Unity Scripting: Beginner

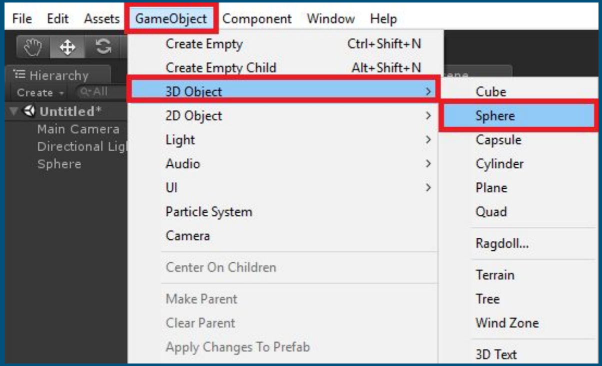
Presented by Virtual Reality Club at UCSD

Unity Scripting: C#

- “Scripting” in Unity is the programming side of game development.
- Unity primarily uses the **C#** language (C Sharp).
  - JavaScript is also available, but is less common.
- 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 are runtime gives us dynamic control over the game.
  - I.e. How can we change things at ***runtime?***

Unity Scripting: What is a Script?

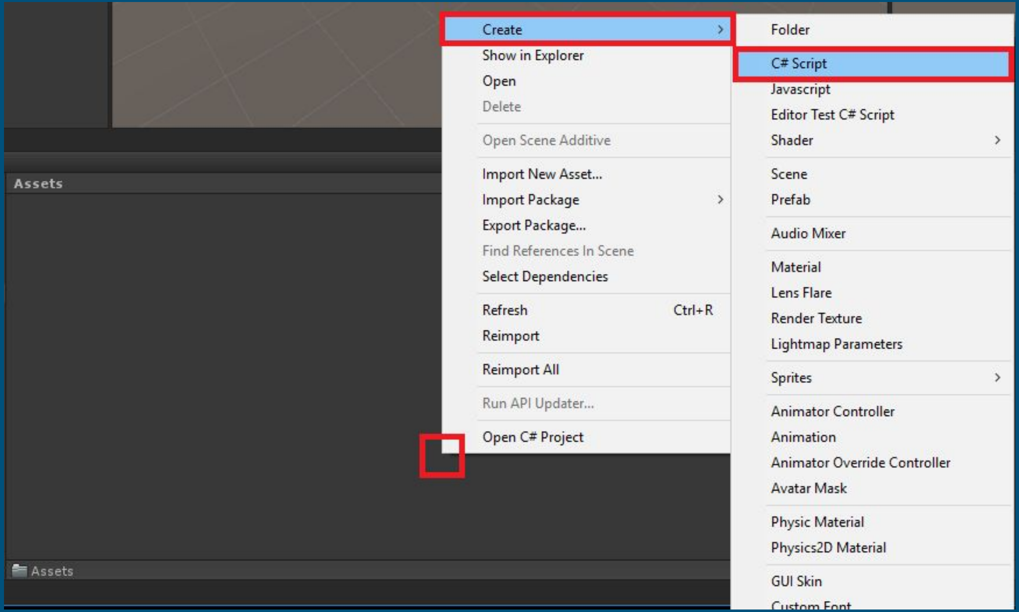
- … but what is a script in Unity?
- Scripts are really just **custom components**!
- When you create a Script, you're creating your very own component.
  - You can give that component behaviour, properties, fields, and values.
- You add scripts to GameObjects just like any other component!
- First, let's make a GameObject to add the script to.



Unity Scripting: Our First Script

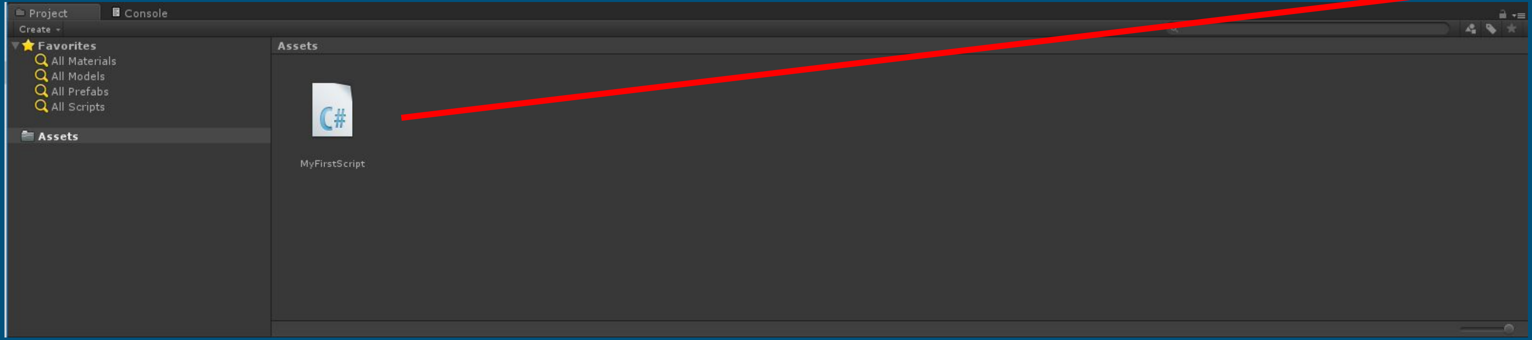
- Now let's create a new C# script in Unity

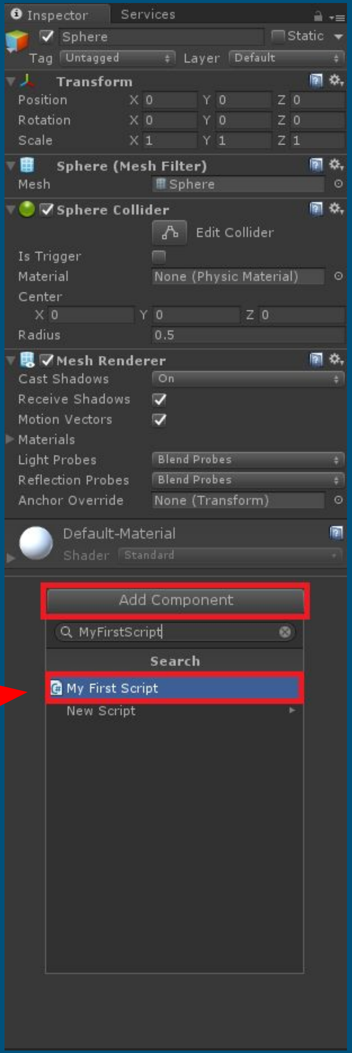
1. Right Click in "Assets" folder
    a. You can also use "Assets" menu
2. Hover over "Create"
3. Click "C# Script"
4. Give it a name!



Unity Scripting: Adding a Script

- Select the object you want to add the script to.
    - In this case, it's our sphere.
- Click "Add Component"
- Add your very own script!
- You can also just drag the script onto the object.





☑ Sphere □ Static
Tag Untagged Layer Default

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

Mesh: Sphere

Is Trigger □
Material: None (Physic Material)
Center: X 0 Y 0 Z 0
Radius: 0.5

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

Default-Material
Shader: Standard

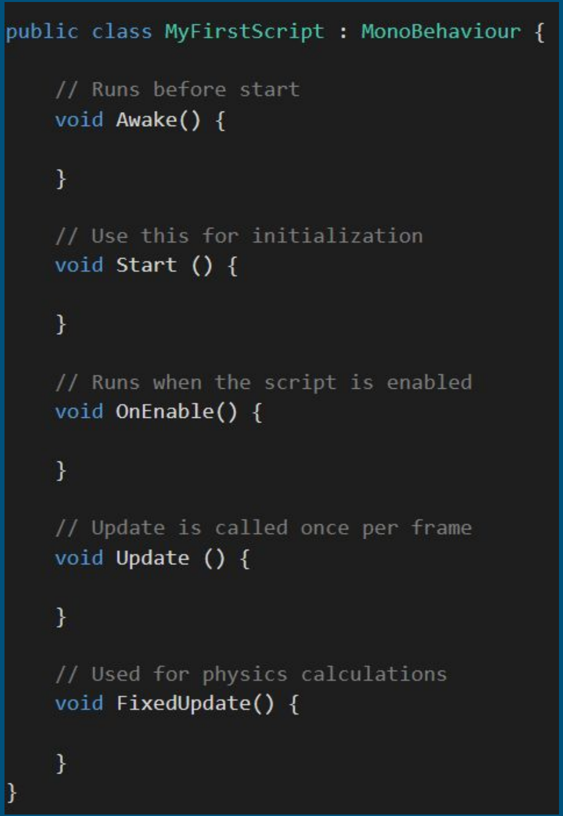
MyFirstScript

Unity Scripting: Opening The Script

- 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 IDE's (Integrated Development Environments) for coding.
- You'll first notice a few things…
    - "MonoBehaviour"
    - "Start()"
    - "Update()"
- A MonoBehaviour is a Unity-specific class that every script **derives.**
- MonoBehaviour scripts are especially useful for game development.

Unity Scripting: MonoBehaviour

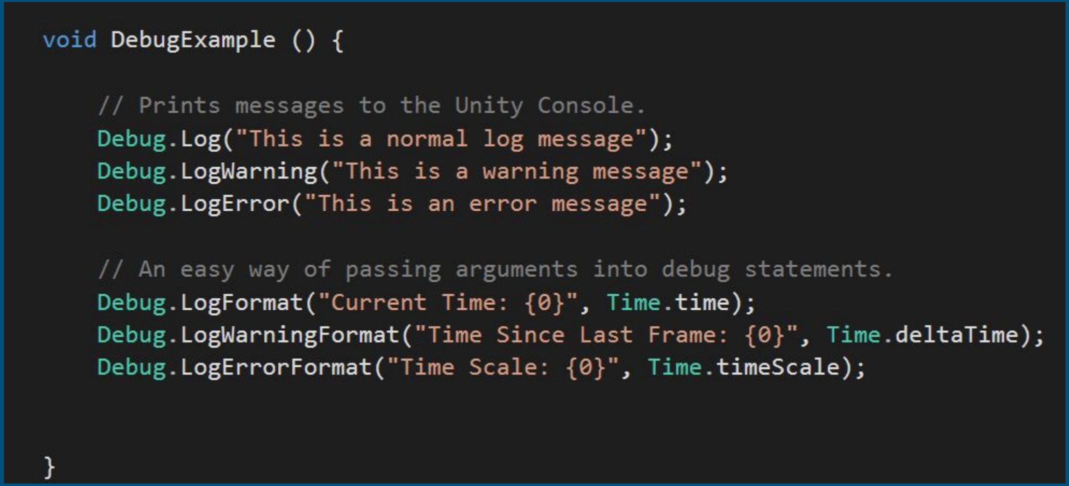
- Start() and Update() are just **methods**.
- **Start():** Runs once when the game begins.
    - Use to initialize script
- **Update():** Runs ***every frame***.
    - A game is divided into “frames”.
        - Think of old-school flipbooks, each page is a “frame”!
    - This method will be called at least 90 times every second.
- Some others:
    - Awake(): Runs before start.
    - OnEnable(): Runs when the script is enabled.
    - FixedUpdate(): Framerate-independent update, for physics.



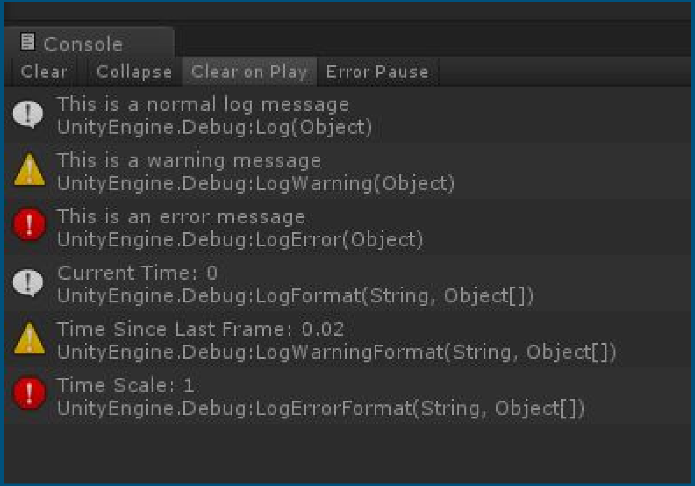
```
public class MyFirstScript : MonoBehaviour {

    // 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() {

    }
}
```

Unity Scripting: Debugging

- To print debug messages in Unity, use Debug.Log(string message)
- Debug messages will appear in the Unity Console



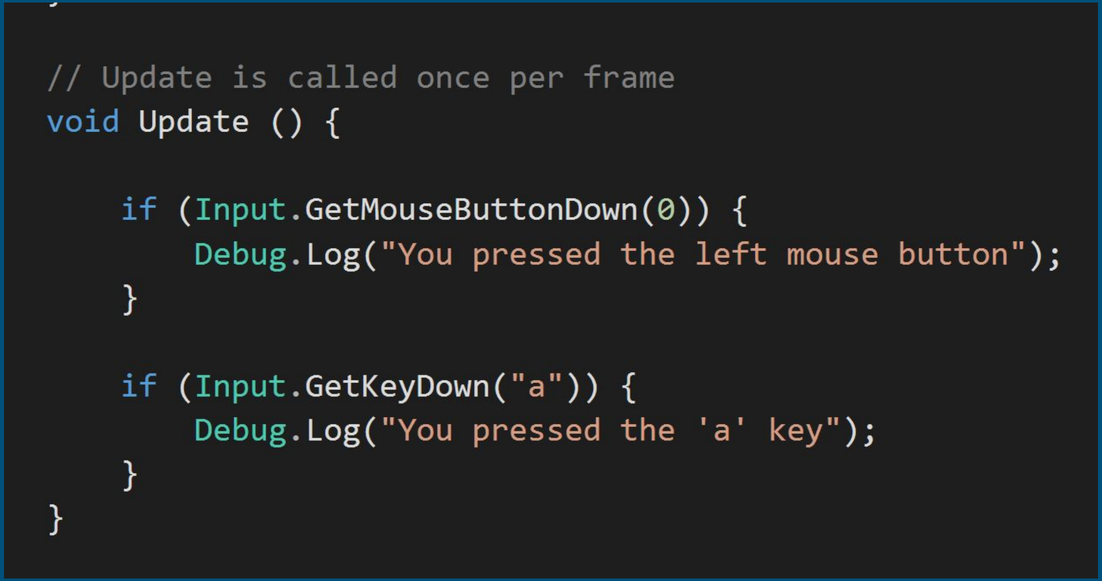
```
void DebugExample () {

    // 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[])
```

Unity Scripting: Keyboard and Mouse Input.

