**Intro to Game Development (Part 2)**

- **Notes:**
  - We’re looking for ways to streamline the XCode process (e.g. not needing to manually change signing team every time)
  - Git repo should not include build, Library, or Logs folders (although this isn’t part of the grade)
  - Don’t use UE4 on iOS until we figure out how to deal with weird Apple Dev problems
  - If using Unreal, they will be much less stressful
    - Only needs Blender (no platform-specific instructions)
    - Will assign if Thursday & have until next Thursday
  - If you don’t have headset, let us know and we’ll get it outside class

---

**RECAP**

- **(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 tests 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, K, animation, etc.
  - First-person character, XR or non-XR... call them Treasure Hunter
    - 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 “MonoBehaviours”... which are also EACH the GO's type (multi-inheritance)
  - Name is one of few users in 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 actually 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

**Texture**

```
class in unity engine / inherits from object / implemented in UnityEngine.Component
```

Unity vs Others

Unity GameObject

- Transform
- Component
- Variables
- Functions

Non-Unity Actor (C++)

- Component
- Variables
- Functions

- (simplest AND most complex structure)

Structure of Unity’s GameObjects

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

UnityGameObject

- Component
- Variables
- Functions

Non-UnityActor (C++)

- Component
- Variables
- Functions

- (simplest AND most complex structure)

“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, OVRCamera, OVRCameraRig, OVRCamera, PostProcessLayer. Outer GO can be grabbed with GetComponent gameObject
  - But can’t be casted to a class that already knows ALL of the components (aka definition)... GetComponent 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

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

- 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 autocompile anything between scripts without the dev linking them manually (manually defining bunch of GetComponents AND checking for nullcount... 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 (multiplexer, 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
Basic Structure of Other Engine GameObjects & Actors

Object-oriented...not much different from other OOP programs you're familiar with

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 abstraction the C++)
  - BP has no syntax... Only “syntax” is physical structure of the graph

No More of This!

Either not deal with C++ at all or streamline compilation

Towards Blueprints & High-Level Programming...

- Simple 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

Social XR: VRChat & Metaverse

- (Brings Unity is better at in addition to large-scale, procedural AR)
- Balance between optimization and runtime versatility
- VRChat/Second Life: user-created worlds linked by Hub, avatars, framework
- Metaverse: user-created or company-created worlds, linked by user “Identity”
- Very difficult to support user-generated content in traditional game engines;
  requires mixing & matching components not known before runtime
- Unity GameObject more flexible; designed to add/remove components during runtime

Finally an example of exposed functional C++ functions available for BP
**UE4 Treasure Setup**

**Events**
- Engine usually calls things in certain sequence
- Called "lifecycle" of GO

**Important Events/Entry Points**
- **BeginPlay/Start:** called when GO component first starts running in VE (usually when game starts playing) (Unity also has **Awake** for the entire GO)
- **Tick/Update:** called beginning of every frame
- **OnCollision[(Enter/Exit)/Hit]:** called when 2 physics objects collide
- **On[Trigger/Overlap][(Enter/Exit):** called when 2 triggers overlap
- **(in UE4) EndPlay:** called when level ends (Unity has similar **OnDestroy**: But no callback for level end specifically). Used to transfer data between levels (Unity programmers usually use Singletons which is not ideal :/)
- **(in UE4) Constructor:** Used to pre-process stuff, pre-attach, etc.
  - Unity MonoBehaviors have a constructor that can be used to initialize components (e.g. if I have a component that relies on Rigidbody physics, then make sure a Rigidbody is also attached to the GO), but b/c of Unity’s engine passes, you’re discouraged from doing anything that involves other GOs
- **(in UE4, no Unity equivalent) Level Blueprints:** Used to define how Actors relate to each other without modifying the Actor code (e.g. specify current camera)

**In Blueprint**
- You can see events specific to a component without going into the API

**Collisions and Triggers**
- **Collisions/Hits:** physical, Newtonian (e.g. objects bump into each other, stopping momentum)
- **Triggers/Overlaps:** some mesh zone that calls the function when some other mesh enters it (e.g. trigger something when room is entered)
The Functions

- I’ll show BP nodes since black box make explaining easier
- OnCollisionEnter/EventHit: called when objects hit each other
- OnCollisionExit: called when they stop hitting each other
  - UE4 doesn’t distinguish between them b/c UE4 assumes, like in real physics, that a real physics collision can’t cause them to overlap. Collision a single event.
  - For things resting on each other, this still doesn’t really make much sense (rarely describe “resting” position with single contact point...more on this in physics chapter).
- OnTriggerEnter/Actor/Component[BeginOverlap]: called when something enters a trigger zone
- OnTriggerExit/Actor/Component[EndOverlap]: called when it exits

Important Difference between Trigger & Collision

- Collisions have collision info (impact force, impact location, impact normal, etc.)
- Trigger/Overlap events only have info about what they triggered.
- Common source of confusion: Collider vs Collision.
  - Collider in Unity lets you get rid of triggered GO by Collider.gameObject. Used for “Triggers.” “Collider” is just collision box component
  - Collision Object contains Collider AND physics info (used for collisions)
  - Also, Unity only handles collision per-component. UE4 can handle per-actor or per-component

Collisions for Multiple Objects

- UE4: “GetOverlappingActors/Components”
- Unity has Physics.OverlapSphere

Physics.OverlapSphere

Ray-Casting: Key to Game Dev

- Shoot a ray from a position in some direction and figure out what it hit
- Solves tons of problems. One of most common technique for understanding the VE
- We have raycasting, spherescasting, cone casting, etc.
- Like Collisions/Hit, raycasting gives some physics info (at which angle did ray hit? Where did it hit? etc.)
- Really useful for XR b/c it allows flexibility
- Allows for simple projection between 3D and 2D
- UE4 calls it “line-tracing”
- Simplified, controlled ray-tracing

Quick Exercise

- With these functions in mind (collision/trigger, raycast), how can we grab collectibles without hardcoding? There are multiple ways!
- OnTriggerEnter between a collision box on collectible and attached to controller (automatic pick-up or press button)
  - If trigger zone & button pressed, then collect
- When user clicks button, spherescast or raycast to where they’re pointing
  - On ButtonPressed, raycast and collect thing that it hits if it’s a collectible

The Tracing Function

UE4 & Unity differ slightly: Unity asks for ray length, UE4 asks for stop point. Return mostly same info
Demo (Implementing Collecting with TreasureHunter)

Intro to Game Development: XR

XR Camera
- XR: special case of game dev with “camera”/head motion controlled by tracking
  - Outside-in: Vive lighthouses, Oculus Sensor
  - Inside-out: HMD-mounted cameras, visual odometry & sensor fusion to detect motion
- Usually specify an “origin” for XR camera to control spawn point wrt VE
  - VR: calibrated by VR system (usually center of tracking space on floor)
  - AR: usually wherever the HMD is when the app starts
  - Without GameObject for origin, camera would always be wrt VE origin... then entire VE would need to shift to user to be in the right place

Thursday: Realtime Programming with Oculus Quest