Intro to Game Development:
GameObjects, Actors, & Game Languages

Information

- A1: come to 0110 Friday or Monday 12-4pm if running into problems
  - Will simultaneously have Zoom room open, but it's hard to debug remotely
  - Trouble-makers seem to be Windows/iOS and UE4/iOS
- A2 will be 3D modelling & UV mapping your face
- Most of the graphics & game dev info will come through experience, don't worry if you don't "get it" right now

Game Development Outline

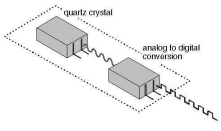
- Fundamental data structures (GameObjects/Actors)
- Logic differences between engines **w/ realtime examples**
- Editors
- Event handling
- Raycasting & collision detection
- **Realtime programming demo** of game engine logic
- How XR is handled in the engine
- **Realtime VR programming demo** with the Oculus Quest (grabbing objects, teleporting, etc.)

Today

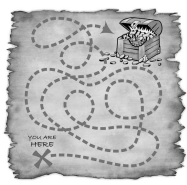
- Most concepts we look at in VR & game dev are similar between engines
- Today, we look at where things **diverge** so when we look at general concepts, you implicitly know implementation differences
- We normally won't distinguish between Unity & UE4 but it's important to understand
  - (engine design somewhat affects why Unity~AR, UE4~VR)
- Look at Unity & Unreal through critical lens instead of roasting Unity

(Most) (Modern) Game Engines

- Essentially realtime systems
  - **clock** (in this case tied to framerate) handles continuous logic
  - **Synchronization** between subsystems (physics, listeners, collision detectors, ray-tracing, etc.)
  - Tries to be as synchronous as possible (e.g. each frame lasts the same amount of time)
  - As opposed to solely reacting to input through **callbacks**, like in most mobile apps, text-based games, etc. which have no continuous logic beyond an actual clock
- Game engines are mostly **fully-featured APIs** containing many sub-APIs
  - Physics, collision logic, rendering, materials, optimized data structures, AI, audio etc.
- **Playground** to experiment with abstract CS concepts like clocks, state machines, data structures, black boxes, etc.



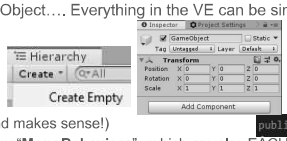
Example Application for the Class



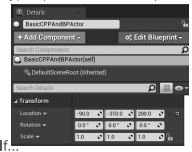
- Treasure Hunt
  - Seems to be a good example with connection to relevant areas of XR
  - Audio, locomotion, logic, haptics, IK, animation, etc.
  - First-person character, XR or non-XR...call them TreasureHunter
    - Can walk around and turn head
    - Can bump into objects in the environment but not go through them
    - Can grab collectible treasure with hands
    - Can hear
  - Treasure
    - 3D mesh that you hide in the VE to be found
    - Has some point value corresponding to how hard it is to find
    - Finding some amount of points wins the game
  - Main menu
    - Game start
    - Settings
    - High score
  - Win/Lose Screen

“Object” in a Game Engine

- Atomic “Object” with a 3D transform
  - 3D equivalent of basic Java Object…. Everything in the VE can be simplified down to this



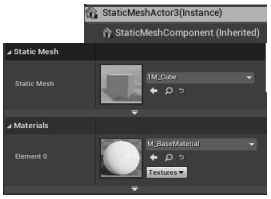
- **Unity: GameObject** (GO)
  - Most common term for this (and makes sense!)
  - List of components described by “**MonoBehaviors**”...which are also EACH the GO’s type (multi-inheritance)
  - Name is one of few ways Unity follows industry standard & Unreal doesn’t



- **UE4: Actor & ActorComponent**
  - **Actor**: UE4’s version of GameObject… should always have 3D root
  - **ActorComponent**: Similar to Actor except always part of Actor...meaningless by itself...
    - E.g. Mesh gives info about physical structure but needs 3D transform to be in game… so a mesh by itself is not too useful and is a COMPONENT of the Actor with transform
    - Might have a relative transform… but could be purely logical like most Unity (Colliders are exception)
  - Based on idea that everything in scene has some action...even if just a static obstacle

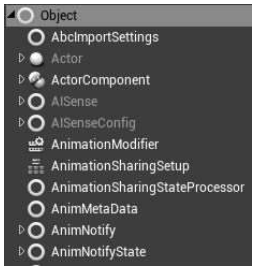
Quick Examples of Actor & ActorComponent

