Unity Bootcamp

by

Yohan Hmaiti

Spring 2026

- 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 (LTS Versions) are available through UCF laptops

For your own computer – ensure you select Personal License when prompted:

- Download and install the Unity Hub: https://unity.com/download
  - Sign in or create an account
  - Go to installs
  - Click install editor and then chose the adequate version
- Download any Unity LTS Version – Unity 6 recommended: https://unity.com/releases/unity-6
  - 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 XR SDKs For Unity **[Part 2 will cover this]**: https://developers.meta.com/horizon/documentation/unity/unity-sdks-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

Basic Concepts

- Project - The project contains all elements that make up the application, including assets, scripts, scenes, settings, and packages.

Think of this as the container for everything you build and ship.

- Scenes - A scene is a collection of GameObjects that define a world, state, or context at runtime.

Scenes organize space and logic.

- Packages - Packages are modular units of

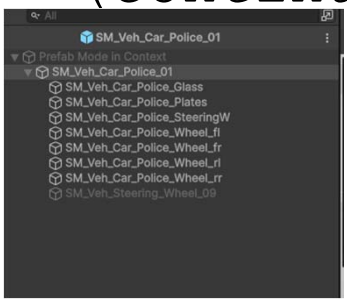
Unity Basic Concepts (continued)

Prefabs: A prefab is a reusable template that defines a **GameObject and its component hierarchy**. Prefabs are used for creating multiple instances of a common object.

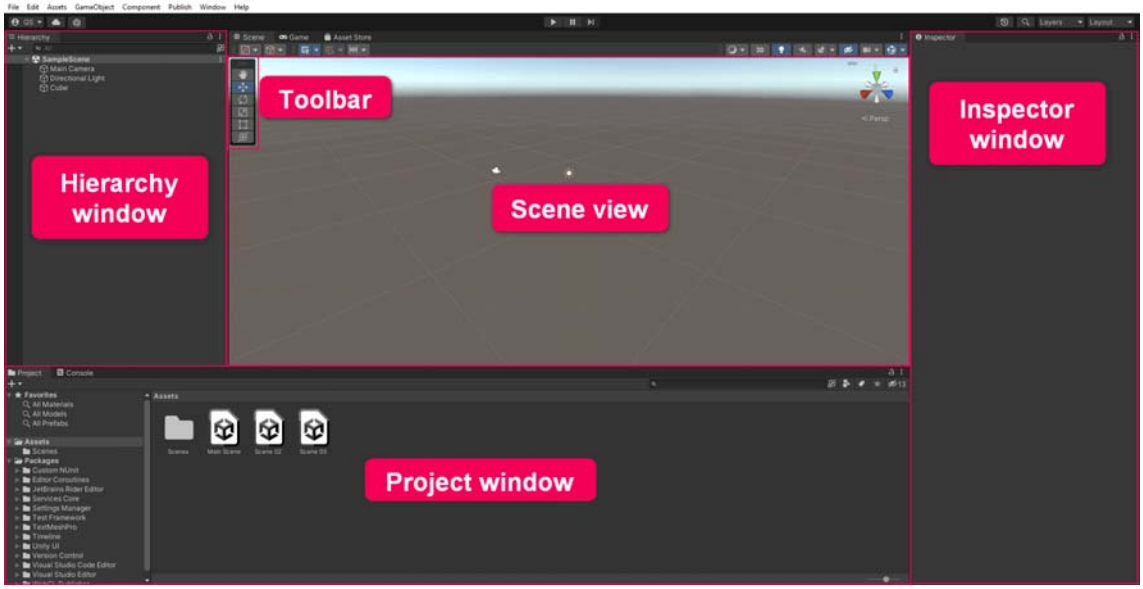
- Used to create multiple instances of a common object (e.g., street lights, trees)

Prefabs can be instantiated at runtime

Unity Basic Concepts (continued)



Overview of the Unity IDE (Tools):



