03 - Game Assets, Scripts, and Debugging

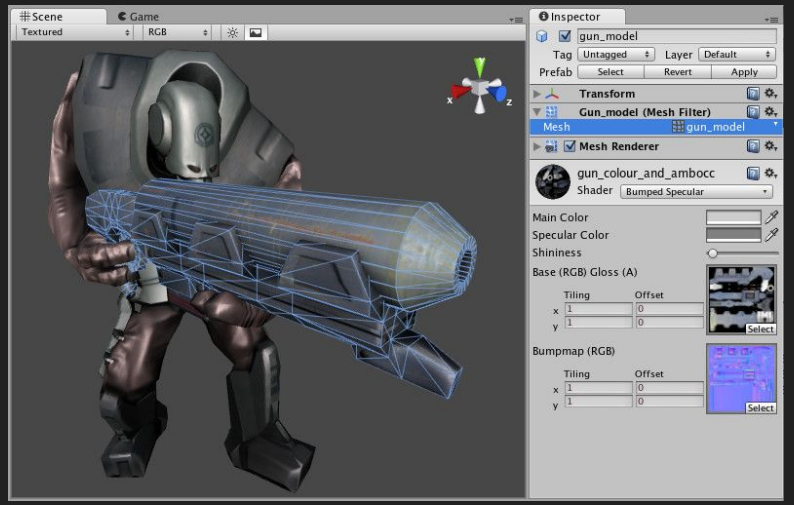
CS 3160 - Game Programming
Max Gilson

Game Assets

- Game assets are all the content and resources used to create a video game's world, characters, interface, and experience:
  - Models and meshes
  - Animations
  - Textures and sprites
  - Shaders
  - Audio files
  - Levels and scenes
  - Libraries
  - Many others, i.e. fonts

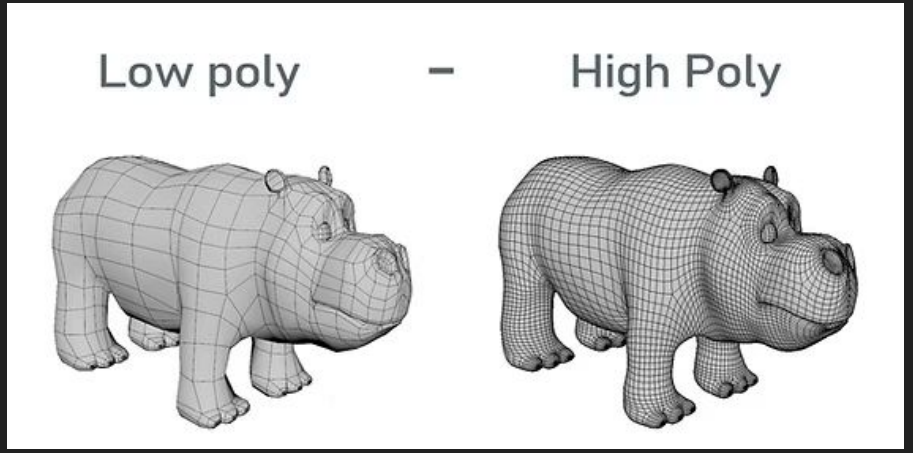
Models and Meshes

- In 3D games there are 3D models to represent objects in the world
- The underlying geometry that represents these 3D models are called meshes
  - Typically made up of tiny triangles referred to as “polygons” or “polys”



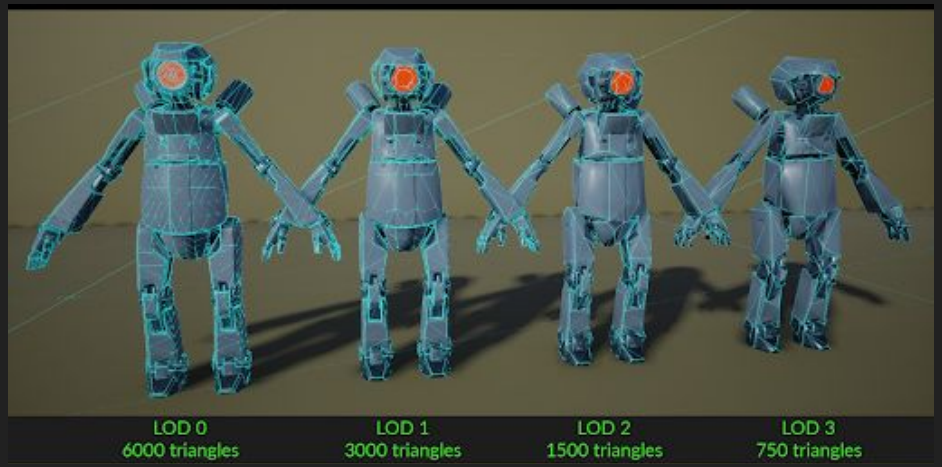
Poly Count of Meshes

- The underlying mesh has a significant impact on performance and appearance
  - Low poly models look worse but perform better
  - High poly models look better but perform worse
- More polys means:
  - Slower rendering on GPU
  - More memory consumption
  - Longer load times
  - Animations are more computationally intensive
  - Games run slower in general



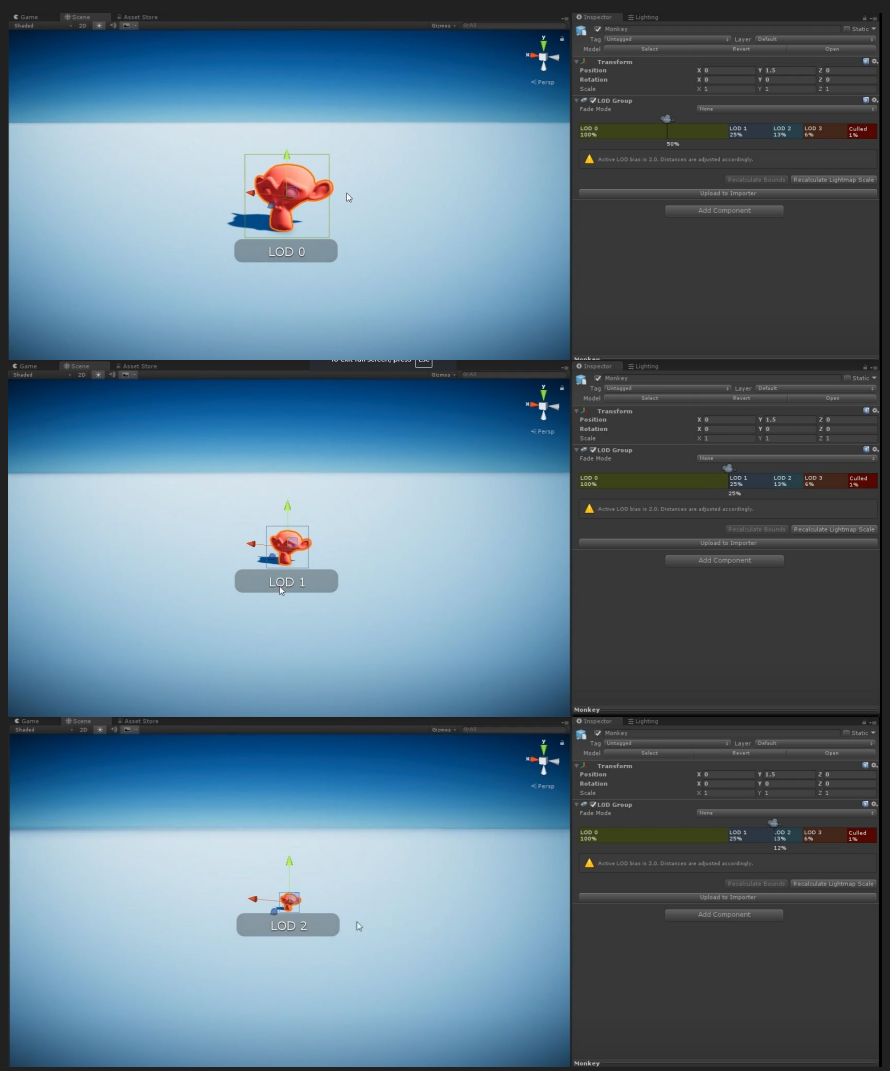
Levels of Detail

- Levels of Detail (LODs) are lower poly versions of the same model
- A game engine can transition between LODs while the game is running to improve performance
    - If this transition is noticed by the player it can be visually distracting
- If a character can only be seen at a distance, there is no reason to render thousands of triangles
- Some game engines can generate dynamic LODs on the fly (Nanite in Unreal Engine)



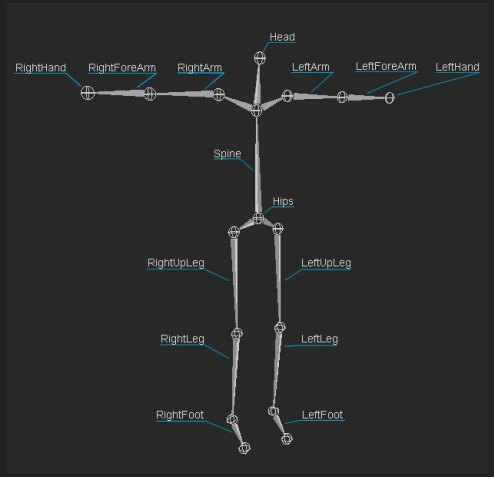
Meshes and LODs in Unity

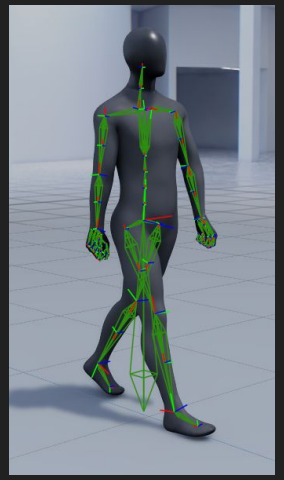
- In Unity, you can add meshes using .fbx , .dae, .dxf, or .obj files
- Drag them into your Project Window, then into your Scene
- Meshes can also have LODs in Unity which requires the LOD Group Component
    - Unity will automatically switch betweens LODs when the GameObject is far enough away from the camera