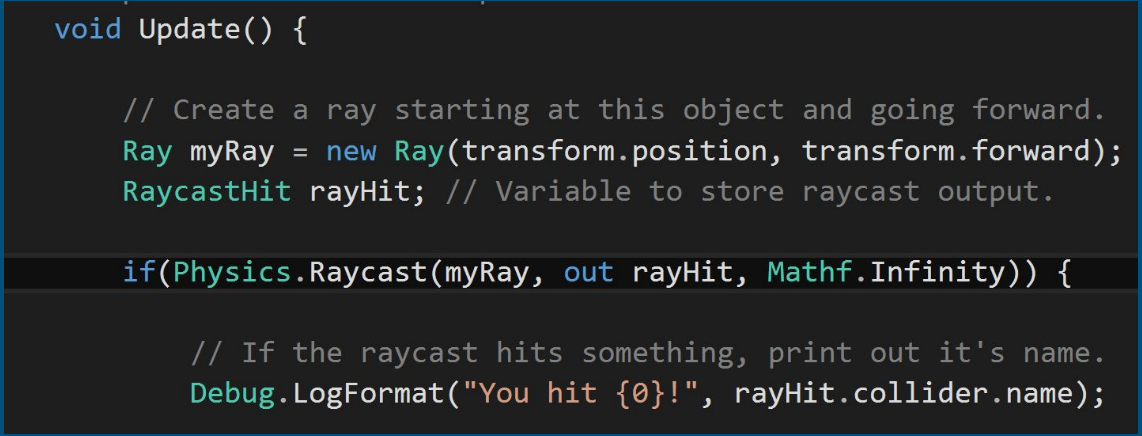
- Useful for testing when you don’t have access to the headset.



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

    if (Input.GetMouseButtonDown(0)) {
        Debug.Log("You pressed the left mouse button");
    }

    if (Input.GetKeyDown("a")) {
        Debug.Log("You pressed the 'a' key");
    }
}
```

Key Concepts: Raycast

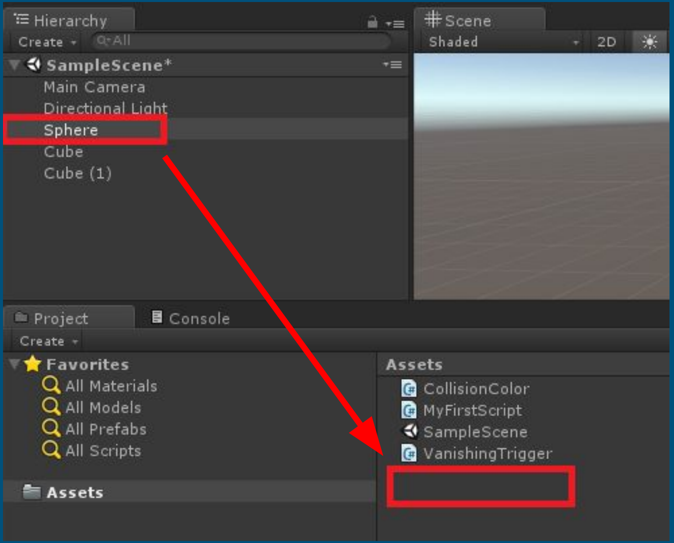
- Physics.Raycast(Vector3 origin, RaycastHit hitInfo, float maxDistance):
  - A “Raycast” is simply a line (or a “ray”) that is projected forward until it hits something
  - Once a raycast hits something, it returns information about what it hits
  - If I raycast forward from my Camera and look at a Cube, I will get a reference to that Cube
  - This is the fundamental aspect of a gaze system, which will be described later.



```
void Update() {

    // Create a ray starting at this object and going forward.
    Ray myRay = new Ray(transform.position, transform.forward);
    RaycastHit rayHit; // Variable to store raycast output.

    if(Physics.Raycast(myRay, out rayHit, Mathf.Infinity)) {

        // If the raycast hits something, print out it's name.
        Debug.LogFormat("You hit {0}!", rayHit.collider.name);
```

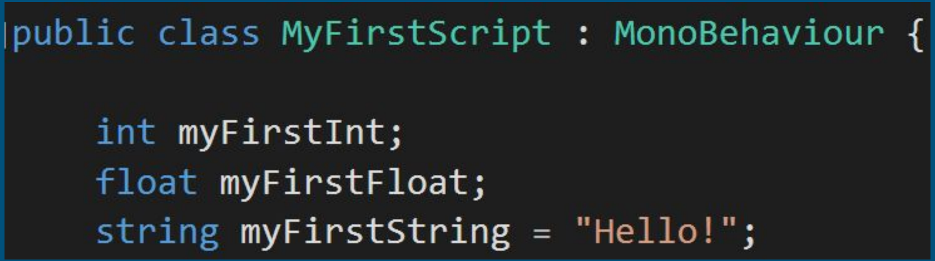
Key Concepts: Instantiation

- Before we move forward, let’s talk about Instantiation.
- Remember what a **Prefab** is?
  - A prefabricated GameObject stored outside of the scene
- **Instantiation** clones prefabs at runtime
  - You can specify **what** is cloned.
  - You can specify **where** they go.
  - You can specify how they’re **rotated.**
- This is necessary for the first 165 assignment!
  - The wall must consist of **instantiated** brick prefabs.
- Create a prefab by dragging from the hierarchy.
  - This effectively clones the object from the hierarchy.
  - All settings, scripts, components, etc… are saved.

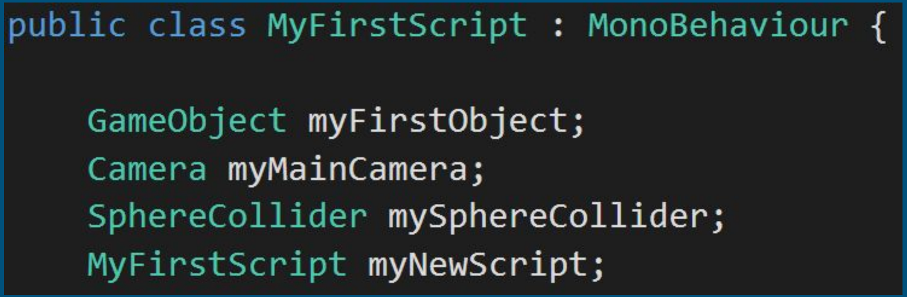


Unity Scripting: Variables

- In C#, we have access to the usual primitive data types.
  - int: Whole integer values
  - float: Precise decimal values (Most common for 3D space)
  - string: Words and characters
  - Etc…
- With MonoBehaviour, we can also use all **components** as types!
  - Including scripts we've written! (Because they're just components, right?)

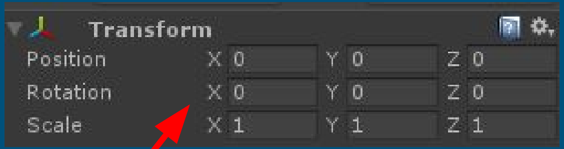


```
public class MyFirstScript : MonoBehaviour {

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



```
public class MyFirstScript : MonoBehaviour {

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

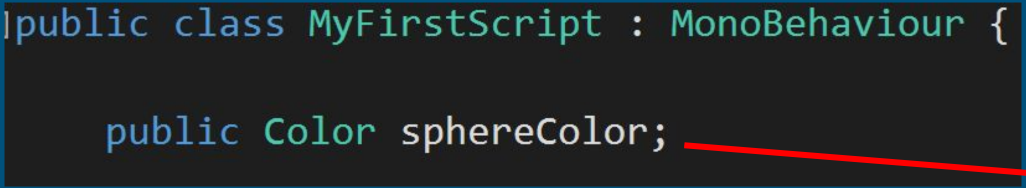
Unity Scripting: Public & Serialized Variables



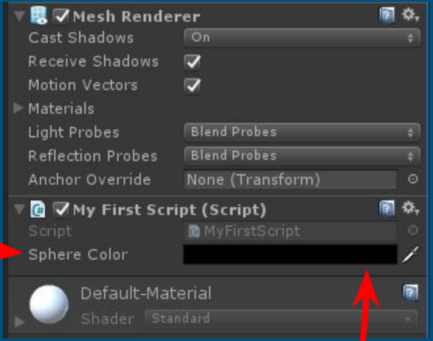
Transform

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

- So… if scripts are just components…
- … then how do we get all those fancy component fields?
- First, just try making a **public** variable!



```
public class MyFirstScript : MonoBehaviour {

    public Color sphereColor;
```



☑ Mesh Renderer

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

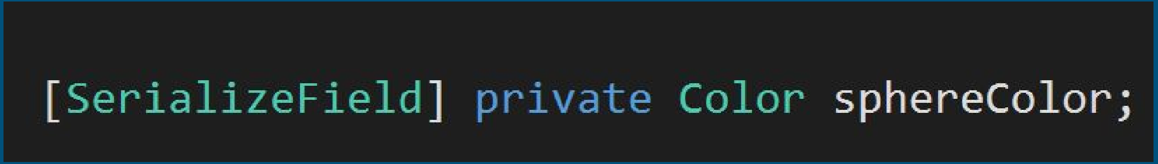
☑ My First Script (Script)

Script	MyFirstScript
Sphere Color	

Default-Material

Shader Standard

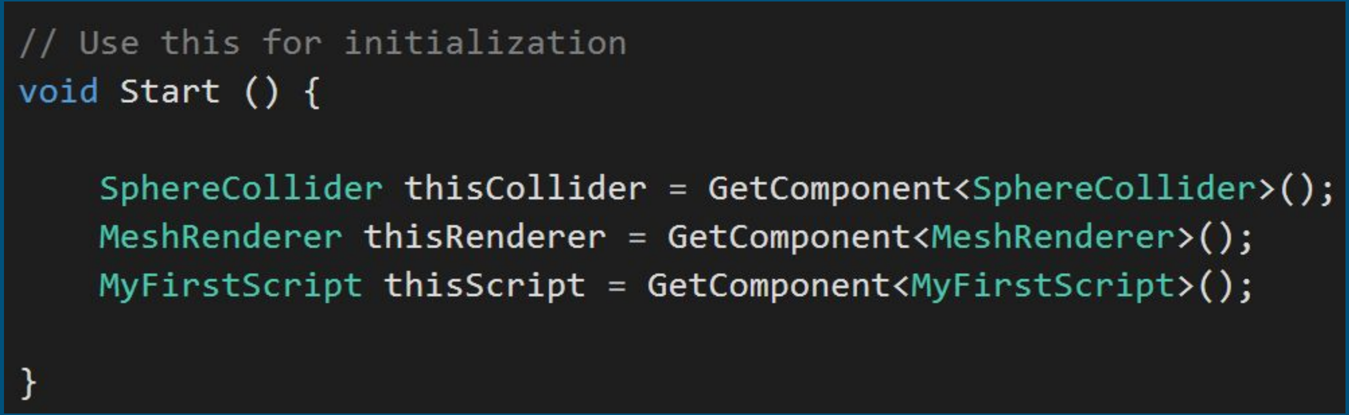
- Now look at it in the inspector!
- This can also be done by adding [SerializeField] before the variable.