- (before we actual get into that part...just to have a high-level understanding)
- **StaticMeshActor** (mesh placeable in environment) is **Actor** with **StaticMeshComponent** (component defined by StaticMesh)
- **DirectionalLight** is **Actor** with **DirectionalLightComponent**
- In Unity, can add DirectionalLightComponent to something, but it can’t **implicitly** define the GO as a light (there can never be a DirectionalLight “object” as far as Unity is concerned… just a GO with DirectionalLight behavior)



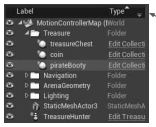
Non-GameObject Objects

- Things without physical representation in VE (Transform has no meaning)
  - Settings, save files, state machines, textures, shaders, etc.
  - Except for Actor & ActorComponent



Fundamental Difference Between Unity & Others

- Plugin structure and other differences are not “fundamental” differences… just implementation differences (e.g. Unity plugin implementation causes strange compiler errors and breaks b/w engine versions)
- **Fundamental difference**: How the GameObject is defined (or in Unity’s case, NOT defined!)
  - **Unity**: GameObject is **composite of unlinked components describing behaviors** (non-3D...just scripts) … but only identifiable by 1 component at once (overall GameObject container has no subtype). GameObject is nothing more than List<Component>
  - **Non-Unity engines**: **Object-oriented.** Actor is **defined composite of variables** (some pointers to ActorComponents & other Actors) **& functions.** Actor containing this info and more is a subclass



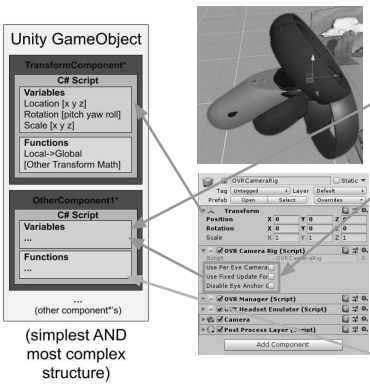
My treasure can’t be typed like this in Unity

For people who already know Unity...

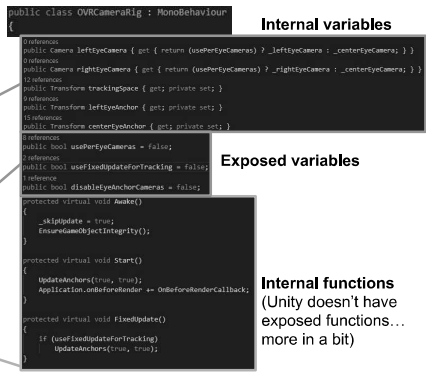
- **Prefabs** don’t count! They aren’t types in C#. Just templates (like prototypes) of something in the Editor or in the scene.
- They don’t solve the problem of lack of real GameObject type…. But they can help a little bit b/c prefab “variant” is kind of like a subclass (but in the JS sense…)
- We’ll come back to prefabs

Structure of Unity’s GameObjects

Similar to **strategy pattern** & core of why Unity’s so “easy”...but also cause of unscalability



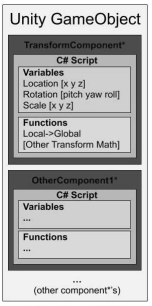
(simplest AND most complex structure)



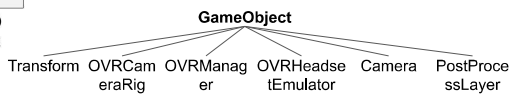
(Unity doesn’t have exposed functions… more in a bit)

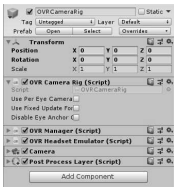
"Finding" Other Components

- Components can find other components by getting their GameObject and asking it to GetComponent<Type> or GetComponents<Type> to return pointers
- This GameObject can be casted to any individual component
  - So OVRCameraRig (just display name in scene or of prefab) IS-A GameObject, Transform, OVRCameraRig, OVRManager, OVRHeadsetEmulator, Camera, PostProcessLayer. Outer GO can be grabbed with typedGO.gameObject
- But can't be casted to a class that already knows ALL of the components (aka definition)...GetComponents returns Type[]
  - No subtype of GameObject that defines the Components
  - So you as the dev must know structure beforehand...
  - While Unity doesn't.... It figures it out at runtime
  - GO is really just a collection of pointers to components

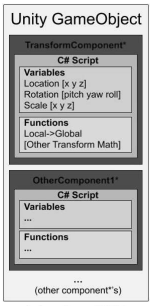


(simplest AND most complex structure)





What do you notice that's different from OO design?