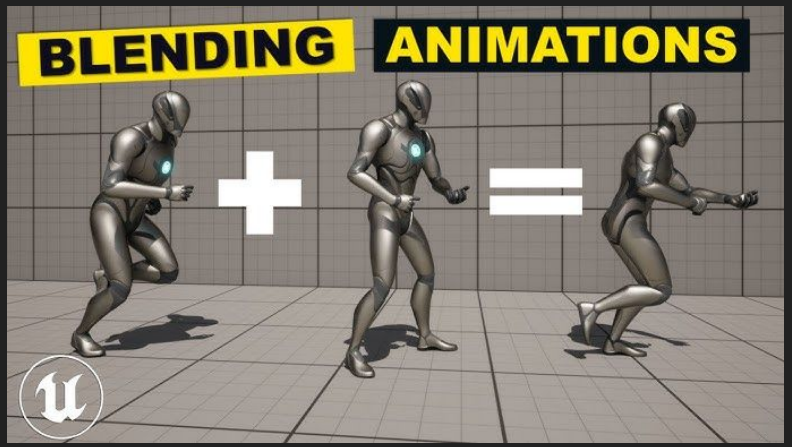


Animations

- Animations allow meshes to change shape while the game is running
  - This allows characters meshes to become animated
- Animations contain various types of data:
  - Skeletal Hierarchy (bones)
  - Keyframe Animation Data
    - Bone position, rotation, and scale over time at a specific framerate
- Animations can also be blending together if the engine allows for it

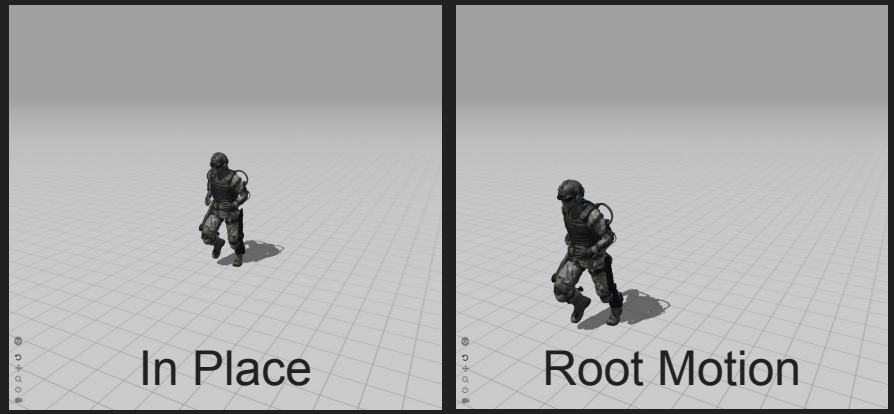






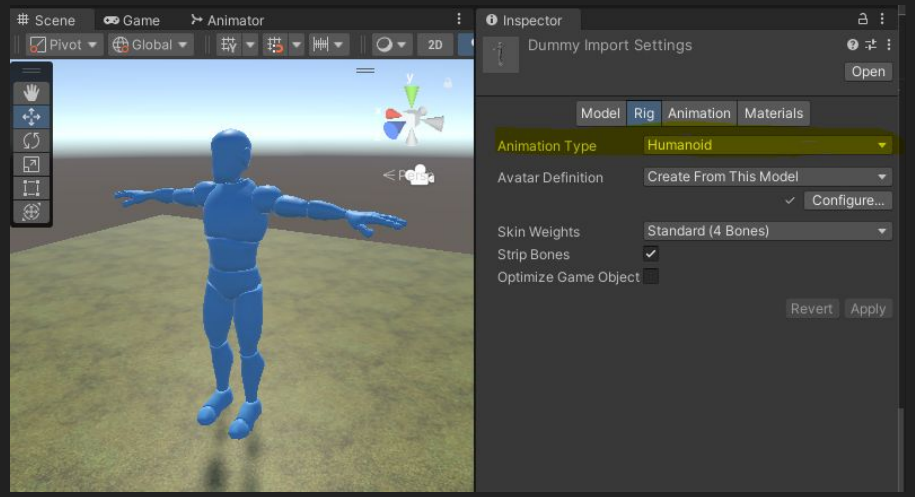
Root Motion Animations

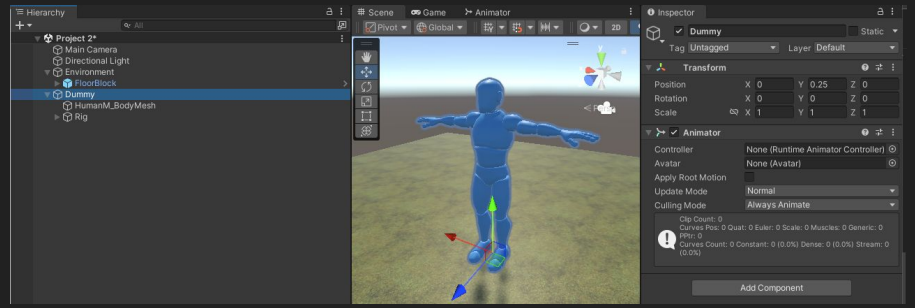
- Animations are either in-place or use root motion
- Root motion allows the animation itself to move the animated character or object
- This makes movement more realistic
  - Also useful for games that require advanced movement and parkour: https://www.youtube.com/watch?v=--sCtgAgIlg



Animations in Unity

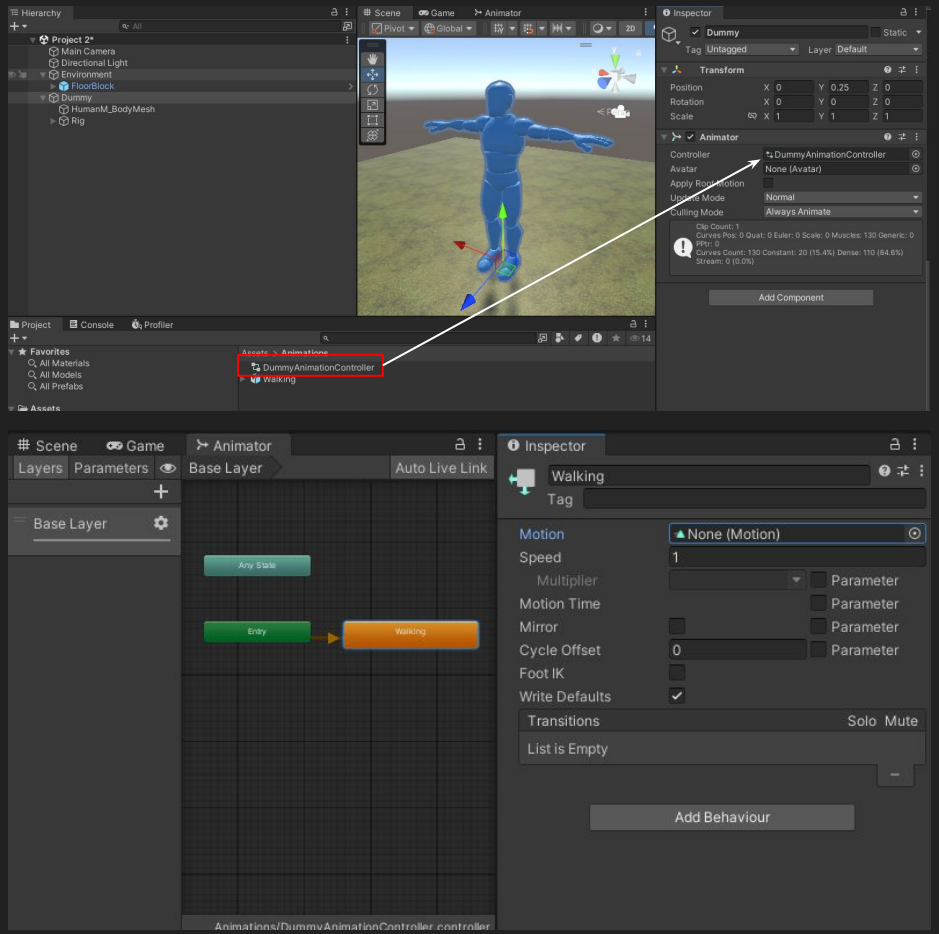
- To add an animation in Unity, first you need a mesh that supports the animation:
  - Set the model's type as "Humanoid"
  - Drag the model into the Hierarchy and add an Animator Component