```
[SerializeField] private Color sphereColor;
```

Unity Scripting: Getting Components

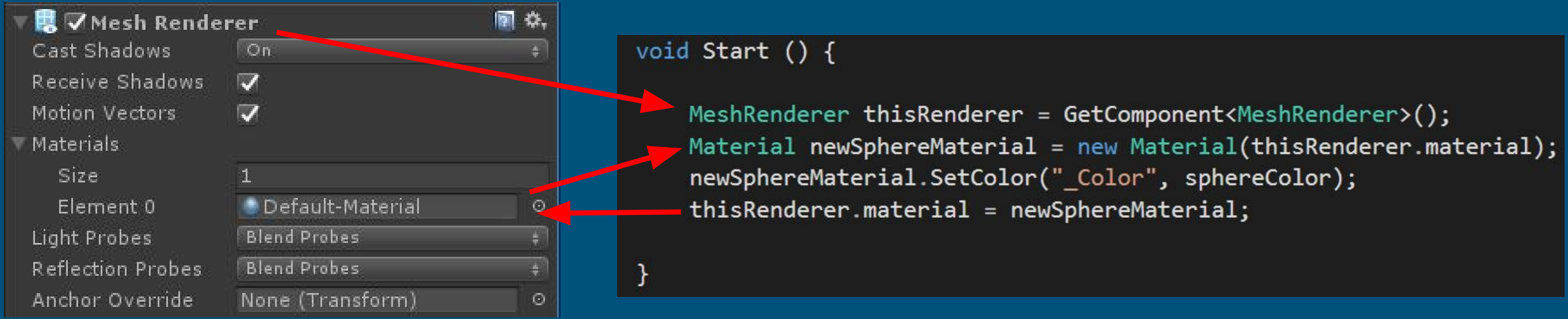
- Most scripting is essentially just modifying object component values.
  - Since all GameObjects consist of Components…
  - … we affect GameObjects by editing their Components at runtime!
- GetComponent<ComponentName>() is a vital method in Unity scripting.
  - This will get the component of specified type on an object.



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

    SphereCollider thisCollider = GetComponent<SphereCollider>();
    MeshRenderer thisRenderer = GetComponent<MeshRenderer>();
    MyFirstScript thisScript = GetComponent<MyFirstScript>();

}
```

Unity Scripting: Modifying Components

- Let’s try something a bit more fun.
- How do we change an object’s color? We modify it’s **Material**!
- The **Material** is a variable of the **MeshRenderer** component!
- So… how can we change the object’s color in a script?



☑ Mesh Renderer

Cast Shadows: On

Receive Shadows: ☑

Motion Vectors: ☑

Materials

Size: 1

Element 0: Default-Material

Light Probes: Blend Probes

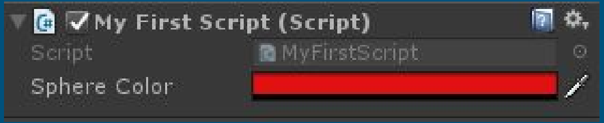
Reflection Probes: Blend Probes

Anchor Override: None (Transform)

```
void Start () {

    MeshRenderer thisRenderer = GetComponent<MeshRenderer>();
    Material newSphereMaterial = new Material(thisRenderer.material);
    newSphereMaterial.SetColor("_Color", sphereColor);
    thisRenderer.material = newSphereMaterial;

}
```

Unity Scripting: Let’s Test It!

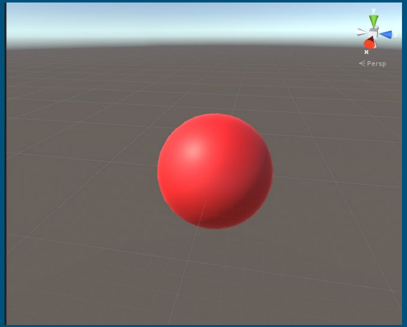
- First, let’s set the value of “sphereColor” in the inspector!



☑ My First Script (Script)

Script MyFirstScript

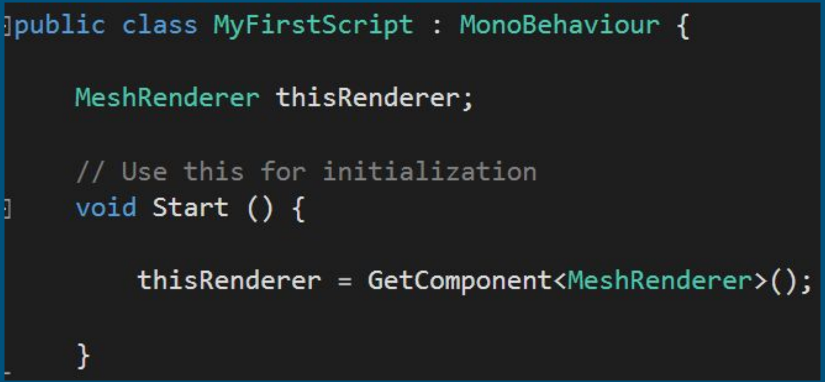
Sphere Color 🟥



- Now, when we Play the game, our sphere should change color!
- We can change the starting “Sphere Color” to have more control over this.

Unity Scripting: A Little More Fun...

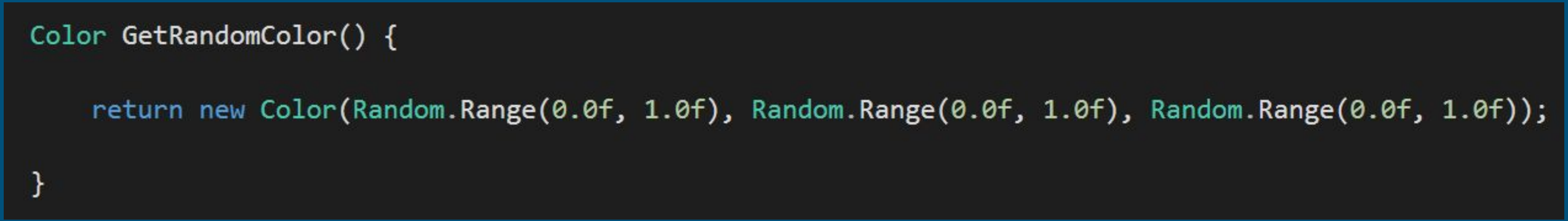
- That was great and all, but we could’ve just made it red in the first place.
- It’s time to use the Update() method to show how awesome scripting is!
- Instead of changing the color once at the beginning in Start()...
- … Let’s change it, randomly, ***every single frame!***
- First, let’s just store the renderer component on Start:



```
public class MyFirstScript : MonoBehaviour {

    MeshRenderer thisRenderer;

    // Use this for initialization
    void Start () {

        thisRenderer = GetComponent<MeshRenderer>();

    }
```

Unity Scripting: A Little More Fun...

- Let’s also set up a method that gives us a random color!



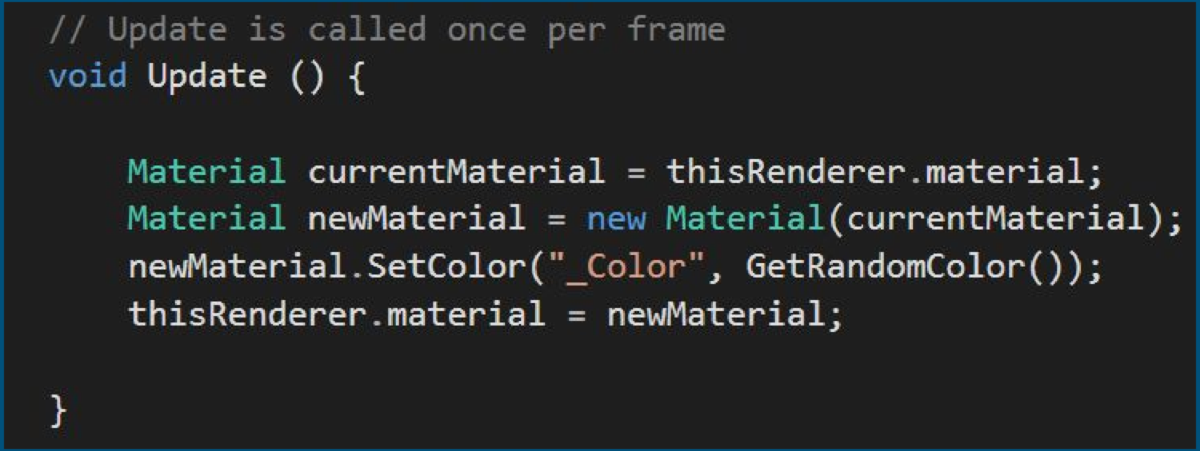
```
Color GetRandomColor() {

    return new Color(Random.Range(0.0f, 1.0f), Random.Range(0.0f, 1.0f), Random.Range(0.0f, 1.0f));

}
```

Note: “Colors” in Unity consist of R (Red), G (Green) and B (Blue) values that are all between 0.0 and 1.0.

- Now...we just need to change the color every frame!

Unity Scripting: A Little More Fun...

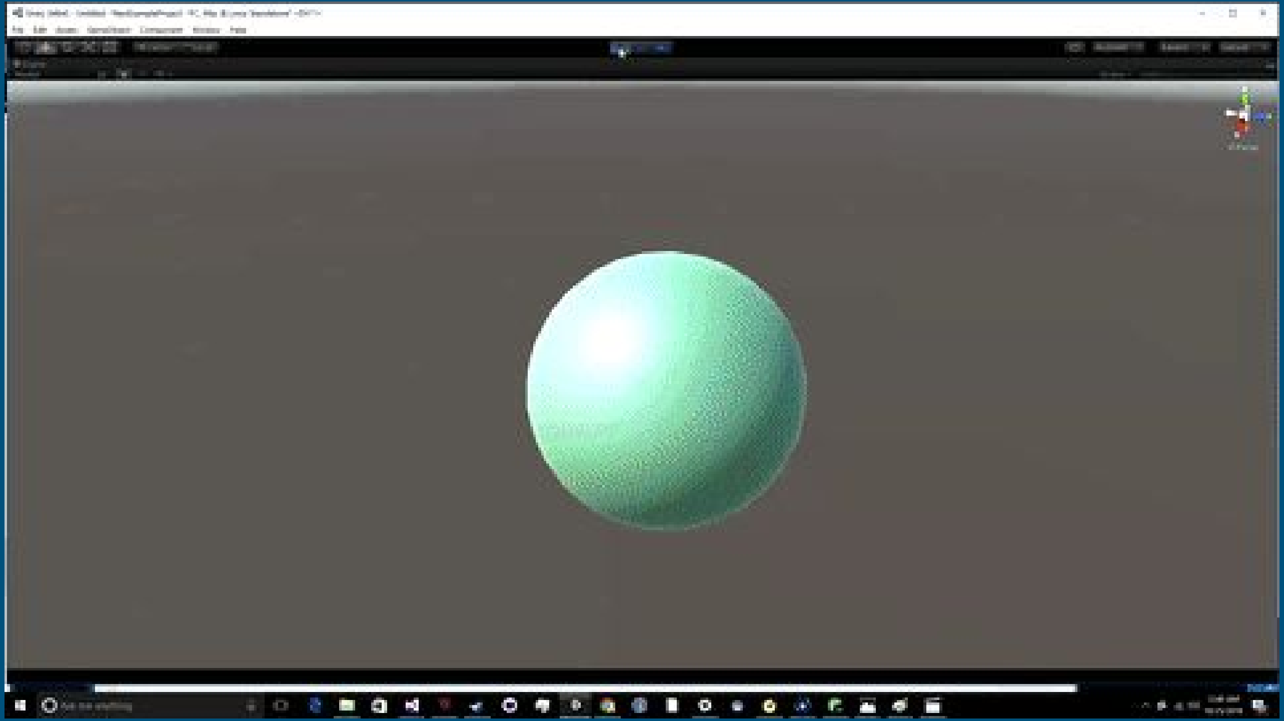
- Step 1: Store the current material on the sphere.
- Step 2: Create a new material from the existing material.
- Step 3: Set the new material’s color randomly
- Step 4: Store the new material back into the Mesh Renderer



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

    Material currentMaterial = thisRenderer.material;
    Material newMaterial = new Material(currentMaterial);
    newMaterial.SetColor("_Color", GetRandomColor());
    thisRenderer.material = newMaterial;

}
```

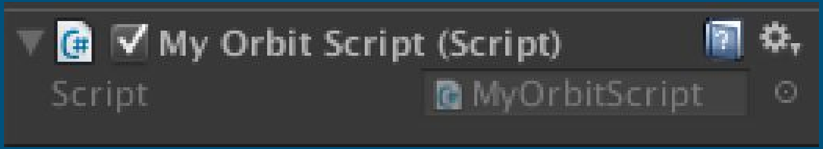
Unity Scripting: A Little More Fun...

- Play!



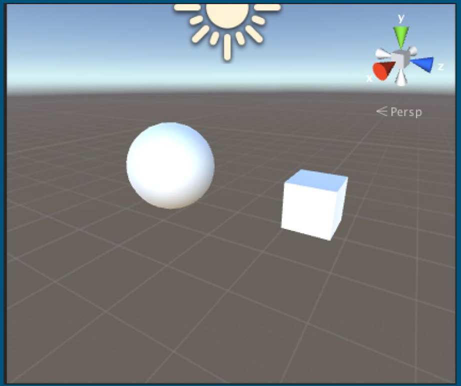
Unity Scripting: Multiple Objects

- Cool, we were able to modify the components on an object!
- But how can we change ***other*** objects’ components in a script?
- Let’s try giving our colorful sphere a moon!
- We’re going to write a script that ***rotates objects around other objects.***
- First, go ahead and just make another GameObject in your scene.
    - Cube, Sphere, Cylinder, etc…
- Once that’s done, make a new script for orbiting
    - I.e. “MyOrbitScript”
- Finally, add your new script to your new object.



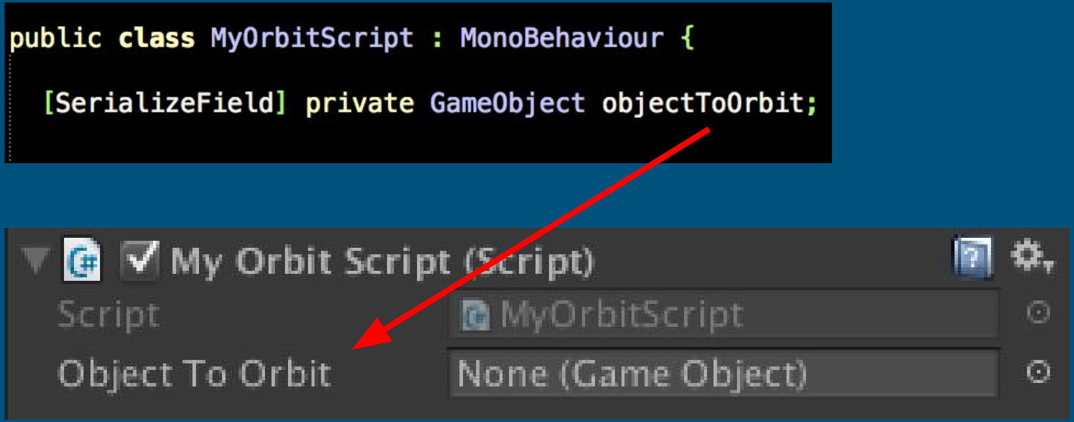
☑ My Orbit Script (Script)

Script MyOrbitScript



Unity Scripting: Multiple Objects

- In our new script, we first need an object to **rotate around.**
- Let’s just make this a field in the inspector!
  - I.e. A public variable or serialized field!