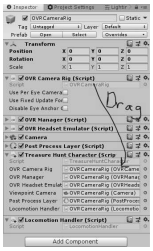


(simplest AND most complex structure)

- There is no dev-editable script linking the Components
  - Outer GameObject cannot be subclassed… not defined any more specifically internally (can't make components member variables)
- So is there any complete definition of "what" this GO is?
  - No! Can create template through prefab but no way to autocomplete anything between scripts without the dev linking them manually (manually defining bunch of GetComponents AND checking for null/count…. Or linking through editor)
- Positives?
  - Might allow more flexibility
  - Might allow GameObject to be more barebones…
    - UE4 Actor contains a lot of data by default (multiplayer, rendering, etc.)…. Can use up RAM quickly
  - Might be simpler to formalize than UE4… it's a tree where components (nodes) can only access each other by going ALL the way back to root and then down to other component
  - Way simpler to understand. The structure never gets more complex

Thinking about Treasure Hunter..

- How do we get a full definition of the GameObject
  - Make a TreasureHunter script
    - i. Manually assign pointers using public keyword (exposes var to editor)
    - ii. GetComponent over and over
    - iii. GetComponents and iterate through all possible classes to figure out what casts correctly (HIGH IQ)



or

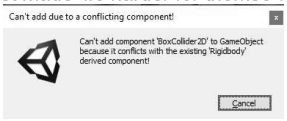
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Rendering.PostProcessing;

0 references
public class TreasureHuntCharacter : MonoBehaviour
{
    1 reference
    OVRCameraRig oVRCameraRig;
    1 reference
    OVRManager oVRManager;
    1 reference
    OVRHeadsetEmulator oVRHeadsetEmulator;
    1 reference
    Camera viewpointCamera;
    1 reference
    PostProcessLayer postProcessLayer;
    1 reference
    LocomotionHandler locomotionHandler;
    // Start is called before the first frame update
    0 references
    void Start()
    {
        oVRCameraRig=this.gameObject.GetComponent<OVRCameraRig>();
        oVRManager=this.gameObject.GetComponent<OVRManager>();
        oVRHeadsetEmulator=this.gameObject.GetComponent<OVRHeadsetEmulator>();
        viewpointCamera=this.gameObject.GetComponent<Camera>();
        postProcessLayer=this.gameObject.GetComponent<PostProcessLayer>();
        locomotionHandler=this.gameObject.GetComponent<LocomotionHandler>();
    }

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

    }
}
```

Strange implications

- You can have multiple GOs with TreasureHunter script with completely different structure (so all IS-A TreasureHunter)…. Wouldn't complain til runtime when something breaks
- In code, you can technically make no difference between some other camera in the scene and TreasureHunter b/c both IS-A camera
  - Might be good in some rendering optimizations
  - But normally just makes things confusing in code

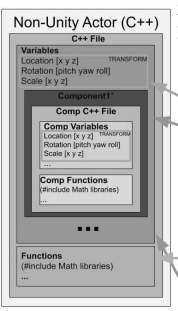
Used to be able to do weird things

- E.g. mix 2D and 3D collisions and physics…. They needed to specifically go in & prevent this… just made life harder for themselves

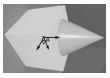


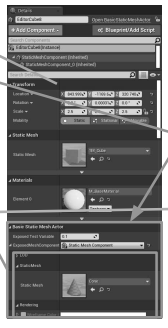
Basic Structure of Other Engine GameObjects & Actors

Object-oriented….not much different than other OO programs you're familiar with



I subclassed StaticMeshActor for simplicity & general C++ example





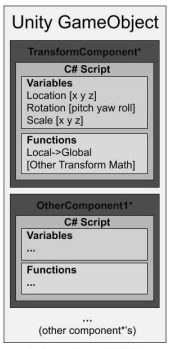
```
UCLASS()
class COMP590A1REWKOWSKI_API ABasicStaticMeshActor : public AStaticMeshActor
{
    GENERATED_BODY()

    UPROPERTY(EditAnywhere)
    float exposedTestVariable = 0.1f;
    UPROPERTY(EditAnywhere, DisplayName="ExposedMeshComponent")
    UStaticMeshComponent* exposedMeshComponent;

    float internalTestVariable = 0.2f;
    UStaticMeshComponent* internalMeshComponent;

    void testInternalFunction(float testVar);
};

