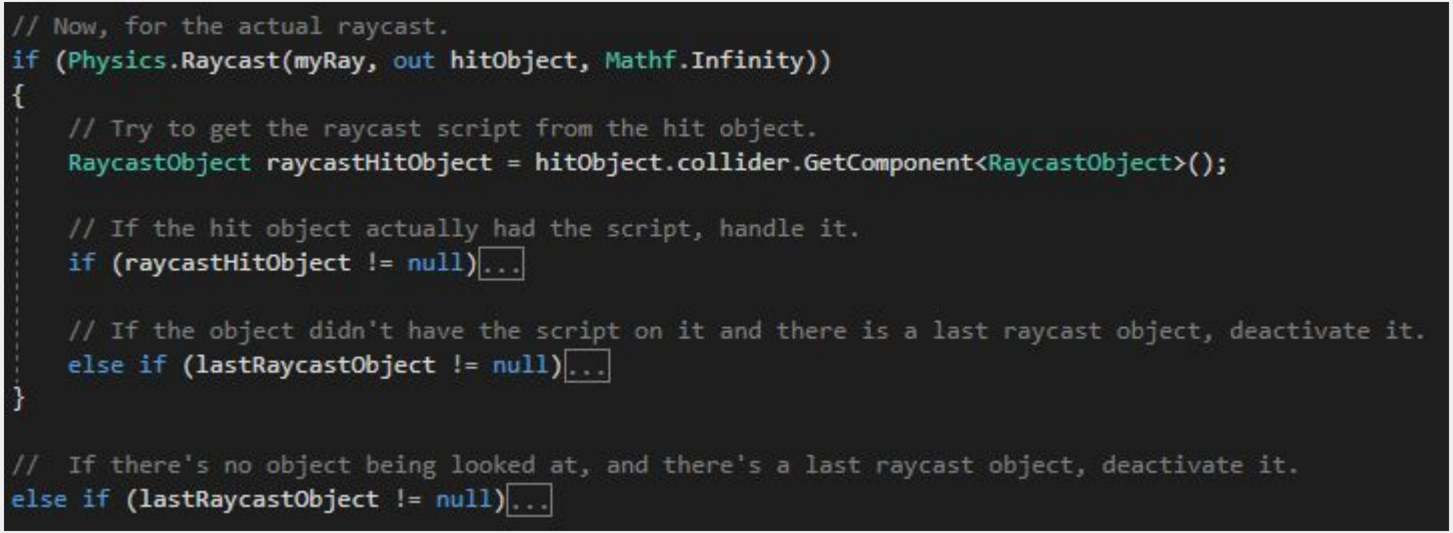
Material None (Physic Materi

Raycasting

- Now, just need to call these RaycastObject functions in our raycast
    - To do this, we also need to keep track of the **last raycast object**
    - Let’s start by looking at a skeleton of all the different conditions



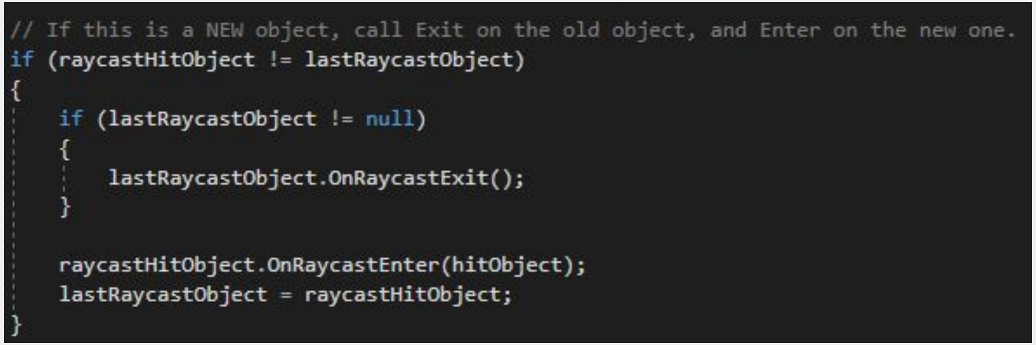
```
// Keeps track of the last raycasted object, if any.
private RaycastObject lastRaycastObject;
```



```
// Now, for the actual raycast.
if (Physics.Raycast(myRay, out hitObject, Mathf.Infinity))
{
    // Try to get the raycast script from the hit object.
    RaycastObject raycastHitObject = hitObject.collider.GetComponent<RaycastObject>();

    // If the hit object actually had the script, handle it.
    if (raycastHitObject != null)...

    // If the object didn't have the script on it and there is a last raycast object, deactivate it.
    else if (lastRaycastObject != null)...
}

//  If there's no object being looked at, and there's a last raycast object, deactivate it.
else if (lastRaycastObject != null)...
```

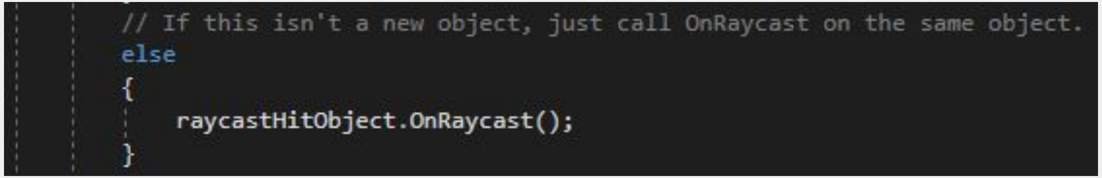
Raycasting

- The first case is when we look at a NEW object for the first time
  - In other words, this object wasn’t being looked at last frame



```
// If this is a NEW object, call Exit on the old object, and Enter on the new one.
if (raycastHitObject != lastRaycastObject)
{
    if (lastRaycastObject != null)
    {
        lastRaycastObject.OnRaycastExit();
    }

    raycastHitObject.OnRaycastEnter(hitObject);
    lastRaycastObject = raycastHitObject;
}
```

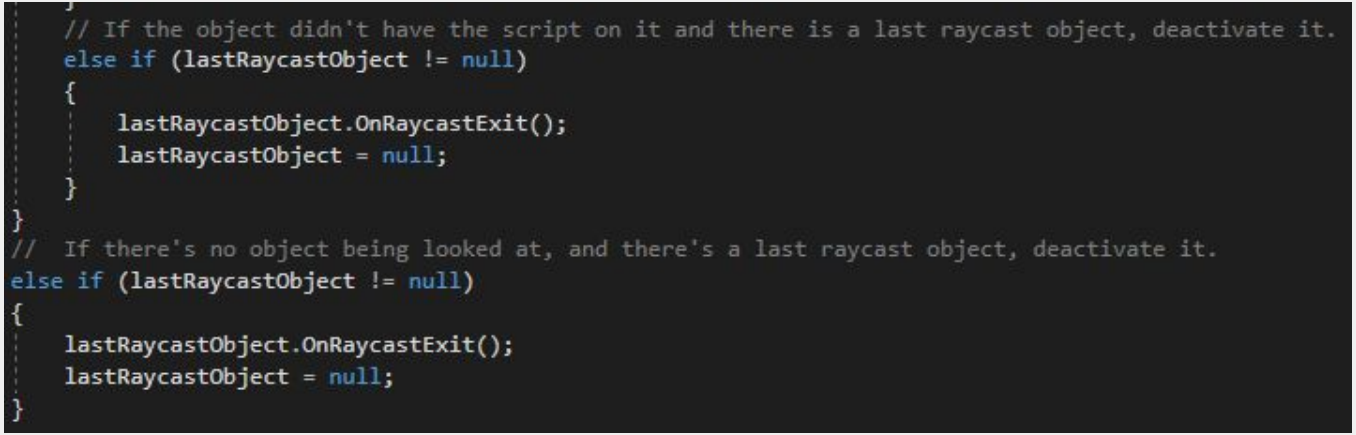
- Otherwise, if it isn’t a new object, just call OnRaycast()
  - OnRaycast should run every frame the object is being looked at/raycasted



