Scorched Fur - Vertical slice with C++

Higher vocational education degree project.

Mikaelah Jenkins
Gothenburg

Yrgo - Game Creator Programmer
Fall-2023

Special thanks to my teammates Max Petersson and Tom Eckerström for sending the best cat gifs during the project.

Table of contents

Introduction

Purpose

For me this project had two main goals. The first was to create a vertical slice/ demo of a soulslike combat game in Unreal Engine 5 (referred to as “UE5” or “Unreal” from this point). The second goal was to learn C++ and its implementation in UE5.

Background

The game idea was originally from a previous project where I brainstormed ideas with Tom. He wanted to make a soulslike while I wanted to play as a squirrel. We then merged these two concepts. Our main inspiration for combat is *From Software* games such as *Elden ring* and *Bloodborne*. At first we wanted the art to be hyper realistic and dark fantasy but we then changed it to the opposite, very stylistic and colourful, because we wanted it to have a more unique quirk than just mimicking the dark souls games. We took the game *conker's bad fur day* as a reference to the artstyle. We then recruited Max who had worked with a very similar project before. Task division was established before the project started: I focused on the boss fight, Tom on the player character, and Max on enemies. The resulting linear demo, featuring combat along a path leading to the boss, showcases our core ideas within the given constraints.

Questions

The main questions I had were:

- What is the workflow for blending C++ and blueprints in UE5
- How do i mimic the core elements of the bosses in *From Softwares* games

Limitations

We have purchased assets due to not having any artists from school for this project and were therefore prepared for limited animations and inconsistent art. All assets, aside from the main character, used in the project have been acquired either through purchases or free downloads from the Unreal Engine Marketplace. The main character was made by an external artist separate from Yrgo.

Due to the deadline and the perceived difficulty of using C++ in UE5 we decided to keep the scope small, with the potential to expand if time allowed.

Implementation

Setting up the project

We decided that each person would choose for themselves how much C++ they wanted to use in the project. Max and I were eager to kickstart the C++ and create our first classes in the engine. However, the entire first day was spent trying to configure Visual Studio 2022 with Unreal. We encountered unrelenting issues with intellisense where it would either have to be disabled completely or give us loads of errors as it failed to recognise Unreal's coding conventions. We found out that this is a common issue developers face during setup of Visual Studio and Unreal but even so it took more than a day to be able to write our own code.
When we finally figured it out and could create our own classes it was time for the next step. I started thinking about what would be needed and which parts I wanted to implement using C++ and what could be done through blueprints. The challenge was finding the optimal balance between the efficient development with blueprints and the slower path of learning C++.

Boss analysis and research

To examine the diverse and effective boss design, I started by watching playthroughs of various boss fights in *Elden Ring* on youtube. Analysing their attacks, speed, what indicators the player would react to, and understanding the triggers for specific attacks. I mainly looked at bosses with animations similar to the ones I had available.

The state machine would require the following states:

- Idle: Passive state before the fight starts.

- Aggro: The player has entered the arena and started the fight. Here it can choose to move towards the player or choose an attack or a whole combination of attacks based mostly on distance to the player. It uses an array of data structures, storing attack state names, maximum distance and attack weight. The max distance is for being able to choose the attack while the weight is used to influence how likely that attack is to be chosen.

- Attack states: Individual states for each type of attack, mostly immobile, triggering a blueprint function in the EntClass to initiate animations. Most of the logic is then run from notifies in the animations through blueprints.

- Staggered: Stops all movement and runs the staggered animation.This state returns to Aggro after either a timer has run out.

- TransitionPhase2: Plays animation and does not move during this. The player can attack but will be hurt from fire damage when close. Additional attacks and combos are now available. Materials and VFXs are changed to their fiery variants.

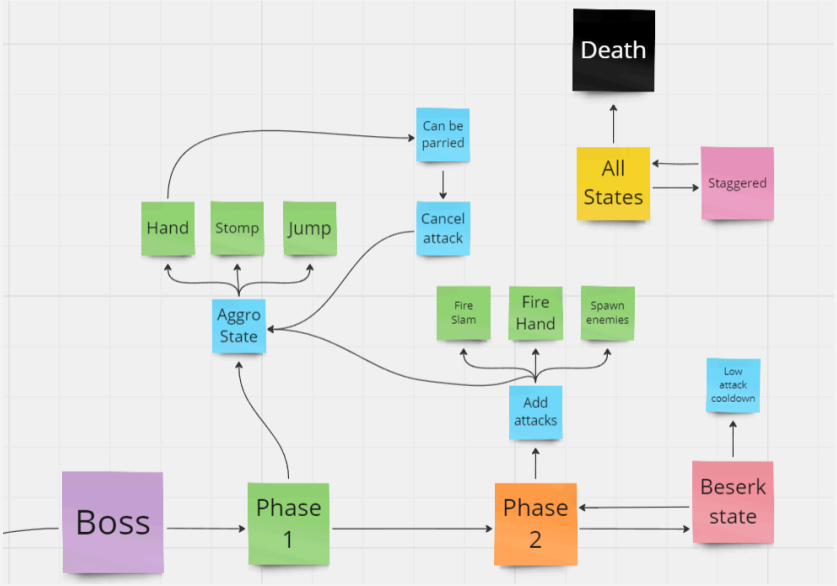
- Death: Starts the death animation and disables movement, ends the demo when finished.

