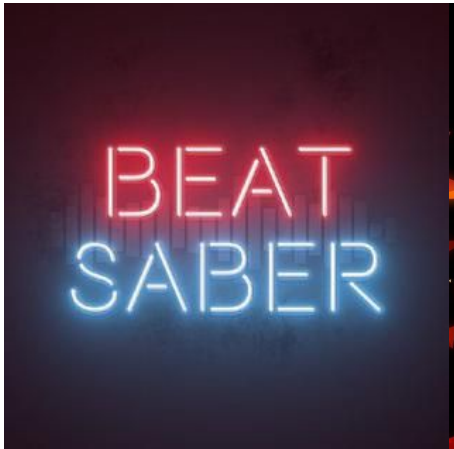
Introduction to Unity

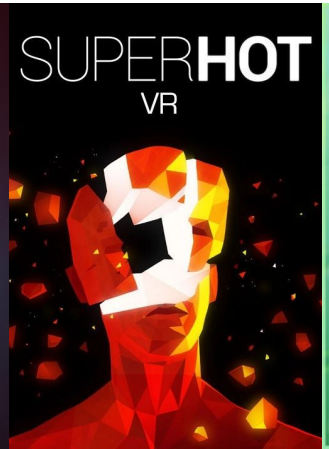
Unity

Unity is a cross-platform game development system

Consists of a game engine and an IDE

Can be used to develop games and applications for many different AR/VR platforms



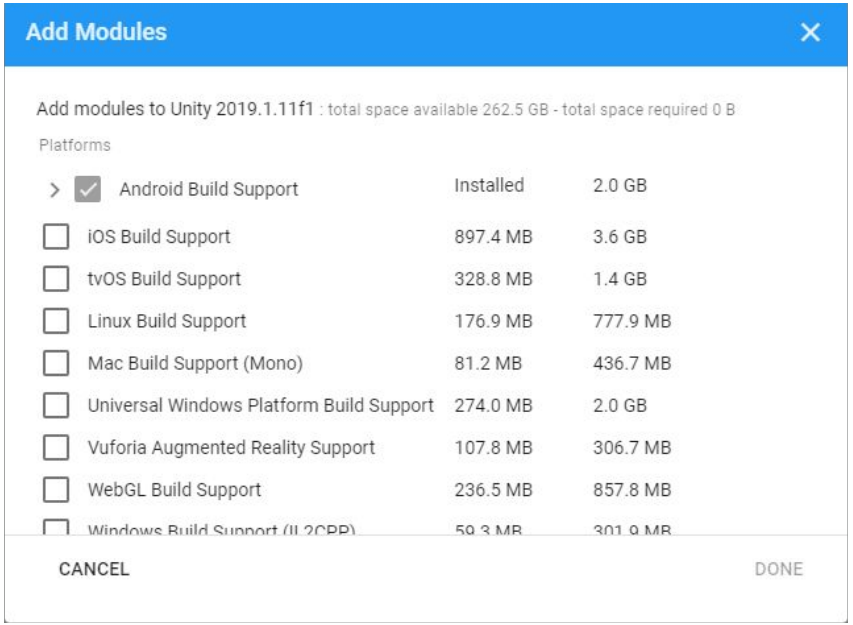


Installation

Unity is already installed on the lab computers in LWSN B131

If you wish to install your own computer:

- Download the ‘Personal Edition’: https://unity3d.com/get-unity
  - Make sure to get Unity version 2019.1.11f1
  - Make sure to add Android Build Support during installation.



Documentation

- Unity User Manual: https://docs.unity3d.com/Manual/index.html
- Scripting API: http://docs.unity3d.com/ScriptReference/index.html
- These pages should become your best friends.
- Also documentation on the OVR Utilities Plugin: https://developer.oculus.com/documentation/unity/unity-utilities-overview/

Unity Official Scripting Videos: These also serve as a good introduction to C#.

- Beginner Scripting Playlist: https://www.youtube.com/watch?v=Z0Z7xc18CcA&list=PLX2vGYjWbI0S9-X2Q021GUtolTqbUBB9B
- Intermediate Scripting Playlist: https://www.youtube.com/watch?v=HzIqrlSbjjU&list=PLX2vGYjWbI0S8YpPPKKvXZayCjkKj4bUP

Unity Basic Concepts

Project - The project contains all the elements that makes up the game, including models, assets, scripts, scenes, and so on.

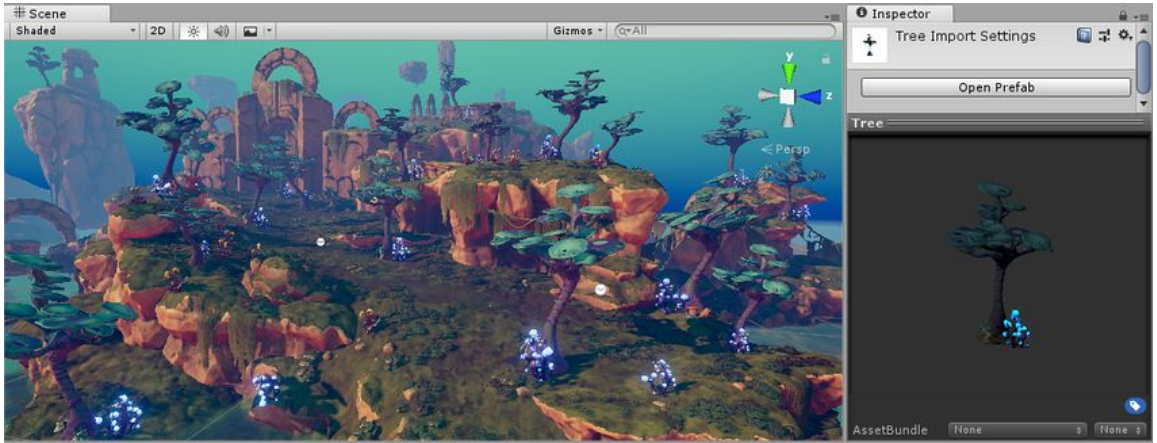
Scenes - A scene contains a collection of game objects that constitute the world that the player sees at any time.

