Spellshooter

Game programming IMT 3601

Members

Name	NTNU ID	Mail	GitHubID	GitLabID
Andreas Blakli	andrbl	andrbl@stud.ntnu.no	TrustworthyBlake	andrbl
Vegard Opktivtne Årnes	vegardoa	vegardoa@stud.ntnu.no	VitriolicTurtle	AlphaBoi
Theo Camille Gascogne	theocg	theocg@stud.ntnu.no	iaminadequate	iaminadequate

Group name: G6

Date: 19.12.2020

Table of Contents

Information

Repo link: https://github.com/VitriolicTurtle/kcnGame

Game engine: Unreal Engine 4.25.4 GitHub release, link:
https://github.com/EpicGames/UnrealEngine

Game theme: FPS multiplayer with dedicated server.

See the readMe.md file in the repo for instructions on how to run, start the server, game and instructions on how to play it.
Direct link to readMe.md in repo:
https://github.com/VitriolicTurtle/kcnGame/blob/main/README.md

Video link for gameplay showing off the important parts of the game: todo
Video link for showing off the code that is tightly integrated with the game engine that is difficult to see from the text of the programming:
https://drive.google.com/file/d/1Fal30wCLYECyP797hIjye-b3Po-RVRSA/view?usp=sharing

Video presentation of gameplay: https://www.youtube.com/watch?v=sGTei1Su9AU

Download links for the game and server executables:
Game:
https://drive.google.com/file/d/1l8zfD6XNvAMluZA1Z6ADgM3W0rV2tlsy/view?usp=sharing
Server: https://drive.google.com/file/d/1uTCwMoHf7vcm8HzHuOw5mwY-15sYNFFu/view?usp=sharing

We highly recommend making a shortcut for the server executable and add -log to the end of the path in the properties section of the shortcut, this makes it easier to close the server and see debug messages.

It is important to note that the server must be manually restarted when the game mode finishes i.e. when a player wins the game mode.

1. Planning

We started our project by deciding what we wanted to make. We had a brainstorming session on discord where we explored different ideas and which engine we wanted to use. We ended up deciding we were going to make a multiplayer fps in the Unreal engine with a dedicated server to host the game.

1.2 Game Design

Backstory

Our idea was a sci-fi game set in the future on the planet Mars, where humans had discovered intelligent life. That intelligent life had access to some incredible powerful magical powers, the humans were amazed, and that amazement turned into jealousy and hate. The humans wanted the newfound amazing powers, but Martians would not share it because they meant humanity were not ready for it, so naturally a war broke out between the Martians and the humans. Poor Martians, human greed has just no end.

Game setting

The game is set on the planet mars where humanity have invaded the Martians and a full-on war between the humans and Martians have developed. You the player spawns in on the map and chooses either to be an alien Martian or the invading human. Humans use normal guns for combat and the Martians use magical abilities.

1.3 Covid-19 and Communication

Due to covid-19 and the social lock down communication between us got a lot more difficult as we decided not to have physical meetings to help limit the spread of covid-19. This meant everything had to be moved to a digital platform. We choose to use Discord as our main communication medium. We mainly communicated with each other through text messages and screenshots to explain what we had been working on, how the logic worked and how it was implemented. We sat up voice sessions and screen share sessions when we felt it was necessary. The limitations of doing all the communication digitally became quite clear early on as the communication got slower compared to a physical session where you can speak directly to the person next to you, use gestures and write on a whiteboard to quickly explain ideas. That being said, we think we managed this situation well after some trial and error.

1.4 Version Controlling and Repository Management

Unreal uses a blueprint system which is a GUI and represent the programming in a graphical way, these files are in a binary format. We quickly realized that merging these blueprint files were going to be an issue so we quickly decided that when a person was editing a blueprint,

they would tell the rest of the group, so everybody was aware of what was being worked on so we wouldn't end up with merging conflict e.g. pushing back old code to the repo. A local backup of each version of the game were created, so in case the repository broke we could always revert it back to a previous working version.

1.5 Organization

Andreas Blakli would be responsible for setting up the dedicated server, UI, menus, and all logic relevant.