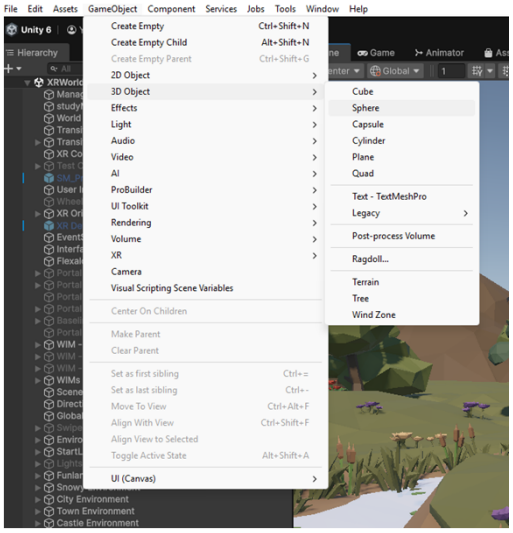
EDITOR CAMERA CONTROLS & SHORTCUTS

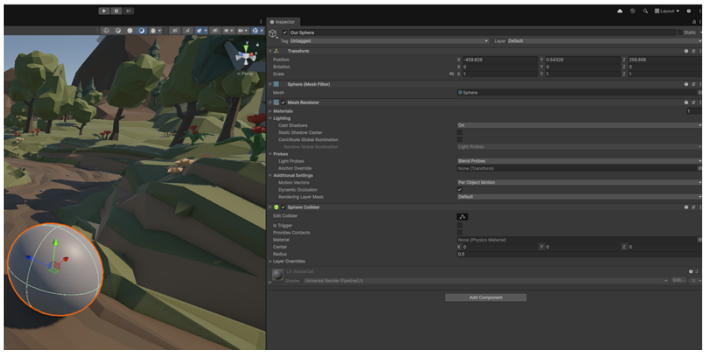
- Controls:
  - Alt + Left Click & Move: Rotate Camera **[or just right click and rotate]**
  - Alt + Right Click & Move (Or Scroll Up/Down): Zoom in and out **[or just right click and WASD]**
  - **Middle Click & Move:** Move camera up/down or left right
- Flythrough Mode:
  - Click and hold right mouse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

Creating Geometry via the Unity Editor

- You can create basic 3D geometry directly in the Unity Editor:
- Go to **GameObject then 3D Object**
- Built-in primitives include: Cube – Sphere – Capsule – Cylinder – Plane -- etc
- These primitives are standard **GameObjects** with:
  - Mesh Filter
  - Mesh Renderer
  - Collider
- Useful for:
  - Prototyping
  - Blockouts
  - Testing interaction and scale