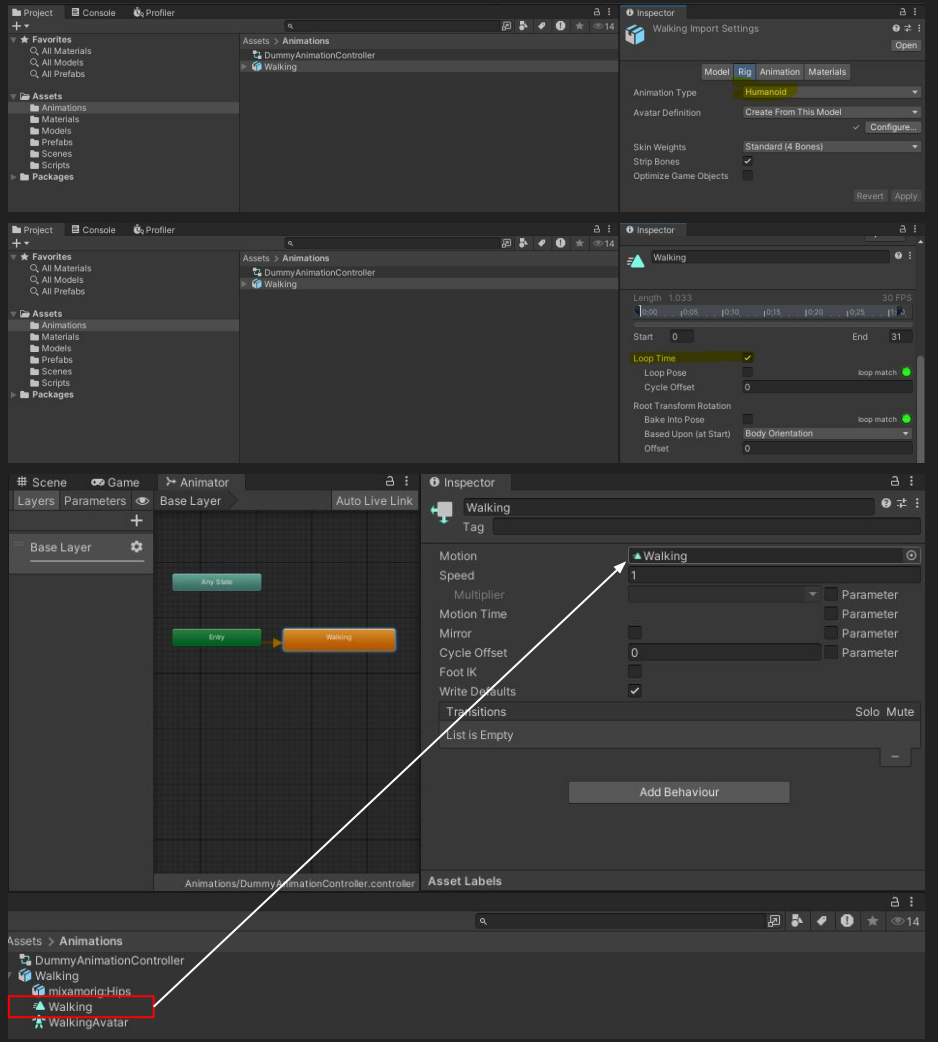
Animations in Unity (cont.)

- Create a new Animator Controller in the Project Window and add it to the Animator Component
- Select the Avatar for this character
- Double click on the Animator Controller and create a new empty state
    - Every additional animation will require a state and some transition to that state



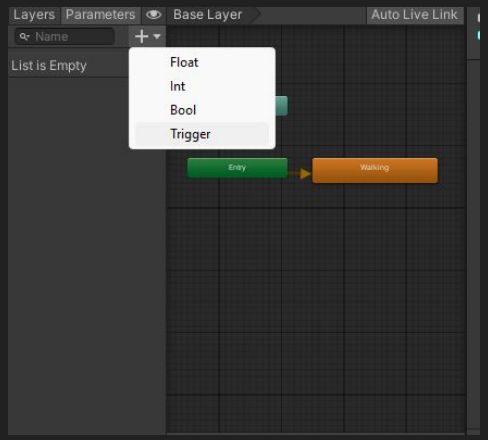
Animations in Unity (cont.)

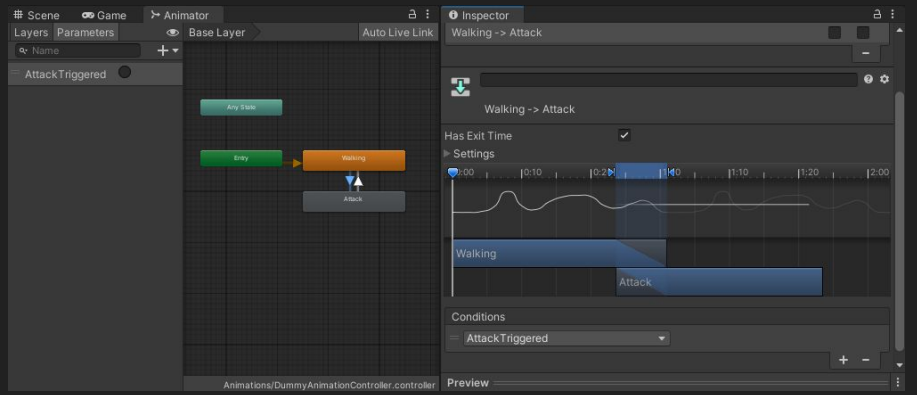
- Add a new animation to the Project Window
  - Animations can be found on https://www.mixamo.com/
  - Set the Animation Type to Humanoid
  - If this animation must loop, toggle “Loop Time”
- Drag the Animation within the Asset into the Motion field of the empty state of the Animator Controller



Animations in Unity (cont.)

- Adding more animations to the character will require some way to trigger the animations
- Create a new state for the Attack animation and transitions from and to Walking
  - On the from transition, add the trigger condition
- To trigger the attack animation use the following:
- `m_Animator.SetTrigger("AttackTriggered");`



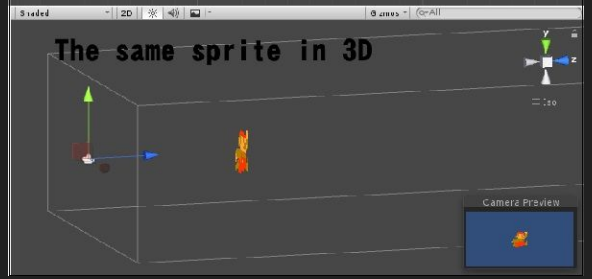


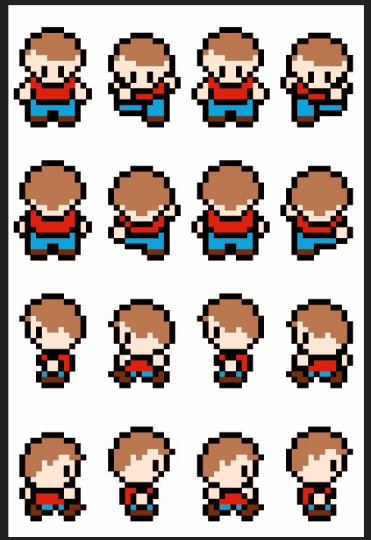
Additional Resources for Meshes and Animations

- LODs:
  - https://www.youtube.com/watch?v=ifNyVS2_6f8
- Animations:
  - https://www.youtube.com/watch?v=tveRasxUabo
  - https://docs.unity3d.com/6000.1/Documentation/ScriptReference/Animator.SetTrigger.html
- Root Motion:
  - https://www.youtube.com/watch?v=mNxEetKzc04

Textures and Sprites

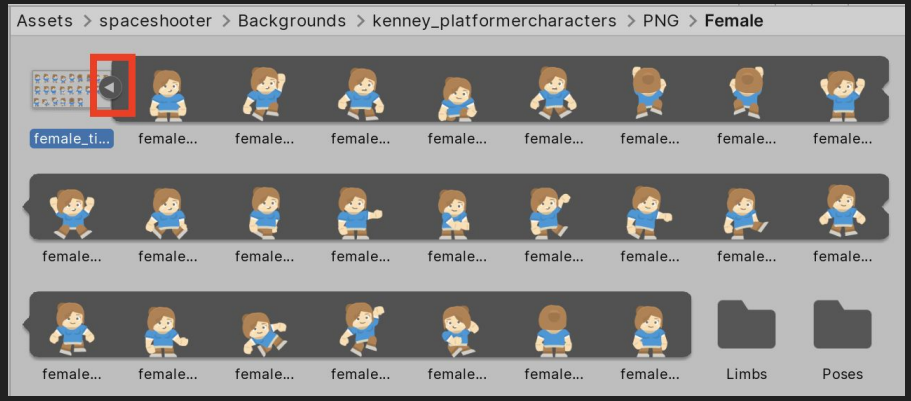
- Textures are images that are applied to 3D objects
- Sprites are images applied to 2D objects
  - Sprites can also be made of “Sprite Sheets” that allow for animating the sprite
  - Sprite sheets are single images that contain multiple variations of the sprite