void ABasicStaticMeshActor::testInternalFunction(float testVar)
{
}
```

Exposed Variables

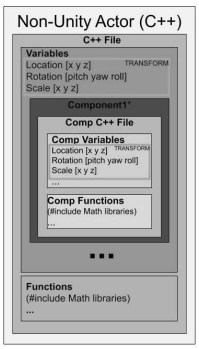
Internal variables

Internal functions
(Exposed functions won't make sense til we look at Blueprint)

Which structure sits better with you?



(simplest AND most complex structure)



Towards Blueprints & High-Level Programming...

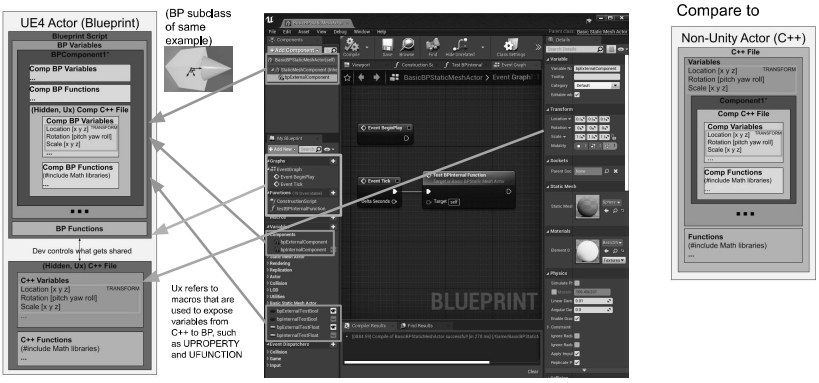
- Works alongside C++. UE4 Actor allows vars/functions to be shared
  - C++ can subclass BP and vice versa!
    - (generally don't subclass BP IN C++ b/c defeats the purpose of abstracting the C++)
  - BP has no syntax…. Only "syntax" is physical structure of the graph!

No More of This!

Either not deal with CPP at all or streamline compilation

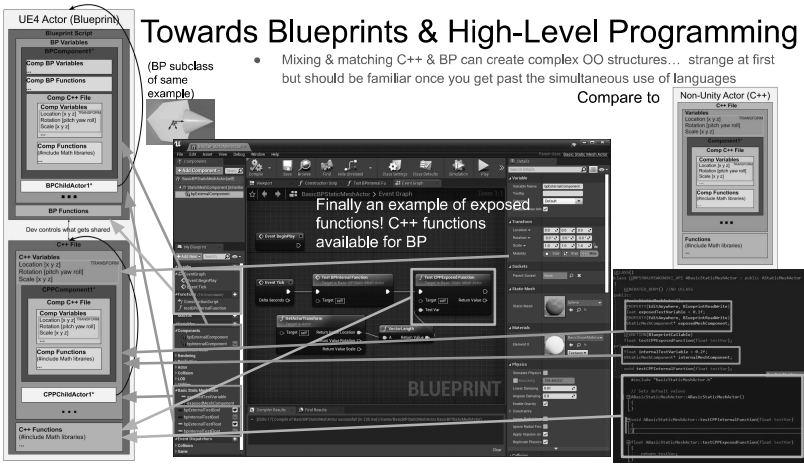
Towards Blueprints & High-Level Programming...

- Simplest BP abstracts ALL C++.... never need to touch Visual Studio!
  - Simply adds BP Variables + Functions on same layer as C++. BP organized structurally



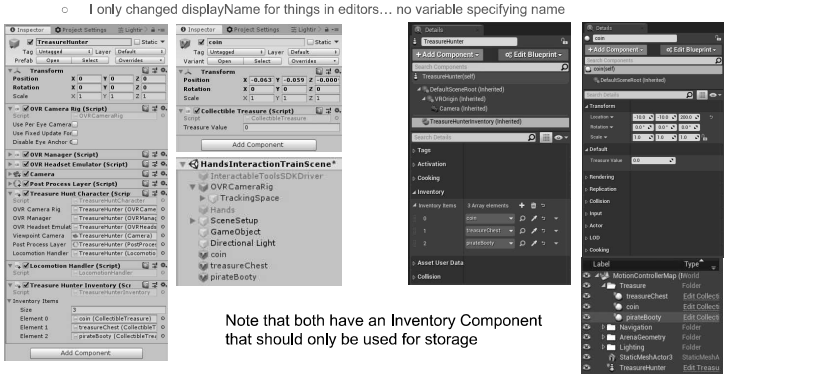