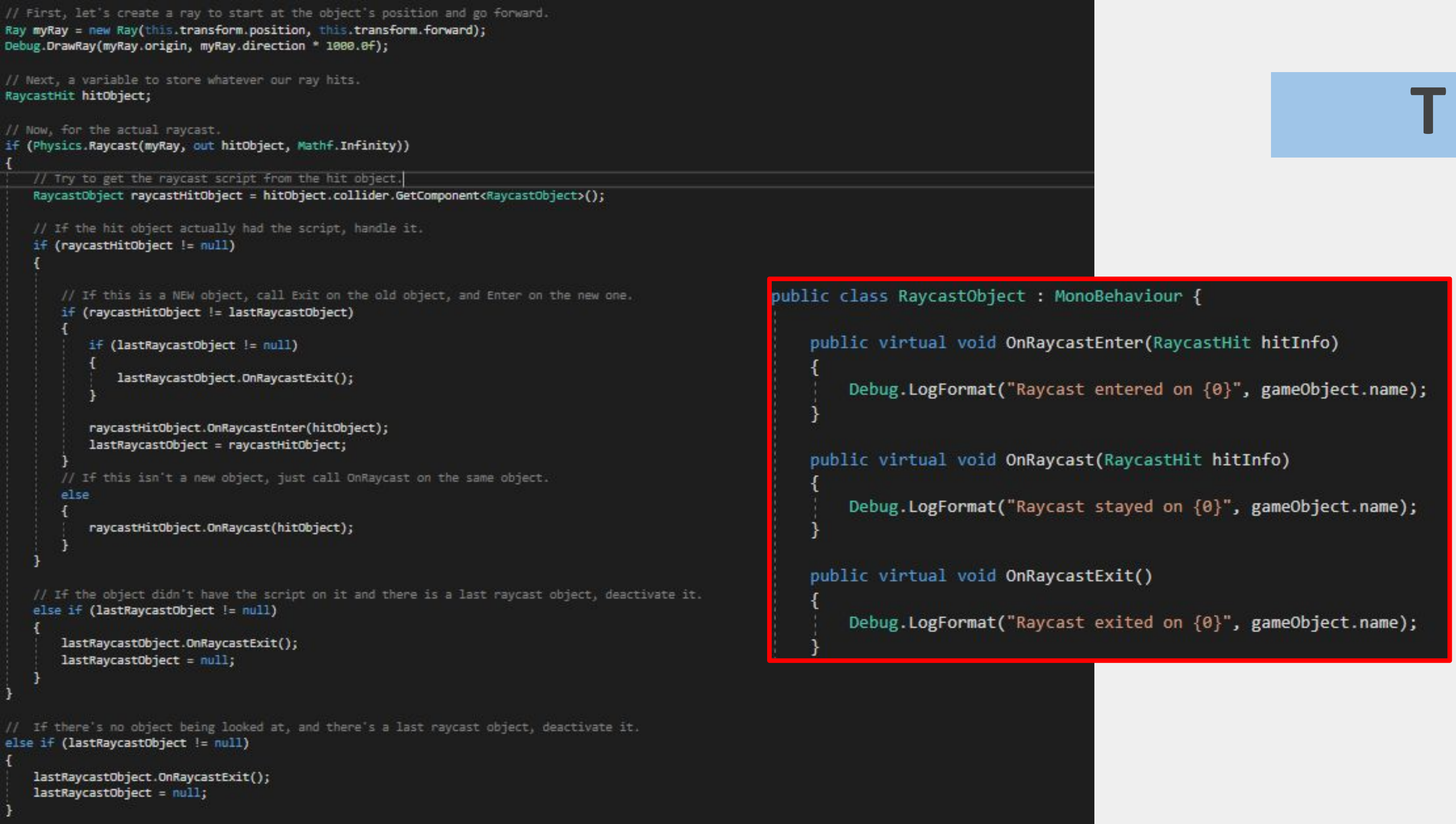
```
        // If this isn't a new object, just call OnRaycast on the same object.
        else
        {
            raycastHitObject.OnRaycast();
        }
```

Raycasting

- Finally, we need cases for when to call OnRaycastExit()
  - This should be called as soon as an object that was raycasted isn’t anymore
  - In other words, if the CURRENT hit object is null or not a raycast object



```
    // If the object didn't have the script on it and there is a last raycast object, deactivate it.
    else if (lastRaycastObject != null)
    {
        lastRaycastObject.OnRaycastExit();
        lastRaycastObject = null;
    }
}
//  If there's no object being looked at, and there's a last raycast object, deactivate it.
else if (lastRaycastObject != null)
{
    lastRaycastObject.OnRaycastExit();
    lastRaycastObject = null;
}
```

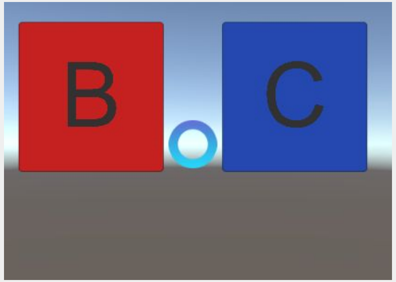


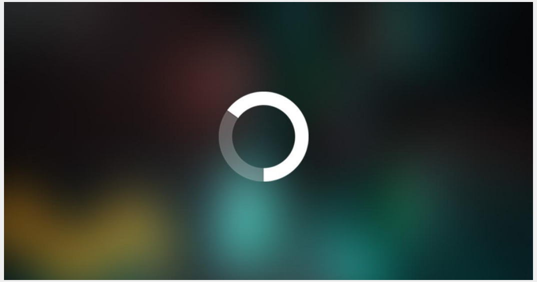
```
// First, let's create a ray to start at the object's position and go forward.
Ray myRay = new Ray(this.transform.position, this.transform.forward);
Debug.DrawRay(myRay.origin, myRay.direction * 1000.0f);

// Next, a variable to store whatever our ray hits.
RaycastHit hitObject;

// Now, for the actual raycast.
if (Physics.Raycast(myRay, out hitObject, Mathf.Infinity))
{
    // Try to get the raycast script from the hit object.
    RaycastObject raycastHitObject = hitObject.collider.GetComponent<RaycastObject>();

    // If the hit object actually had the script, handle it.
    if (raycastHitObject != null)
    {

        // If this is a NEW object, call Exit on the old object, and Enter on the new one.
        if (raycastHitObject != lastRaycastObject)
        {
            if (lastRaycastObject != null)
            {
                lastRaycastObject.OnRaycastExit();
            }

            raycastHitObject.OnRaycastEnter(hitObject);
            lastRaycastObject = raycastHitObject;
        }
        // If this isn't a new object, just call OnRaycast on the same object.
        else
        {
            raycastHitObject.OnRaycast(hitObject);
        }
    }

    // If the object didn't have the script on it and there is a last raycast object, deactivate it.
    else if (lastRaycastObject != null)
    {
        lastRaycastObject.OnRaycastExit();
        lastRaycastObject = null;
    }
}

//  If there's no object being looked at, and there's a last raycast object, deactivate it.
else if (lastRaycastObject != null)
{
    lastRaycastObject.OnRaycastExit();
    lastRaycastObject = null;
}
```

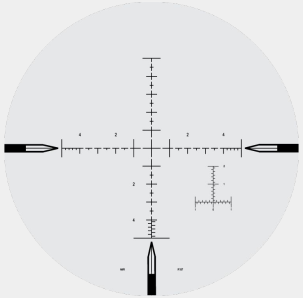
```
public class RaycastObject : MonoBehaviour {

    public virtual void OnRaycastEnter(RaycastHit hitInfo)
    {
        Debug.LogFormat("Raycast entered on {0}", gameObject.name);
    }

    public virtual void OnRaycast(RaycastHit hitInfo)
    {
        Debug.LogFormat("Raycast stayed on {0}", gameObject.name);
    }

    public virtual void OnRaycastExit()
    {
        Debug.LogFormat("Raycast exited on {0}", gameObject.name);
    }
```

Gaze Interaction

- Raycasting is a critical component of **gaze interaction**
  - Gaze interaction involves modifying objects just by looking at them!
  - Most useful for VR devices that don’t have controllers
- For gaze interaction, you generally also want a **gaze cursor**
  - This will signify what the user is looking at at any given time
  - Can also fill/change depending on how long the user is looking at something
  - The concept of this delay before activating is called “dwelling”







