C++ For Games Designers – The Oath of Truth: Technical Breakdown

The Oath of Truth Overview

The project made for this module is a simple stylized, medieval-themed platformer that is focused on movement, dodging hazards, and utilizing various pickups found throughout the world. The key features implemented in C++ are turrets, projectiles, various pickups such as health, speed and keys, doors of different types including automatic and locked, launch plates, moving platforms and lava. Player progress such as overlapping a checkpoint or level end actor is all managed through the use of a Progress Manager.

C++ and Blueprint Integration

For my project, I put an emphasis on keeping core gameplay functionality within C++, while exposing select variables and events to keep to the industry standard of allowing designers to tweak these without having access to logic that could result in breaks or inconsistent gameplay elements. I did this in various ways through my code; the first one being having RespawnPlayer and HandlePlayerDeath within the game mode have the BlueprintCallable and BlueprintImplementableEvent specifiers respectively that allow for tweaking what happens when the player dies within blueprint editor, including widgets and timer, and then calling respawn. The core elements of respawn and death are managed in C++, and blueprint is used for finetuning these functions and how they are displayed to the player.

Another example is in pickups – in my BasePickup actor, a function with the specifier BlueprintImplementableEvent exists so that designers could add in niagara, SFX, or UI. Another example is through exposed UPROPERTY's - in the turret and projectile, variables with the specifiers EditAnywhere and BlueprintReadWrite are present so that the fire rate, damage amount, lifespan are editable. The TArray for the moving platform path points is also visible so platform movement can be tweaked on an instanced basis.

Another area is combining ABP with variables in the PlayerCharacter – bIsInAir and Speed are defined in CPP and then referenced in the ABP. Research was conducted into what is the best practice for combining blueprints and C++. Epic Games documentation recommends that the core functionality of a class is coded within C++, and then is tweaked by designers in blueprint. This is shown in documentation[1] where Epic Games demonstrates creating core functionality for a light switch and exposing a DesiredIntensity variable with the EditAnywhere and BlueprintReadWrite specifiers for a designer to tweak.

In terms of some of the main advantages of C++ compared to blueprint, performance wise C++ code runs faster at runtime than blueprint, as it compiles down to the native machine code.

Another reason is that blueprint can become visually cluttered, and hard to sort through. C++ also gives you access to features not available in blueprint, some of these being custom class inheritance, writing custom memory management and optimization of data structures. The main advantages in blueprint are the use of rapid prototyping and iteration – this was used in my project to quickly prototype mechanics and gameplay ideas before then converting it to C++. Blueprint is also arguably more intuitive and easier to understand – the node-based interface makes it easier to grasp logic compared to C++ which can be harder to understand due to syntax and compiler errors. Blueprint being more visual, although can cause clutter, makes working with visual aspects of game development such as UI and animation much more efficient, and works well when working with smaller systems or functionality.

Quality of C++ Code

In terms of class structure, base classes were created for gameplay elements that would have multiple variations. For example, a BasePickup was made which includes all base pickup functionality, and then children would override the PickUpEffect function, including their own functionality using the PickUpInterface.

The main areas in which I refactored my code was through the removal of debug logs and including comments. As a beginner, I found the use of both debug logs and comments very useful, and I made sure to utilize them throughout my code. Once the project was complete, I went back through my code and added in additional comments to ensure anyone reading my code would understand what functionality does. This is especially important for actors such as the ProgressManager or the PlayerCharacter where they are referenced a lot in other classes. I personally found it a bit tricky to remember what functions go where, however once I started commenting on my code more it made it much easier to return to my project and pick up where I had left off. One area where my comments could have improved is separating blocks of code into smaller chunks – I focused more so on commenting on what functionality does, but I should have also used code to separate functions and components from one another.

In terms of naming my functions, I made sure to choose sensible names for them which describe what they do. I used a previous project where I focused on systems design in blueprint as the base for my code, and followed the naming conventions used within that project which are all descriptive and indicate the purpose for functions at a glance.

Throughout the development of my game, I made a consistent effort to read up on memory management techniques and to reuse the knowledge of what I know from studying blueprint code. The first thing I looked at was the MinimalAPI specifier – I researched it and then applied it to my Pickup Interface which reduces binary size and cuts compile times. I followed a tutorial to create a projectile and afterwards found EpicGames documentation that allowed me to incorporate SetLifeSpan into the projectile which ensures the game has efficient performance and memory management. I made an effort to turn ticks off for actors that do not need it. I also made sure to use an interface for my pickups, as casting to the player is expensive at runtime.

Quality of Game Feel and Mechanics

For most interactable objects within the game, I made sure that a designer friendly workflow was used - sound, niagara and UI was used where necessary in a mixture of both CPP and blueprint. All pickups use exposed variables which allows designers to alter their values, in speed, health or number of coins picked up. The turret is also editable by designers, the fire rate editable alongside toggling a bool which determines if it fires on beginplay, allowing for flexibility in its design. For doors, the door open speed is editable. There are also blueprintimplementableevent specifiers used throughout, to allow designers to add functionality with cosmetics. For example, I practiced this by handling Niagara within blueprint within the checkpoint and pickups. The use of these makes the game feel more interactable for the player and makes it easier for designers to tweak only the necessary variables without accidentally causing inconsistencies within actors. For example, the health pickup is BlueprintReadOnly to avoid inconsistent behaviour and encourages designers to instead create children that have the required values, rather than just having one actor with lots of different values that would make the game harder to manage and feel bad for players.

Overall Quality of Work Produced

Overall, I am very satisfied with what I was able to produce for this project. As someone who has enjoyed learning and improving their skills in blueprint, I have always wanted to learn another programming language, yet my lack of knowledge in programming made me feel intimidated, as I have never done any coding besides blueprint. This project has opened the door for me to begin utilizing C++ in my work and will also aid me as a designer in the industry to work closely with tech and understand their workflow better. I plan to expand upon the project I have created from this module or create a new one, in which I can practice more complex coding to boost my employability. I think the biggest obstacle for me was the progress manager, as I found the number of actors and components linked to it overwhelming at times to keep track of. I found that writing more

comments within the code, even ones that felt redundant, helped me to better understand all the functionality when coming back to continue to work on it.

In the progress manager, there was also a lot to consider, for example when it came to SFX, UI and niagara, I only wanted these to trigger if the checkpoint was a part of the progress manager's array – a lot of these aspects were time consuming and took a lot of trial and error to get right. I found simply adding more debug logs helped – I had an issue with the sprint function working properly, where the speed boost pickup would work as intended, yet when I toggled sprint, it would overwrite the changes to the character movement speed. I found the solution by adding a debug log before and after both the pickup and the sprint toggle. This showed me that the sprint toggle was resetting the speed back to its default value. To fix this, I created a new function within the player that consolidated all the functionality for calculating the player's movement speed in one place and calling it whenever the character's movement speed was changed, either by pickups or sprinting. Not only did this solve my problem, but it also made my code more concise by having all movement variables calculated within a singular function which improves readability.

I initially also found aspects like memory pointers confusing; however, I ensured that I spent time researching them using class content, online resources and traditional methods. I found the book 'Modern C++ for Absolute Beginners' very helpful as the exercises allowed me to put what I had learned into practice, and I soon became confident working with these new concepts introduced. I also expanded upon the topics taught in class; with some I posted to my work forum thread using Epic Games documentation and tutorials, and my own trial and error.