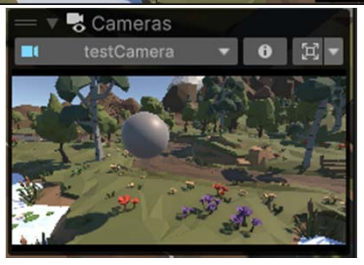
Creating Geometry via the Unity Editor



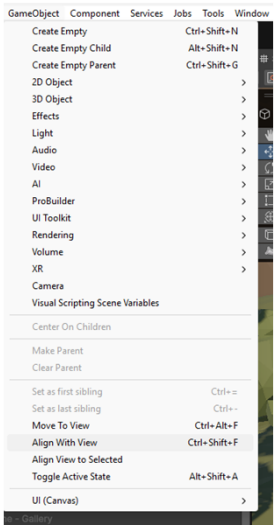


Setting Up The Scene Camera

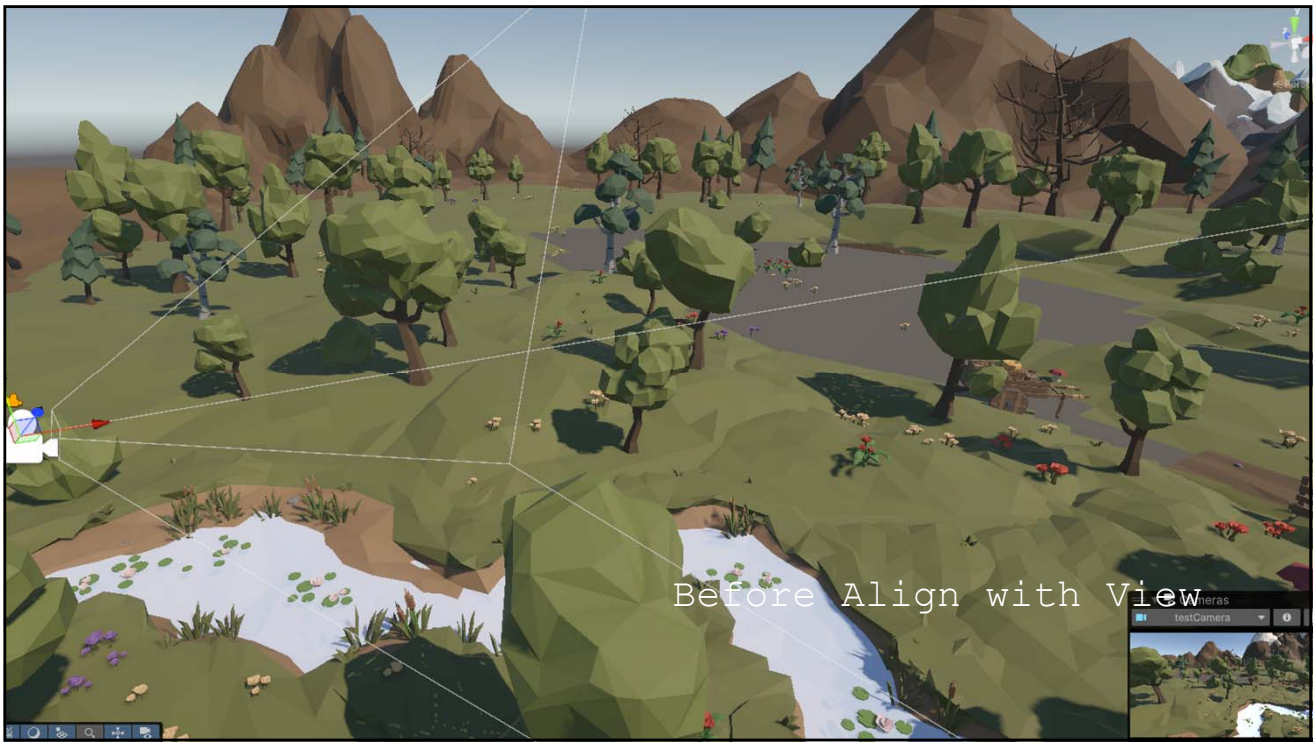
- Do not confuse the scene camera with the editor camera.
- Unity scenes by default come with a “Main Camera.” Notice the tag of “MainCamera” in the inspector, this will be useful for accessing the camera from your scripts.
- “Camera Preview” box is useful to see what your camera can see.
- “Camera Preview”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