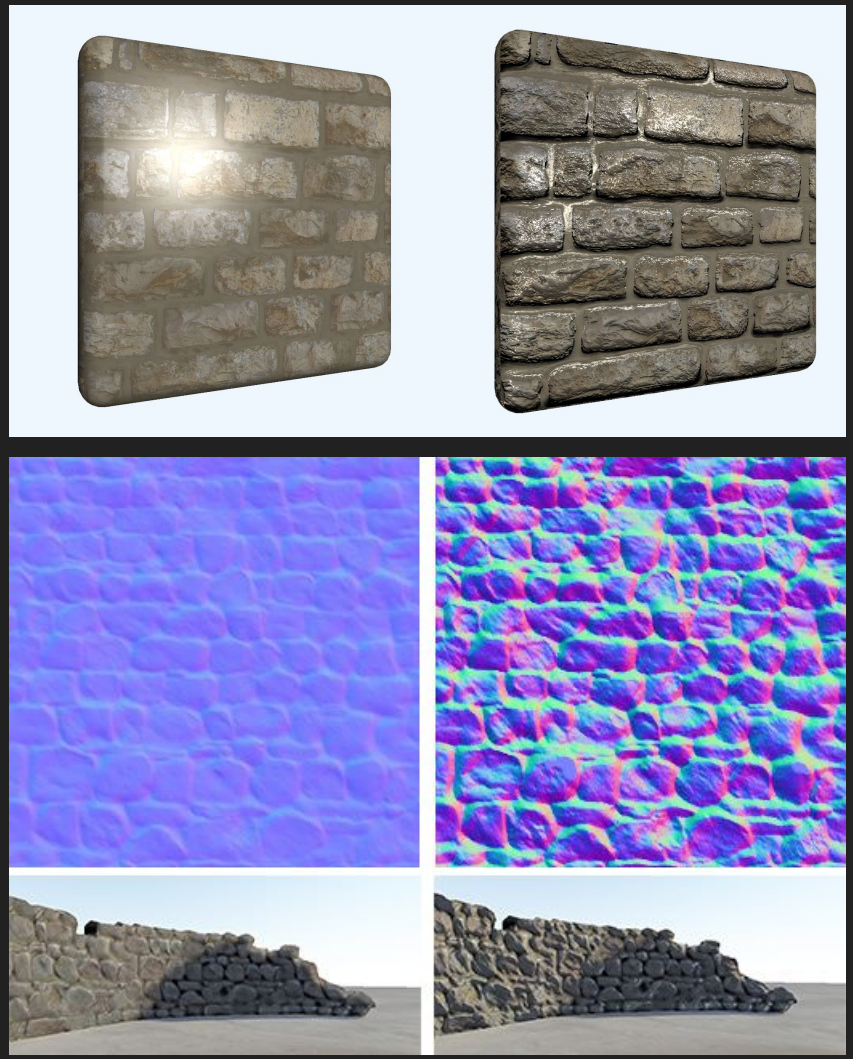
Sprite Sheets in Unity

- Unity has advanced sprite sheet functionality
- This allows you to animate sprites and to utilize multiple sprites stored within 1 image file
- More info:
  - https://learn.unity.com/tutorial/introduction-to-sprite-editor-and-sheets#
  - https://www.youtube.com/watch?v=FXXc0hTWIMs



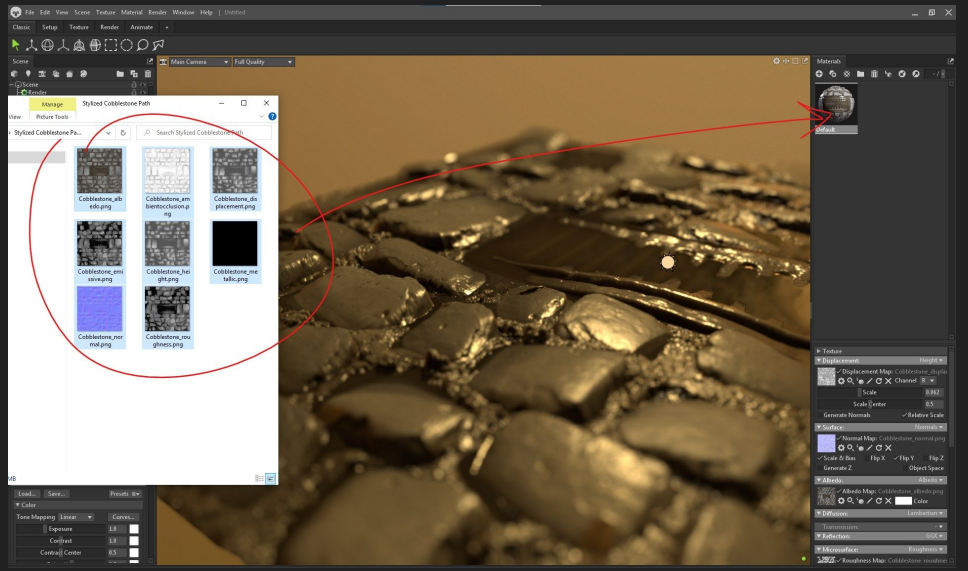
Textures and Maps

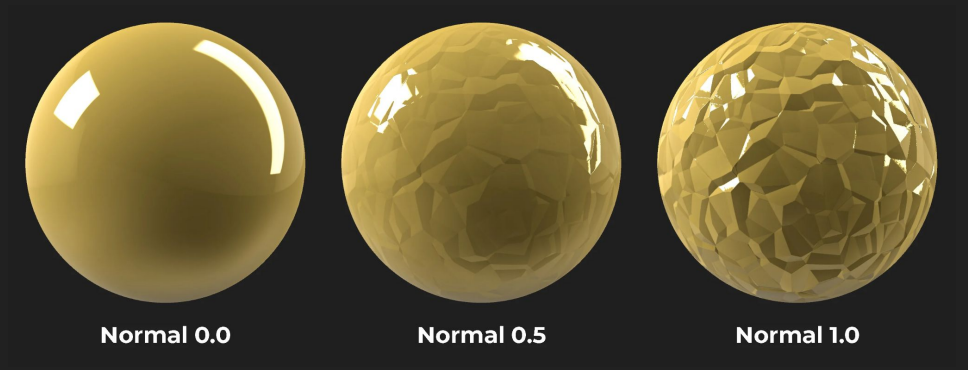
- Textures can have multiple “maps” or images:
  - Diffuse/albedo map
  - Normal map (bump map)
  - Specular map
  - Metal map
  - Roughness map
  - Ambient occlusion map
  - Height map
- These different maps can give textures color, depth, and reflectivity



PBR Textures

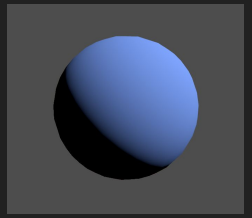
- A physically based rendering texture (PBR) aims to render textures in a way that mimics how light interacts with real-world materials
- PBR textures do not require more polys in the model
  - The details are baked into the textures!
  - This forces the GPU to work harder
- In Unity, it would be best to use HDRP if PBR textures are being used
  - Allows for ray tracing and better quality lighting

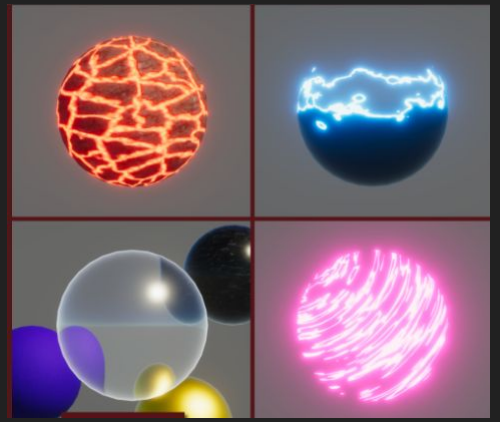


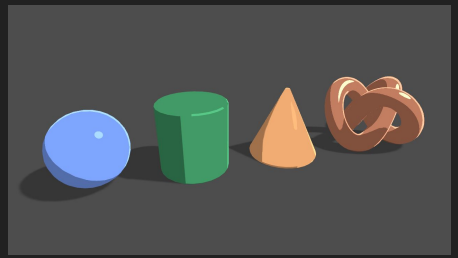


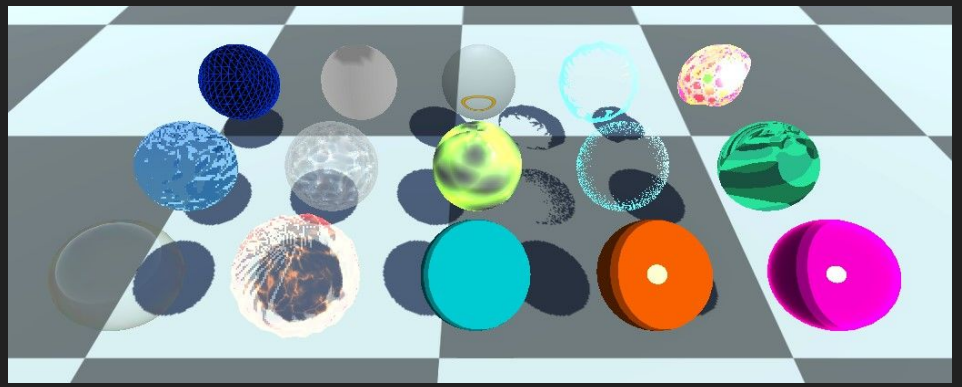
Shaders

- Shaders are small programs that run on the GPU to control how graphics are rendered for a given texture or material
  - Positions of vertices
  - Pixel color
  - How lighting affects
  - Special effects
- Shaders are written in their own programming language
  - HLSL is popular
- Shaders can also be written to compute things other than graphics like physics
  - Called “compute shaders”