```
public class MyOrbitScript : MonoBehaviour {

  [SerializeField] private GameObject objectToOrbit;
```

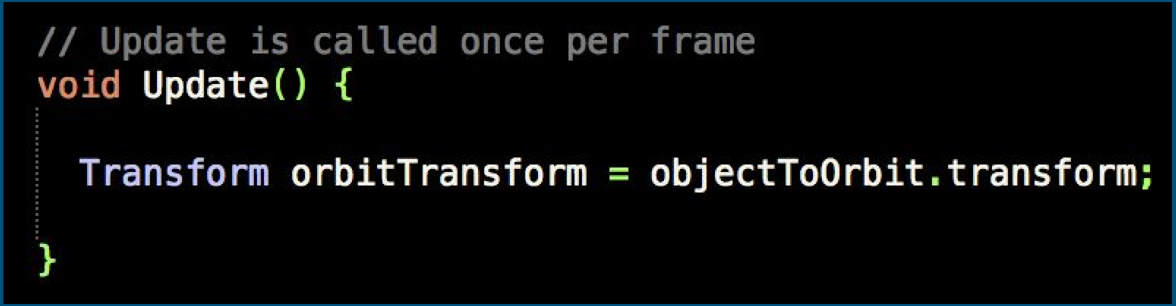
☑ My Orbit Script (Script)

Script MyOrbitScript

Object To Orbit None (Game Object)

Unity Scripting: Multiple Objects

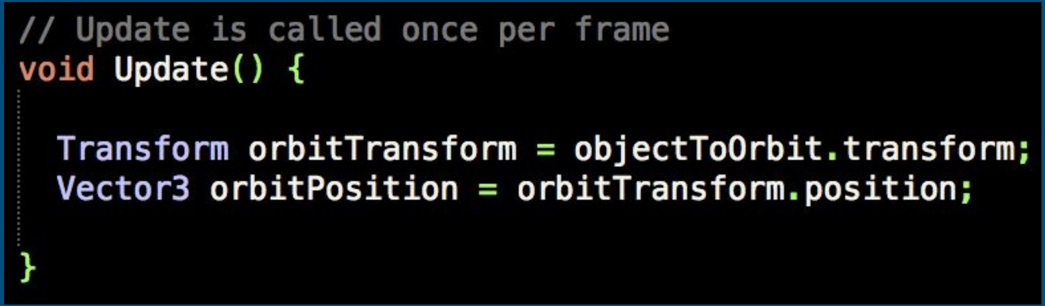
- So, how do we actually orbit around an object?
- First, we need to get the position to rotate around.
- What component of our “objectToOrbit” has it’s position?
- Remember: **All** GameObject’s have Transform components
    - Since this is true, we can get the transform component without GetComponent<>()!
    - Thanks, Unity!



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

    Transform orbitTransform = objectToOrbit.transform;

}
```

Unity Scripting: Multiple Objects

- We have the Transform component, but how do we get the position?
- “Position” is just a value of the Transform component!

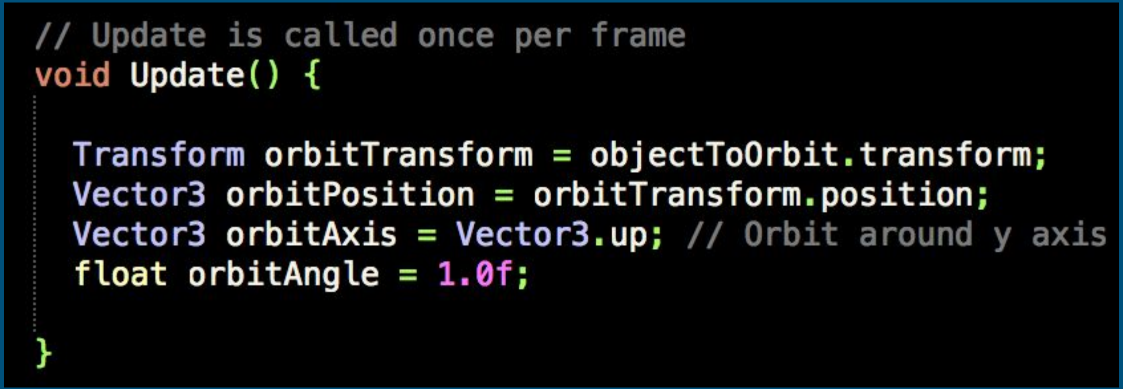


```
// Update is called once per frame
void Update() {

  Transform orbitTransform = objectToOrbit.transform;
  Vector3 orbitPosition = orbitTransform.position;

}
```

- Positions, Rotations, and Scales are all stored as “Vector3”
- Vector3’s are just data structures with an x, y, and z value.
- Useful for 3D space!

Unity Scripting: Multiple Objects

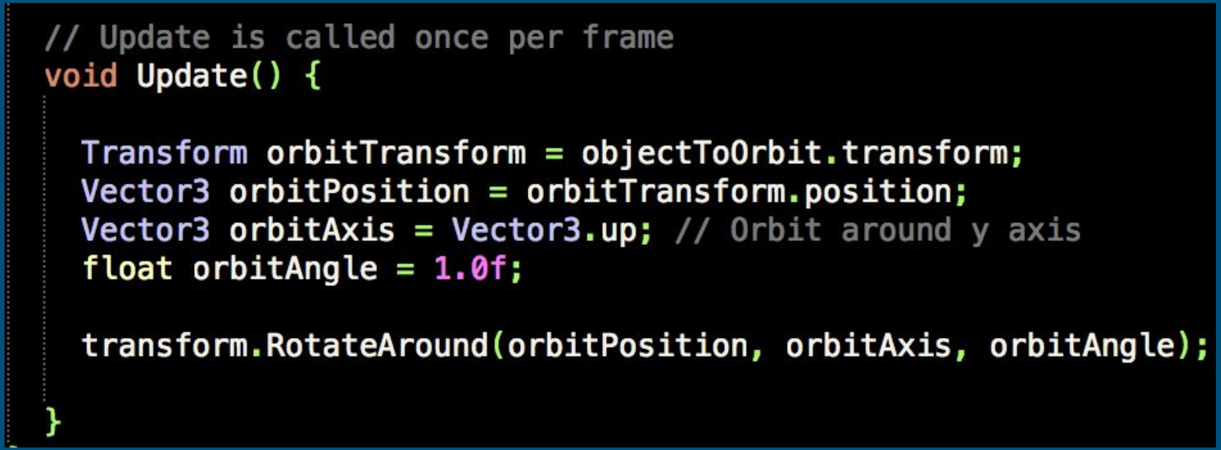
- Next, we must know an axis to rotate around.
    - Y axis: Vector3.up & Vector3.down
    - X axis: Vector3.right & Vector3.left
    - Z axis: Vector3.forward & Vector3.back
- Finally, we must know how ***much*** the object should rotate each frame.
    - This will be an angle. i.e. the value “5” would rotate the object 5 degrees.



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

  Transform orbitTransform = objectToOrbit.transform;
  Vector3 orbitPosition = orbitTransform.position;
  Vector3 orbitAxis = Vector3.up; // Orbit around y axis
  float orbitAngle = 1.0f;

}
```

Unity Scripting: Multiple Objects

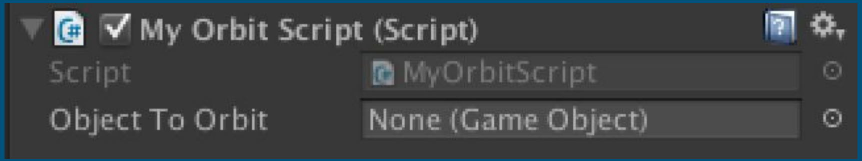
- We have all the pieces we need!
- Transform components have a nice “RotateAround()” method.
- Let’s modify the ***current*** object’s transform so that it rotates!
- Method: RotateAround(Vector3 point, Vector3 axis, float angle);



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

    Transform orbitTransform = objectToOrbit.transform;
    Vector3 orbitPosition = orbitTransform.position;
    Vector3 orbitAxis = Vector3.up; // Orbit around y axis
    float orbitAngle = 1.0f;

    transform.RotateAround(orbitPosition, orbitAxis, orbitAngle);

}
```

Unity Scripting: Multiple Objects

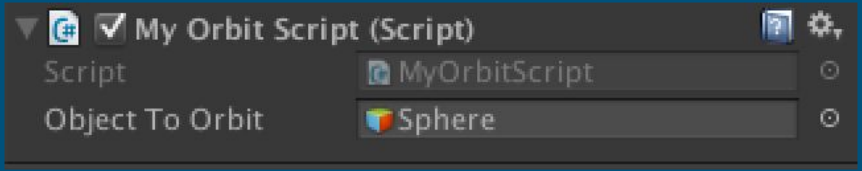
- Awesome! All that’s left is to drag in what we want to orbit.
- Drag your colorful sphere from hierarchy into the “objectToOrbit” field.



☑ My Orbit Script (Script)

Script: MyOrbitScript

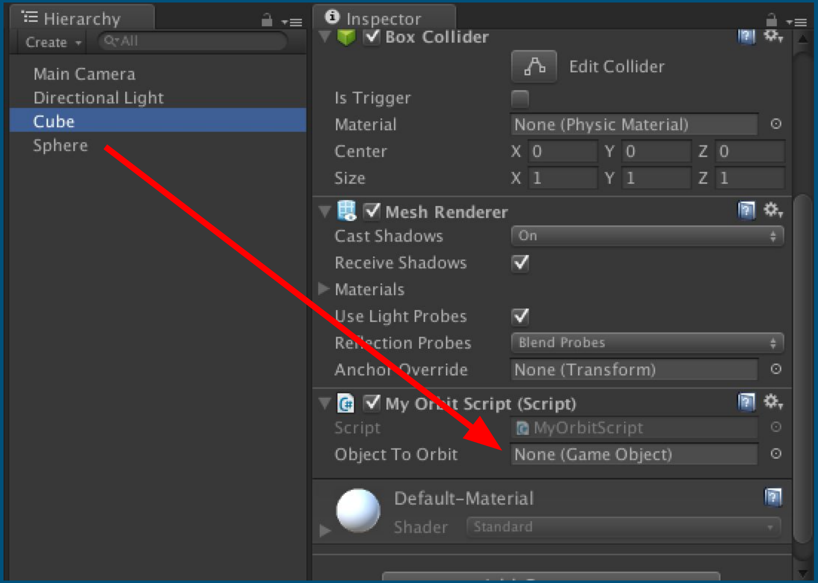
Object To Orbit: None (Game Object)



☑ My Orbit Script (Script)

Script: MyOrbitScript

Object To Orbit: Sphere



☑ Box Collider

Edit Collider

Is Trigger: □

Material: None (Physic Material)

Center: X 0 Y 0 Z 0

Size: X 1 Y 1 Z 1

☑ Mesh Renderer

Cast Shadows: On

Receive Shadows: ☑

Materials

Use Light Probes: ☑

Reflection Probes: Blend Probes

Anchor Override: None (Transform)

☑ My Orbit Script (Script)