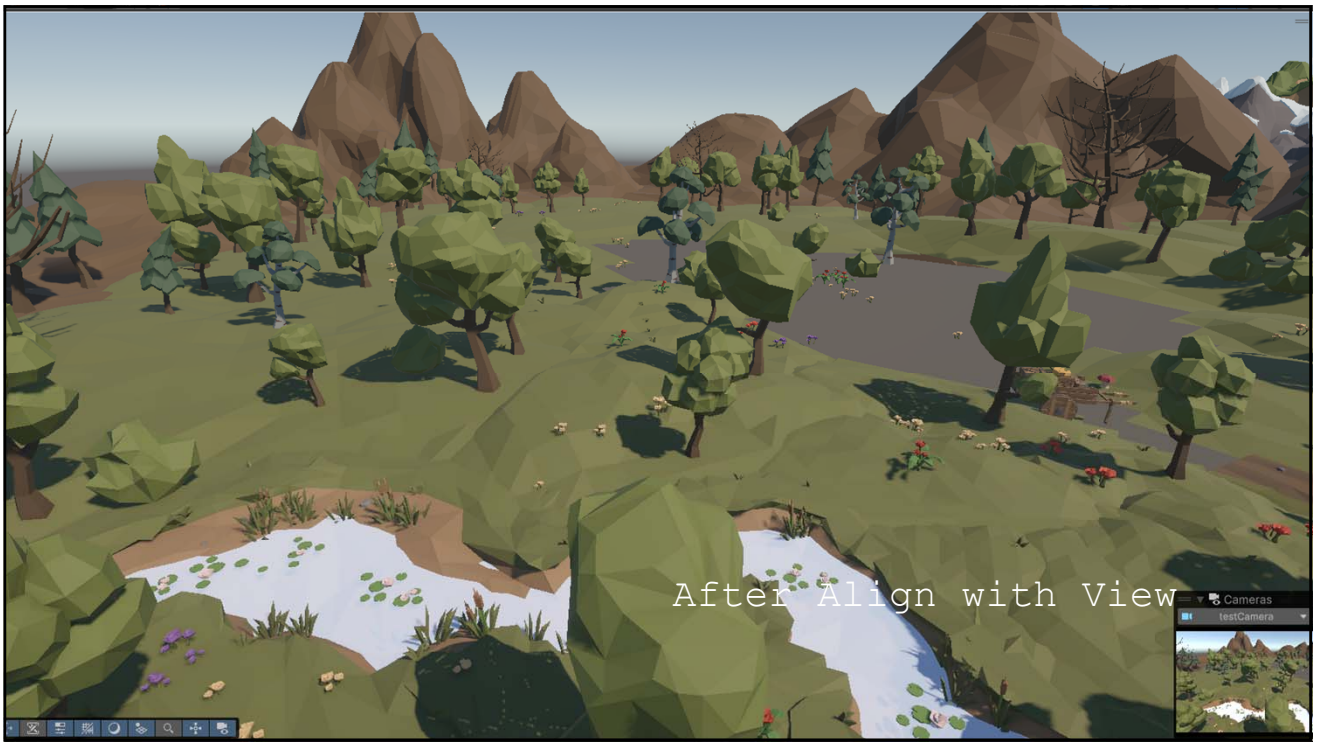


Setting Up The Scene Camera



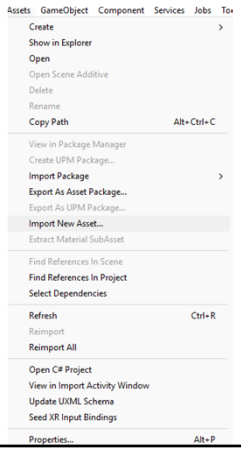
- Moving the scene camera can be done manually by changing the position/rotation/scale in the Inspector.
- Or you can move the editor camera around as mentioned earlier, select the camera and align the camera to the view. Note that your camera should be selected before doing this.

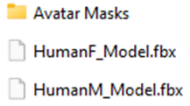




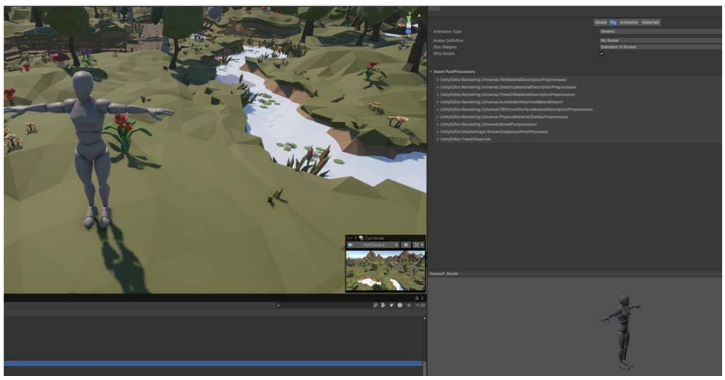
Import External Objects

- Create and export an object from Maya/Blender/3ds Max as either an *.OBJ or a *.FBX. You can save this anywhere.
  - Or download from a website an asset that exists already
- 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.