Gaze Cursors

T

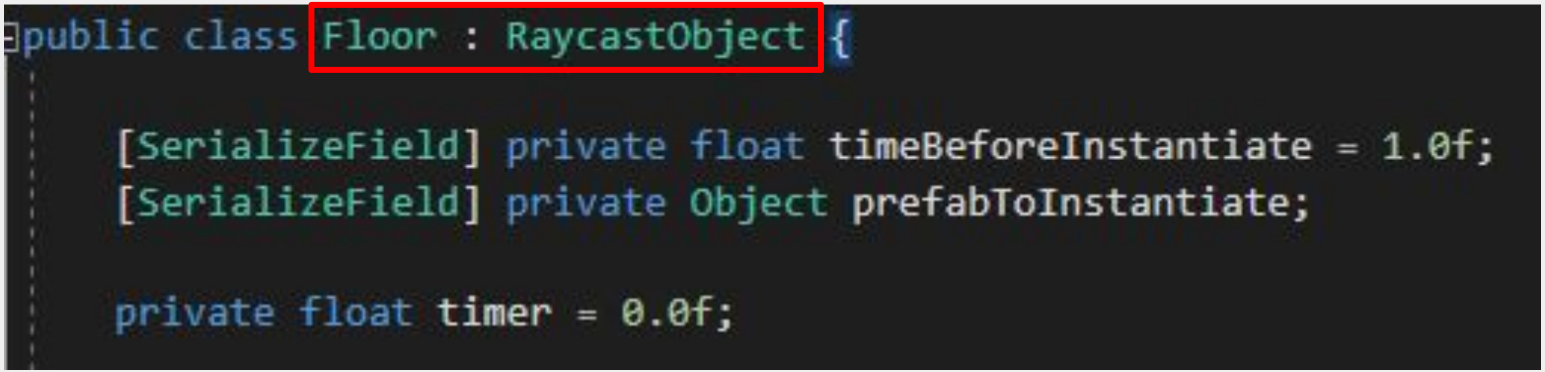
- Create a UI -> Image object and use whatever image you want!
  - A canvas object should be automatically created
  - Set the canvas Render Mode to “World Space”
- Attach and place the cursor in front of the camera
  - This can be done by parenting the canvas to the camera, or using a script
- To always render the cursor over everything, you need a custom shader
  - This shader should go on the actual cursor object (specifically, it’s material)
  - https://answers.unity.com/questions/878667/world-space-canvas-on-top-of-everything.html

Instantiation

- **Instantiation** involves creating GameObjects at runtime
  - Example: Creating cans as the pop out of a soda machine
  - Example: Spawning enemies when you enter a room
  - Example: Firing bullets out of a gun
- This is done by cloning a **Prefab** or an existing GameObject
- The function for instantiation is **GameObject.Instantiate**
  - https://docs.unity3d.com/ScriptReference/Object.Instantiate.html
  - Technically, all you need for this is the prefab/GameObject
  - You can also provide a location (Vector3) and rotation (Quaternion)

Instantiation

- Let’s try using our existing raycast functionality to instantiate objects
  - When we look at the floor of a room, spawn a specified prefab
  - Wait a period of time (dwell) before spawning
- To do this, we’ll need a **Floor.cs** script

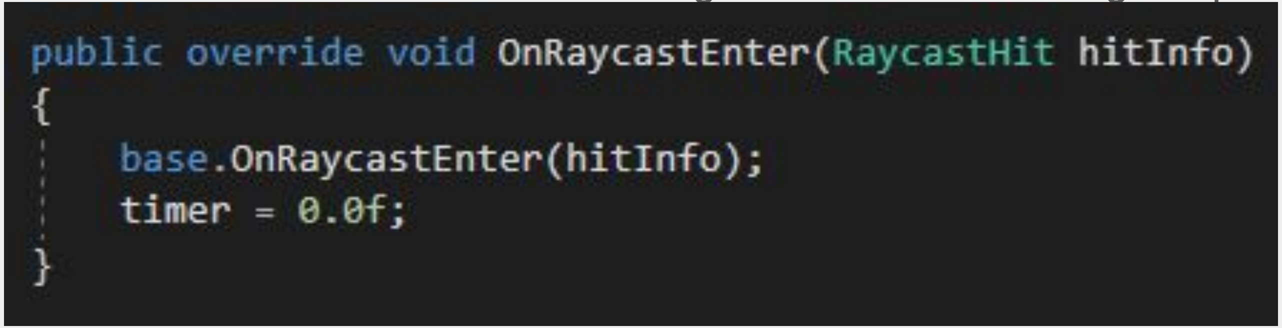


```
public class Floor : RaycastObject {

    [SerializeField] private float timeBeforeInstantiate = 1.0f;
    [SerializeField] private Object prefabToInstantiate;

    private float timer = 0.0f;
```

Instantiation

T

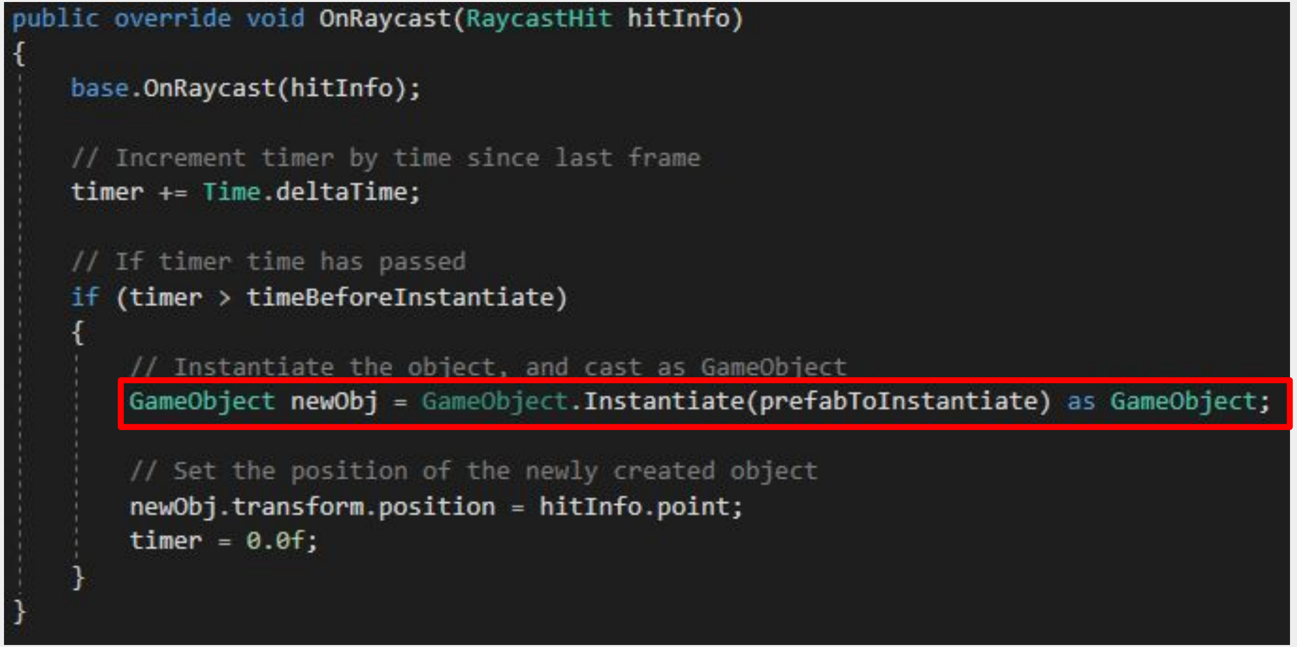
- What should happen when the user first looks at the floor?
    - Just reset the timer!
    - Also need to call “base.OnRaycastEnter” to maintain parent functionality
    - Note the “override” in the signature too. Overriding the parent



```
public override void OnRaycastEnter(RaycastHit hitInfo)
{
    base.OnRaycastEnter(hitInfo);
    timer = 0.0f;
}
```

Instantiation

- Now, what’s the core functionality of spawning objects on the floor?
  - Need to check the timer… then **Instantiate** if time has passed!

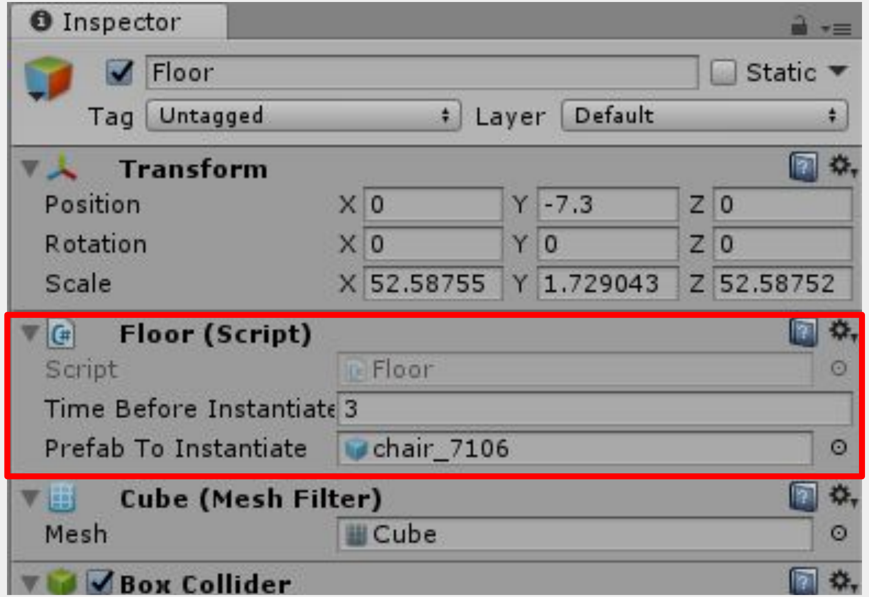


```
public override void OnRaycast(RaycastHit hitInfo)
{
    base.OnRaycast(hitInfo);

    // Increment timer by time since last frame
    timer += Time.deltaTime;

    // If timer time has passed
    if (timer > timeBeforeInstantiate)
    {
        // Instantiate the object, and cast as GameObject
        GameObject newObj = GameObject.Instantiate(prefabToInstantiate) as GameObject;

        // Set the position of the newly created object
        newObj.transform.position = hitInfo.point;
        timer = 0.0f;
    }
}
```