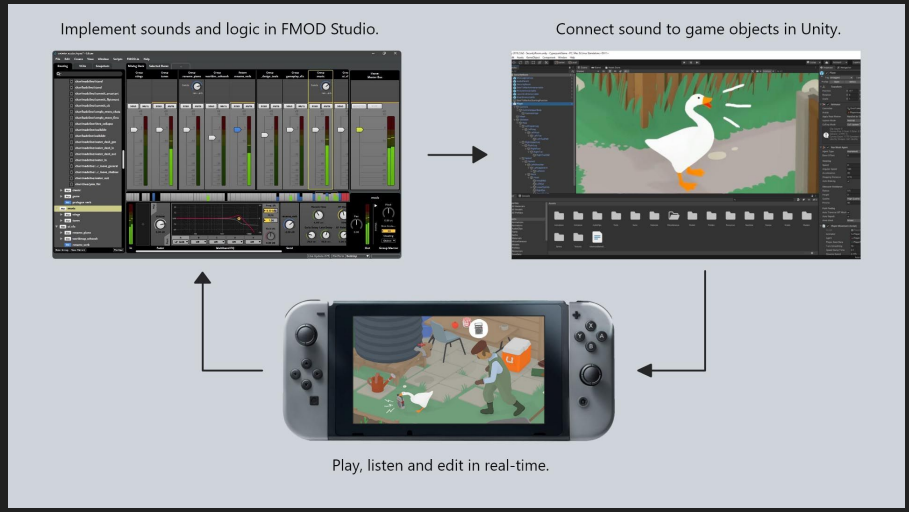






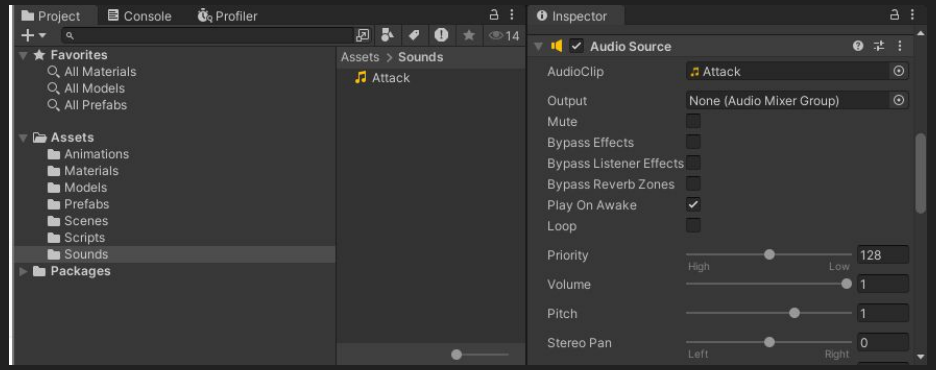
Audio Files

- Audio files contain sound information for music and sound effects
- Audio design in games is an art and requires many techniques:
  - Spatial audio affected by 3D environment
  - Reverberation and echoes
  - Procedurally generated audio
  - Audio mixing
- Some plugins/libraries can make this easier
  - FMOD



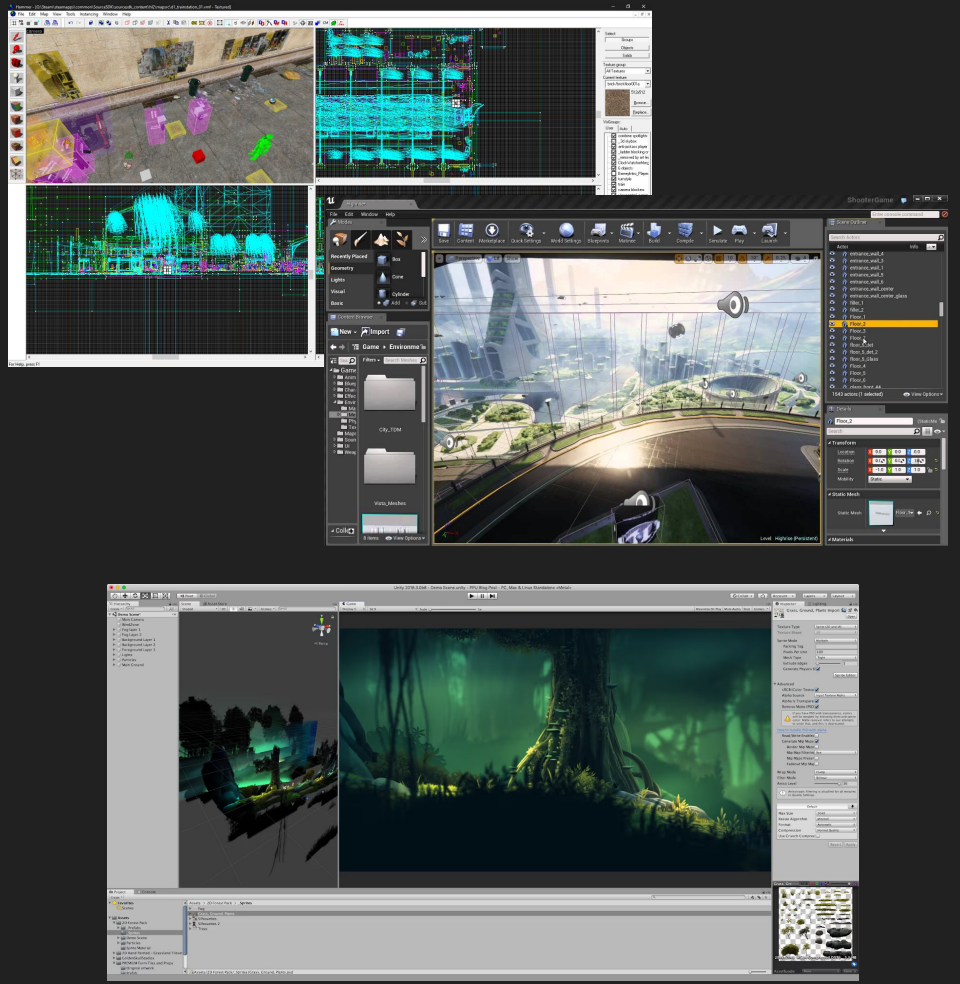
Playing Audio with Unity

- Audio files can be played in Unity using the Audio Source Component
  - Add the Audio Source Component on the GameObject that should play the sound
- Assign the AudioClip field in the Audio Source via a script or by dragging and dropping in the Editor
- This can be triggered via a script using the following:
  - `m_MyAudioSource.Play();`
- More info:
  - https://docs.unity3d.com/ScriptReference/AudioSource.html



Levels and Scenes

- Levels, scenes, or maps are environments in a game
  - Made up of various objects in the game
  - Designed by manually placing objects or procedurally generating objects
- In 2D games, levels are layers of sprites

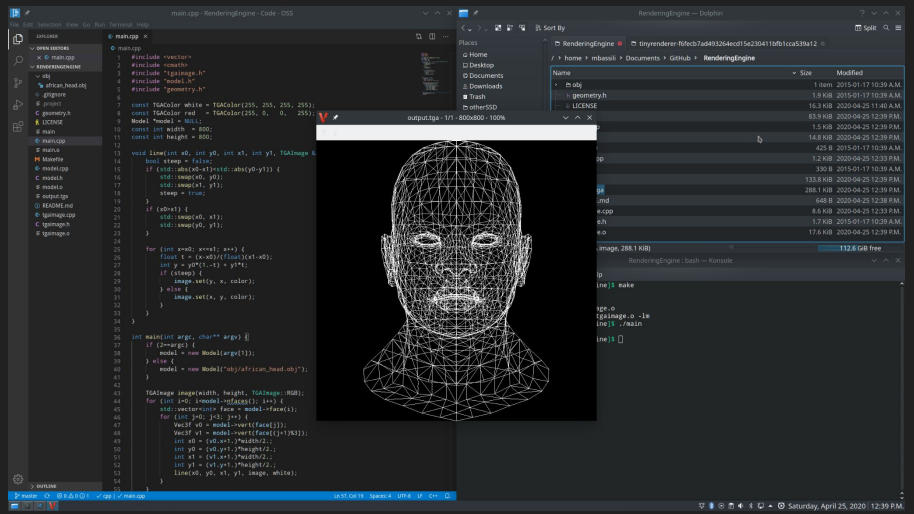


Loading Scenes in Unity

- If you have created multiple scenes, you can load into them using:
  - Overwrite the current scene:
    - SceneManager.LoadScene("YourScene", LoadSceneMode.Single);
  - Add to the current scene:
    - SceneManager.LoadScene("YourScene", LoadSceneMode.Additive);
  - https://docs.unity3d.com/6000.1/Documentation/ScriptReference/SceneManagement.SceneManager.LoadScene.html
- If you have one of more GameObjects that must persist between Scene loading, use this:
  - DontDestroyOnLoad(this.gameObject);
  - https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html