- Writing all the logic for the multiplayer
- Projectiles, spawning, projectile vector calculations, player health etc.
- Synchronization and replication of data between clients and dedicated server
- Game mode e.g. free for all.
- UI elements: Player HUD, pause menu, victory screen, death screen, winning player name broadcast to all connected clients.
- Main Menu map and UI.
- Lobby map for clients to be placed in, until enough clients have connected to the server for then start the game.

Vegard Opkvitne Årnes would take responsibility for gameplay abilities, human weapons (rifles), alien weapons (magic), weapon selection UI, and human animations.

- Creating initial playable & animated human character
- A creating versatile controller universal to human/alien
- Adding bone transformation based on pitch input from mouse (look up/down)
- UI element: Adding an item/weapon selection menu with different content based on race
- Adding weapon and magic assets which are dynamically equipable through weapon selection UI.
- Create main menu background.

Theo Camille Gascogne would be responsible for the caster class, setting up a test map, basic caster animations and class change.

- Implementing controls and logic for the caster class
- Animations for caster
- Implementing UI that allows class change on the fly

- Switch between what actor the player is currently controlling
- A basic map for testing

2 Making the Game

2.1 Assets

Persons responsible: Theo Camille Gascogne, Vegard Opkvitne Årnes

Assets from the game were found on mixamo, where 3d-models and animations can be imported to any unreal project. This also includes textures for objects and said models. Spell effect were found on the unreal marketplace for free. The map required no textures.
Also, while First Person Shooter games often have a set of arms only visible to player with a full body only visible to other players, we decided to go for full body model as both what player sees and what other players see, this was due to lack of alternatives with only arms.

Challenges

Challenges with the assets was getting them to work properly. Animations were wonky due to mixamo animations having no proper root bone, any animation that requires movement would awkwardly center around the hip bone while the body would shake around it when the hipbone stood perfectly static. Rigging the animation to the payer's input was rather non intuitive. Unreal engine has an interface where the user can set animation based on directional changes, and the animation would transition between the animation states. Setting forward sprint to the positive extreme while backwards running to the opposite would create a bug where the model would attempt to play both animations simultaniously.

Reflection

In the end, the animations were set up in an adequate manner. Had we known of the improper root bone problem before implementing the 3d models into our project, we could have saved the trouble of trying to manually fix the animations. While the fixes was quite trivial to implement trough check boxes in the animation blueprints, this serves as a lesson to properly check what the tools and interfaces does in unreal before putting in effort to fix the issues.

2.2 Level Design

Persons responsible: Theo Camille Gascogne

Nothing much to say here. The main idea was to add different obstacles to test player movement and mobility as well as jumping mobility with some platforms. The map is purely geometrical with the basic building blocks Unreal engine offers. For aesthetic purposes we added a orange space skybox and some lighting.

Challenges

Challenges here include what functionality the map should have to be able to properly test player movement

Reflection

The map was a low priority for us as gameplay was more important, however the map is still a part of gameplay so we should have given it more priority

2.3 Gameplay Abilities, Weapons and Projectiles

Persons responsible: Andreas Blakli, Vegard Opkvitne Årnes

2.3.1 Projectiles

Person responsible: Andreas Blakli

Projectiles are quite widely used in this game since it is an FPS. We sat up the main logic for the projectiles in C++ so that all we would need to do was make a new blueprint of the C++ code to make new projectiles with different velocities, damage values and shapes (models).

Challenges

The main challenge we encountered with projectiles were the multiplayer part of it which we discuss in section 2.4 Networking and Multiplayer. Some minor challenges where to figure out how to do the collision detection and destruction of the objects on impact.

Reflection

We felt this aspect of the game went well and the code is set up in a modular way, so it is easy to create new projectiles with different properties. As mentioned earlier the problems came when integrating the projectiles with the dedicated server.

2.3.2 Classes and control change

Persons responsible: Theo Camille Gascogne

Our idea of the game was to have the humans as shooters and aliens as casters and let the player pick which one they wanted. In the game, pressing f will make a menu appear that asks what class the player wants to be. After picking one, the player will re-spawn as their chosen class. This was implemented trough blueprints and some C++ coding.