Towards Blueprints & High-Level Programming

- Mixing & matching C++ & BP can create complex OO structures… strange at first but should be familiar once you get past the simultaneous use of languages



Example of Programming Differences

- Problem statement: I want to get info from the TreasureHunter…. I want to calc their score and save it in main TreasureHunter class! Needs to be calculated in **main class** b/c this function needs to be used for other things (e.g. UI display)
  - I only changed displayName for things in editors… no variable specifying name

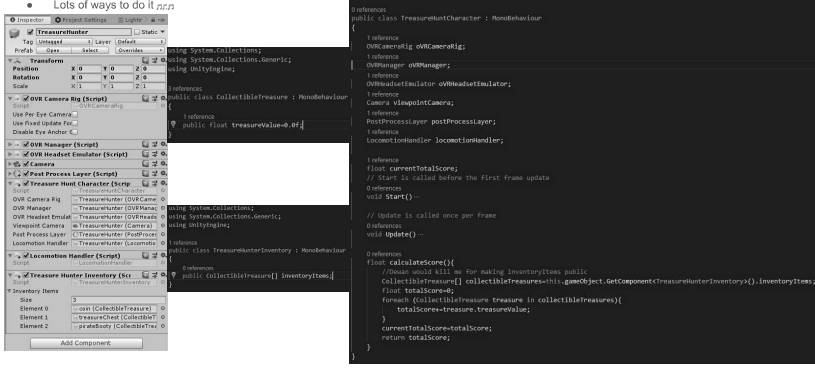


Note that both have an Inventory Component that should only be used for storage

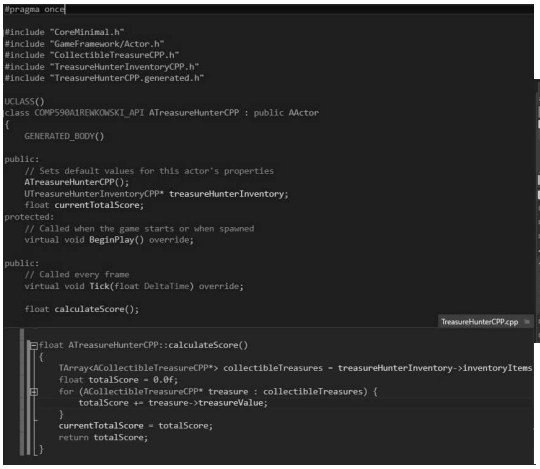
This is how we do it 🎵🎵🎵

Unity C#

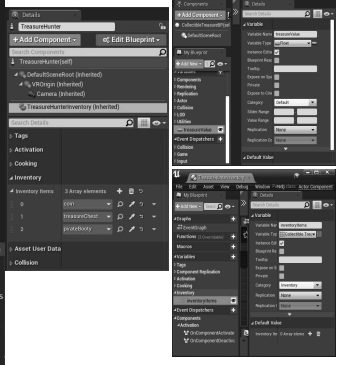
- Remember that inventory should only be used for storage of pointers
- Lots of ways to do it ¯_(ツ)_/¯

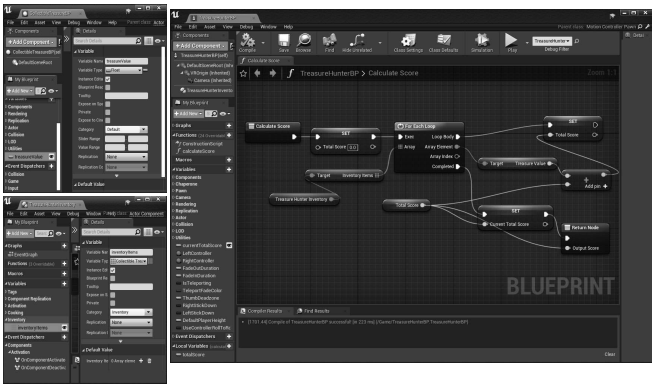


Unreal C++



Unreal BP





Anyone not convinced these are the same? Thoughts?

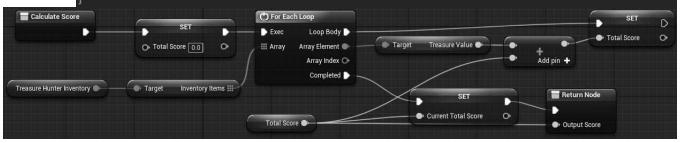
C#