Packages: A package is an aggregation of game objects and their associated metadata

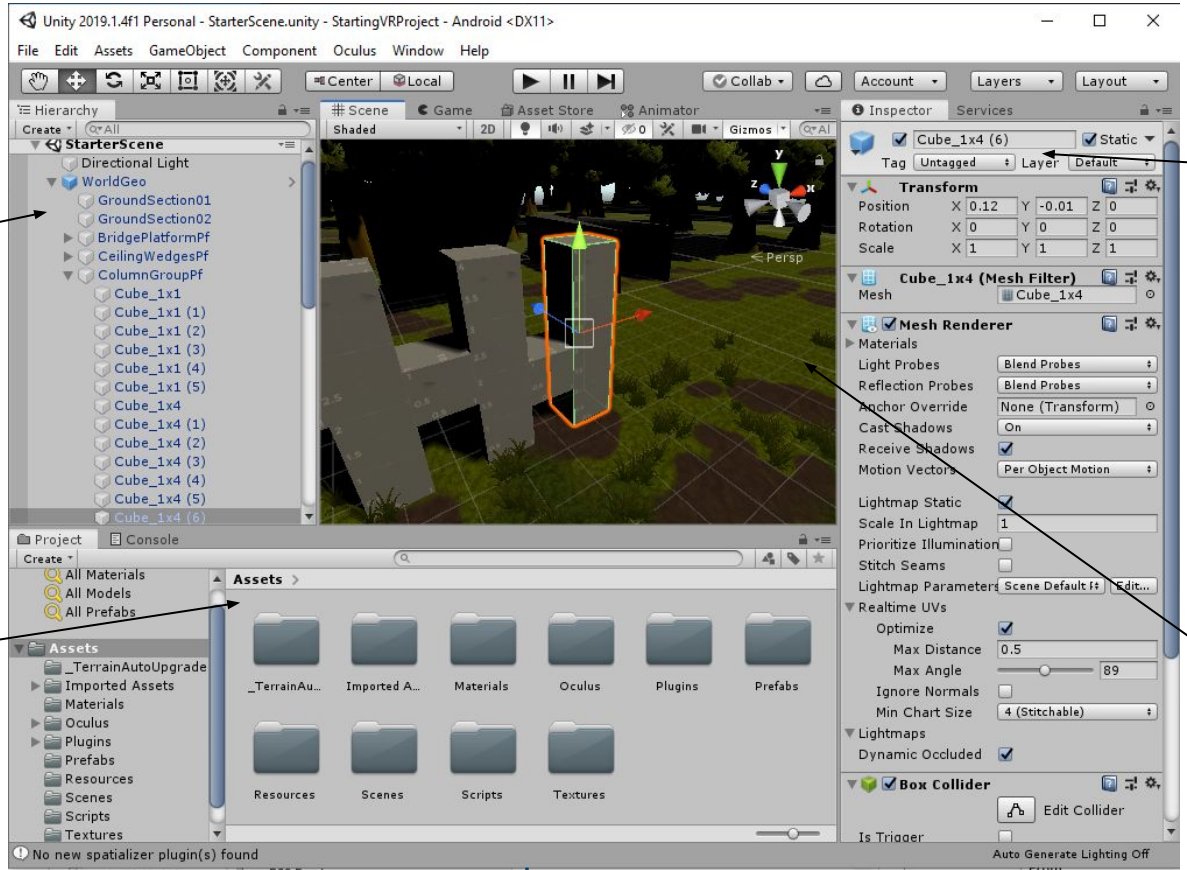
Unity Basic Concepts (continued)

Prefabs: A prefab is a template for grouping various assets under a single header.

- Prefabs are used for creating multiple instances of a common object.
- For example, you may have a large number of copies of a single element (e.g., street lights, trees)
- Prefabs can be instantiated during runtime



Overview of the Unity IDE:



Object hierarchy

Project assets

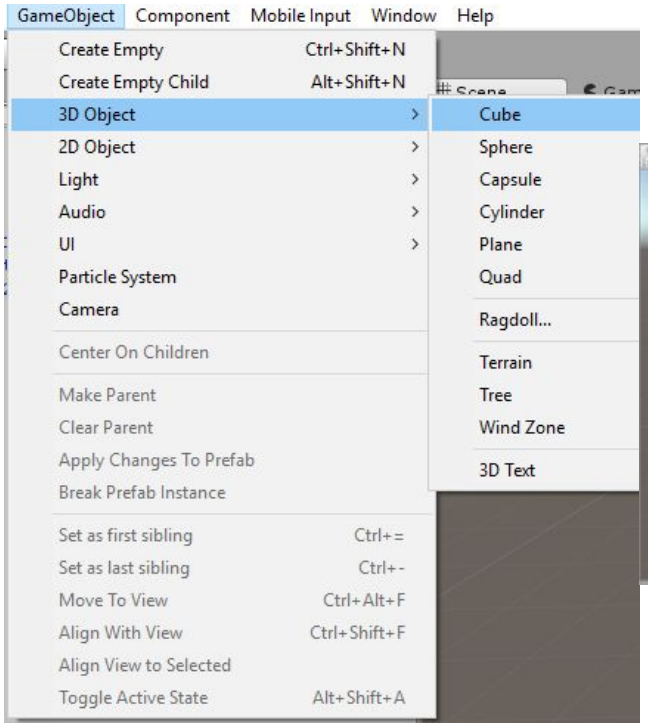
Game object Inspector

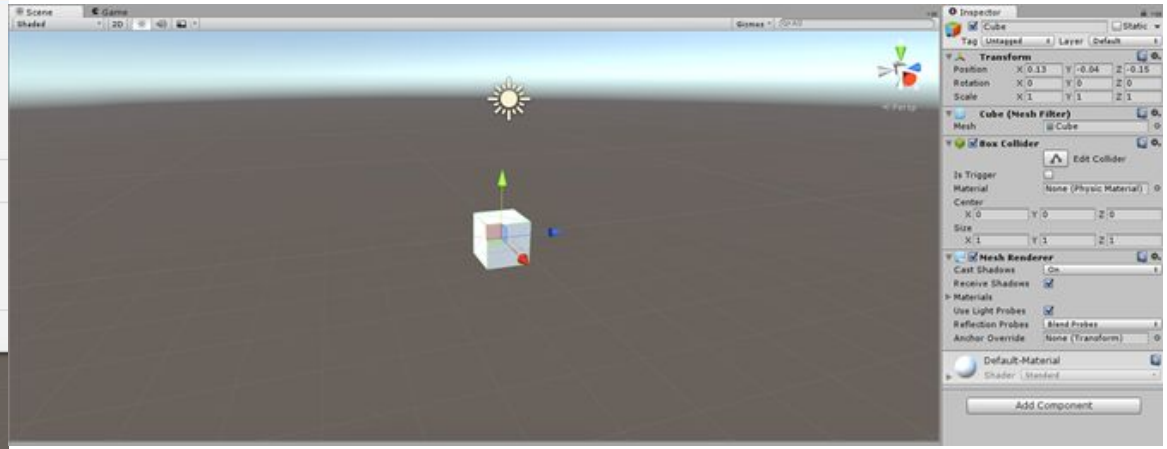
Scene View

Editor Camera Controls

- Controls:
  - Alt + Left Click & Move: Rotate Camera
  - Alt + Right Click & Move (Or Scroll Up/Down): Zoom in and out
  - Alt + Middle Click & Move: Move camera up/down or left right
- Flythrough Mode:
  - Click and hold right mouse button and now you can use FPS-like controls to move around through the scene (WASD, Q/E to move up down).
- Unity Documentation: http://docs.unity3d.com/Manual/SceneViewNavigation.html