ie assets library into your scene hierarchy and it

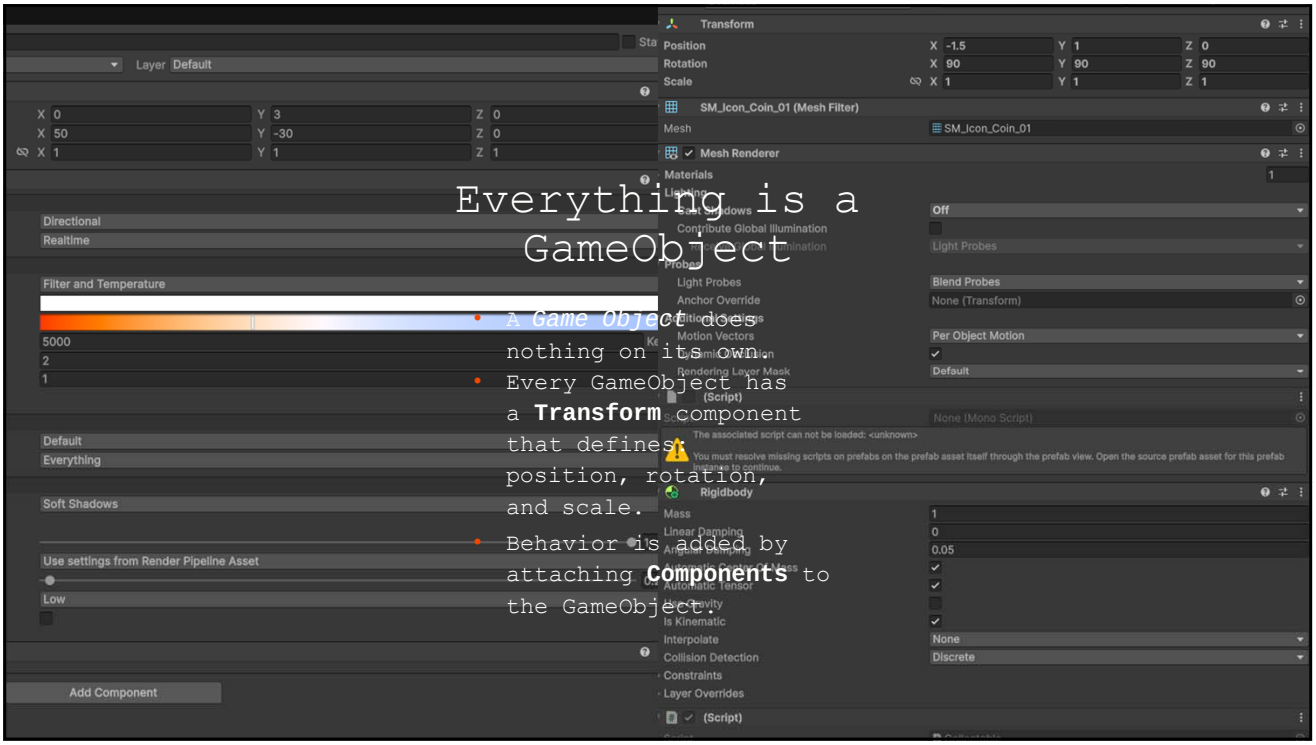


Game Objects

GameObjects are the fundamental building blocks of a Unity scene.
They represent all "things" that exist in the scene.

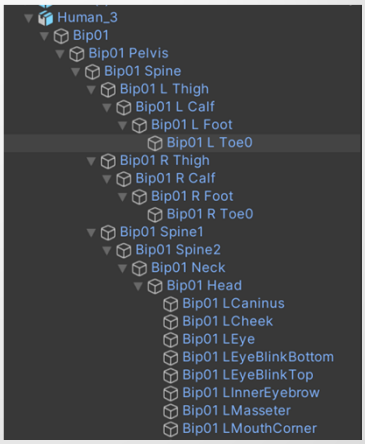
- Audio sources
- Cameras
- Gameplay Logic
- User Interface
- Light sources
- And so on

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



Scene Hierarchy

- Parent game objects hold child game objects



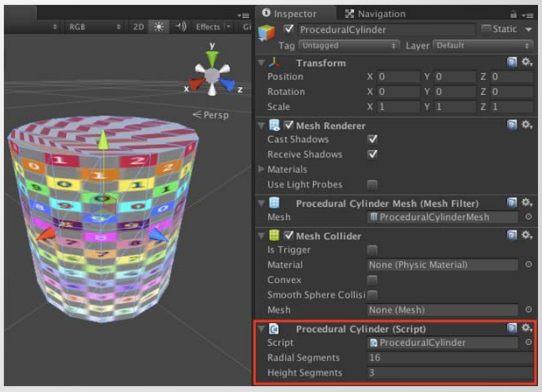
Components

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

- Many components already exist within Unity:
  - Mesh Filter
  - Mesh Renderer
  - Rigidbody
  - Collidiers
  - VideoPlayer

- You can also create your own components.

Custom components are **scripts** that inherit from **MonoBehaviour**.



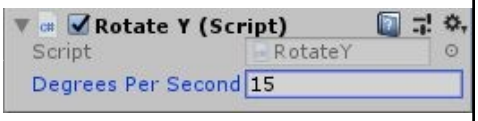
Scripts

- Public variables and serialized fields can be modified by the inspector

- The MonoBehaviour class is defined by Unity to help you transform your class into a component for game objects

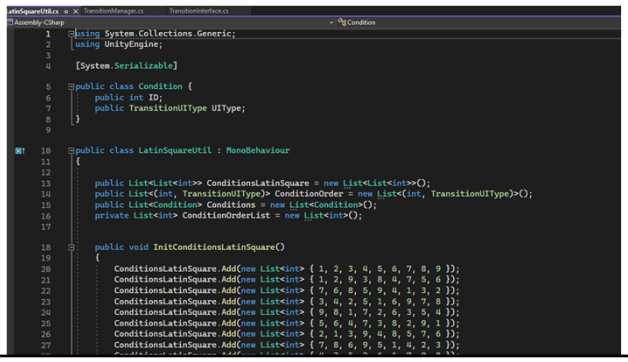
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

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

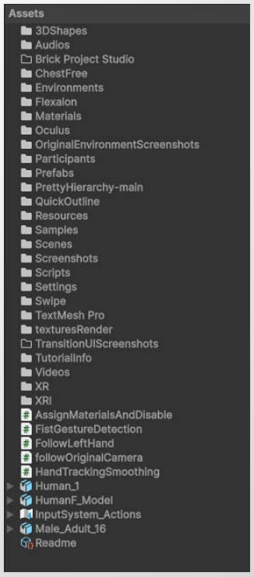
- GameObject: http://docs.unity3d.com/ScriptReference/GameObject.html
- MonoBehaviour: http://docs.unity3d.com/ScriptReference/MonoBehaviour.html
- Drag and Drop Script onto the GameObject in the “Inspector” or manually add it by going to:
  - Add Component > Scripts > YOUR_SCRIPT_NAME_HERE