Instantiation

T

- Now, create a floor object and put this new script on it!
  - You can just use a regular cube, and scale it to create a floor
  - Be sure to drag in the prefab you want to instantiate and set a timer val



Inspector

☑ Floor □ Static

Tag Untagged Layer Default

Transform

Position X 0 Y -7.3 Z 0

Rotation X 0 Y 0 Z 0

Scale X 52.58755 Y 1.729043 Z 52.58752

Floor (Script)

Script Floor

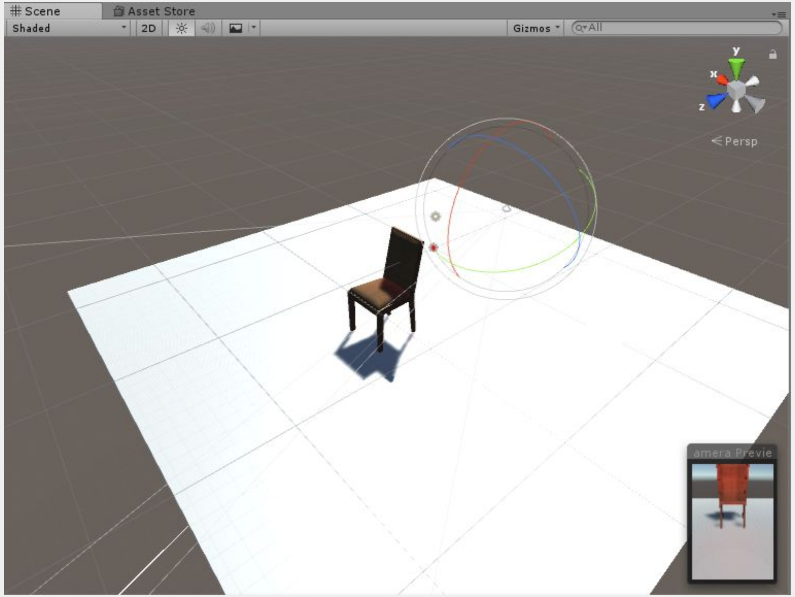
Time Before Instantiate 3

Prefab To Instantiate chair_7106

Cube (Mesh Filter)

Mesh Cube

☑ Box Collider

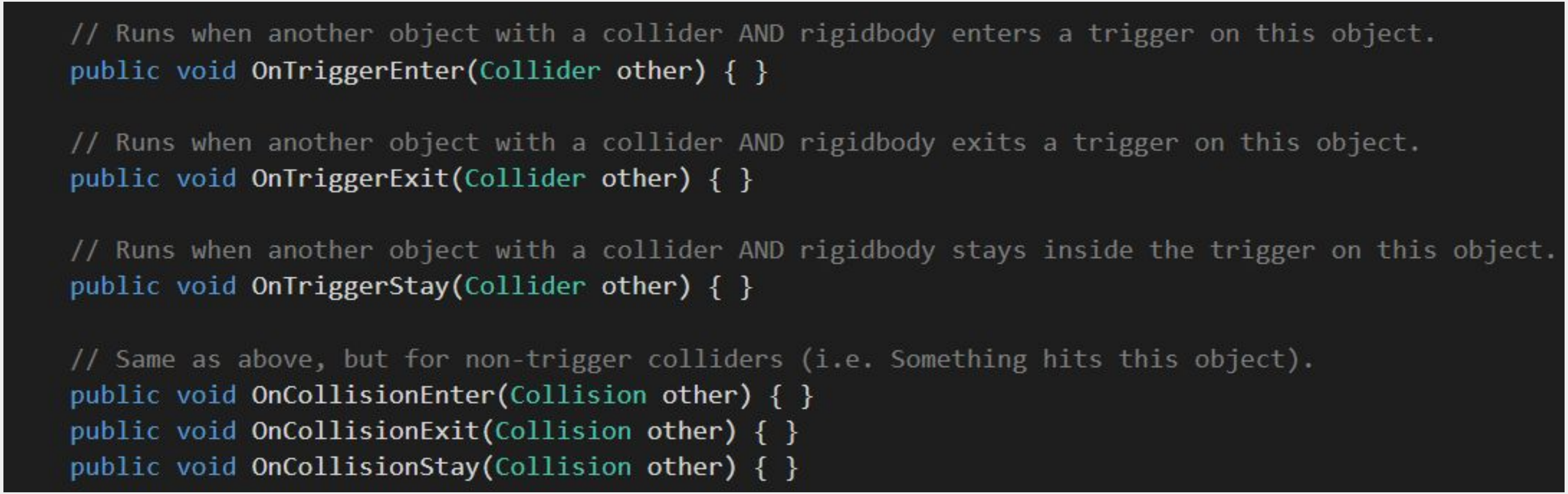


Colliders and Triggers

- Colliders do more than just cause collisions and physics interactions!
- Whenever two colliders “collide”, collision data is sent to components
- We gain access to both colliding objects and collision information
- This also works with colliders that are “triggers”
  - When a trigger comes into contact with another collider
  - No physical interactions, but data still sent to script
- We can use collision and trigger events to add more complex behaviour
- These events come in the form of more MonoBehaviour functions
  - OnCollisionEnter(Collision other) is called when a collision happens
  - OnTriggerEnter(Collider other) is called when a trigger is entered

Colliders and Triggers

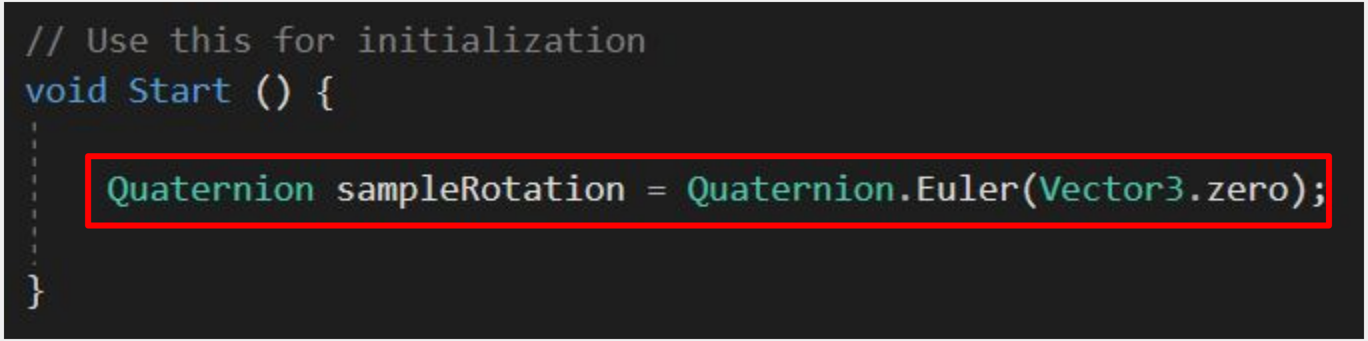
- There are a wide variety of collider/trigger functions that can be used
- Note the method signatures. They have to match exactly!



```
// Runs when another object with a collider AND rigidbody enters a trigger on this object.
public void OnTriggerEnter(Collider other) { }

// Runs when another object with a collider AND rigidbody exits a trigger on this object.
public void OnTriggerExit(Collider other) { }

// Runs when another object with a collider AND rigidbody stays inside the trigger on this object.
public void OnTriggerStay(Collider other) { }

// Same as above, but for non-trigger colliders (i.e. Something hits this object).
public void OnCollisionEnter(Collision other) { }
public void OnCollisionExit(Collision other) { }
public void OnCollisionStay(Collision other) { }
```

Quaternions

- Unity rotations are stored as something called **Quaternions**
- Whoa whoa whoa, wait, what’s that “Quaternion” thing?!?!
- Quaternions contain x, y, z, and **w** values.
- Quaternions are **complex** numbers. X, y, z are NOT the actual rotations!
- However, we can think of Quaternions in terms of **Euler Angles**
- **Euler Angles** are the rotations that we’re familiar with
  - Angles in the X, Y, and Z axis. I.e. Rotated 90 degrees in x axis is (90, 0, 0).