Not required:

- ComboAttacks: Gets a set sequence of attacks and runs through it going through each state in succession until it is completed. Can only be interrupted by staggering or getting the Ent to 50% health where it will start its transition state instead.
  Combos were not essential but I felt they added some consistency to where the player could possibly learn combinations and manoeuvre around them.

- Enrage/ Berserk: Has a chance to go into this state when close to death. Does almost random rapid attacks which makes it unpredictable. A sort of last stand.
  This state was a bonus only to be added if there was time for it, it was not necessary for the demo to work but would give some valued variety to the fight.

Flowchart of the states:



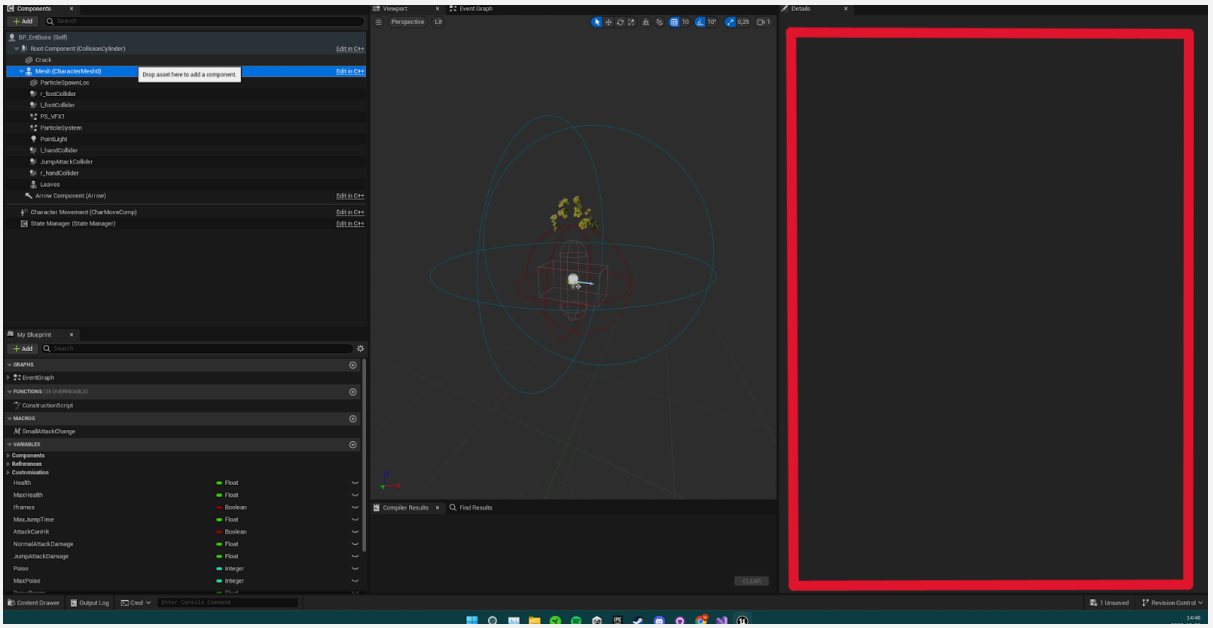
The state machine

After establishing the base states that would be needed the next step was to start implementing them. All states are derived from one state, the EntStateBase. This state has the base functions that all states will have: OnEnterState, OnExitState and TickState. The first two execute on state entry and exit respectively. The TickState is supposed to run continuously on tick but I choose to not use this in any state as the few things I needed to do on tick could be easier managed in the C++ class of the Ent.

Max and I followed a tutorial for a state manager and tried to figure it out together. While a lot from this video had to be scrapped soon after due to some issues, we could finally set up the state manager correctly. This manager built the foundation for our separate state machines all of which could interact with the same state manager. The manager handled the state transitions, allowing classes to call a function with a string key of the chosen states name to switch state. This is very modular and it's easy to add all the states needed as you only need to add a name and drag and drop its state in the state manager component, shown in image.

Behaviour development

I then created the C++ EntClass, which would have the functions and references that the state classes would refer to and access. Once this setup was done I decided to take a small break from the C++ and the problems UE5 had with it and started working on some base functionality in the Ents blueprint class.
I soon discovered the importance of correctly specifying which functions and variables would be blueprint accessible and how they needed to be set up. A C++ function could not be accessed by the blueprint unless specified while a blueprint function could not be accessed from its C++ class at all. Therefore I decided to add most of the functions in C++ because it allowed me to call them from its states when needed. For example each attack state would call the function in the EntClass to start its respective attack, this function would be run in the blueprint class where it disables movement and tells its animation blueprint to start the animation. The animations then trigger notifies for when it should hit the player, start a particle system or when the animation ends. These notifies mostly check bools so that either the EntClass or its blueprint class knows when to trigger certain events, for example when to send damage to the player. In hindsight I would have wanted to do more of this in the Ents C++ class. It would have been a lot cleaner and there are some functions in the animation blueprint that would've made more sense to have in the C++ class. For example the spawn enemy attack that it has. Some of the logic for spawning in roots is in the animation blueprint while some is in the C++ file and some is in the blueprint class, this is something I might reorganise in my freetime to improve the clarity and maintainability.
A problem that happened twice to me and once to Tom was that somehow the blueprint got disconnected to its parent C++ class. The Ent would look like the picture below in the viewport and I couldn't edit the state machine nor any of the main components such as the character mesh or the root capsule.