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

[System.Serializable]

public class Condition {
    public int ID;
    public TransitionUIType UIType;
}


public class LatinSquareUtil : MonoBehaviour
{

    public List<List<int>> ConditionsLatinSquare = new List<List<int>>();
    public List<(int, TransitionUIType)> ConditionOrder = new List<(int, TransitionUIType)>();
    public List<Condition> Conditions = new List<Condition>();
    private List<int> ConditionOrderList = new List<int>();


    public void InitConditionsLatinSquare()
    {
        ConditionsLatinSquare.Add(new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 });
        ConditionsLatinSquare.Add(new List<int> { 1, 2, 9, 3, 8, 4, 7, 5, 6 });
        ConditionsLatinSquare.Add(new List<int> { 7, 6, 8, 5, 9, 4, 1, 3, 2 });
        ConditionsLatinSquare.Add(new List<int> { 3, 4, 2, 5, 1, 6, 9, 7, 8 });
        ConditionsLatinSquare.Add(new List<int> { 9, 8, 1, 7, 2, 6, 3, 5, 4 });
        ConditionsLatinSquare.Add(new List<int> { 5, 6, 4, 7, 3, 8, 2, 9, 1 });
        ConditionsLatinSquare.Add(new List<int> { 2, 1, 3, 9, 4, 8, 5, 7, 6 });
        ConditionsLatinSquare.Add(new List<int> { 7, 8, 6, 9, 5, 1, 4, 2, 3 });
```

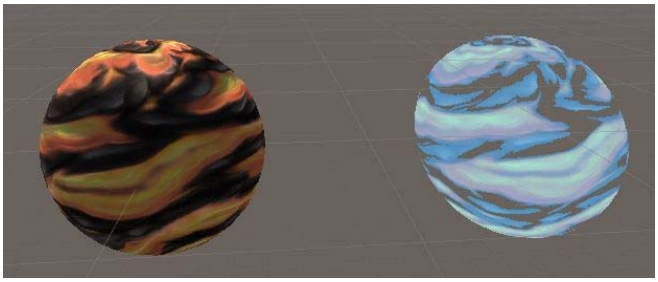
Assets

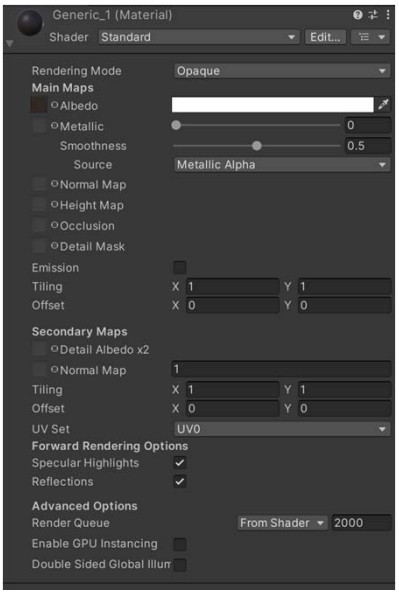
- An asset is any resource that will be
- used as part of an object's component
  - "Prefabs"
  - Scripts
  - Textures
  - Animations
  - Models
  - Particles
  - Sprites
  - Etc.

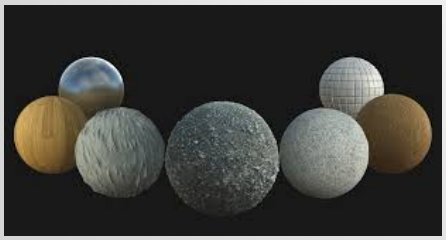


Shading and Materials

- 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
- Standard Shader Documentation: http://docs.unity3d.com/Manual/shader-StandardShader.html
- Materials Documentation: http://docs.unity3d.com/Manual/Materials.html







Importing Textures

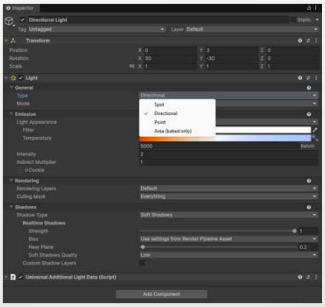
- Process is the same as importing an external object. This time, instead of selecting an *.FBX or *.OBJ, select a *.PNG, *.JPG, etc. You can also place the images inside the Assets folder manually.

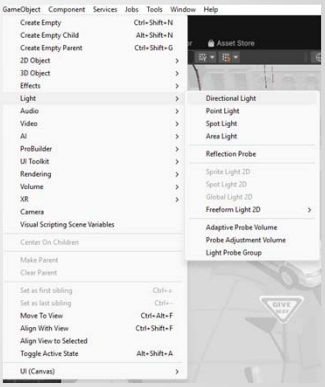
Using Textures

- 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 is shown below:

```
using UnityEngine;

0 references
public class Instantiate_example : MonoBehaviour
{
    //Start is called before the first frame Update
    0 references
    void Start()
    {

    }


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

    }
}
```

Fundamental Classes: MonoBehaviour

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

The MonoBehaviour class is defined by Unity to help you transform your class into a component for game objects

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

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

Script LifeCYCLE

All components that derive from the MonoBehaviour class executes a number of event functions in a predetermined order. They're the following:

- **Awake()** - This function is called after a game object has been instantiated.
- **OnEnable()** - This function is called when a game object is enabled.
- **Start()** - This function is called before the first frame update.
- **Update()** - This function is called on every frame.
- **LateUpdate()** - This function is called on every frame **after** the Update() function is called.
- **OnBecameVisible()** - This function is called when the current game object becomes visible via any camera.
- **OnBecameInvisible()** - This function is called when the current game object no longer becomes visible via any camera.
- **OnDrawGizmos()** - This function is called for drawing gizmos in the **scene** window.
- **OnGUI()** - This function is called multiple times for GUI events.
- **OnApplicationPause()** - This function is called when the game is paused via the Unity editor.
- **OnDisable()** - This function is called when the game object is disabled.
- **OnDestroy()** - This function is called when the game object is destroyed.

There's a lifecycle function called FixedUpdate, which is called a fixed number of times

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");`
- `GameObject player = GameObject.FindWithTag( "Rohith");`
- `GameObject [] enemies = GameObject.FindGameObjectsWithTag( "Professor");`

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*), scale (***transform.scale***), and rotation (*transform.eulerAngles*)

Vector3

Vector3 is Unity's structure for representing **3D positions and directions.**

It is used through Unity for Positions, Directions, Movements, Forces, etc

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

For more information, see the documentation:
https://docs.unity3d.com/ScriptReference/Vector3.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 (Physics Events):

- Events in Unity can be:
  - User-driven (e.g., input)
  - Time-based (e.g., Update())
  - Physics-based (collisions and triggers)

- **Colliders** represent physical objects that should not overlap.