Script: MyOrbitScript

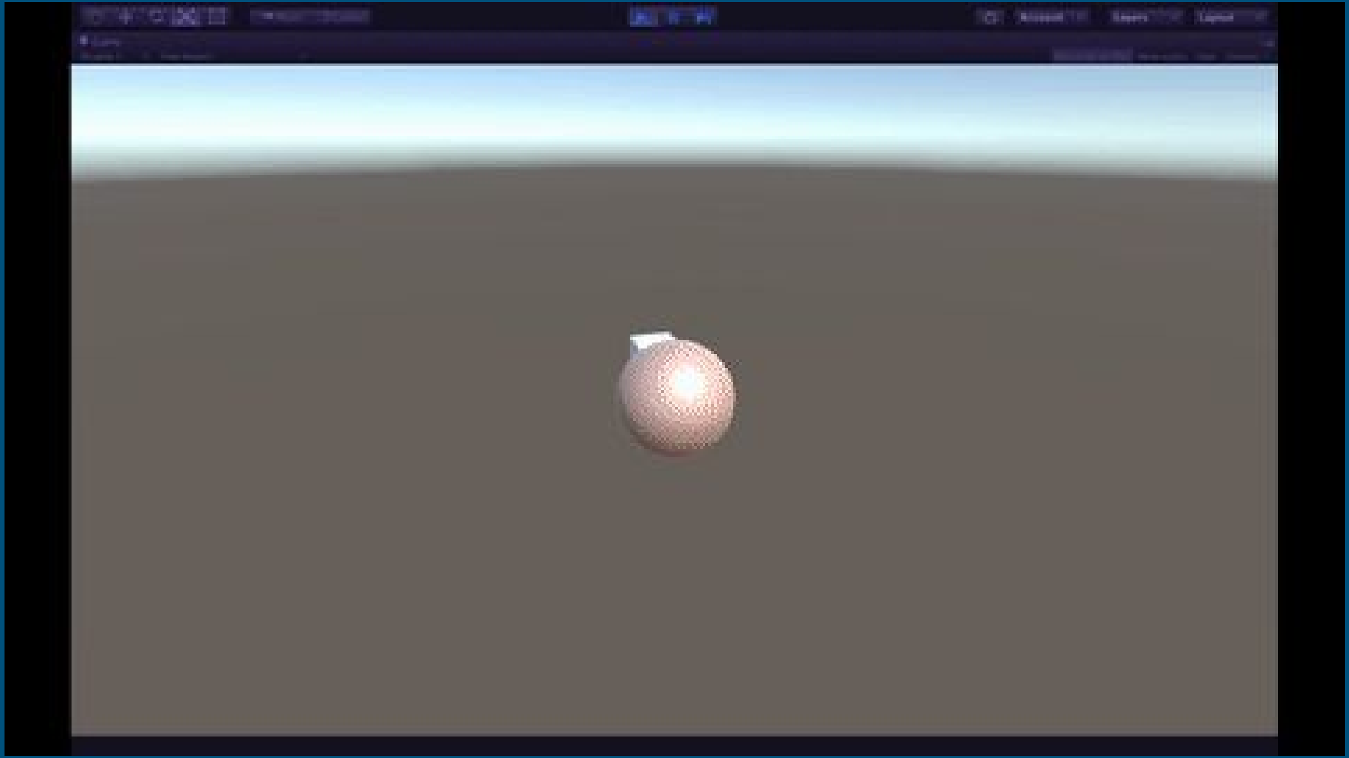
Object To Orbit: None (Game Object)

Default-Material

Shader: Standard

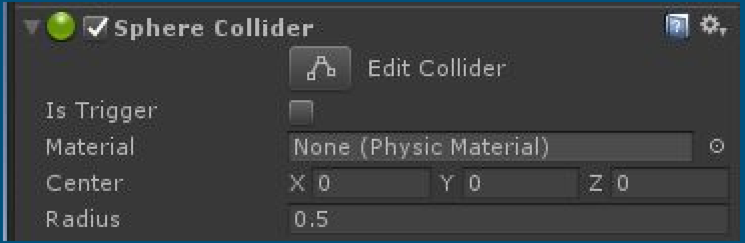
Unity Scripting: Multiple Objects

- Play!



Unity Scripting: Colliders and Triggers

- Remember that “Collider” component from before?
- Colliders do more than just cause physical collisions!
- Whenever two colliders “collide”, collision data is actually sent to script.
- We gain access to both colliding objects, exact collision points, physics, etc.
- This also works with colliders that are “Triggers”
  - When a trigger comes in contact with another collider!
- We can use collision and trigger events in script to have more control!



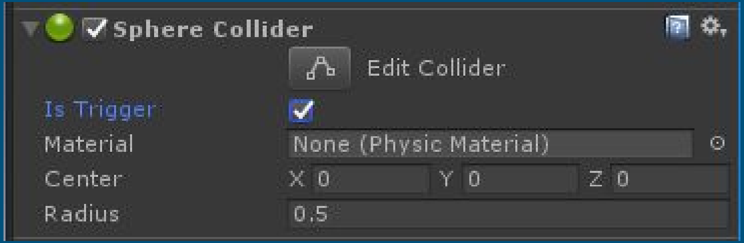
☑ Sphere Collider

Is Trigger ☐

Material None (Physic Material)

Center X 0 Y 0 Z 0

Radius 0.5



☑ Sphere Collider

Is Trigger ☑

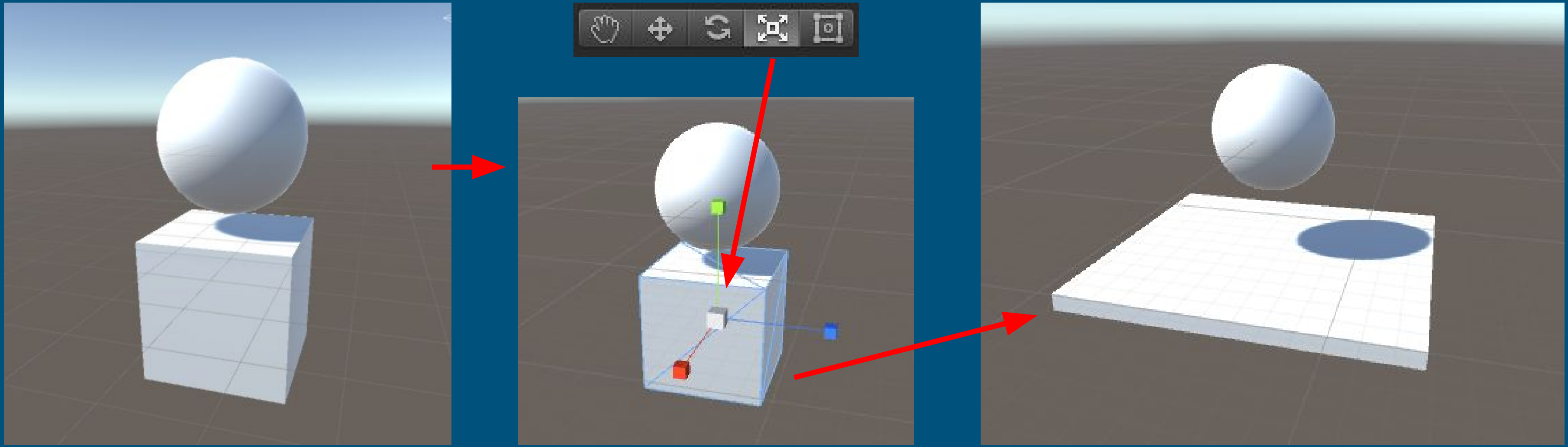
Material None (Physic Material)

Center X 0 Y 0 Z 0

Radius 0.5

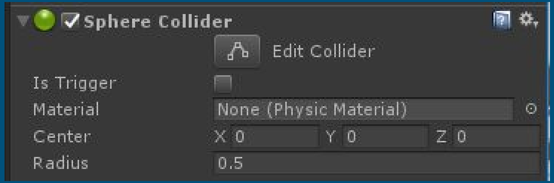
Unity Scripting: Colliders and Triggers

- First, let’s give our colorful sphere something to collide with.
- Go ahead and make a new cube, this will be our floor.



Unity Scripting: Colliders and Triggers

- Great, we have a floor! Now, also notice that both objects have colliders!
  - Default GameObject’s in Unity come with them - but it’s important to note.



☑ Sphere Collider

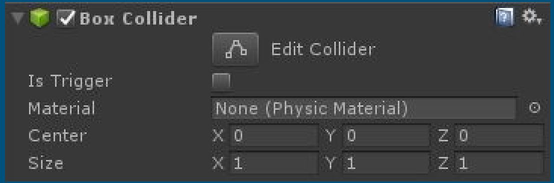
Edit Collider

Is Trigger □

Material: None (Physic Material)

Center: X 0 Y 0 Z 0

Radius: 0.5



☑ Box Collider

Edit Collider

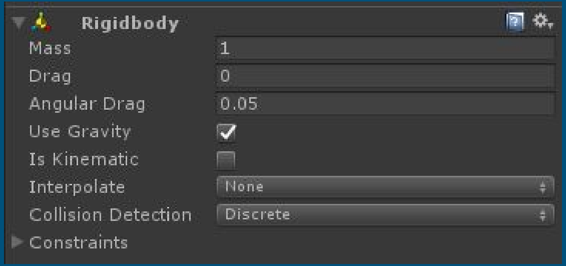
Is Trigger □

Material: None (Physic Material)

Center: X 0 Y 0 Z 0

Size: X 1 Y 1 Z 1

- To make physics work, we also need to add a **Rigidbody** to the sphere!
- Add Component -> Rigidbody
- Do not add one to the floor!
  - We don’t want the floor falling!
  - We also don’t want the impact to move the floor!



Rigidbody

Mass: 1

Drag: 0

Angular Drag: 0.05

Use Gravity ☑

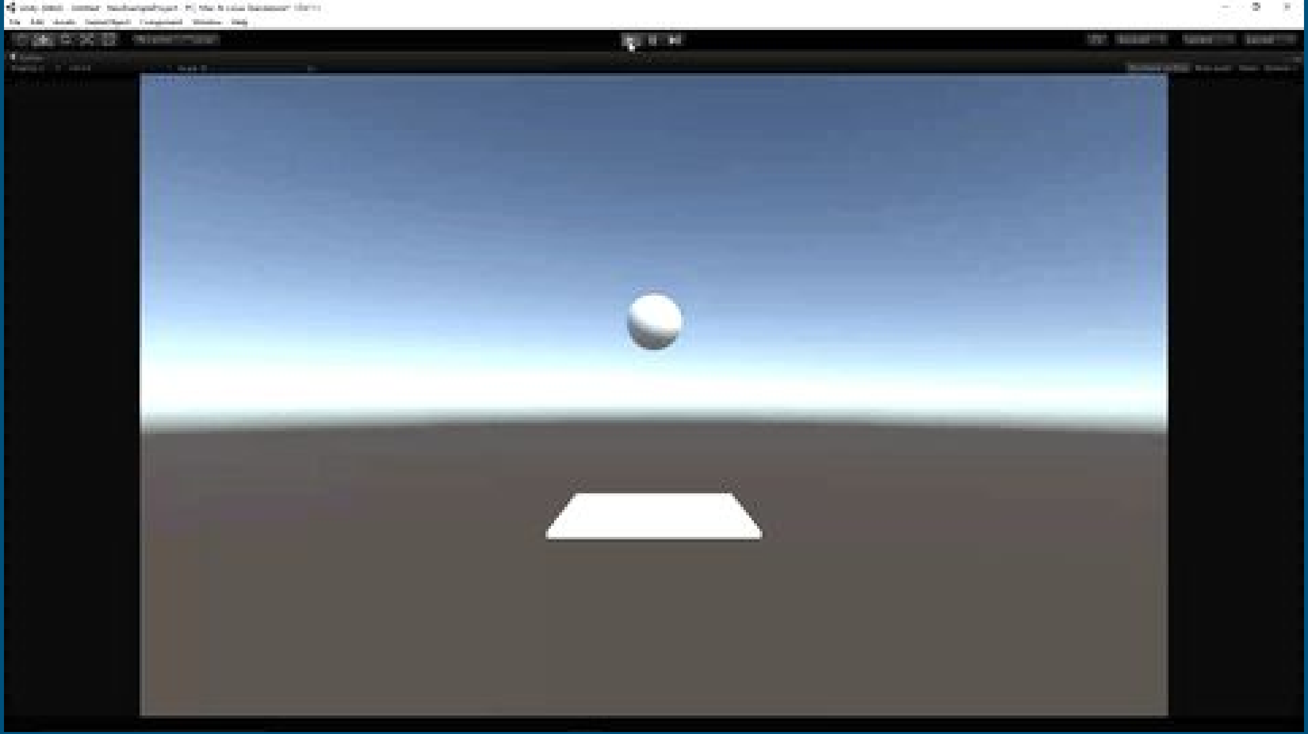
Is Kinematic □

Interpolate: None

Collision Detection: Discrete

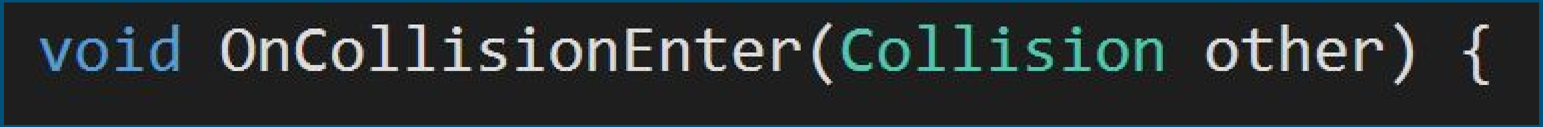
▸ Constraints

Unity Scripting: Colliders and Triggers



Unity Scripting: Colliders and Triggers

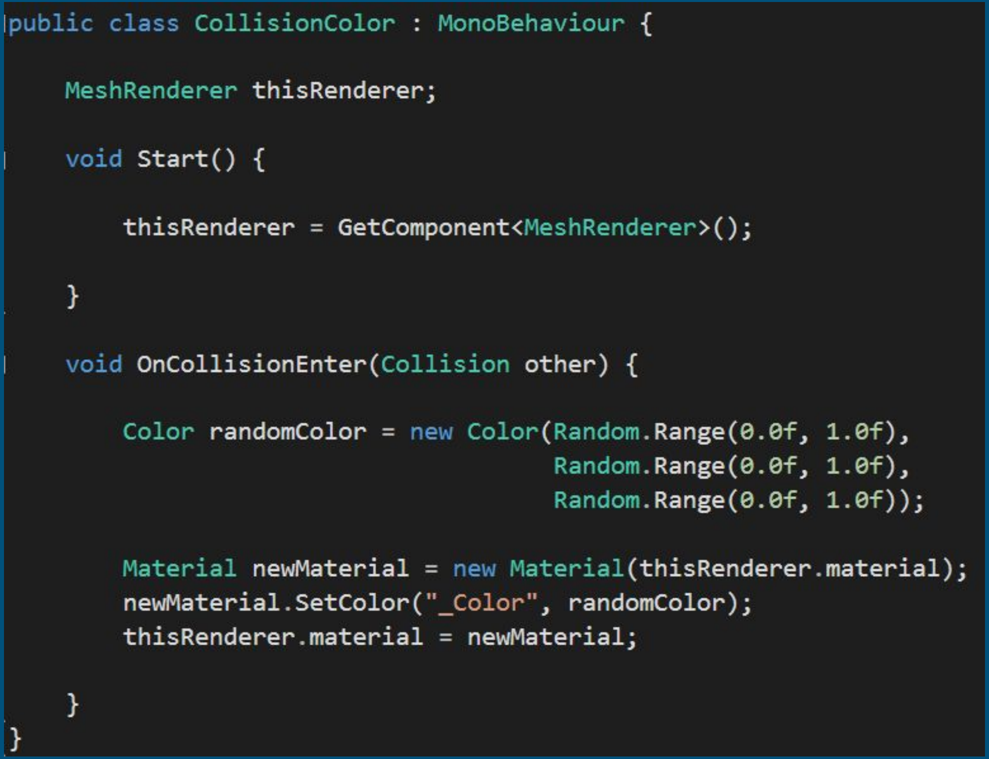
- Now let’s have something happen when the collision actually occurs!
- We’re going to change the floor color when something collides with it.
- Create a new script, i.e. “CollisionColor”
- The key for collisions in scripts is a very specific method…
- We’re going to use OnCollisionEnter(Collision other)
- This method will run automatically (Just like Start and Update)
    - Only when there’s a collision, though!
- The “other” parameter contains collision data
    - We can even get the colliding object from this!



```
void OnCollisionEnter(Collision other) {
```