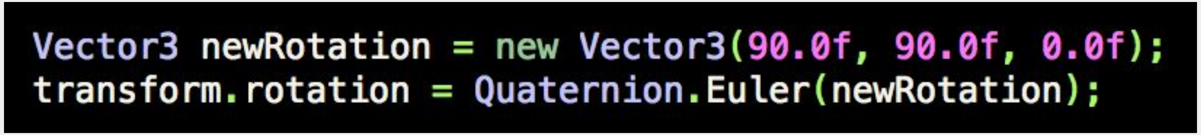


```
// Use this for initialization
void Start () {

    Quaternion sampleRotation = Quaternion.Euler(Vector3.zero);

}
```

Quaternions

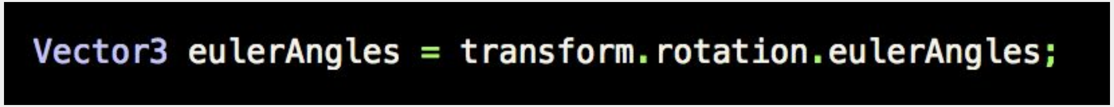
T

- Thankfully, Unity can convert between Euler and Quaternions
- Easy method to use: Quaternion.Euler(Vector3 angles)
    - Returns a Quaternion using the specified angles

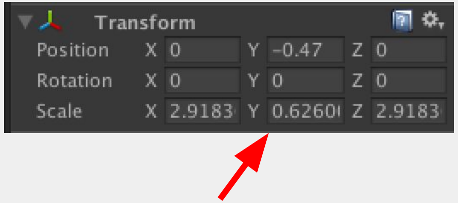


```
Vector3 newRotation = new Vector3(90.0f, 90.0f, 0.0f);
transform.rotation = Quaternion.Euler(newRotation);
```

- We can also get Quaternions as Euler Angles
    - Just use quaternionValue.eulerAngles



```
Vector3 eulerAngles = transform.rotation.eulerAngles;
```



Transform

Position X 0 Y -0.47 Z 0

Rotation X 0 Y 0 Z 0

Scale X 2.9183 Y 0.6260 Z 2.9183

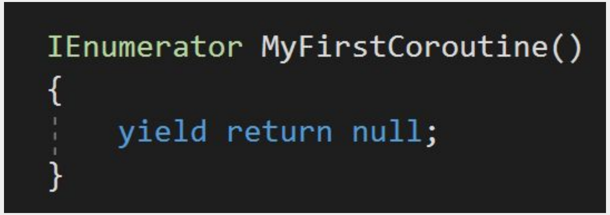
- Note: The rotation values in the inspector are thankfully **Euler Angles**

Coroutines

- Normally, a function runs to completion and returns...
- Coroutines are special Unity functions that can **pause** and **resume** later
- The pause step in the Coroutine always starts with a “yield”
- There are several different types of yields…
  - yield return null - Wait for the next frame
  - yield return new WaitForSeconds(float seconds) - Wait for a period of time
  - yield return new WaitForEndOfFrame() - Wait until everything else runs
  - These are just the most common!
- When the yield condition is met, the function will resume where it left off

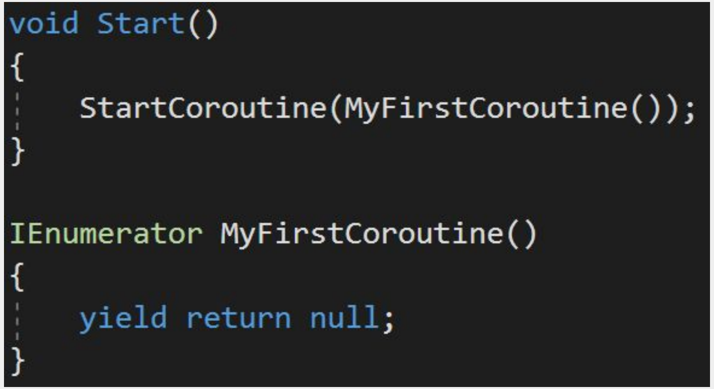
Coroutines

- Coroutines are methods with a return type of **IEnumerator**



```
IEnumerator MyFirstCoroutine()
{
    yield return null;
}
```

- To call a Coroutine, we use the StartCoroutine() method

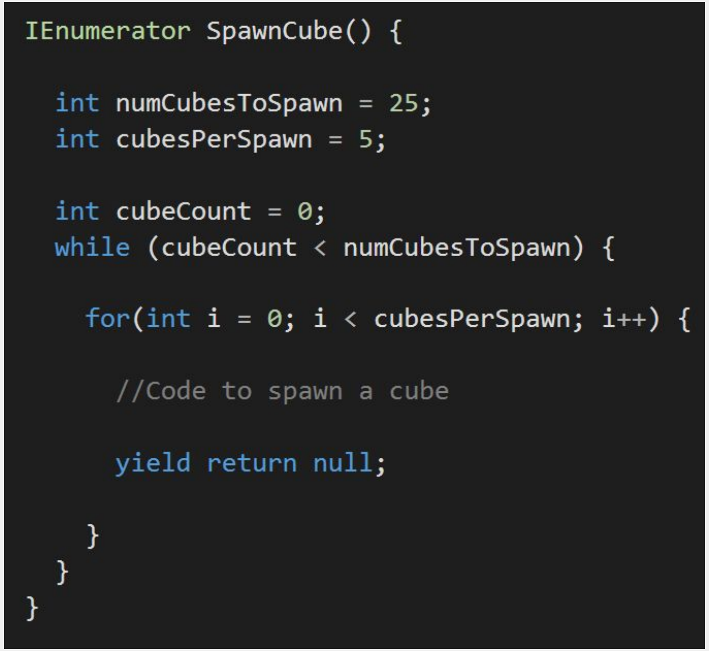


```
void Start()
{
    StartCoroutine(MyFirstCoroutine());
}

IEnumerator MyFirstCoroutine()
{
    yield return null;
}
```

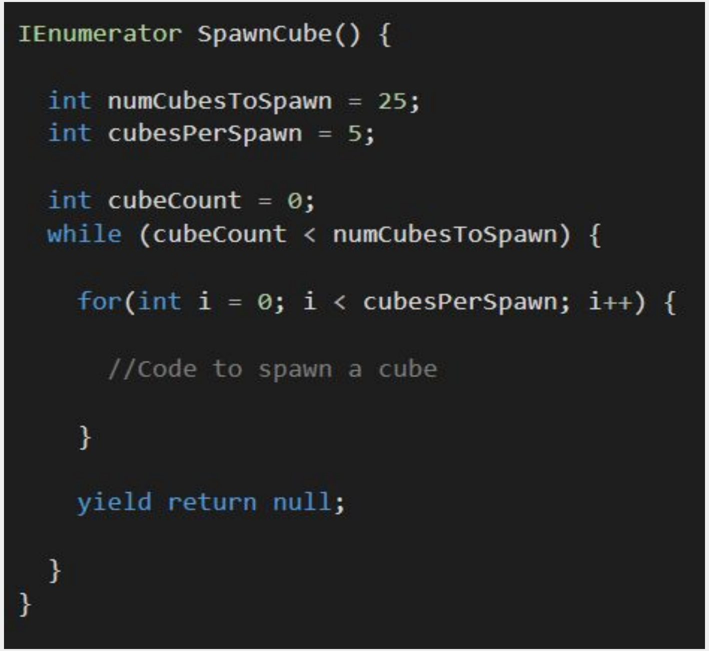
Coroutines

T

- What’s the difference between these two Coroutines?