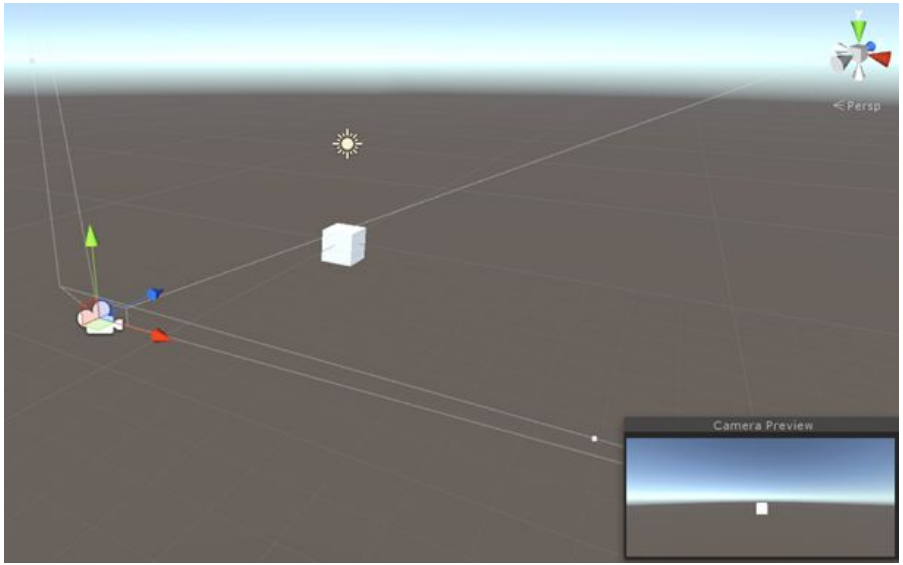
Creating Geometry via the Unity Editor





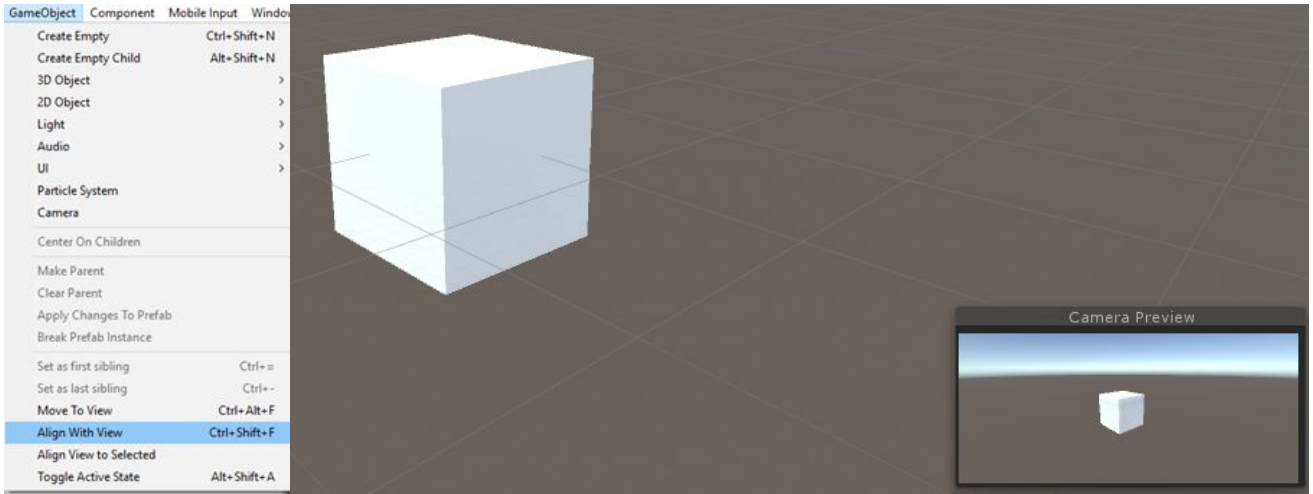
Setting Up The Scene Camera

- Do not confuse the scene camera with the editor camera.
- Unity scenes by default come with a “Main Camera.” Notice the tag of “MainCamera” in the inspector, this will be useful for accessing the camera from your scripts.
- “Camera Preview” box is useful to see what your camera can see.
- “Camera Preview” is what you will see when you hit Play.



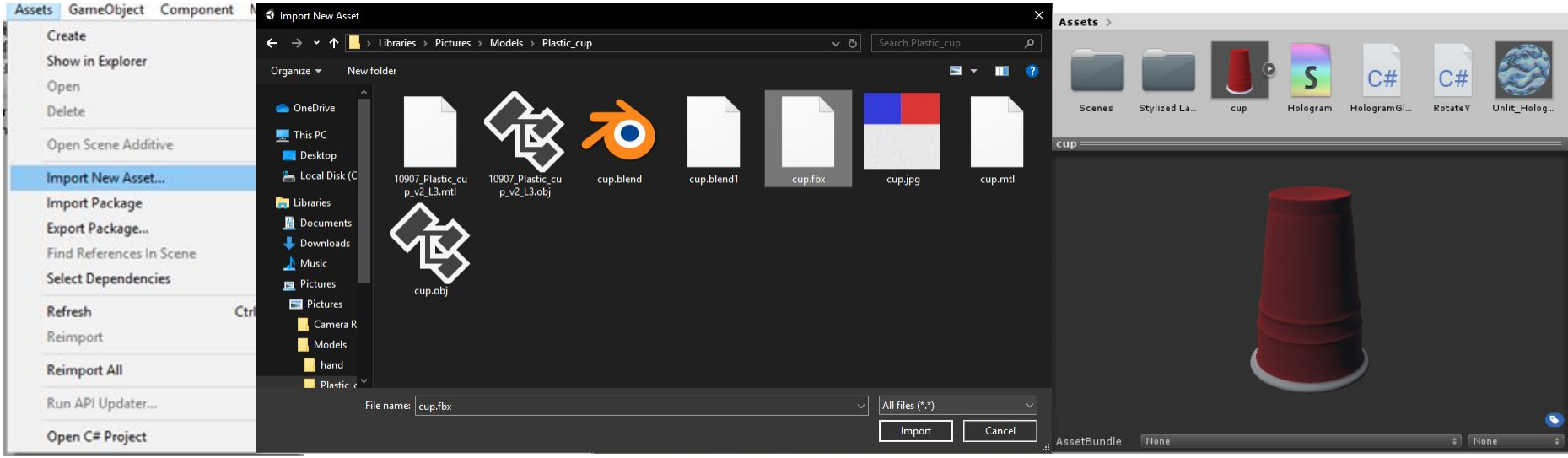
Setting Up The Scene Camera

- Moving the scene camera can be done manually by changing the position/rotation/scale in the Inspector.
- Or you can move the editor camera around as mentioned earlier, select the camera and align the camera to the view. Note that your camera should be selected before doing this.