```
float calculateScore(){
    //Dewan would kill me for making inventoryItems public
    CollectibleTreasure[] collectibleTreasures=this.gameObject.GetComponent<TreasureHunterInventory>().inventoryItems;
    float totalScore=0;
    foreach (CollectibleTreasure treasure in collectibleTreasures){
        totalScore+=treasure.treasureValue;
    }
    currentTotalScore=totalScore;
    return totalScore;
}
```

C++

```
float ATreasureHunterCPP::calculateScore()
{
    TArray<ACollectibleTreasureCPP*> collectibleTreasures = treasureHunterInventory->inventoryItems;
    float totalScore = 0.0f;
    for (ACollectibleTreasureCPP* treasure : collectibleTreasures) {
        totalScore += treasure->treasureValue;
    }
    currentTotalScore = totalScore;
    return totalScore;
}
```

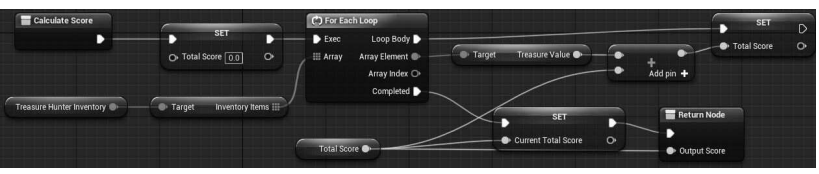
BP



In Action

Nice things about BP

- Helps you understand the visual structure of logic
  - Even if you don't like UE4, BP is good for building logical understanding!
- Abstracts the annoyances of syntax
- Compiles <1s
- Search what you think something is called, like in Blender



Main Purposes of BP

- No syntax
- Compiles crazy fast
- Trivial to create pointers
- API importing handled automatically

Main Purposes of C++

- Way faster at loops (BP has similar computational limitations to Python)
- Can interface with APIs not specifically made for game dev

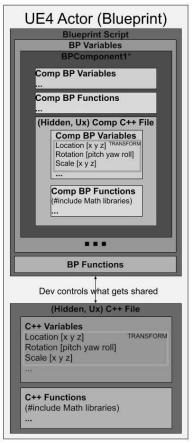
Intro to Game Development:
Game Engine Editors, Events, & Raycasting

General Flow in UE4

- Use C++ to expose other APIs to UE4 Editor & BP
- Use C++ to handle slow operations (e.g. mesh iteration)
- Use BP for everything else (unless you prefer using ONLY C++)

Relationship between BP & C++