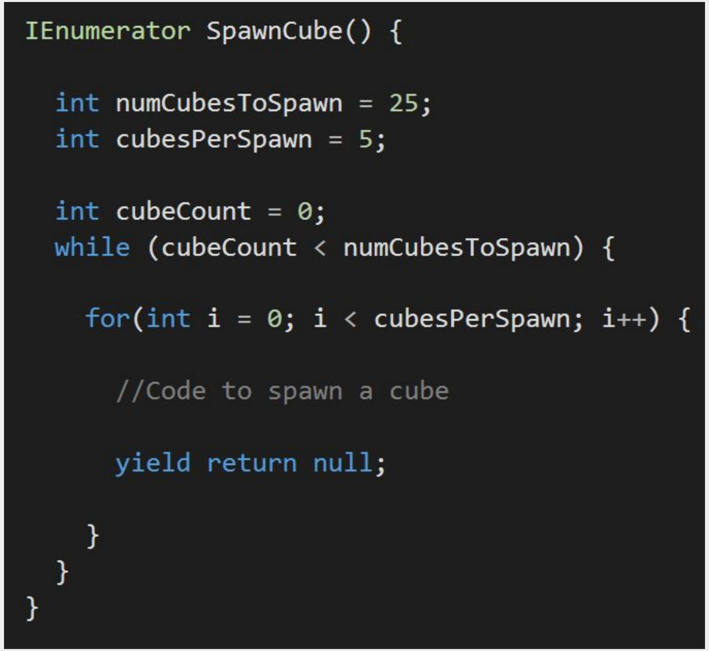


```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

      yield return null;

    }
  }
}
```



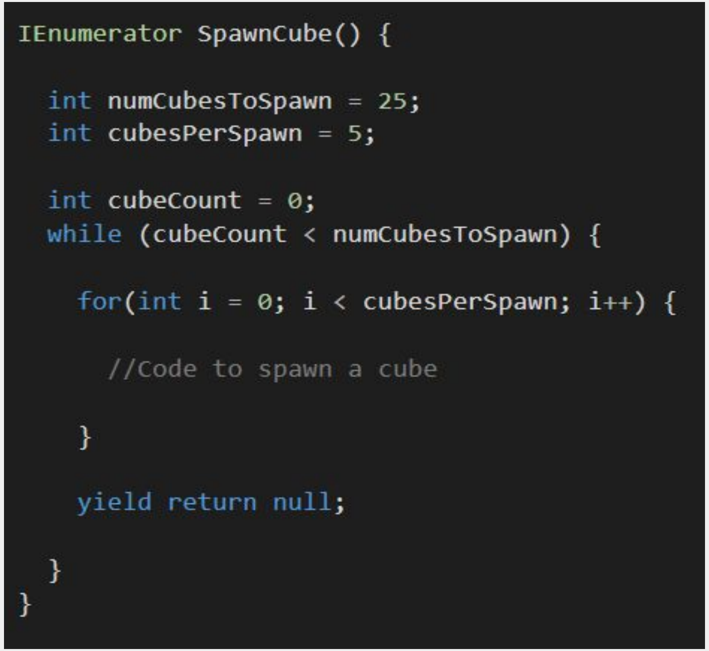
```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

    }

    yield return null;

  }
}
```

Coroutines



```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

      yield return null;

    }
  }
}
```

Spawns 1 cube every frame, 25 times



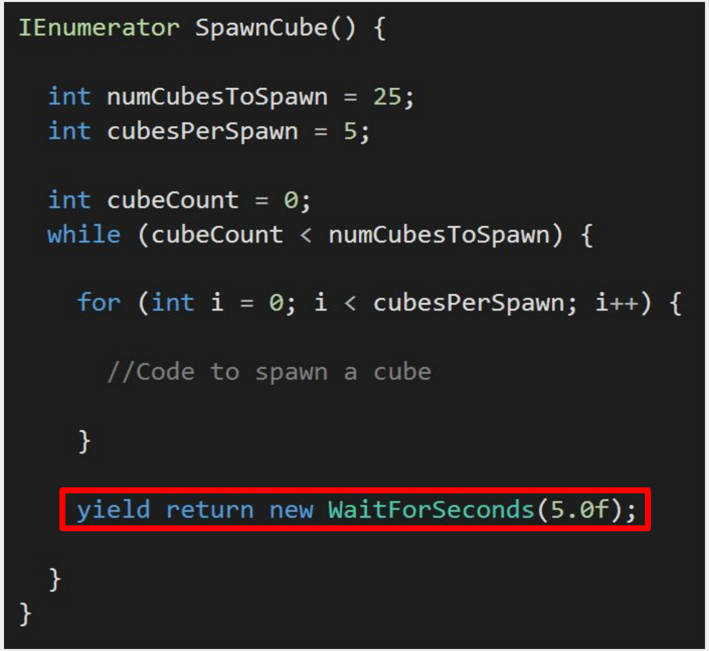
```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

    }

    yield return null;

  }
}
```

Spawns 5 cubes every frame, 5 times

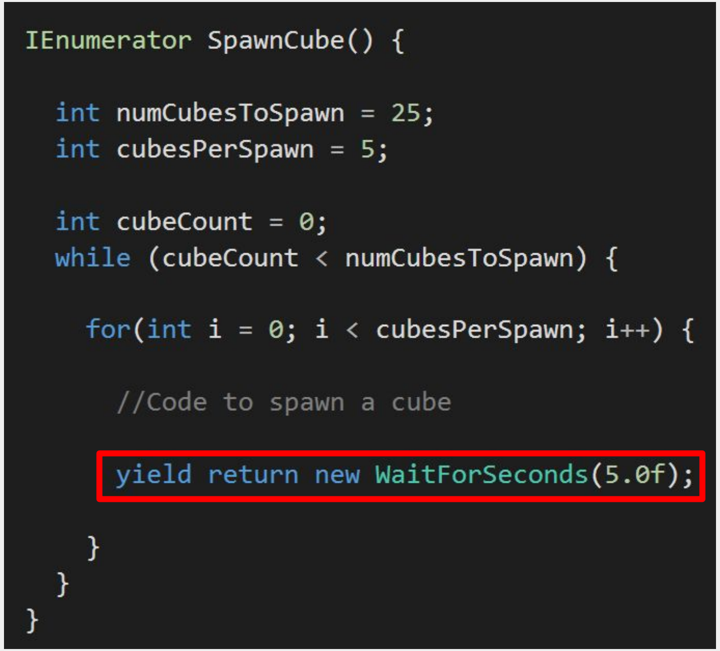
Coroutines

T

- We often want to make code **framerate independent** by using time

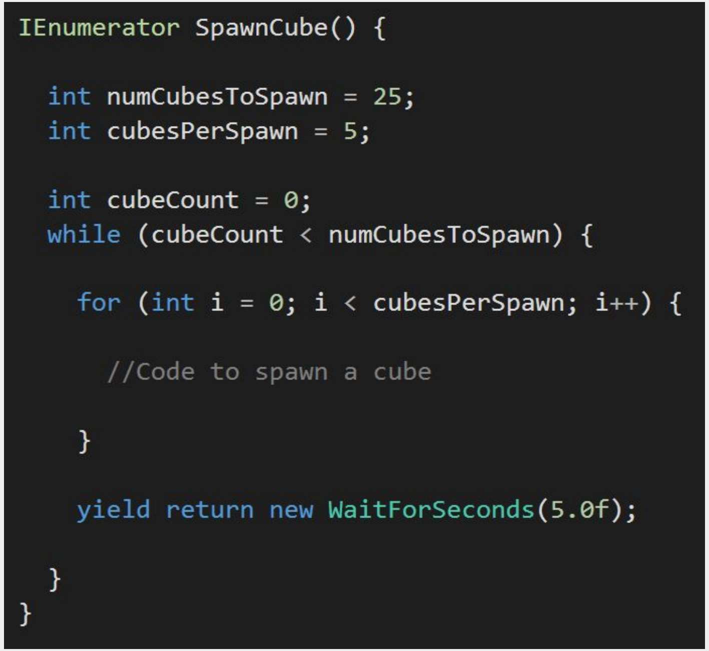


```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for (int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

    }

    yield return new WaitForSeconds(5.0f);

  }
}
```



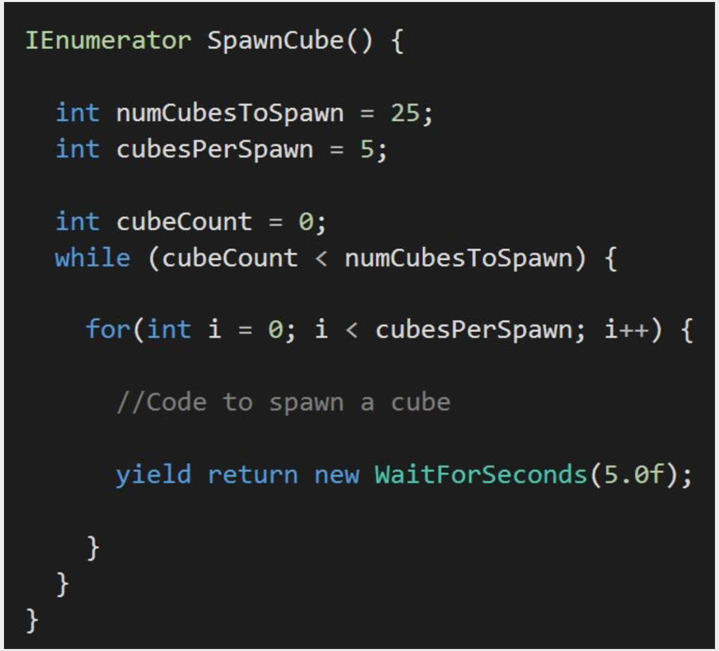
```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

      yield return new WaitForSeconds(5.0f);

    }
  }
}
```

