C++ gameplay programming in Unreal Engine 4

Samuel Schimmel

samuel.schimmel@digipen.edu

samuelschimmel.com

Official YouTube tutorials

- Live training videos
  - C++ Tanks vs Zombies
  - Creating a Match 3 Game
- Tutorial series videos
  - Introduction to Blueprints
  - Blueprint 3rd Person Game
  - BP Twin Stick Shooter
  - BP Time Attack Racer
  - C++ 3rd Person Battery Collector
  - UMG UI Inventory
  - Introduction to Materials/Level Creation/Cascade

Perforce

- Add these files to your depot:
  - Content
  - Source
  - Config
  - .uproject
- Don’t add these (automatically generated) files to your depot:
  - .sln
  - .vs
  - Binaries
  - Intermediate
  - Saved (client data, like game saves and preferences)
- No “ignore” file in Perforce, so just delete the automatically generated files before adding your project to the depot
  - This is also a troubleshooting step
- Connecting to source control in the editor doesn’t handle C++ files
  - You have to add, check out, and submit C++ files in P4V
  - New C++ classes are not automatically added to the depot

Class naming conventions

- Required prefixes
  - U for C++ classes derived from UObject (but not AActor or UStruct)
  - A for C++ classes derived from AActor
  - F for C++ classes derived from UStruct
- Recommended prefixes
  - BP_ for class blueprints
  - https://github.com/Allar/ue4-style-guide

C++ file management

- Files can only be deleted/moved/renamed from the file explorer
  - Regenerate your Visual Studio solution after doing this
- Give each C++ class its own folder, with the same name as the class, located in the folder of its base class
- Split up large classes among multiple .cpp files

Compiling

- Compile from Visual Studio, not the editor, so you see output
- https://answers.unrealengine.com/questions/3647/how-to-improve-compile-times-for-a-c-project.html

UClass, UObject, and UStruct

- UClass derives from UObject
  - Relationship is inheritance, not instantiation
    - Counterintuitively, UObject is the base class
  - You can have a UClass class, a UClass object, a UObject class, and a UObject object
  - If you derive from UClass instead of UObject, compilation will succeed and then the editor will crash
- UStruct derives from UField, which derives from UObject
  - UObject and UStruct are very different

C++ versus Blueprint

- Use both
- Some things can only be done in Blueprint
  - Level blueprints
  - Material blueprints
  - Animation blueprints
  - Behavior trees
- Some things can probably only be done in C++
- Most things can be done in Blueprint or C++

C++ versus Blueprint

- All of your actors will be instances of class blueprints. You can derive your class blueprints from Unreal's classes (AActor, ACharacter), but it's better to derive them from custom C++ classes which are derived from Unreal's classes.
- Example 1 without C++
  - UObject
  - AActor
  - APawn
  - ACharacter
  - *BP_PlayerCharacter*
- Example 2 with C++
  - UObject
  - AActor
  - APawn
  - ACharacter
  - **AAgent**
  - **APlayerCharacter**
  - *BP_PlayerCharacter*

C++ versus Blueprint

- Gameplay programmer writes logic/mechanics/systems in AAgent and APlayerCharacter
  - Possible performance benefits (see blueprint nativization)
  - Massive complexity management benefits
  - Gameplay logic can be implemented in blueprint, but it will be harder to write, harder to read, and much more verbose
  - Text is also easier to merge, backup, view outside the editor/IDE, move between UE4 projects of different versions, adapt to other engines, etc.
- Content creator adds feedback in BP_PlayerCharacter
  - Audio, particle systems, decals, camera animations, setting material instance parameters, etc.
  - Feedback can be implemented in C++, but doing so complicates C++ code, makes the content creator highly dependent on the gameplay programmer, and introduces asset management challenges
- Dividing code between C++ and Blueprint simplifies both and ensures gameplay programmers and content creators never have to work on the same files