Import External Objects

- Create and export an object from Maya/Blender/3ds Max as either an *.OBJ or a *.FBX. You can save this anywhere.
- Then import this asset into Unity. Unity will take care of everything for you.
- Alternatively, you can just save your *.OBJ or *.FBX inside the “Assets” folder.
    - You will need to right click on the folder it is in and click “Refresh” to get it to show up.
- Click and drag the object from the assets library into your scene hierarchy and it should now show up!



Game Objects

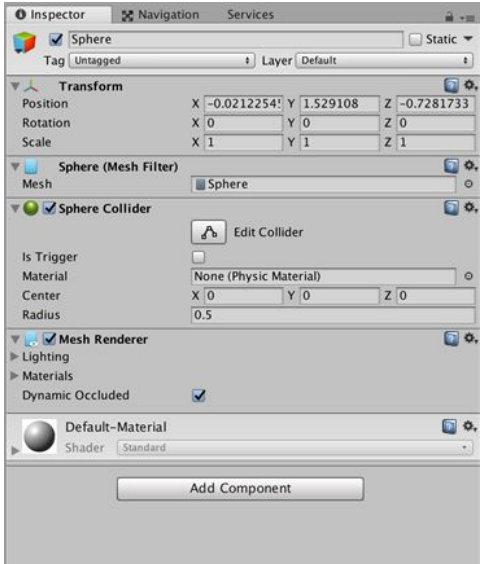
Game Objects: The *game objects* are all the “things” that constitute your scene.

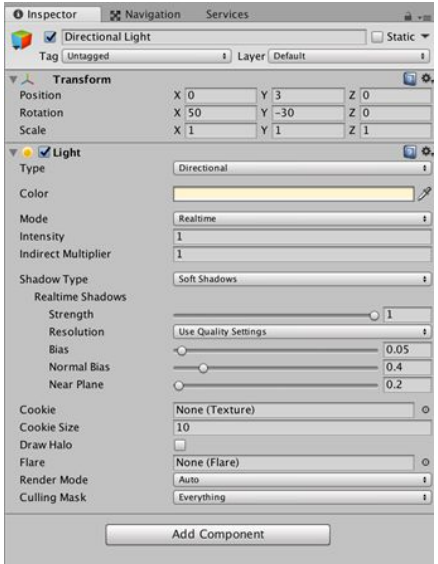
- Light sources
- Audio sources
- Cameras
- Gameplay Logic
- User Interface
- Etc.

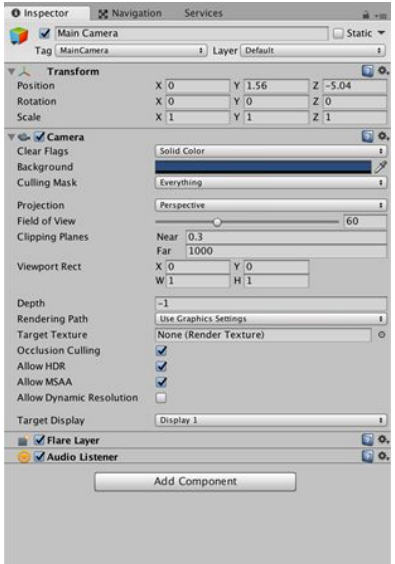
GameObject: http://docs.unity3d.com/ScriptReference/GameObject.html

Everything is a “GameObject”

- A *Game Object* does nothing on its own.
- *Game Objects* always have a *Transform* component which has a position/rotation/scale.
- Must add *Components* to the *Game Object* to give it some behavior.





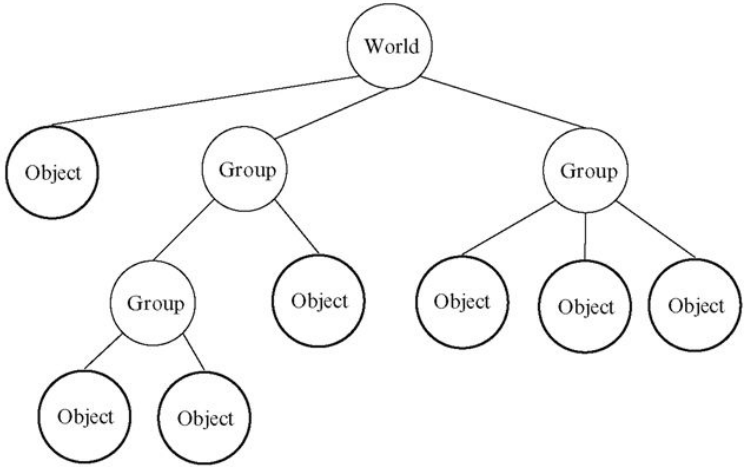


Scene graph

A scene graph is a collection of nodes in a graph or tree structure.

In Unity all tree nodes have only a single parent but may have many children.

Operations applied to a parent are applied to all its child nodes.