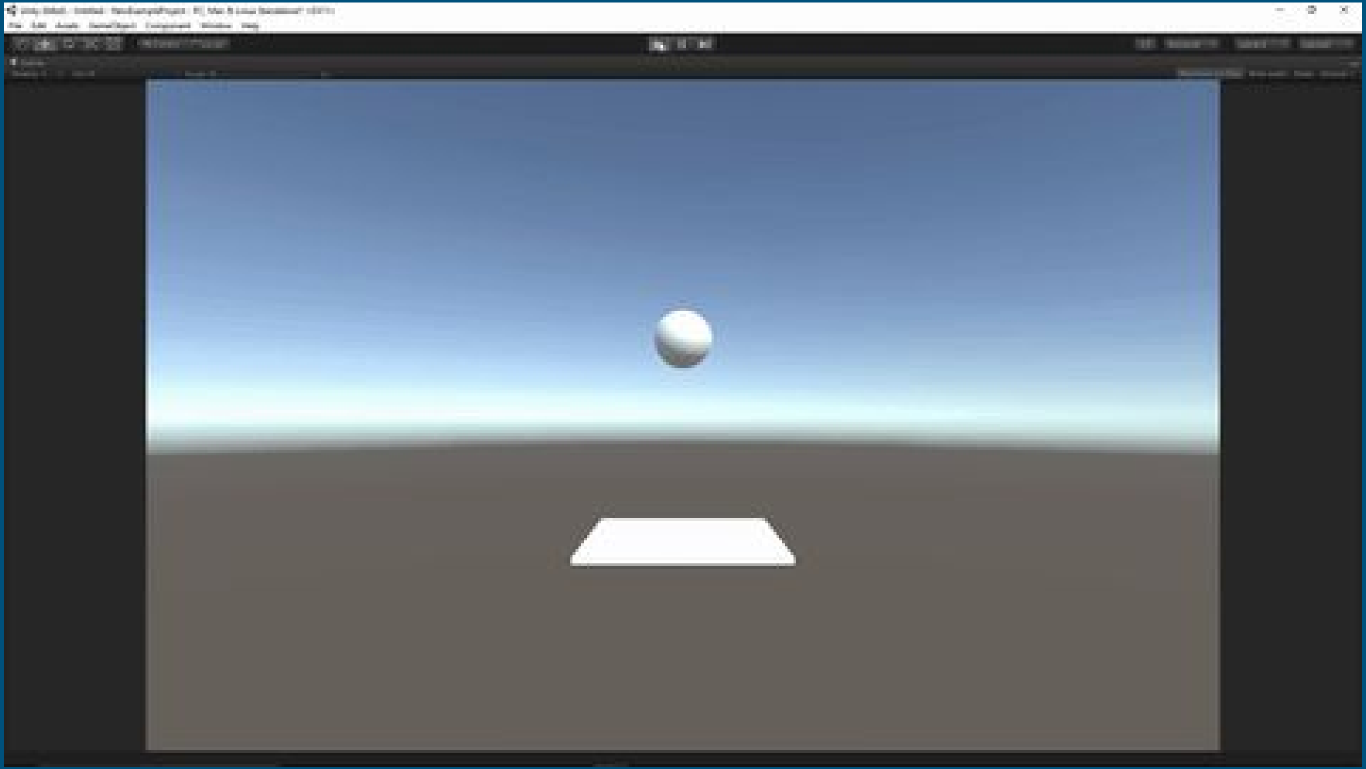
Unity Scripting: Colliders and Triggers

- Using color code from before…
- Just put into new method!
    - OnCollisionEnter(...)
- Will run when something collides



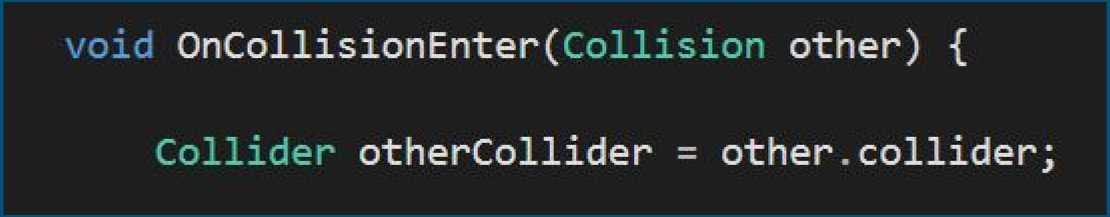
```
public class CollisionColor : MonoBehaviour {

    MeshRenderer thisRenderer;

    void Start() {

        thisRenderer = GetComponent<MeshRenderer>();

    }

    void OnCollisionEnter(Collision other) {

        Color randomColor = new Color(Random.Range(0.0f, 1.0f),
                                      Random.Range(0.0f, 1.0f),
                                      Random.Range(0.0f, 1.0f));

        Material newMaterial = new Material(thisRenderer.material);
        newMaterial.SetColor("_Color", randomColor);
        thisRenderer.material = newMaterial;

    }
}
```

Unity Scripting: Colliders and Triggers



Unity Scripting: Colliders and Triggers

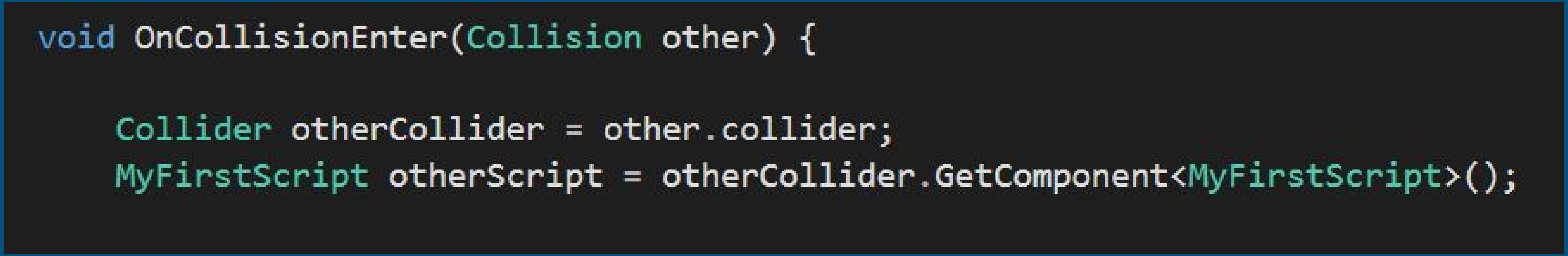
- Awesome, we changed the platform using a collision event!
- So what if we want to change the colliding object?
- Let’s use the collision data to disable the color changing on the sphere!
- Remember, the color changing script on the sphere is just a component!
- Using collision data, we get access to the other object’s **collider**.
- A collider is just a component as well!
- This component will be our gateway to all the other components!



```
void OnCollisionEnter(Collision other) {

    Collider otherCollider = other.collider;
```

Unity Scripting: Colliders and Triggers

- Cool, we have the collider, and that’s all we need!
- From there, let’s just use GetComponent to get the script on the sphere!
- We called in MyFirstScript in this tutorial, so…

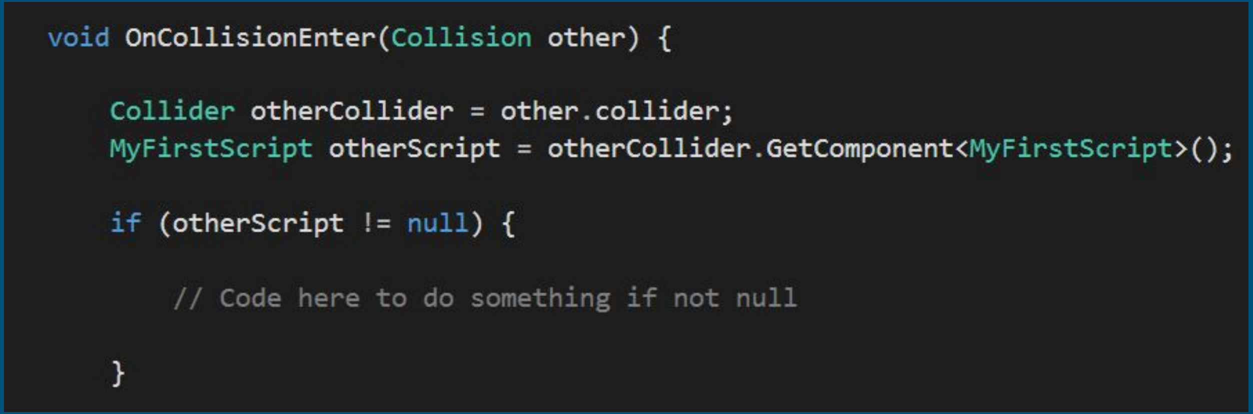


```
void OnCollisionEnter(Collision other) {

    Collider otherCollider = other.collider;
    MyFirstScript otherScript = otherCollider.GetComponent<MyFirstScript>();
```

- Now we have the script! (If it exists…)

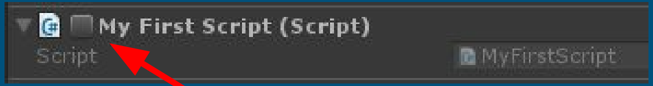
Unity Scripting: Colliders and Triggers

- Before we move on, important question:
- What happens if the colliding object DOESN’T have “MyFirstScript”
  - Uh oh! We shouldn’t try to do anything with it!
- Generally when using GetComponent, it’s a good idea to check for **null**
  - Null = nothing = nonexistent



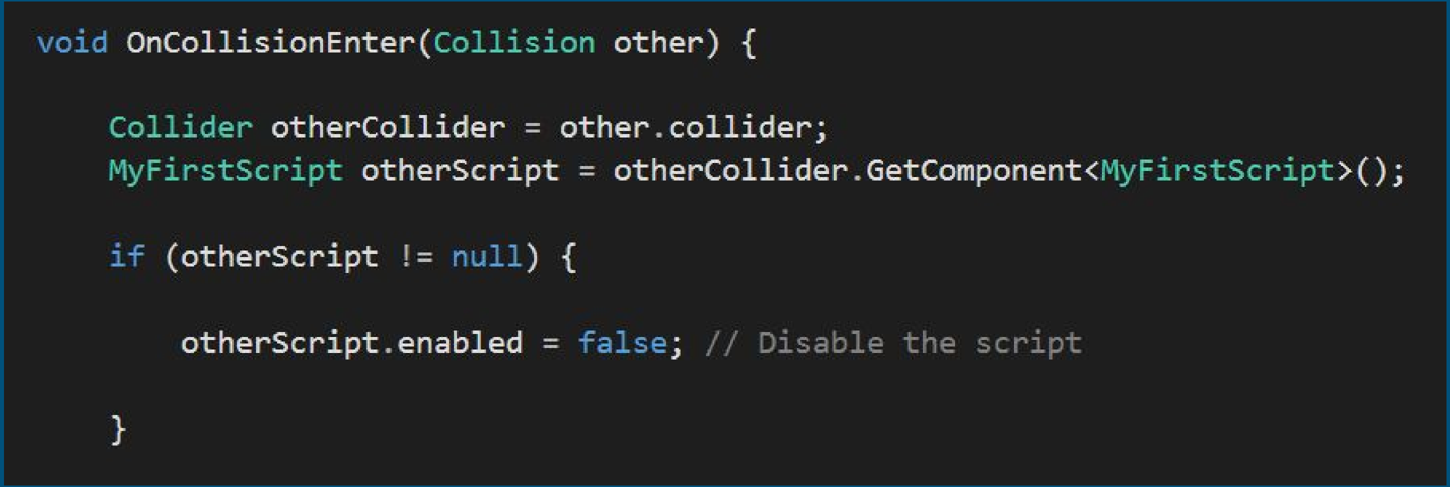
```
void OnCollisionEnter(Collision other) {

    Collider otherCollider = other.collider;
    MyFirstScript otherScript = otherCollider.GetComponent<MyFirstScript>();

    if (otherScript != null) {

        // Code here to do something if not null

    }
```