Marked in red is where I should be able to edit the component. We couldn't find any solutions for this problem except for creating a new blueprint or going back a few branches in GitHub. It was to either redoing the blueprint copying and pasting components and nodes or going back and redoing things that would go lost.

For the Ents spawn enemy attack I made some stationary roots that come out of the ground to attack the player.
As I had started to get more familiar with Unreals C++ I chose to do almost all the roots logic in its C++ class. I also felt safer to do more of this in C++ because it was not really an essential nor large class, if nothing worked I could easily change to blueprints and would not have to redo as much as I would have to if the Ent broke down. I also made its animation blueprint from a C++ class so that I could easily reference it and could go directly between the RootClass and its animation class and skip using its blueprint class as a middleman.The roots also use the state manager with its own states and works basically the same as the Ent. I had almost no references or variables in blueprints for the roots at first as this was much cleaner and easier to keep track of. This became an issue quite fast though as it turns out a C++ reference to the animation C++ instance changes in the blueprint view to a “Livecoding” reference as soon as i compile the code using livecoding.

This is how it should look but after compiling it would say LIVECODING Root Anim Instance as the variable type.
This then breaks the blueprint references I had as they are no longer the animation reference, it is now a livecoding animation reference. This is solved by first compiling, closing Unreal, deleting the binaries folder, start the project where it will rebuild the binaries. Then it will work again, until the next time you use live compiling. I did not find any other solutions for this when working with it but there may be a fix out there that I didn't spend enough time to find. This was only a problem with references to either the animation class instance or the root class, I assume this is because the livecoding somehow makes its own instance of the reference and then doesn't understand that it is the same thing as before it was compiled.

We made a collective C++ interface for all the actors to use for taking damage. It declares which parameters would be needed to send when doing damage.

```
UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "DamageInterface")
void TakeDamage(float DamageTaken, float Poise, bool FireDamage, float KnockbackValue, FVector KnockbackSource);
```

This could easily be implemented in all classes and could be used in both C++ and blueprints, this was very convenient as we could individually choose to implement it in C++ or blueprints. I did this in blueprints on the Ent and C++ in the RootClass, looking back I would have wanted to do this in C++ on the Ent as well but this would have taken more time and would leave less time for the polishing of the project. This is also something I might look into changing in my freetime.

Result

The main point of this project was to make a vertical slice/ demo of a soulslike combat experience and I'm very happy with our end product. I'm especially happy about my part in this. While the boss of course could be better and more refined it still turned out better than I expected and it is a solid foundation.

I managed to combine both C++ and blueprints which was the other primary goal and we learned more about the workflow of C++ development in Unreal Engine 5. This said both me and Max had many issues that seemed like things a lot of people have problems with but no real fixes have come from the UE5 developers. For example the projects binaries folder. Whenever there is an error without a clear path in the code the first step has been to close down Unreal, delete the binaries folder, restart Unreal. Both this and the minor errors that will crash Unreal result in multiple restarts per day. This might not be the typical workflow for studios in the engine, but we simply didn't have the time to find out if this is the real world workflow or not. Making the demo feel complete was of higher priority for me at least.

Having an artist with animation skills on the team could have further improved the end product. A lot of time has been spent on cutting and merging parts from different animations together to create new ones. I've also spent many hours on VFX and figuring out Unreals Niagara particle system. Even though the time could've been spent on refining the logic it has still given me very valuable experience with the engine. I've gone from basic knowledge about Niagara systems to being able to easily manipulate them to fit my purposes. The same goes for UE5s animation blueprint systems. I now know how to blend animations and can reuse parts of assets to build new animations for different purposes.

In conclusion, this project has been a valuable learning experience. The insights into C++ and UE5 will aid me massively in future projects and games, especially but not exclusively in UE5. Every day has been something new and if I were to restart the project with the knowledge I have now it would have gone even further. I might work on it more in my freetime to achieve the "perfect" boss, but I am extremely proud of my team and the resulting product as it is. This demo has been intense but, above all, very fun to create.

Bibliography

State Machine Tutorial:

State Machine From Scratch Tutorial for Unreal Engine (C++) - Heavy Beat - ...

Unreal Engine Documentation:

https://docs.unrealengine.com/5.3/en-US/

Unreal Engine Forums:

https://forums.unrealengine.com/categories?tag=unreal-engine