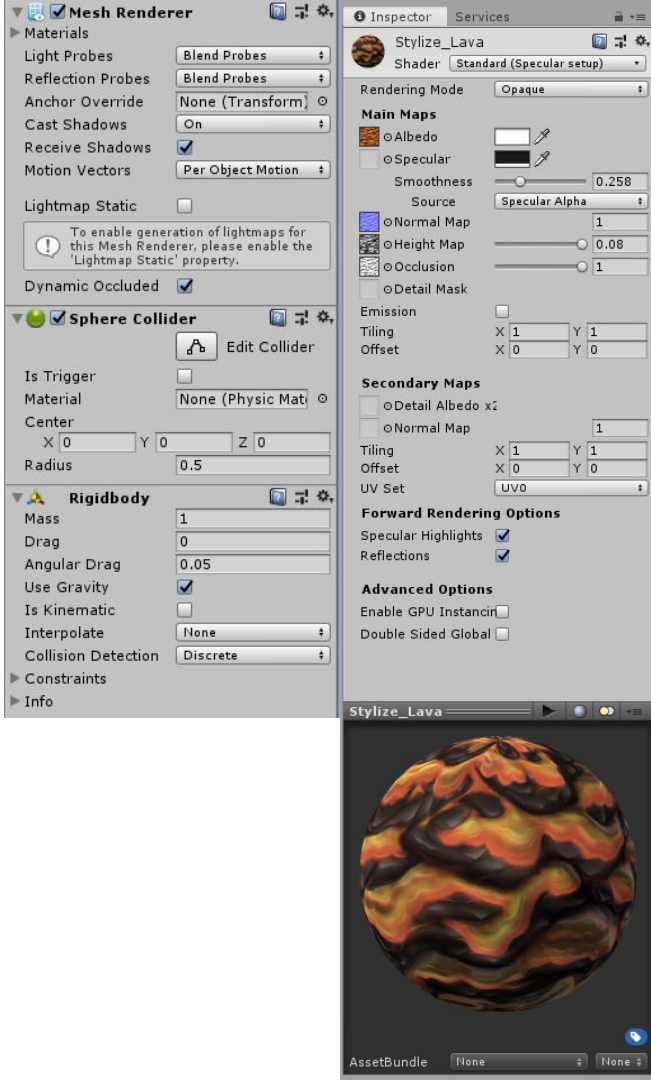
Components

Game Objects have *Components* to give it some behavior.

Many components already exist within Unity:

- Mesh Filter
- Mesh Renderer
- Rigidbody
- Collidiers
- VideoPlayer

But you will also need create your own. These are your scripts that inherit from *MonoBehaviour*.

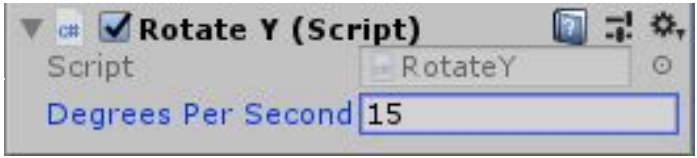


Scripts

- Many components already exist! But you will also need create your own. These are your scripts that inherit from *MonoBehaviour*.
- Public variables will show up in the Inspector. A variable that is a Component can also be modified by the inspector

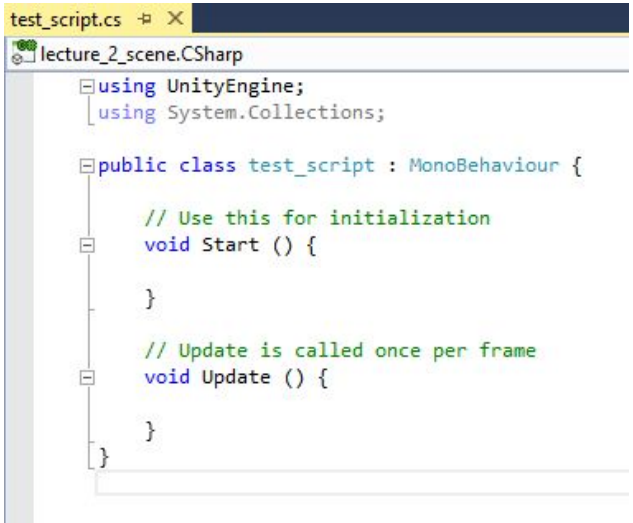
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RotateY : MonoBehaviour
{

    // Degrees to rotate around Y-Axis
    public float rotationRate = 5.0f; // This value is overwritten by any changes to the script in the inspector!

    // Update is called once per frame
    void Update()
    {
        // Define the axis of rotation
        Vector3 axis = new Vector3(0, 1, 0);

        // Calculate the amount to rotate the object this frame
        float amountToRotate = rotationRate * Time.deltaTime;