UE4 gameplay framework

- UObject
  - AActor
    - AController
      - APlayerController
      - AAIController
    - APawn
      - ACharacter

Sample custom gameplay framework

- UObject
  - **ULevelManager**, **USaveManager**, **UQuestManager**, etc.
  - AActor
    - AController
      - APlayerController
        - **AMyPlayerController**
      - AAIController
        - **AAgentAIController (same one for all NPCs, as well as player for autoplay)**
    - APawn
      - ACharacter
        - **AAgent**
          - **APlayerCharacter**
          - **ANPC**
    - **AInteractiveObject** (anything that the player can target and use)
      - **AWeapon**, **ADoor**, **AItem**, **AContainer**, **ALadder**, etc.
    - **AProjectile**, **ATrigger**, **ADamageVolume**, **ASpawner**, etc.

Other bases classes worth overriding in C++ (and optionally, in Blueprint as well)

- AHUD
- AGameModeBase
  - Persists for lifetime of world
  - AGameMode adds additional multiplayer features
- UGameInstance
  - Persists for lifetime of application
  - You can override Init(), which is called when the application starts
    - Reset static data here, as it will persist between PIE sessions
- ALevelScriptActor
  - Override to add UPROPERTYs to the level blueprint
- USaveGame
  - UGameplayStatics::SaveGameToMemory
  - UGameplayStatics::SaveGameToSlot
  - UGameplayStatics::LoadGameFromSlot
- UGameUserSettings
  - Allows you to add custom settings to Saved/Config/Windows/GameUserSettings.ini
  - You must edit DefaultEngine.ini to use your derived class
  - Scalability settings are already implemented and exposed to blueprint
  - Key remapping: GetKeysForAxis/Action() and AddAxis/ActionMapping()
- UCheatManager
  - UFUNCTION(exec) for console commands

UPROPERTY macros and specifiers

- Edit/Visible/Anywhere + DefaultsOnly/InstanceOnly
  - Typically used for asset pointers and data fields, but these may get reset when you change and recompile the header file containing them
  - “Components should be VisibleAnywhere, VisibleInstanceOnly or VisibleDefaultsOnly, never editable. Components being pointers, an editable component means you can edit the **pointer** to the component itself rather than the **contents** of the component, for instance allowing the component to be nullified.”
    - https://forums.unrealengine.com/development-discussion/c-gameplay-programming/28537-uproperty-member-vars-reset-to-null-by-objectinitializer
- BlueprintAssignable is used for dynamic multicast delegates
- The reflection system is also used by the garbage collector

UFUNCTION macros and specifiers

- BlueprintPure for const functions
  - Nodes will not have execution pins
- BlueprintCallable for non-const functions
  - Nodes will have execution pins
- Exec for console commands

Dynamic multicast delegates

- In Delegates.h:
  - DECLARE_DYNAMIC_MULTICAST_DELEGATE(FJump);
- In Agent.h:
  - UPROPERTY(BlueprintAssignable) FJump OnJump;
- In Agent.cpp:
  - OnJump.Broadcast();

Dynamic multicast delegate parameters

- DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FDelegateName, paramType, paramName);
- You must pass TArrays by const reference

C++/Blueprint communication

- Dynamic multicast delegates
  - C++ telling BP that something happened
  - All of your game logic should be implemented in C++, so expect to have dozens of these
  - To promote consistency, consider having a header file just for delegates
  - As delegates are the cornerstone of your C++/BP interface, their design deserves special consideration as well as communication with content designers
    - Delegates should be general enough to give content creators flexibility
    - But specific enough that the delegate handlers in BP don't require lots of additional logic
- BlueprintCallable UFUNCTIONs
  - BP telling C++ that something happened
  - Expect to have few of these, usually for telling C++ that an animation finished
- BlueprintPure UFUNCTIONs
  - Const getters for BP

Data and asset management

- Asset pointers and data fields exposed to the editor using the UPROPERTY macro may get reset when you change and recompile the header file containing them
- Data
  - Write a class to deserialize design data from spreadsheets
- Assets
  - Use ConstructorHelpers::FObjectFinder to get asset pointers in C++
  - Can only be used in constructors
  - Wrap in a static function
    - Include error handling
    - Cache results for efficiency
    - Templatize to support BP classes, static meshes, etc.