- You probably already understand this relationship! Same as these relationships:
  - **XCode**: relation between Swift & Obj-C
  - **C++**: relation between C++ & C (extern “C”)
  - **ML work**: relation between Python & C++ (numpy is mostly C++ exposed to python!)
  - **C**: relation between C and x86 (_asm_ volatile (“[some ASM commands]”))
- BP is an interesting case b/c it’s basically the most high-level we can get (that is actually feasible!)
  - It’s also pretty weird, but probably have seen similar APIs
  - MIT Scratch, GameMaker Studio, RPG Maker, Blender Cycles, Maya PBR, etc.



Reasons not to use UE4 & other high-end engines

- UE4 is PBR by default, meaning it kills the GPU
- UE4 also uses pseudo-ray-tracing called “voxelized cone-tracing”.... Not as slow as ray-tracing, but still slower than traditional method
- Uses a TON of RAM (lot of data propagates through subclasses)
- Overkill for many tasks (Cuphead would probably look no different but perform poorly in UE4)
- UE4 treats ALL Actors as physics-based
  - Another resource-killer if not careful

Program Toolbar

-Settings

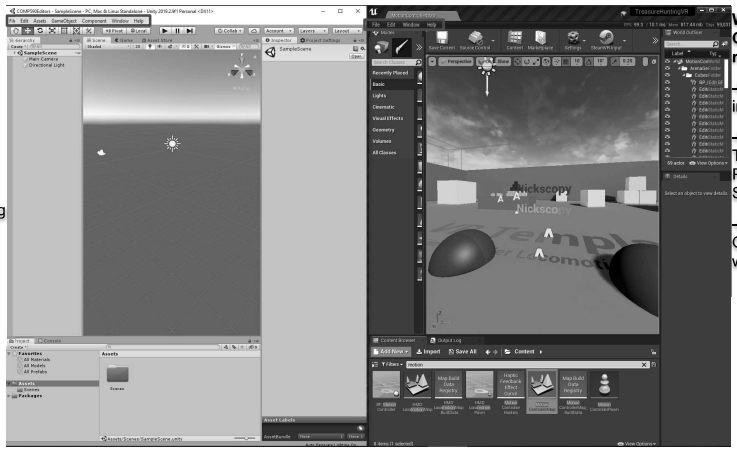
-Windows

-Help

-Meta-stuff

-(in Unity) Plugin config

-Info about engine version, Project name, etc

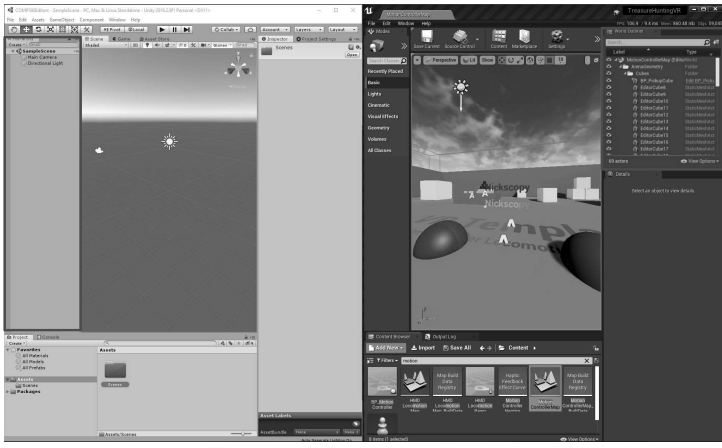


World Outliner/Hierarchy

-List of GOs in VE

-Physical/ Temporary Parenting Structure

-(In UE4) Organize with Folders

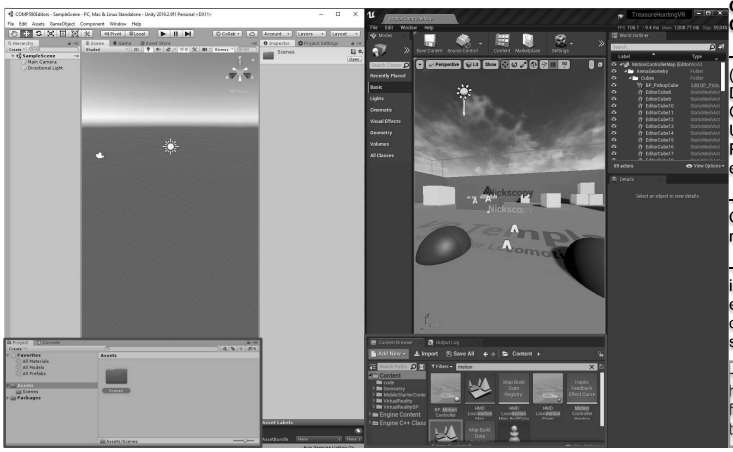


Project (Assets)/ Content Browser

-List of folders & assets (classes, materials, levels, etc.)

-Also has Plugin content, favorites, etc.

-(in UE4) color-coded types



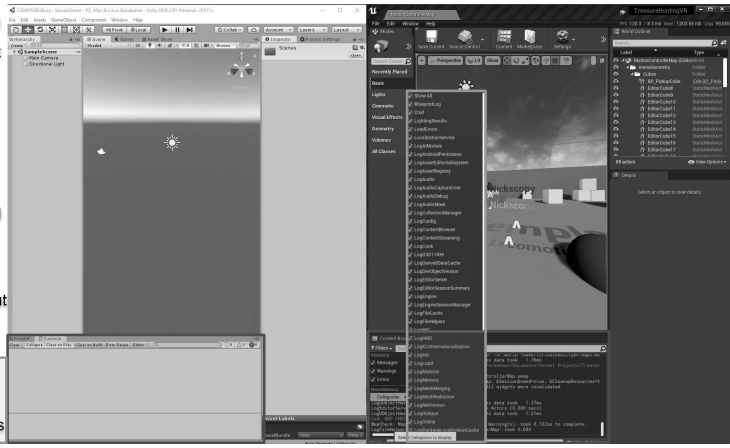
Console/ Output Log

-Your output (when Debug.Log, GLog->Log, UE_Log, Print String, etc.)

-(in UE4) Color-coded messages