        // Access the transform component of this object and rotate
        this.transform.Rotate(axis, amountToRotate);
    }
}
```

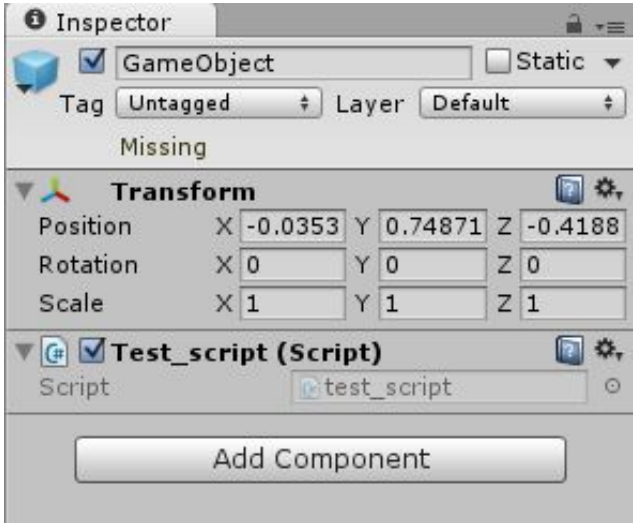
overwrites



Adding Components to Game Objects

- GameObject: http://docs.unity3d.com/ScriptReference/GameObject.html
- MonoBehaviour: http://docs.unity3d.com/ScriptReference/MonoBehaviour.html
- Drag and Drop Script onto the GameObject in the “Inspector” or manually add it by going to:
  - Add Component > Scripts > YOUR_SCRIPT_NAME_HERE

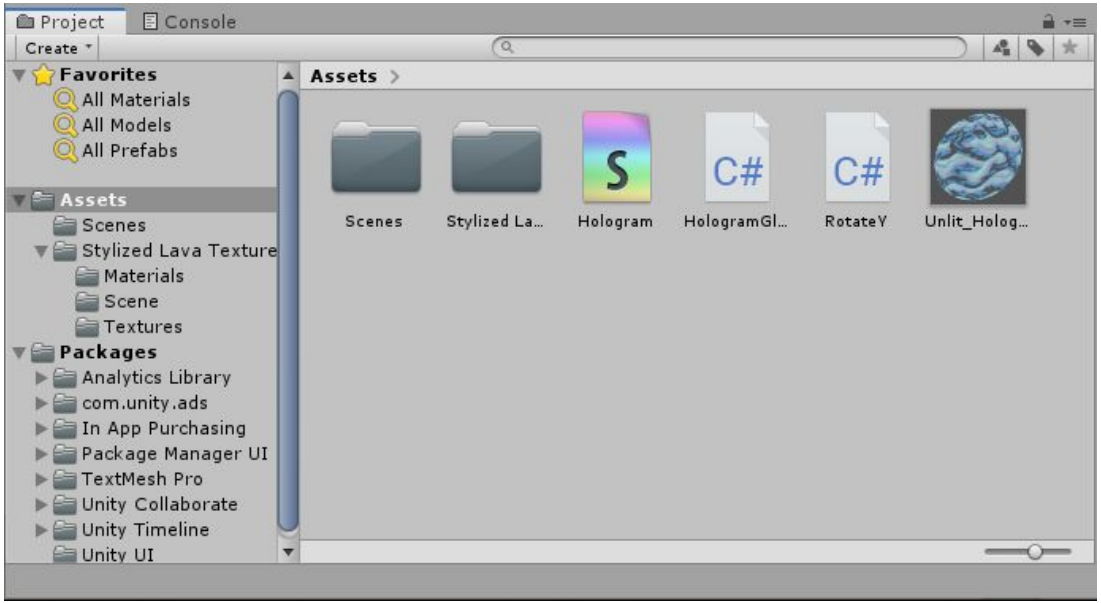




Assets

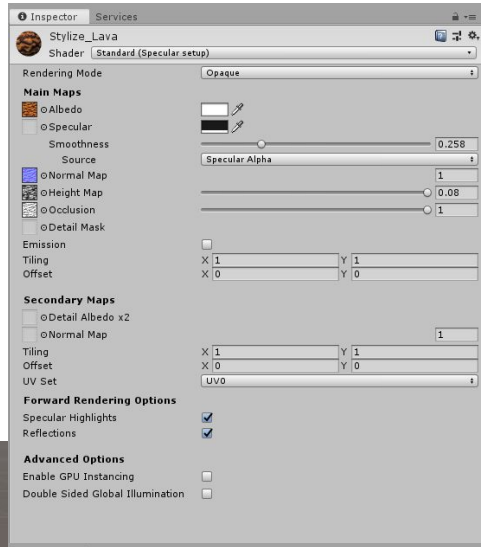
An asset is any resource that will be used as part of an object's component

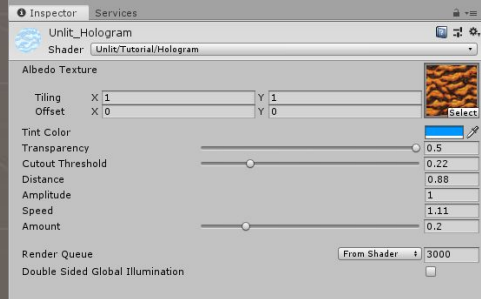
- Scenes
- "Prefabs"
- Scripts
- Textures
- Animations
- Models
- Particles
- Sprites
- Etc.

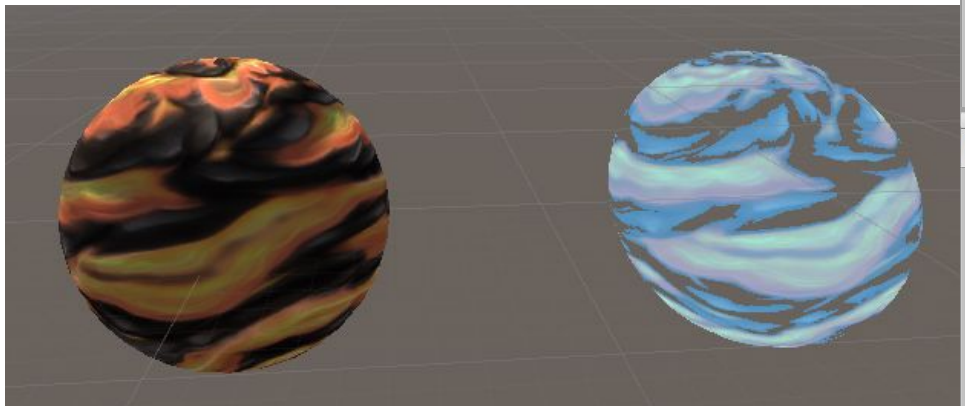


Shading and Materials

- Unity provides several built-in shaders
  - Unity Standard shader
  - Can also write your own shader
    - Shaders are written in Cg/HLSL and wrapped in ShaderLab
- Manual Shader Documentation: http://docs.unity3d.com/Manual/ShadersOverview.html
- Standard Shader Documentation: http://docs.unity3d.com/Manual/shader-StandardShader.html
- Materials Documentation: http://docs.unity3d.com/Manual/Materials.html

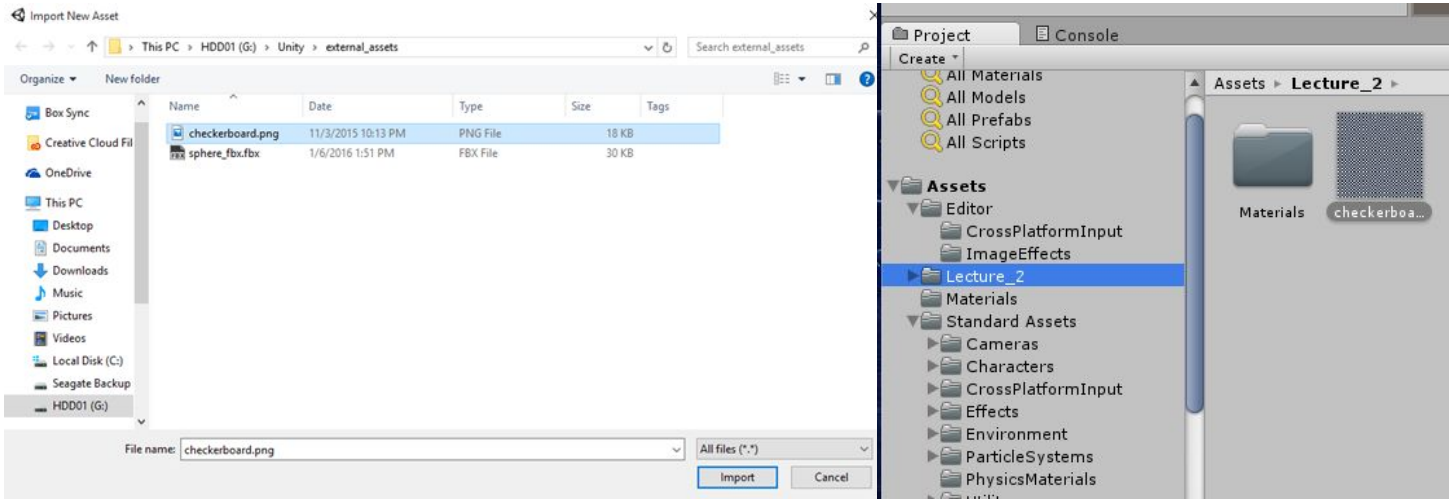






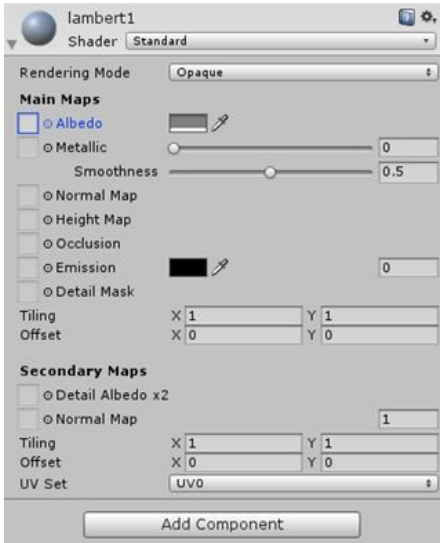
Importing Textures

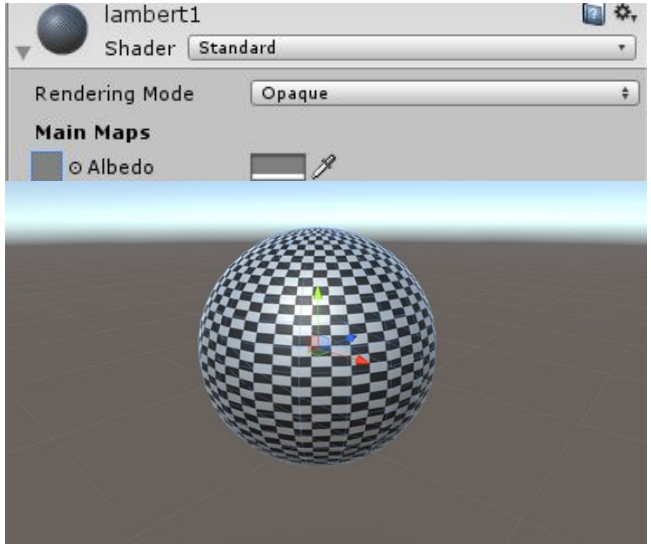
- Process is the same as importing an external object. This time, instead of selecting an *.FBX or *.OBJ, select a *.PNG, *.JPG, etc. You can also place the images inside the Assets folder manually.



Using Textures

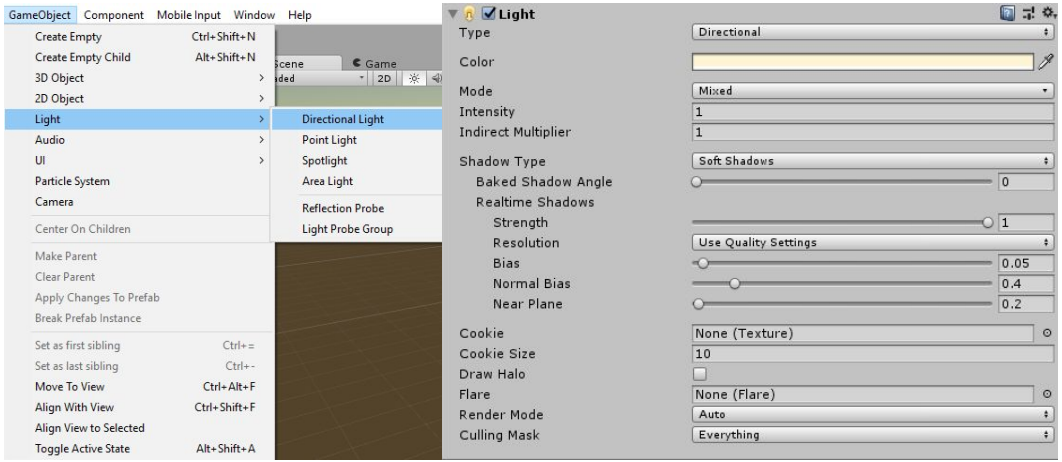
- Click on the object you imported in the scene hierarchy and expand the shader properties in the Inspector.
- Click and drag the imported texture onto the square next to “Albedo” and your object should now have a texture on it.





Lighting

- Lighting Documentation: http://docs.unity3d.com/Manual/Lighting.html
- Global Illumination Documentation: http://docs.unity3d.com/Manual/GlobalIllumination.html
- Lighting is accomplished with the “Light” component.
  - Directional
  - Point
  - Spot
  - Area (baked only)



Scripting in Unity

Scripting in Unity is done in C#

Scripts are an example of a component that is associated with a game object

The skeletal structure of a typical script called *MyGameObject* is shown below:

Script example