Templatized asset lookup with caching

```
template<typename T = UClass>
static T *GetAsset(const UObject *object, const FString &path)
{
	if (!object) return nullptr;
	if (object->GetName().StartsWith("Default__")) return nullptr;
	static TMap<FString, T *> assets;
	T *asset = nullptr;

	if (assets.Contains(path))
	{
		asset = *assets.Find(path);
	}
	else
	{
		const ConstructorHelpers::FObjectFinder<T> finder(*path);
		asset = finder.Object;
		if (asset) assets.Add(path, asset);
		else Log(FString::Printf(TEXT("Failed to find asset: %s"), *path));
	}

	return asset;
}
```

Templatized asset lookup with caching

```
const auto agentClass = GetBPClassName(this);
const auto prefix = "DT_";
const auto asset = prefix + agentClass + "Melee";
const auto root = "Class'/Game/MyContent/DamageType/";
const FString path = root + asset + "." + asset + "_C'";

meleeDamageType = GetAsset(this, path);
```

Getting the BP class name of an actor

```
FString GetBPClassName(const UObject *object)
{
	if (!object) return FString();
	auto name = object->GetClass()->GetName();
	name.RemoveFromStart("SKEL_");
	name.RemoveFromEnd("_C");
	return name;
}
```

Getting the instance name of an actor

```
FString GetInstanceName(const UObject *object)
{
	if (!object) return FString();
	auto name = object->GetName();
	name.RemoveFromStart("BP_");
	const auto index = name.Find("_", ESearchCase::IgnoreCase, ESearchDir::FromEnd, name.Len());
	if (index != INDEX_NONE && name.Right(name.Len() - index - 1).IsNumeric()) name = name.Left(index);
	return name;
}
```

Memory management

- Don’t use new or delete
  - Create objects with NewObject, SpawnActor, or SpawnActorDeferred
- UPROPERTY macro
  - An object will not be garbage collected if at least one UObject has a UPROPERTY pointer to it
  - If an object is destroyed, all UPROPERTY pointers to it will be nullified
  - May have performance implications
- TWeakObjectPtr
  - Simliar to UObject pointer with UPROPERTY macro, but will not prevent objects from being garbage collected
- Containers
  - TArray, TMap, TSet
  - Don’t modify TArray in range-based for loop

Coroutines and tick

- Unlike Unity C#, UE4 C++ doesn’t natively support coroutines
- Plugins like SkookumScript can provide them
- If you need to interpolate a value over multiple frames, you must use Tick() or ideally a function called from Tick()
- https://www.gamasutra.com/blogs/ZachBurke/20170310/293425/The_Death_of_Tick_UE4__The_Future_of_Programming.php

Behavior tree AI

- Perception
  - Use for vision and hearing
  - Debug with ‘ and numpad
  - UPawnSensingComponent is deprecated, use UAIPerceptionComponent instead
- Write nodes in C++ and compose trees out of them in the behavior tree editor
- The built-in AIMoveTo node will not work with behavior trees, so you’ll have to make your own if you want to combine AI with level BP scripting
  - https://wiki.unrealengine.com/Creating_Asynchronous_Blueprint_Nodes

Behavior tree AI

- Tasks derive from BT_TaskBlackboardBase
  - virtual EBTNodeResult::Type ExecuteTask(UBehaviorTreeComponent &ownerComponent, uint8 *nodeMemory);
- Services derive from BT_ServiceBlackboardBase
  - virtual void TickNode(UBehaviorTreeComponent &ownerComponent, uint8 *nodeMemory, float deltaSeconds);