Coroutines



```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for (int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

    }

    yield return new WaitForSeconds(5.0f);

  }
}
```

Spawns 5 cubes every 5 seconds, 5 times

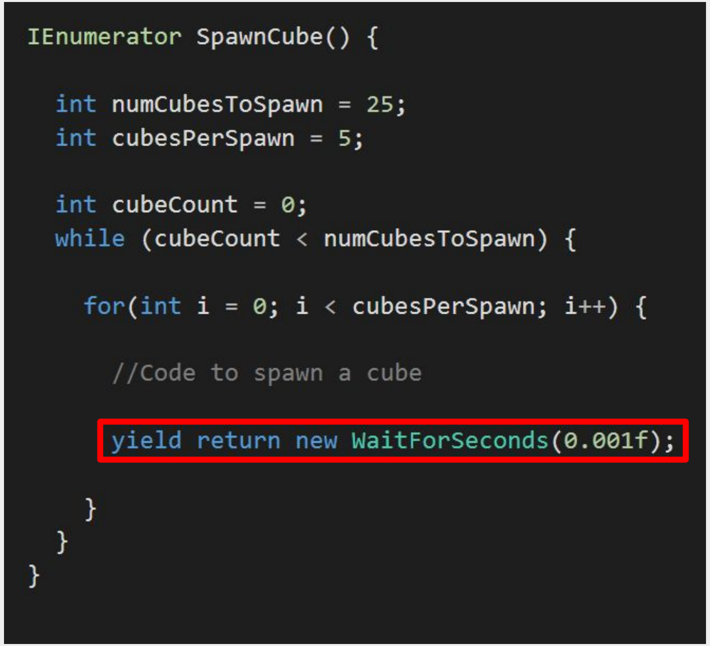


```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

      yield return new WaitForSeconds(5.0f);

    }
  }
}
```

Spawns 1 cube every 5 seconds, 25 times

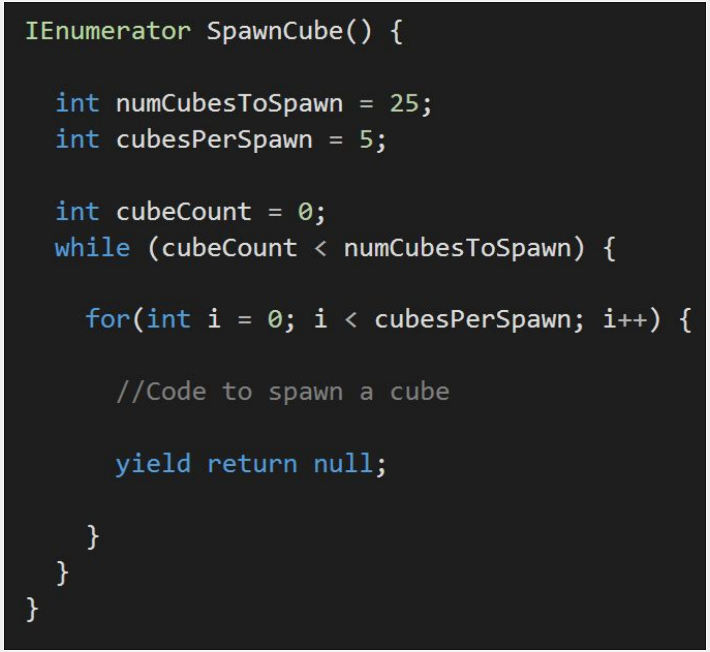
Coroutines

- Be careful! If we’re getting 90FPS, that’s ~ 1 frame per 0.01 seconds…
- Code on left will still run every frame, since wait time is faster than FPS



```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

      yield return new WaitForSeconds(0.001f);

    }
  }
}
```

==



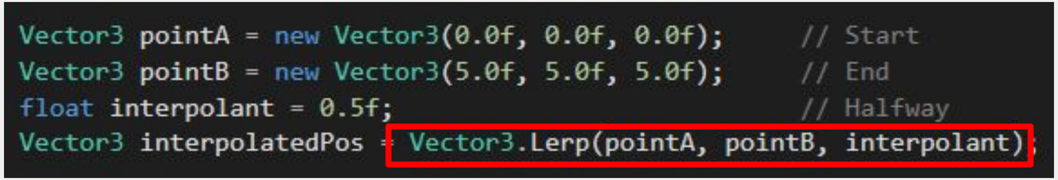
```
IEnumerator SpawnCube() {

  int numCubesToSpawn = 25;
  int cubesPerSpawn = 5;

  int cubeCount = 0;
  while (cubeCount < numCubesToSpawn) {

    for(int i = 0; i < cubesPerSpawn; i++) {

      //Code to spawn a cube

      yield return null;

    }
  }
}
```

Linear Interpolation

- Linear Interpolation, or “Lerp”, helps us change values over time
- Lerp calculates intermediate values between two points
  - I.e. A fraction between A and B, starting from A and going to B
- The **Interpolant** is the interval between the two points that we want
  - If Point A = 0.0, Point B = 10.0, and I want the value ¼ between the two…
  - The interpolant should be 0.25, yielding 2.5
- This is an important process in game developing and in Unity
  - Using Lerp, we can move objects and values gradually between two points

Linear Interpolation

- In Unity, we can lerp across different types of values
  - Most common is a Vector3… the same type as position, rotation, and scale
  - We can also lerp between colors!
- The Lerp function in Unity is as follows:
  - For Vectors: Vector3.Lerp(Vector3 A, Vector3 B, float interpolant)
  - For Colors: Color.Lerp(Color A, Color B, float interpolant)

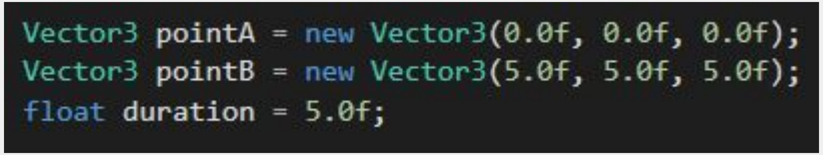


```
Vector3 pointA = new Vector3(0.0f, 0.0f, 0.0f);     // Start
Vector3 pointB = new Vector3(5.0f, 5.0f, 5.0f);     // End
float interpolant = 0.5f;                           // Halfway
Vector3 interpolatedPos = Vector3.Lerp(pointA, pointB, interpolant);
```

- Using Lerp, we can effectively change values across frames!
- … but how do we calculate the interpolant at each frame?
- Coroutines are a huge help!

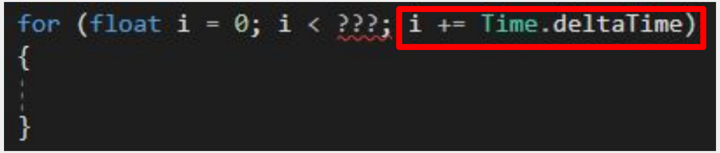
Linear Interpolation

- What we need for a simple lerp:
  - A duration value
  - A start point and an end point



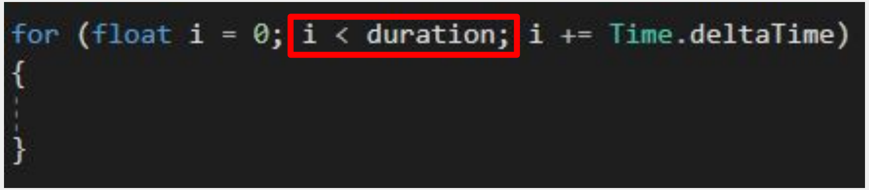
```
Vector3 pointA = new Vector3(0.0f, 0.0f, 0.0f);
Vector3 pointB = new Vector3(5.0f, 5.0f, 5.0f);
float duration = 5.0f;
```

- We want the lerp to happen over time, rather than frames…



```
for (float i = 0; i < ???; i += Time.deltaTime)
{

}
```

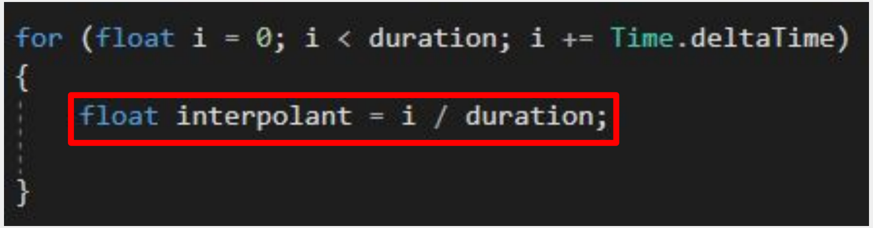
- This will increment “i” by the amount of scaled time since last frame
- We want the loop to terminate when we’ve reached our desired duration



```
for (float i = 0; i < duration; i += Time.deltaTime)
{

}
```

Linear Interpolation

- Now how do we calculate the interpolant using these values?
  - We have a time counter stored in i …
  - And we have our total desired time stored in duration!
- The interpolant should be i/duration.
  - If our change should last 20 seconds, and it’s been 7 seconds…
  - … then interpolant is just 7 / 20!