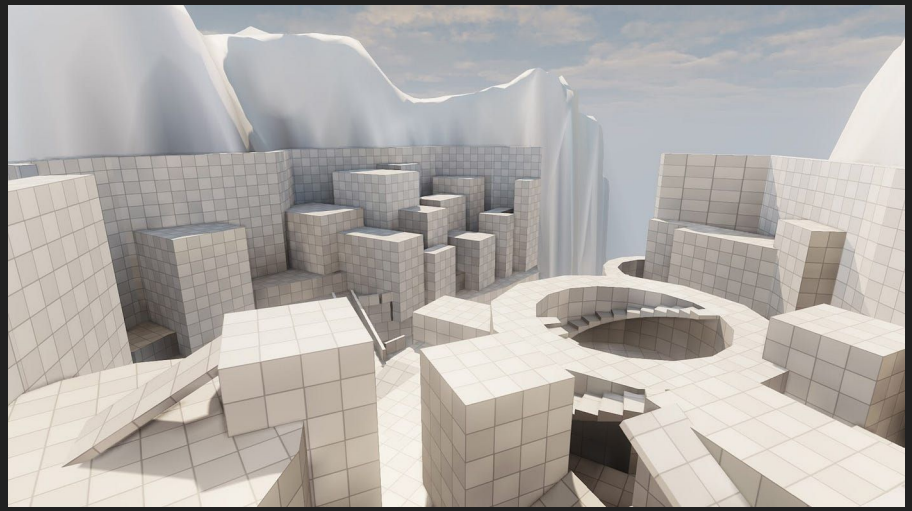
Libraries

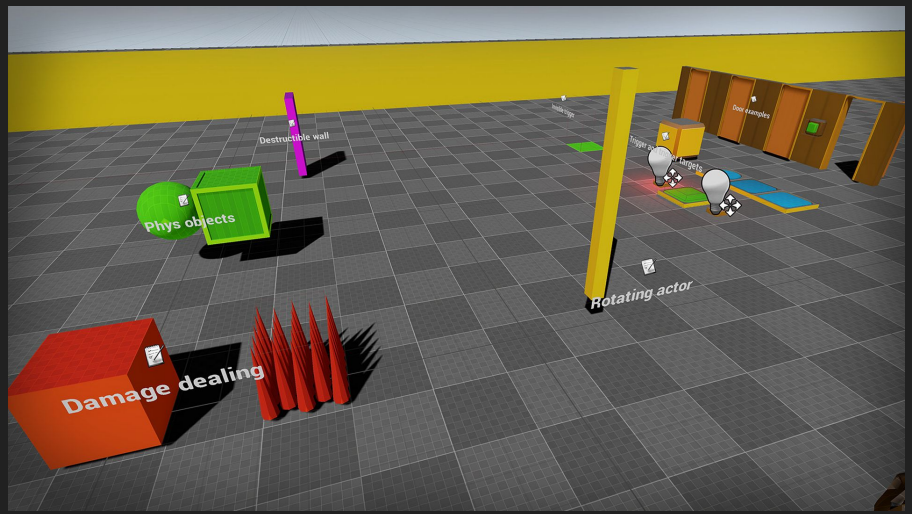
- Libraries provide additional functionality that you can plug into your engine
- Libraries specifically target some specialized functionality:
  - Rendering (OpenGL)
  - Physics (PhysX)
  - Audio (FMOD)
- Without libraries you would have to program everything from scratch!



Greyboxing

- Greyboxing is designing levels and systems using placeholder assets and grey boxes to test your game
- When programming a game, you do not need good graphics or high end assets
  - All you need is enough to test your ideas and make sure your code works and scale is correct
  - Flashy assets can be added later





Scripts and Hot Reload

- Scripts
  - Config files, languages, usages
  - Modding
  - Scriptable objects
  - Blueprints (Unreal)
  - Finite State Machine (unity animations)

Unity Scripts

- In Unity we write “Scripts”, but this is a bit different than what we’re about to talk about
- These scripts are actually compiled before runtime
  - A “just-in-time” compiler is also used in C#
  - Also allows for dynamic code loading
- In the game development world, the word “script” can mean various things
  - Usually means a text/code file that can be modified without needing to recompile the entire game

Config Files

- Config files give access to settings to a game or allow for “configuring” a game
- Config files are human-readable text files
- These can allow an advanced user to make minor tweaks
  - Giving all users access to these inside the game may overwhelm players
  - Typically, you have to manually create an interface for loading in and interpreting config files
- Config files are written in various languages and formats:
  - INI format (Unreal, Skyrim, Fallout)
  - JSON format (Minecraft)
  - CFG format (Doom, Quake, Half-Life)
  - YAML format (Unity)

Config File Examples

Fallout.ini

```
[Audio]
fASFadeInTime=2.0
fASFadeOutTime=10.0
fRegionLoopFadeInTime=5.0
fRegionLoopFadeOutTime=5.0
fAudioDebugDelay=0.0000
bEnableAudio=1
bEnableAudioCache=1
bMultiThreadAudio=0
bEnableEnviroEffectsOnPC=0
iAudioCacheSize=2048
iMaxSizeForCachedSound=256
bUseAudioDebugInformation=1
fDefaultMasterVolume=1.0000
fDefaultFootVolume=0.5000
fDefaultMusicVolume=0.6000
fDefaultRadioVolume=0.5000
fDefaultEffectsVolume=1.0000
fDefaultVoiceVolume=0.6000
iMaxImpactSoundCount=32
fMaxFootstepDistance=1100.0000
fPlayerFootVolume=0.6500
iCollisionSoundTimeDelta=150
```

config.cfg

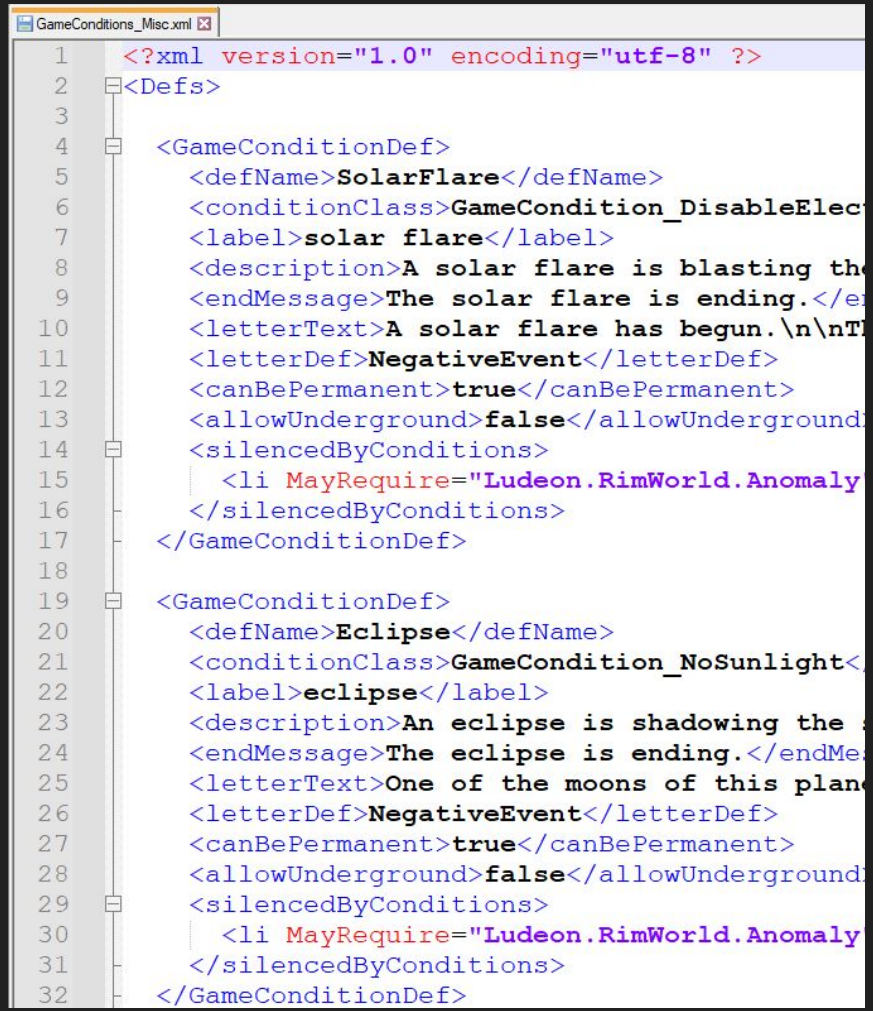
```
cl_timeout "60"
cl_updaterate "60"
cl_vsmoothing "0.05"
con_color "255 180 30"
con_mono "0"
console "1.000000"
crosshair "1.000000"
default_fov "90"
ex_interp "0.1"
fps_max "500"
fps_override "0"
gamma "2.500000"
gl_ansio "16"
gl_dither "1"
gl_flipmatrix "0"
gl_fog "1"
gl_keeptjunctions "1"
gl_lightholes "1"
gl_lowlatency "0"
gl_lowlatency_debugoutput "0"
gl_lowlatency_maxslop_ms "5"
gl_lowlatency_minslop_ms "2"
```

New Scene.unity

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011
--- !u!29 &1
OcclusionCullingSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 2
  m_OcclusionBakeSettings:
    smallestOccluder: 5
    smallestHole: 0.25
    backfaceThreshold: 100
  m_SceneGUID: 00000000000000
  m_OcclusionCullingData: {fi
--- !u!104 &2
RenderSettings:
  m_ObjectHideFlags: 0
  serializedVersion: 9
  m_Fog: 0
  m_FogColor: {r: 0.5, g: 0.5
  m_FogMode: 3
  m_FogDensity: 0.01
  m_LinearFogStart: 0
  m_LinearFogEnd: 300
```

Scripts for Modding

- Giving players the ability to modify scripts allows players to customize, or modify (mod), their game
- Example: RimWorld is a Unity game that loads various information about the game from XML files
  - You can modify these as you like to change how the game plays
- This functionality does not come by default with Unity, you would have to program it in



Scripts for Modding (cont.)

- Some scripts use more advanced scripting languages that allow for code to be injected into your game at runtime
  - Lua (Roblox, Garry's Mod, Elden Ring's AI)
  - Papyrus (Custom scripting language for Skyrim)