Unity Scripting: Colliders and Triggers



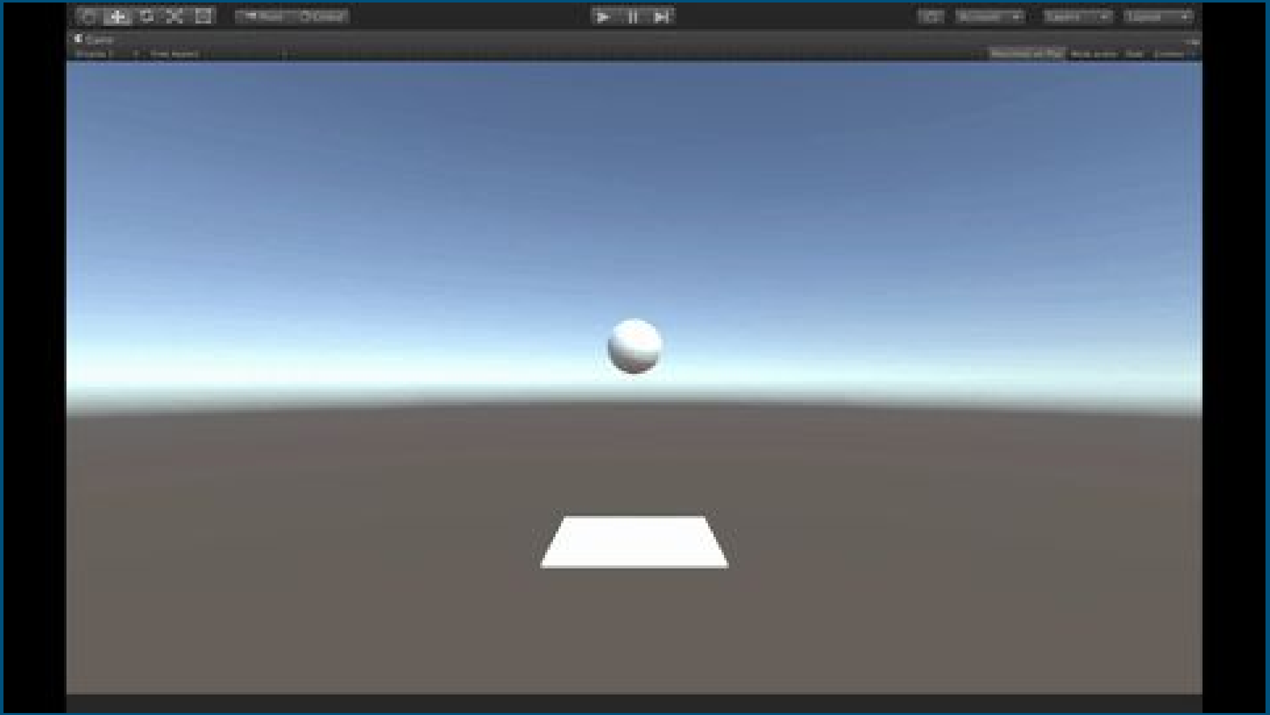
Script MyFirstScript

- Finally, let’s stop that other script from running!
- All components have “**enabled**” values! (It’s the little checkbox in Inspector)
- We can just disable it, or set it’s **enabled** value to false.



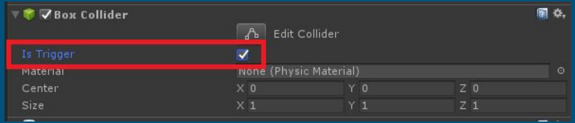
```
void OnCollisionEnter(Collision other) {

    Collider otherCollider = other.collider;
    MyFirstScript otherScript = otherCollider.GetComponent<MyFirstScript>();

    if (otherScript != null) {

        otherScript.enabled = false; // Disable the script

    }
```

Unity Scripting: Colliders and Triggers



Unity Scripting: Colliders and Triggers

- This is awesome, we can change any colliding object’s components!
- Now what about ***triggers***?
- They’re actually very similar!



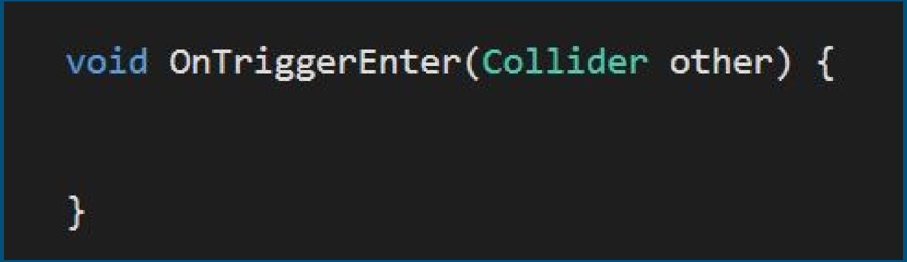
Is Trigger ☑

Material None (Physic Material)

Center X 0 Y 0 Z 0

Size X 1 Y 1 Z 1

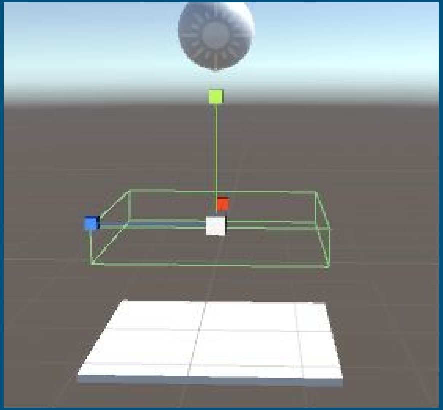
- A **trigger** just doesn’t have any physical collision (i.e. passes right through!)
  - NOTE that the object still needs a rigidbody to collide with a trigger!
- The method for triggers is OnTriggerEnter(Collider other)
  - We get the collider directly this time since there’s no actual collision data



```
void OnTriggerEnter(Collider other) {


}
```

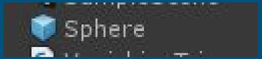
Unity Scripting: Colliders and Triggers

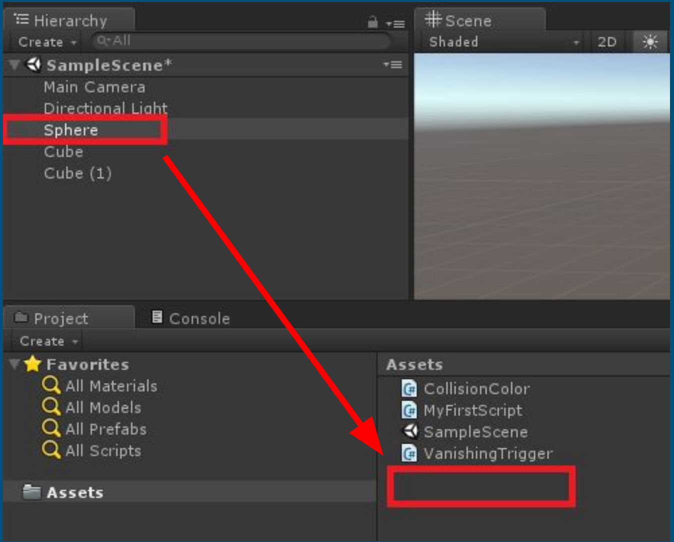
- To make this work, we first must set up a trigger.
- The easiest way to do this is to start with a cube
  - Easy to visualize the trigger space
- Resize the cube to be your trigger area, and make the collider a trigger
- Now, something should happen when anything goes in the trigger area!
- Let’s make our falling sphere game a bit more fun…
- … and have it spawn more spheres!



Unity Scripting: Instantiation

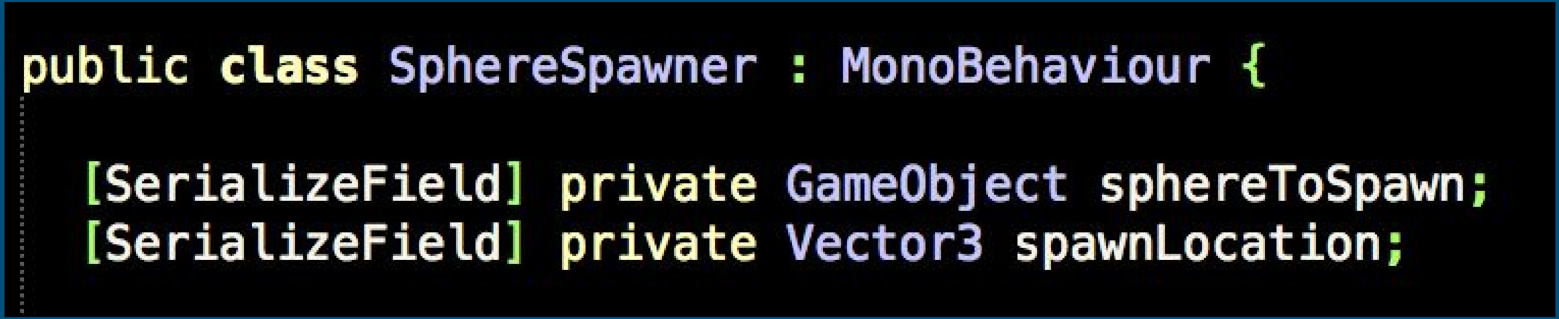
- Before we move forward, let’s talk about **Instantiation.**
- Remember what a **Prefab** is?
    - A prefabricated GameObject stored outside of the scene
- **Instantiation** clones prefabs at runtime
    - You can specify **what** is cloned.
    - You can specify **where** they go.
    - You can specify how they’re **rotated.**





Unity Scripting: Instantiation

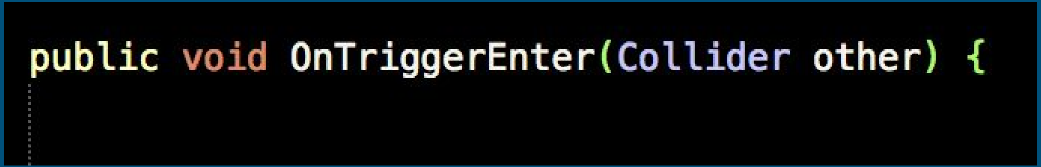
- Start by making a new script for our instantiation
  - I.e. “SphereSpawner”
- Let’s give this script some variables we can modify in the inspector!
- 1. A GameObject field for the prefab we want to instantiate
- 2. A Vector3 field for the location where the object should spawn



```
public class SphereSpawner : MonoBehaviour {

    [SerializeField] private GameObject sphereToSpawn;
    [SerializeField] private Vector3 spawnLocation;
```

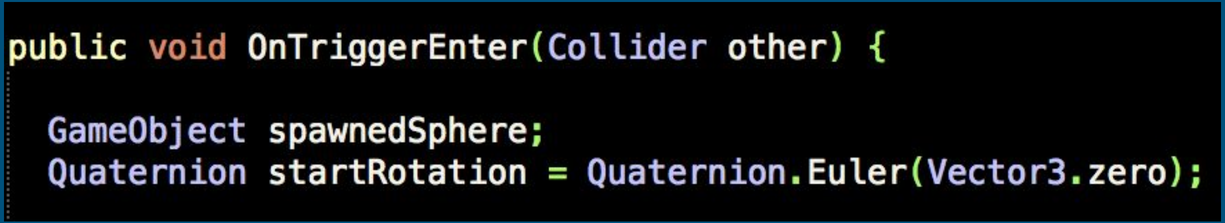
Unity Scripting: Colliders and Triggers

- Awesome! Now, we know we want our object to spawn OnTriggerEnter



```
public void OnTriggerEnter(Collider other) {
```

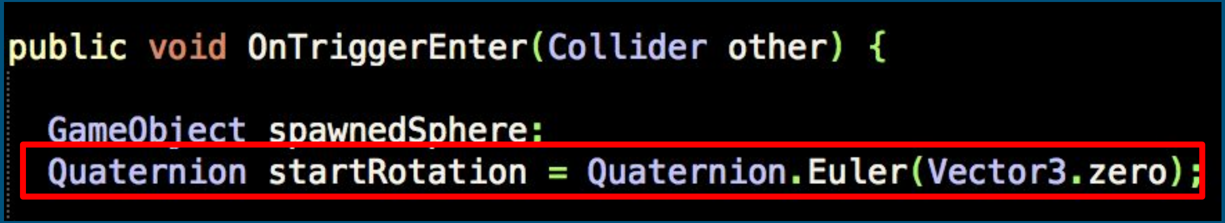
- Let’s set up two additional variables:
    - One to store the spawned GameObject after instantiation
    - Another to determine what rotation the object should start at



```
public void OnTriggerEnter(Collider other) {

  GameObject spawnedSphere;
  Quaternion startRotation = Quaternion.Euler(Vector3.zero);
```

Unity Scripting: A Digression on Quaternions

- Whoa whoa whoa, wait, what's that "Quaternion" thing?!?!