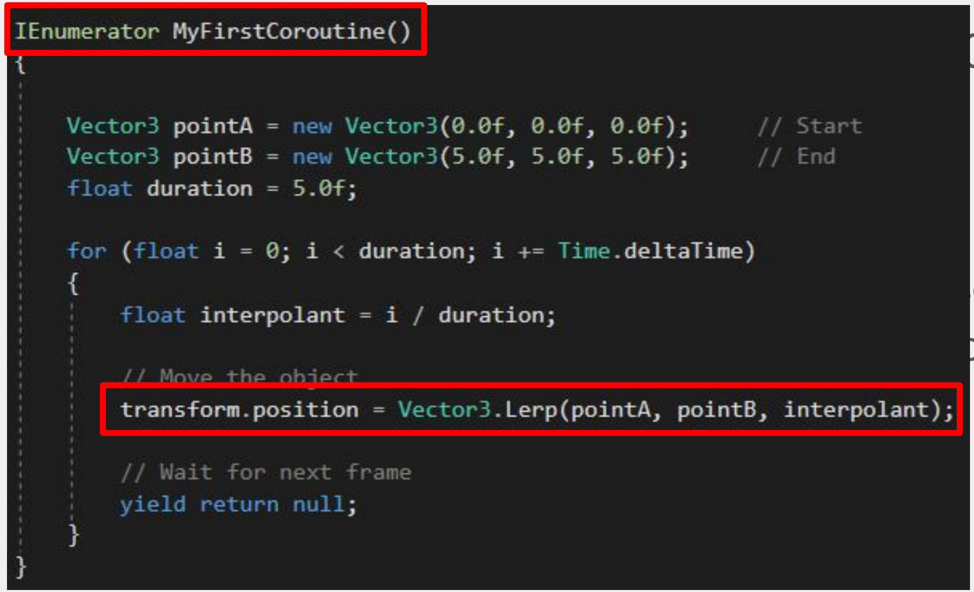


```
for (float i = 0; i < duration; i += Time.deltaTime)
{
    float interpolant = i / duration;

}
```

- So what values are Point A and Point B?
  - Point A is usually just the starting value of whatever we’re changing
  - Point B is just the desired/destination value

Linear Interpolation

T



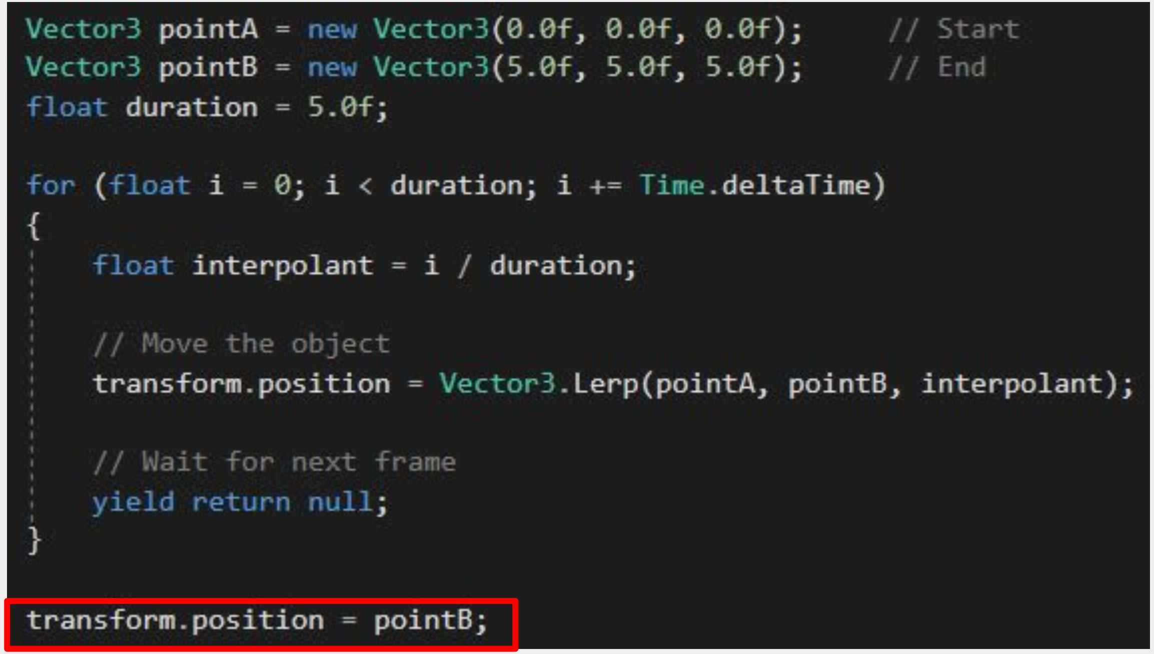
```
IEnumerator MyFirstCoroutine()
{
    Vector3 pointA = new Vector3(0.0f, 0.0f, 0.0f);     // Start
    Vector3 pointB = new Vector3(5.0f, 5.0f, 5.0f);     // End
    float duration = 5.0f;

    for (float i = 0; i < duration; i += Time.deltaTime)
    {
        float interpolant = i / duration;

        // Move the object
        transform.position = Vector3.Lerp(pointA, pointB, interpolant);

        // Wait for next frame
        yield return null;
    }
}
```

0, so position goes from A

o it won't ever be 1.0!

on at the end

Linear Interpolation



```
Vector3 pointA = new Vector3(0.0f, 0.0f, 0.0f);     // Start
Vector3 pointB = new Vector3(5.0f, 5.0f, 5.0f);     // End
float duration = 5.0f;

for (float i = 0; i < duration; i += Time.deltaTime)
{
    float interpolant = i / duration;

    // Move the object
    transform.position = Vector3.Lerp(pointA, pointB, interpolant);

    // Wait for next frame
    yield return null;
}

transform.position = pointB;
```

- Easy fix: Just snap position to pointB at the end
- Always makes sure the object reaches it’s final value

Edited by Connor and Robin

General Unity Tutorials

Total Tutorial Time, Excluding Projects & Extras: ~8 hours
Total Tutorial Time, Including Projects (No Extras): ~13 hours

Interface Essentials (30 minutes)

https://unity3d.com/learn/tutorials/topics/interface-essentials

- All of "Using The Unity Interface", except for #8 (22 minutes)
- All of "Essential Unity Concepts" (8 minutes)

Beginner Scripting (90 minutes)

https://unity3d.com/learn/tutorials/s/scripting

- "Beginner Gameplay Scripting", 1-3, 5-27 (90 minutes)

Lighting and Rendering (60 minutes)

https://unity3d.com/learn/tutorials/topics/graphics

- "Rendering and Shading", 1 - 6 (47 minutes)
- "Cameras and Effects", Just #1 (8 minutes)
- All of "Geometry in Unity" (5 minutes)

Edited by Robin Xu

Edited by Connor and Robin

General Unity Tutorials

Total Tutorial Time, Excluding Projects & Extras: ~8 hours
Total Tutorial Time, Including Projects (No Extras): ~13 hours

UI (35 minutes)

https://unity3d.com/learn/tutorials/topics/user-interface-ui

- “UI Components”, 1-2, 4-6, 11 (35 minutes)

Audio (50 minutes)

https://unity3d.com/learn/tutorials/topics/audio

- All of “Audio Setup” tutorials (4 minutes)
- “Live Trainings on Audio”, just #1 (46 minutes)

Physics (45 minutes)

https://unity3d.com/learn/tutorials/topics/physics

- All of “3D Physics” (30 minutes)
- #3 of “Assignments”: the “Brick Shooter” game (15 minutes)

Project: Roll A Ball (75 minutes)

- https://unity3d.com/learn/tutorials/projects/roll-ball-tutorial (75 minutes)

Edited by Robin Xu

Thanks!

CSE165: 3D User Interaction
Robin Xu

Visit Triton XR Community for more at https://tritonxr.ucsd.edu/