```
using UnityEngine;          // basic Unity-Engine objects
using System.Collections; // basic structures (ArrayList, HashTable,...)

public class MyGameObject : MonoBehaviour {
  void Start () {
    // ... initializations (like a constructor in Java)
  }
  void Update () {
    // ... insert code to be repeated every update cycle
  }
}
```

Fundamental Classes: MonoBehavior

When you create a script in Unity, Unity creates a class that extends MonoBehaviour.

Contains functions and events that are available to standard scripts attached to Game Objects

- Awake, Start, Update, FixedUpdate
- OnCollisionEnter, OnCollisionStay, OnCollisionExit
- GetComponent, SendMessage, BroadcastMessage
- Destroy, Instantiate

For a full list of methods and documentation, see:
https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Fundamental Classes: GameObject

GameObject: A generic type from which all game objects are derived. This corresponds to anything that could be placed in your scene hierarchy.

GameObjects have an associated *name* and *tag*. You can find other gameObjects with *Find, FindWithTag, FindGameObjectsWithTag,* etc.

Here is an example of how to obtain the main camera reference by its name:

```
GameObject camera = GameObject.Find ( "Main Camera");
```

Suppose that we assign the tag “Player” with the player object and “Enemy” with the various enemy objects. We could access the object(s) through the tag using the following commands:

```
GameObject player = GameObject.FindWithTag( "Player");
GameObject [] enemies = GameObject.FindGameObjectsWithTag( "Enemy");
```

Fundamental Classes: Transform

Transform: Every *game object* in Unity is associated with an object called its transform.

This object stores the position, rotation, and scale of the object. You can use the transform object to query the object's current position (*transform.position*) and rotation (*transform.eulerAngles*)

Vector3

Structure in Unity for representing 3D vectors and points.

This structure is used throughout Unity to pass 3D positions and directions around. It also contains functions for doing common vector operations.

Other classes can be used to manipulate vectors and points as well. For example the Quaternion and the Matrix4x4 classes are useful for rotating or transforming vectors and points.

Common methods: Cross, Dot, Normalize, Lerp, Reflect, Distance

For more information, see the documentation:
https://docs.unity3d.com/ScriptReference/Vector3.html

Quaternion

Quaternions are used internally by Unity to represent rotations.

There are some advantages to using quaternions over euler angles (gimbal lock, can be interpolated easily, etc)

Have x,y,z,w components and are non-commutative. Likely will never need to modify these components individually.

Instead use these to create/manipulate Quaternions: Quaternion.LookRotation, Quaternion.Angle, Quaternion.Euler, Quaternion.Slerp, Quaternion.FromToRotation, Quaternion.identity

For more information, see: https://docs.unity3d.com/ScriptReference/Quaternion.html

Matrix4x4

Structure for a 4x4 transformation matrix

Can perform translation, rotation, scale, shear, and perspective transformations using homogeneous transformations.

Column major: for the expression ***mat[a, b]***, ***a*** refers to the row index, while ***b*** refers to the column index

In Unity, Matrix4x4 is used by several Transform, Camera, Material and GL functions.

Common methods/properties: determinant, inverse, transpose, LookAt, Ortho, Perspective, Rotate, Scale, Translate, TRS

For more information, see: https://docs.unity3d.com/ScriptReference/Matrix4x4.html

Accessing Components:

It is often desirable to modify the values of components at run time.

Unity defines class types for each of the possible components, and you can access and modify this information from within a script.

To access public variables/methods from a component, use *GetComponent*.

Example:

```
// Get rigidbody component of this game object
Rigidbody rb = GetComponent <Rigidbody>();

// change this body's mass
rb.mass = 10f;
```

Accessing Members of Other Scripts

Often, game objects need to access members variables in other game objects.

Can use *GetComponent* to access public variables/methods in other scripts.

```
public class PlayerController : MonoBehaviour {
     public void DecreaseHealth () { ... }  // decrease player 's health
}

public class EnemyController : MonoBehaviour {
     public GameObject player;  // the player object
     void Start () {
          GameObject player = GameObject.Find( "Player");
     }
     void Attack () { // inflict health loss on player
          player.GetComponent<PlayerController>().DecreaseHealth();
     }
}
```

Colliders and Triggers:

Some events are generated by the user (e.g., input), some occur at regular time intervals (e.g.,Update()), and finally others are generated within the game itself.

Typically, colliders are physical objects that should not overlap, whereas triggers are invisible barriers that send a signal when crossed.

There are various event functions for detecting when an object enters, stays within, or exits,collider/trigger region. These include, for example:

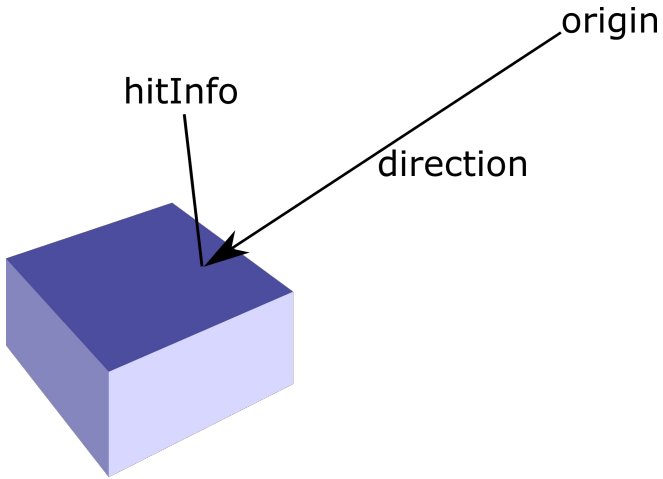
- For colliders: void OnCollisionEnter(),void OnCollisionStay(),void OnCollisionExit()
- For triggers: void OnTriggerEnter(),void OnTriggerStay(),void OnTriggerExit()

Example: Rotate script

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

public class RotateYFinal : MonoBehaviour
{
	// Degrees to rotate around Y-Axis per second
	public float rotationRate = 5.0f;

	// Update is called once per frame
	void Update()
	{
		// Define the axis of rotation
		Vector3 axis = new Vector3(0, 1, 0);
		// Equivalently you could use Vector3.up

		// Calculate the amount to rotate the object this frame
		float amountToRotate = rotationRate * Time.deltaTime;

		// Access the transform component of this object and rotate
		this.transform.Rotate(axis, amountToRotate);
	}
}
```

Raycasting:

```
public static bool Raycast(
     Vector3 origin,
     Vector3 direction,
     out RaycastHit hitInfo,
     float maxDistance,
     int layerMask,
     QueryTriggerInteraction queryTriggerInteraction
);
```



Casts a ray, from point *origin*, in direction *direction*, of length *maxDistance*, against all colliders in the Scene.

You may optionally provide a LayerMask, to filter out any Colliders you aren't interested in generating collisions with.

Documentation: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

Oculus Utilities for Unity

OVRCameraRig is a Component that controls stereo rendering and head tracking. It maintains three child “anchor” Transforms at the poses of the left and right eyes, as well as a virtual “center” eye that is halfway between them.

This Component is the main interface between Unity and the cameras. It is attached to a prefab that makes it easy to add comfortable VR support to a scene.

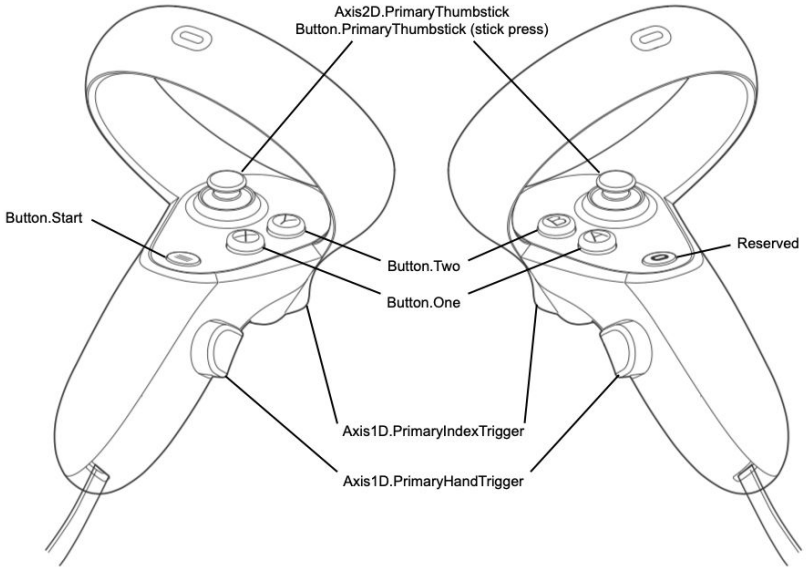
Public Members:

1. Updated Anchors - Allows clients to filter the poses set by tracking. Used to modify or ignore positional tracking.

Game Object Structure:

1. TrackingSpace - A Game Object that defines the reference frame used by tracking. You can move this relative to the OVRCameraRig for use cases in which the rig needs to respond to tracker input. For example, OVRPlayerController changes the position and rotation of TrackingSpace to make the character controller follow the yaw of the current head pose.

OVRInput


Axis2D.PrimaryThumbstick
Button.PrimaryThumbstick (stick press)
Button.Start
Button.Two
Button.One
Reserved
Axis1D.PrimaryIndexTrigger
Axis1D.PrimaryHandTrigger