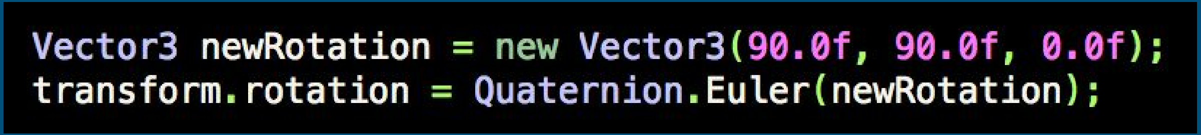


```
public void OnTriggerEnter(Collider other) {

  GameObject spawnedSphere;
  Quaternion startRotation = Quaternion.Euler(Vector3.zero);
```

- Rotations in Unity are stored as **Quaternions**
- Quaternions contain x, y, z, and **w** values.
- Quaternions are **complex** numbers… x, y, z are NOT the actual rotations!!
- However, thankfully, 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).

Unity Scripting: A Digression on Quaternions

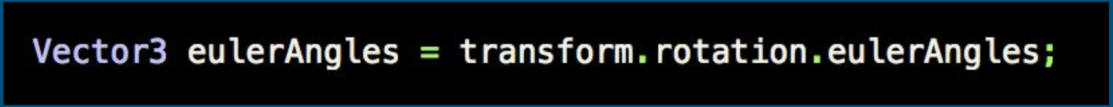
- Since Unity is our friend, it auto converts between Euler and Quaternions
- Easy method to use: Quaternion.Euler(Vector3 angles)
  - Returns a Quaternion using the specified angles



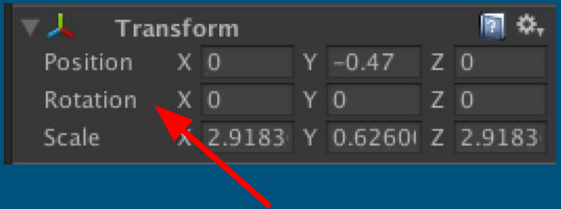
```
Vector3 newRotation = new Vector3(90.0f, 90.0f, 0.0f);
transform.rotation = Quaternion.Euler(newRotation);
```

Sets rotation to 90, 90, 0

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



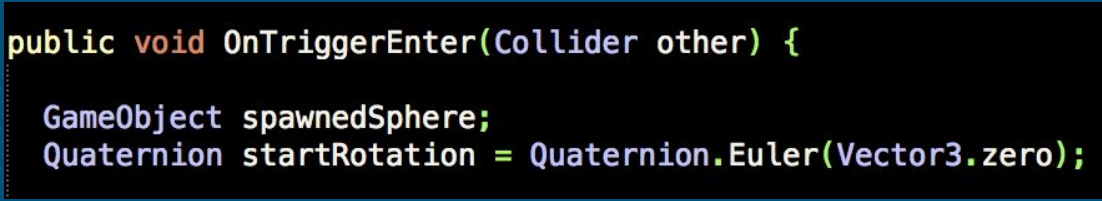
```
Vector3 eulerAngles = transform.rotation.eulerAngles;
```



```
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
```

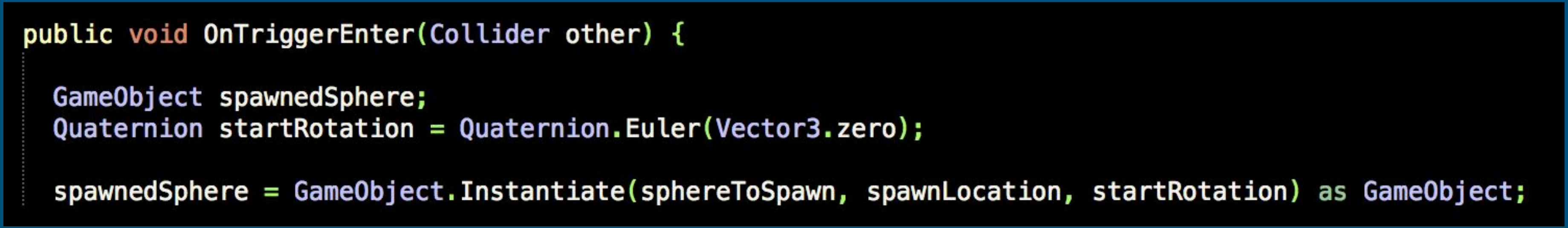
- Note: The rotation values in the inspector are thankfully **Euler Angles**

Unity Scripting: Instantiation



```
public void OnTriggerEnter(Collider other) {

  GameObject spawnedSphere;
  Quaternion startRotation = Quaternion.Euler(Vector3.zero);
```

- Back on track…
- Now just to instantiate!
- **GameObject.Instantiate(Object prefab, Vector3 pos, Quaternion rotation)**
- We can use this method to make our new sphere!

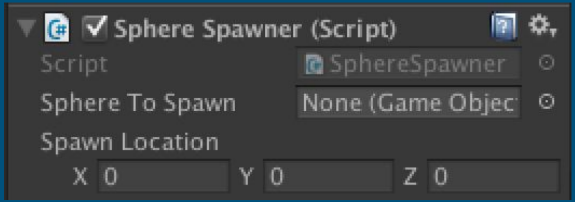


```
public void OnTriggerEnter(Collider other) {

  GameObject spawnedSphere;
  Quaternion startRotation = Quaternion.Euler(Vector3.zero);

  spawnedSphere = GameObject.Instantiate(sphereToSpawn, spawnLocation, startRotation) as GameObject;
```

- Note: We use the “as GameObject” keyword to cast result as a GameObject
- Instantiate returns an “Object”, so we need to cast to the Component we want

Unity Scripting: Colliders and Triggers

- Finally, let’s fill in those Inspector fields!
- Drag your colorful sphere from the hierarchy to the “Sphere to Spawn” field
- Let’s change the “Y” value of the spawn location so that spheres spawn high

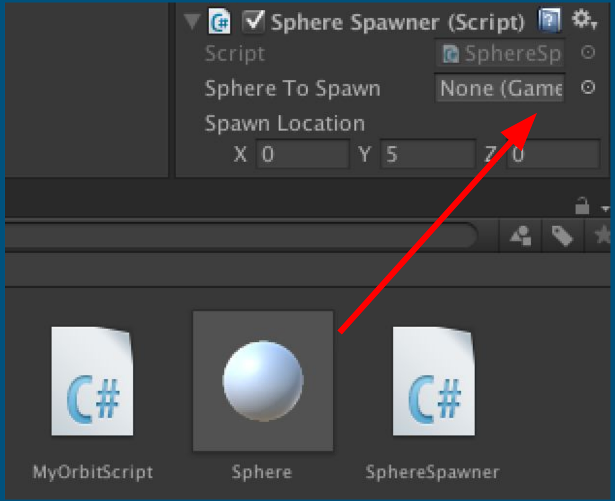


☑ Sphere Spawner (Script)

Script: SphereSpawner

Sphere To Spawn: None (Game Objec

Spawn Location: X 0 Y 0 Z 0

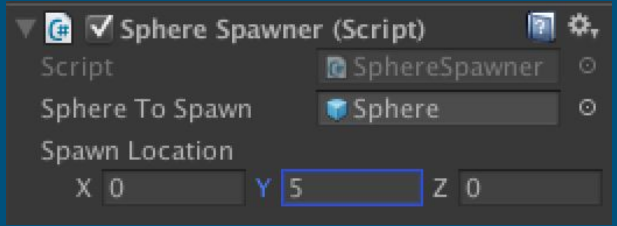


☑ Sphere Spawner (Script)

Script: SphereSp

Sphere To Spawn: None (Game

Spawn Location: X 0 Y 5 Z 0



☑ Sphere Spawner (Script)

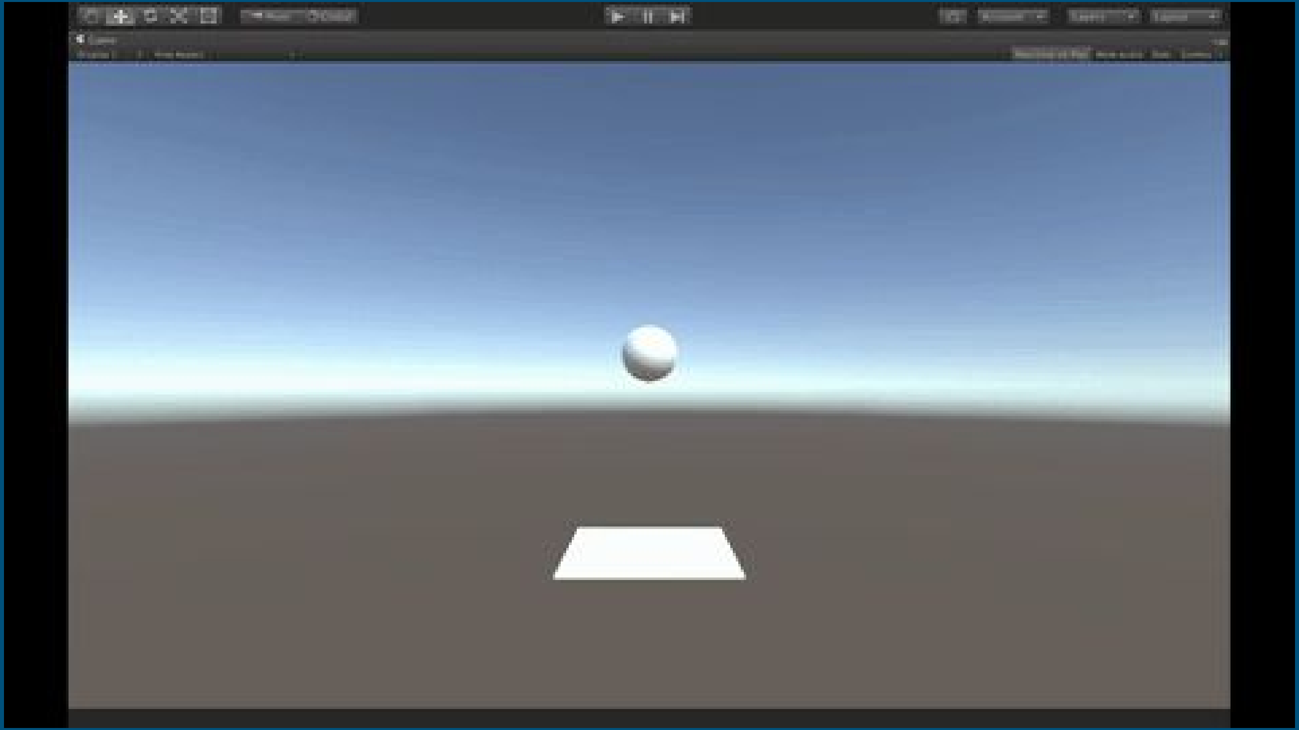
Script: SphereSpawner

Sphere To Spawn: Sphere

Spawn Location: X 0 Y 5 Z 0

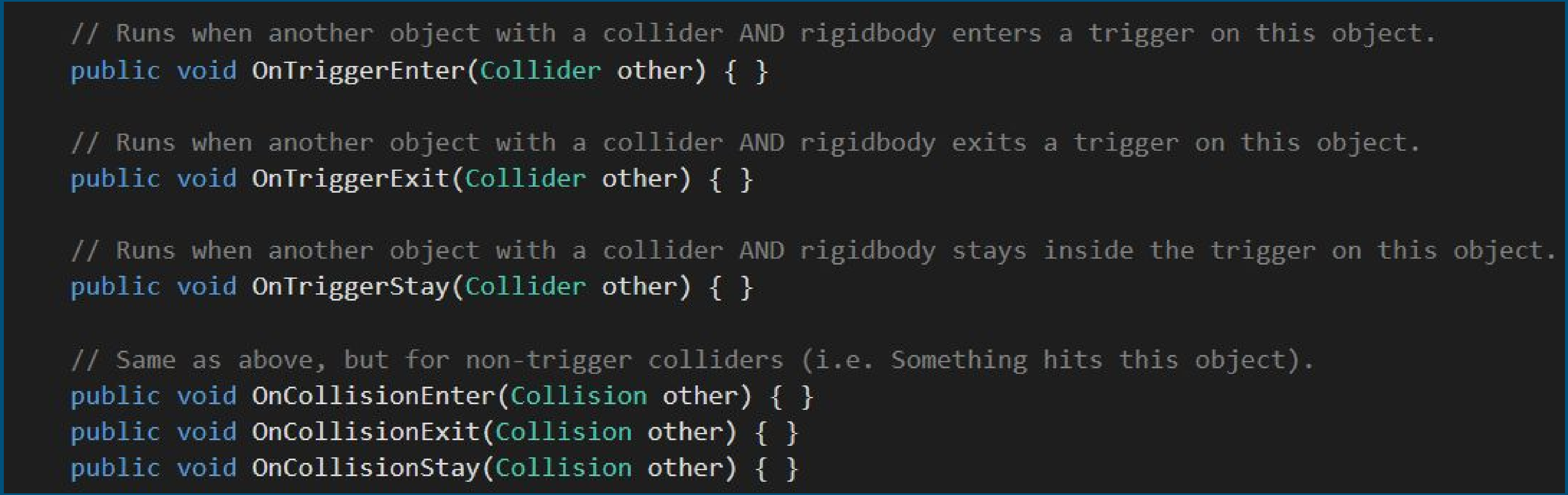
Unity Scripting:

- Play!



Unity Scripting: Collisions and Triggers

- There’s some other functions that accomplish similar things:



```
// 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) { }
```

Unity Scripting!

- Congrats!
- Brief review of what we’ve learned:
  - C# and Monobehaviours
  - Debugging
  - Public & Serialized variable fields
  - GetComponent
  - Component modification
  - Colliders and Triggers
  - Instantiation
- These are some basic tools that can build a game.
  - And definitely the most common!

Thanks!