- **Triggers** are non-physical volumes that detect when objects pass through them.
- Unity provides event functions to detect when an object enters, stays in, or exits a collider or trigger:

- **Triggers**
  - OnTriggerEnter()
  - OnTriggerStay()
  - OnTriggerExit()
- **Colliders**
  - OnCollisionEnter()
  - OnCollisionStay()
  - OnCollisionExit()

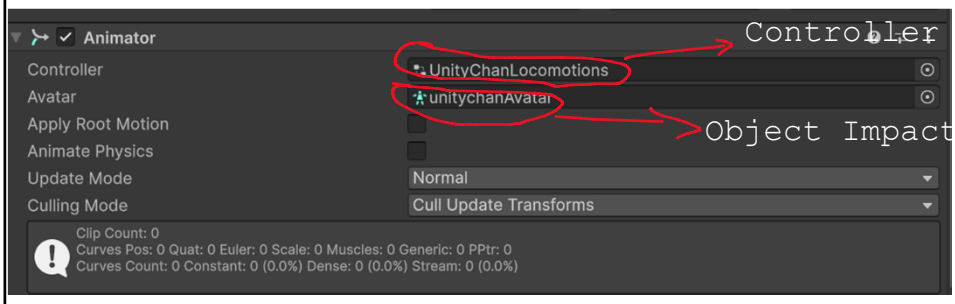
RIGIDBODIES

- A **Rigidbody** is a component that enables **physics simulation** on a GameObject.
- When attached, Unity's physics engine controls how the object:
  - Moves
  - Falls due to gravity
  - Collides with other objects
  - Responds to forces
- **Physics Behavior**
  - **Gravity:** objects fall naturally
  - **Forces:** objects can be pushed or pulled
  - **Collisions:** objects collide and react realistically
  - **Constraints:** limit movement or rotation on specific axes
- **Common Properties**
  - **Mass:** how heavy the object is
  - **Drag:** resistance to linear motion
  - **Angular Drag:** resistance to rotation
  - **Constraints:** lock position or rotation axes
- **Scripting with Rigidbodies**
  - Rigidbodies can be controlled via scripts
  - Common methods:
    - AddForce()
    - velocity
  - Used for characters, projectiles, vehicles, and physical interactions
- **My Recommendation To You [Saves Time]:**
  Use **physics forces** (not transform.Translate) when moving Rigidbody objects.

Animations Controller

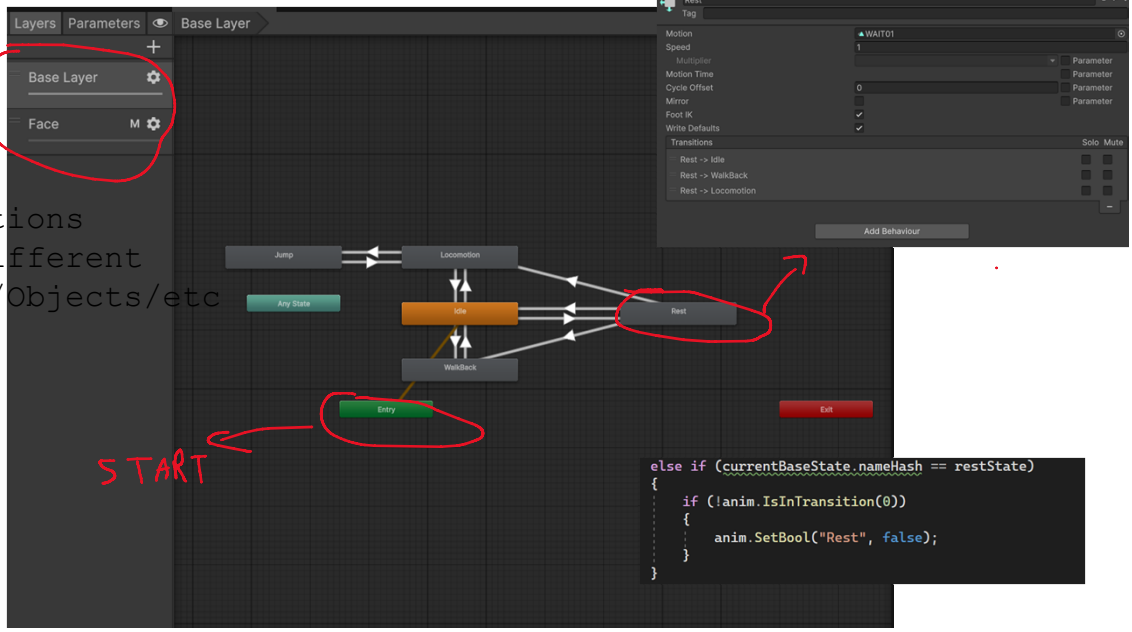
- Unity uses the **Animator Controller** to manage **animations and transitions** for a GameObject.
- An Animator Controller:
  - References one or more **Animation Clips**
  - Defines **transitions** between animations
  - Uses a **state machine** (e.g., Idle → Walk → Jump)
- **Important recommended practice for this course:**
  - We **do not record animations directly in Unity**.
  - Instead:
    - Use **pre-made animations** (Asset Store / online libraries), or
    - Create animations in **Blender** (or similar tools)
    - Import them into Unity and control them via an **Animator Controller**
  - The Animator Controller is responsible for **when** animations play — not **how** they are created.

Animations Controller (2)



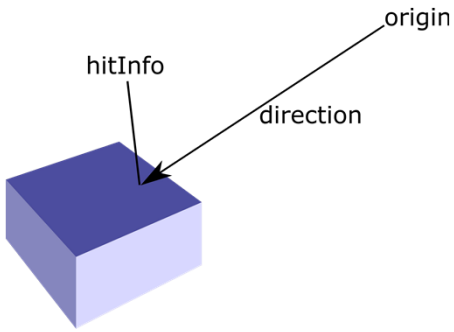
Animations Controller (3)

Animations For Different Parts/Objects/etc



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

What is NEXT

Feedback?

Version Control – A problem and why?

Demo – Simple Unity Starter Project and Code Can Be Found Here:
https://github.com/YHmaiti/UnityBootcamp

Thursday -> Second Part Includes Demo and Tutorial For VR Development
+ additional handy tools like Profilers

Acknowledgment

-Purdue University Tutorials
-Unity Learn Tutorials

For additional resources:
https://zerotomastery.io/cheatsheets/unity-cheat-sheet/