-(in UE4) info from engine about compilation, states, etc.)

-(in UE4) huge list of filters for types of logs



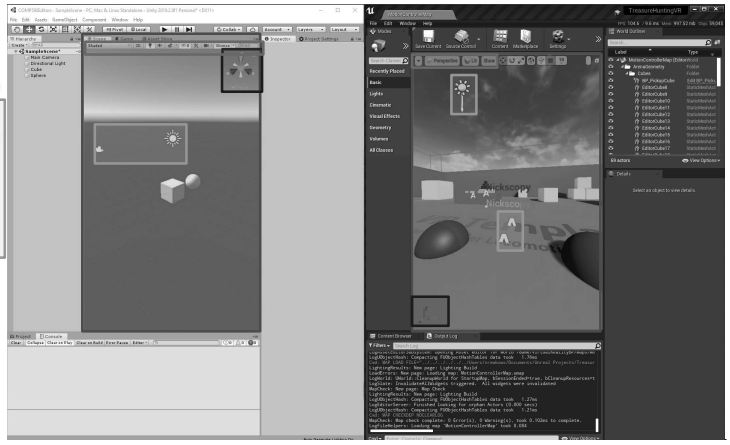
3D View of Scene

-Physical layout of scene (GOs)

-Symbols representing non-visible GOs (light, cameras, etc) (Unity calls these "Gizmos")

-3D axes

-Meta-info (what to show, shading, etc.)

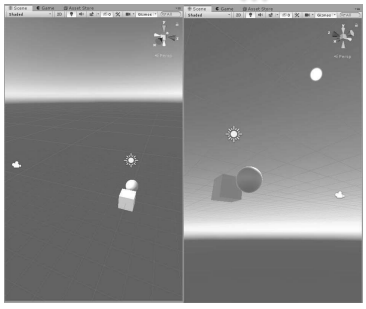


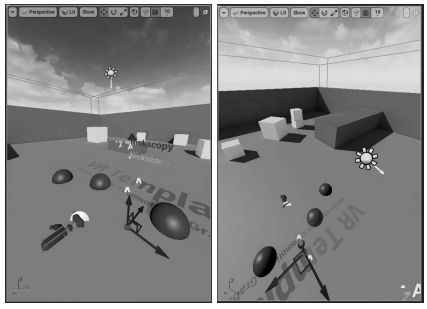
SHORT RANT

-WHY can't you easily see the transform of gizmos that have orientation in Unity? The camera & lights looks the same no matter which angle. They couldn't add a simple vector like UE4??? How do I know where the camera is facing without clicking it?

The camera icon itself isn't even facing the right way

VVV





Details/ Inspector

-Info about GO and Components

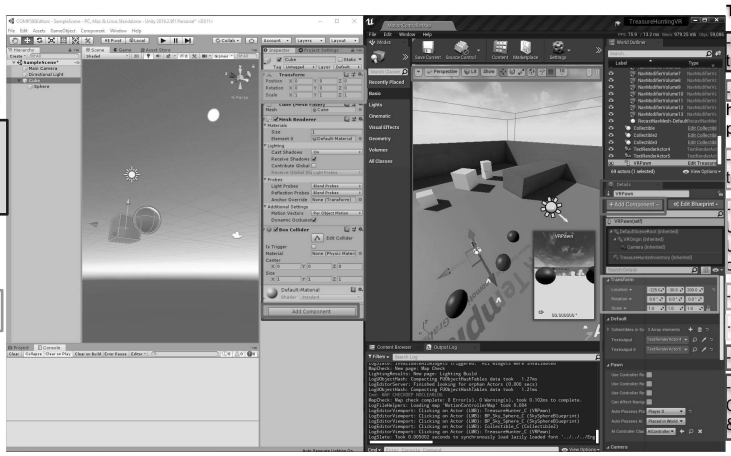
-(in UE4) Parenting structure of SceneComponents

-Add Component button

-Transform info

-Exposed variables

-(in UE4) metadata (replication)



Toolbar & misc

-Relative or global transform

-(In Unity) Pivot (UE4 handles this in component parenting structure)

-(in UE4) Snap values for transforming

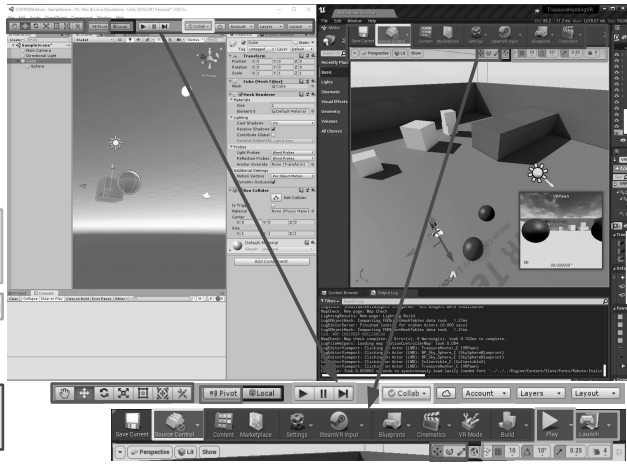
-Play/Pause Game (in Unity, only in Editor) (in UE4, Play in VR & other options)

-(in UE4) Launch to device

-Source control

-Navigation/Transform

-(in UE4), Compile, Cinematics, Build Lighting & Navigation,etc



UE4-Specific Bar: Modes

-Place mode (placing Actors)

-Paint mode (vertex/ texture paint, weights for animation, etc.)

-Landscape (shape of ground) (Unity has Terrain tool elsewhere)

-Foliage painting (add plants, grass, etc.) (Unity has a simpler variant)

-Geometry (edit UE4 meshes/brushes) (Unity recently added ProBuilder)