```
7   ------------------------------------------------------------]]
8
9   -- These files get sent to the client
10
11  AddCSLuaFile( "cl_hints.lua" )
12  AddCSLuaFile( "cl_init.lua" )
13  AddCSLuaFile( "cl_notice.lua" )
14  AddCSLuaFile( "cl_search_models.lua" )
15  AddCSLuaFile( "cl_spawnmenu.lua" )
16  AddCSLuaFile( "cl_worldtips.lua" )
17  AddCSLuaFile( "persistence.lua" )
18  AddCSLuaFile( "player_extension.lua" )
19  AddCSLuaFile( "save_load.lua" )
20  AddCSLuaFile( "shared.lua" )
21  AddCSLuaFile( "gui/IconEditor.lua" )
22
23  include( 'shared.lua' )
24  include( 'commands.lua' )
25  include( 'player.lua' )
26  include( 'spawnmenu/init.lua' )
27
28  --
29  -- Make BaseClass available
30  --
31  DEFINE_BASECLASS( "gamemode_base" )
32
33  --[[------------------------------------------------------------
34      Name: gamemode:PlayerSpawn()
35      Desc: Called when a player spawns
36  ------------------------------------------------------------]]
37  function GM:PlayerSpawn( pl, transiton )
38
39      player_manager.SetPlayerClass( pl, "player_sandbox" )
40
41      BaseClass.PlayerSpawn( self, pl, transiton )
42
43  end
44
```

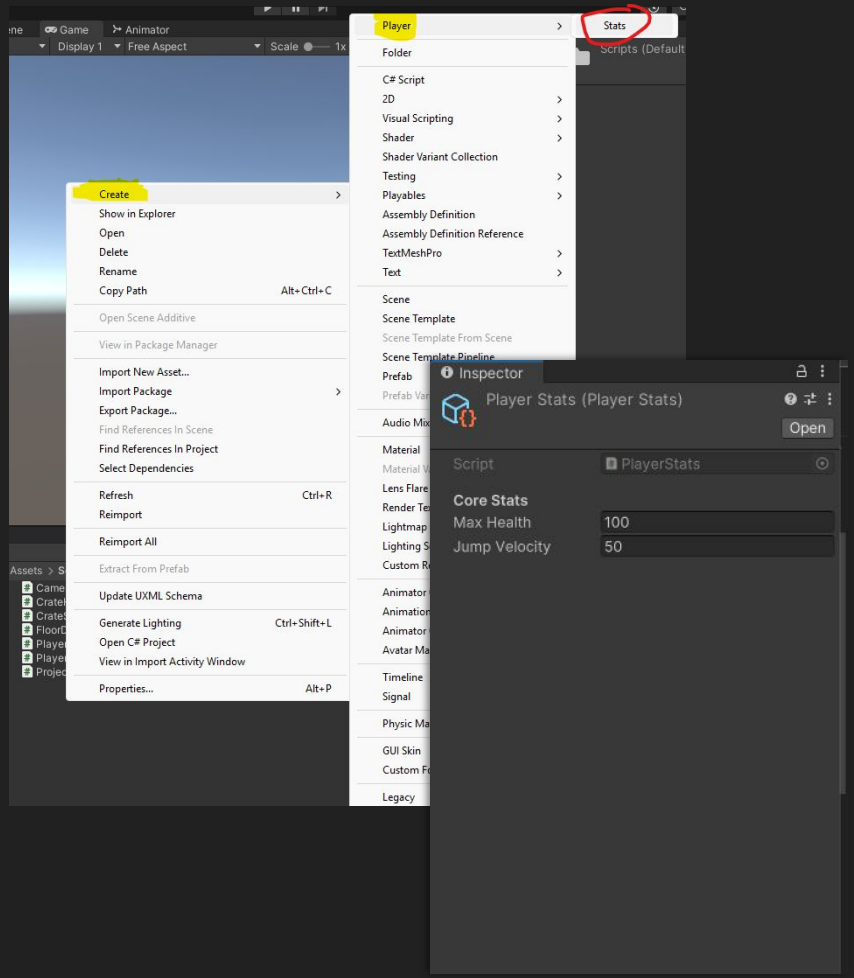
Using Config Files or Moddable Scripts

- Including config files or moddable scripts requires more advanced techniques and is non-trivial
  - If your scripts allow for adding code, and not just changing settings, this is even more difficult

1. Decide on what format/language you will use
2. Find a way to load the files off of the storage device
3. Parse the text in the files
   a. Analyze the text structure of the file and convert it into a data structure that the program can work with
4. Add functionality in your game that uses that data structure

Scriptable Objects

- In Unity, Scriptable Objects (SO) are custom classes that allow you to create your own assets that hold data
- SO's can hold onto data that is shared between multiple scenes and can be changed at runtime
- Example: Create SO's that hold item, character, or weapon stats for your game
    - Next, create multiple instances of this SO inside the Project Window so that you can have items, characters, or weapons with a variety of stats



Scriptable Objects (cont.)

PlayerStats.cs:

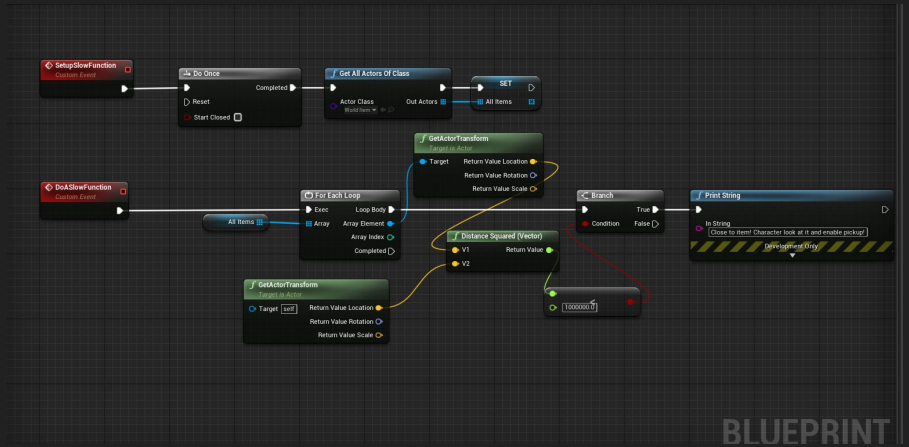
```
using UnityEngine;

[CreateAssetMenu(fileName = "PlayerStats", menuName = "Player/Stats")]
public class PlayerStats : ScriptableObject
{
    [Header("Core Stats")]
    public int maxHealth;
    public float jumpVelocity;
}
```

Player.cs:

```
using UnityEngine;

public class Player : MonoBehaviour
{
    public PlayerStats stats;

    private int currentHealth;
    private Rigidbody rigidBody;

    void Start()
    {
        currentHealth = stats.maxHealth;
…
    }

    // FixedUpdate is called once per fixed frame
    private void FixedUpdate()
    {
…
        if (jumpWasPressed && isTouchingFloor)
        {
            var force = new Vector3(0f, stats.jumpVelocity, 0f);
            rigidBody.AddForce(force, ForceMode.VelocityChange);
            jumpWasPressed = false;
        }
…
    }
```

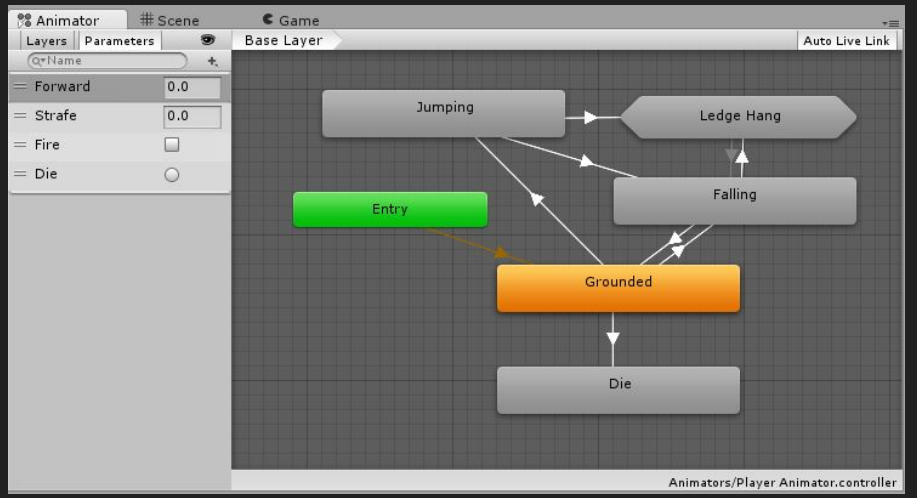
Blueprints (Unreal)

- Blueprints are not necessarily “scripts” but they are a way of creating game logic without writing code in the Unreal Engine
- There is a lot of functionality that can be implemented using Blueprints
    - Entire games can be made with Blueprints without ever touching code



Animation Controller FSM (Unity)

- The Unity equivalent of Blueprints is the Animation Controller which is only used for defining how animations play
- States are the animations that play
- Transitions allow for transitioning to other states
- Can get just as messy as Blueprints



Debugging

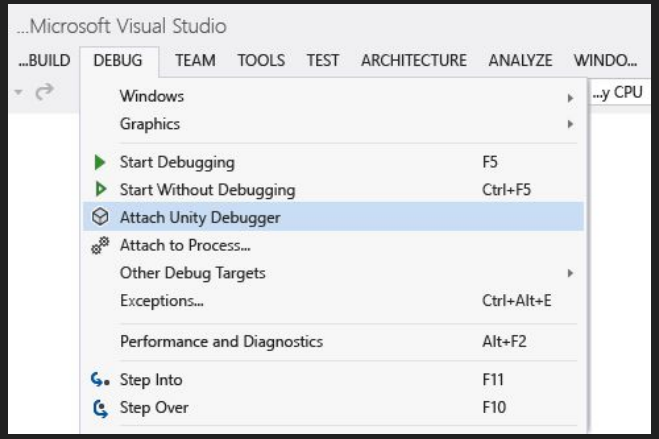
- Debugger
- User bugs and reproducing
  - Logging
- Unit tests and assertions
- Tools for debugging
- Version control and gitflow

Debugger

- The debugger is a tool that lets you set breakpoints and inspect the execution of your code while it is running
  - We will use this in Unity starting now!
- Very important tools to have:
  - Setting breakpoints
  - Conditional breakpoints
  - Checking the call stack

Debugger (cont.)

- The debugger is an easy way to figure out what might be wrong with your script
- If you have an if statement that's not working properly, you'll be able to investigate with ease
- All variables/fields can be inspected to check for issues




```
1   using UnityEngine;
2
3   public class DebugScript : MonoBehaviour
4   {
5       // Start is called before the first frame update
6       void Start()
7       {
8           Debug.Log("Executing the Breakpoint Test");
9           int numOne = 1;
10          int numTwo = 2;
11          int total = numOne + numTwo;
12          Debug.Log("Application Resumed");
13          Debug.Log("Total: "+total);
14      }
15
16      // Update is called once per frame
17      void Update()
18      {
19
20      }
21  }
```

User Bugs

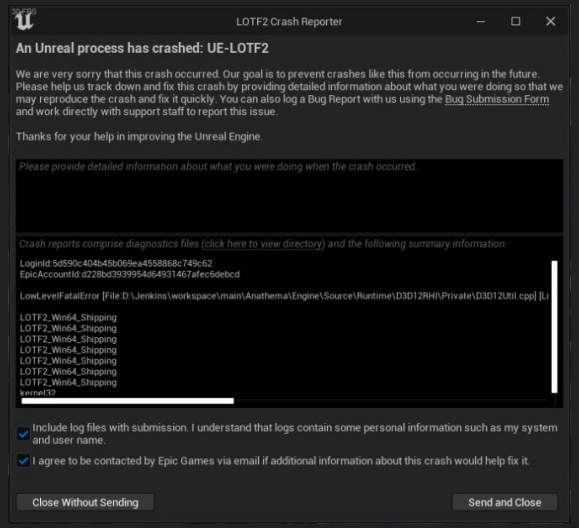
- You will not find all of your bugs
- Your QA department will not find all your bugs
  - If you have 10 testers testing 8 hours a day, 5 days a week, 52 weeks a year, that’s 20,800 hours of testing
- Your player will find your bugs
  - If you have 10,000 players it will take 3 hours to hit 30,000 hours of testing all the bugs you missed
- Aim to minimize the amount of bugs your users find
  - Don’t test in production

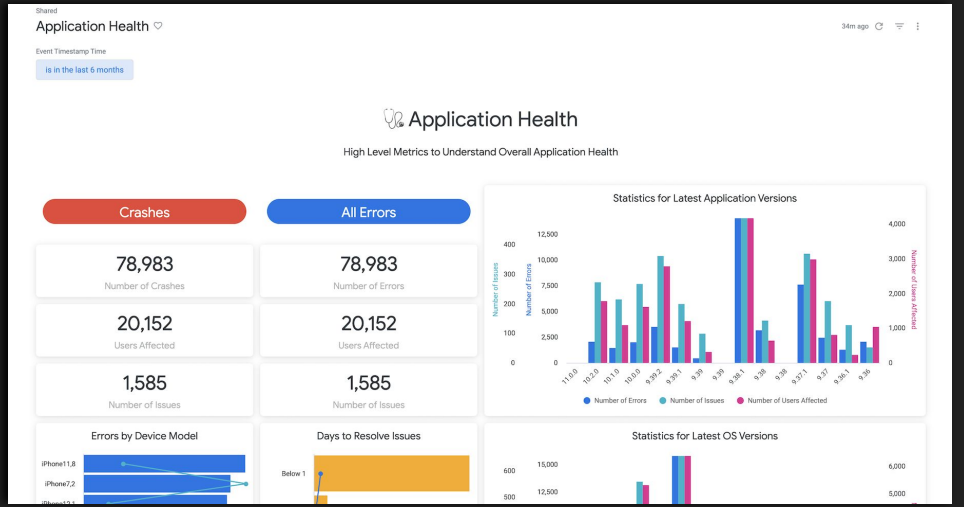
User Bugs (cont.)

- How to reduce bugs?
  - Test rigorously and early
  - Keep code clean and organized
  - Document as much as possible
  - Use version control (tracks when/where bugs pop up)
  - Do play testing with actual players
  - Avoid spaghetti code
    - Follow the “Single Responsibility Principle”
    - Write modular code

Tracking and Reproducing User Bugs

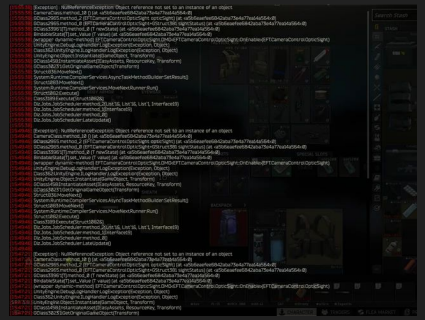
- You cannot fix a player's bug if you cannot figure out how the bug occurs
- You can always ask the player "how did the game crash?" but they might not always have a great answer
  - Add an error reporting tool in your game
  - Automated error/crash logging is extremely important





Tracking and Reproducing User Bugs (cont.)

- Some tools offer reporting tools that will automating send you logs when users crash
  - This can help you reproduce errors and fix bugs
  - Beware! Bugs in games might not involve errors or crashes!
- Have your games generate a log file that is sent to you if an error or crash occurs
- Once your error/crash reporting tools are in place, let players beta test your game



Unit Tests

- Unit tests allow you to test the smallest pieces of code
  - Tests a variety of inputs to functions
  - Useful for testing static code
    - Menus, login windows, server communication, etc.
- Unit tests are not as common in game development as in traditional software development
  - It is very hard to test real world scenarios over time and space
  - Your players will do things you never thought to test for

Unity Unit Tests (requires Test Framework package)

Calculator.cs:

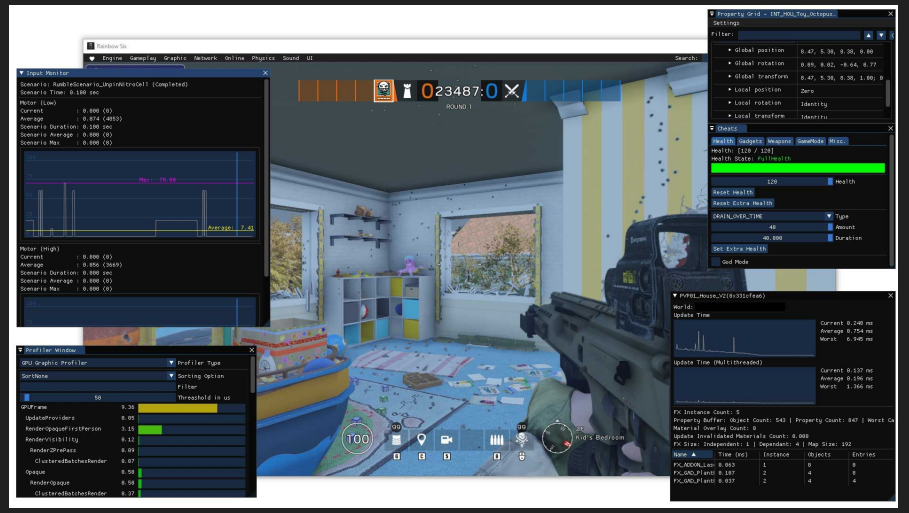
```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}
```

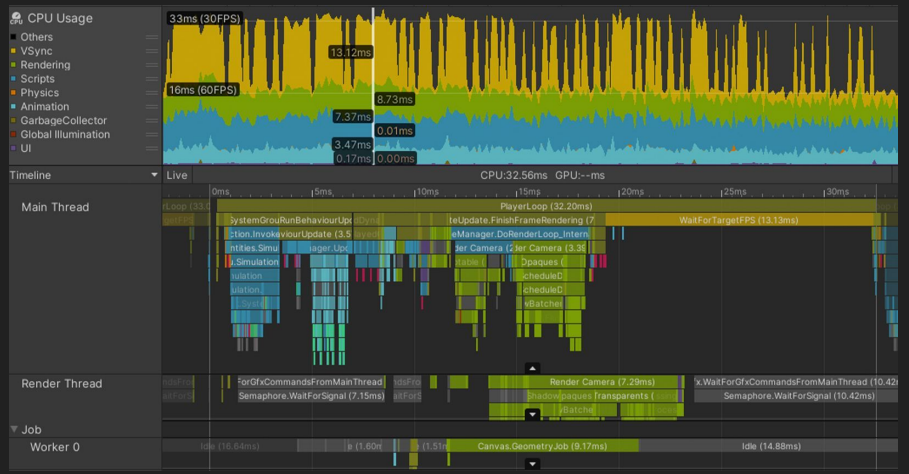
CalculatorTest.cs:

```
using NUnit.Framework;

public class CalculatorTests
{
    [Test]
    public void Add_TwoPositiveNumbers_ReturnsCorrectSum()
    {
        // Arrange
        var calculator = new Calculator();

        // Act
        int result = calculator.Add(2, 3);

        // Assert
        Assert.AreEqual(5, result);
    }

    [Test]
    public void Add_WithNegativeNumber_ReturnsCorrectSum()
    {
        var calculator = new Calculator();
        int result = calculator.Add(-2, 5);
        Assert.AreEqual(3, result);
    }
}
```

Tools for Debugging

- Other than “the” debugger there are other tools that help you debug issues
  - Profiler (performance)
  - IMGUI (displays debug info on top of game)
    - Gizmos, overlays, text
  - RenderDoc (allows you to deconstruct individually rendered frames)
    - Helps for debugging graphical issues





Version Control

- What happens if you fix a bug, release a new version of your game, and your players experience new bugs?
  - Are these bugs related to your fix or not?
- What happens if you add an awesome new feature and your players experience new bugs?
  - Are these bugs related to your new feature or not?
- What happens when you build a game with more than 1 person?
  - It’s impossible to track everyone’s changes manually
- You need version control to keep track
- Version control, like Git + GitHub is critical for maintaining your code and fixing and tracking bugs
  - Download Github Desktop and create an account now!

Version Control Like a Pro

- You don’t need to be a command line warrior to use version control
  - Github Desktop + Github Account is all you need
- Follow standard branching structures like GitFlow
  - This makes it 10000x easier to add features, isolate issues, collaborate with friends, create releases, etc.