Challenges

Setting up the class change outside of servers was very trivial. A simple blueprint logic was enough. But this became a rather large problem once clients and server were introduced. It turned out that certain aspects of unreal engine will work on server but not on clients and vice versa. To figure out which blueprint logic goes where was quite difficult and unintuitive, most of the time there was a problem with what blueprint class one could cast to or get the controller of. An example would be widget and the player class. Widgets are client-side only,

so UI elements go there, but then how do you register this choice and send this to the server when the registration is client-side and casting is not an option. Such problems also appeared in the other direction, where the server would duplicate entities more than once for each client.

Reflection

Looking at the logic now after completion, it is very easy to implement once we figured out what logic goes in client or server. Learning this and knowing what goes where was not a simple task due to poor documentation of Unreal's client and server-side blueprints and lack of tutorials focusing specifically on the issues we dealt with.

2.3.3 Gameplay Abilities/Weapons

Person responsible: Vegard Opkvitne Årnes

Weapons can be gained in the game by selecting them from the specific weapon selection and the system is therefore heavily tied to the UI logic as can be seen in section 2.6.1. Each of the weapons you are able to select in the Item selection UI is tied to a custom-made "Data Asset" with a parent C++ class named "spellshooterItemWeapon". The Data Asset system is a versatile and structured manner of storing multiple variables and is therefore responsible for holding the features we want it to have. An example of this can be seen in the AK-47 weapon:

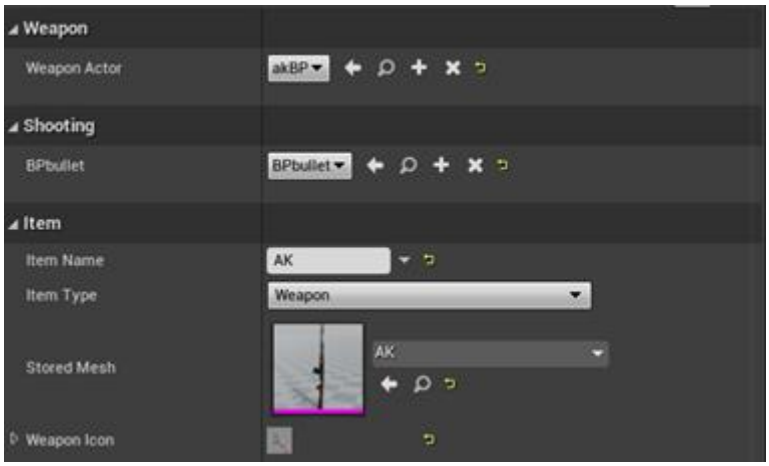


Figure 1

Upon selecting an item from the shop, the Data Asset is passed to the player controller spellshooterPlayerController. The controller takes the stored "Weapon Actor" and "BPbullet" actors, then spawns the "Weapon Actor" in the appropriate skeleton sockets on either human or alien, based on what they chose. The "BPbullet" actor is passed into the bullet variable stored in the human/alien classed and is thereafter shootable through the functionality created explained in part 2.3.1. As both "Weapon Actor" and "BPbullet" take blueprint actor classes,

the opportunities are endless in terms of what they can provide to the player as each separate blueprint can hold any functionality compatible with actor. However, if the “Weapon Actor” happens to have Mesh which is supposed to spawn, it is currently only equipable in the weapon socket.

Challenges

Initially the challenge was how the weapon system would differentiate between human and alien in terms of Weapon Actor type, and second how the different weapon types could have the most variation in aesthetics and damage type. Differentiation between human/alien in the logic got solved through making the functionality of weapons/bullets universal for humans and aliens by not having the logic surrounding equipping weapons in either of their blueprints, instead it was done by having the player controller handle it. Through this the player controller checks the controller pawn to decides what skeleton socket to spawn weapon in. The challenge of different attack types, in particular the difference between magic and firearm bullet, was solved through the versatility for the aforementioned Bullet class which all bullet/magic attacks are derived from. In the case of magic attacks, the attack Blueprint, e.g. “elecBP” has a child actor which is spawned with the bullet actor, which holds the more complex graphical elements as well as possible additional functionalities. A minor challenge that is also worth mentioning was to make the different rifles fit the human model, in particular make the models left hand properly hold the weapon grip. The solution idea was to give each weapon skeleton a socket the arm would automatically attach to so that it would fit each weapon, but in the end the solution was to give the left hand a set position and adjust weapon position to look acceptable. The consequence with that solution is that the weapon floats above left hand slightly when walking sideways, as well as the “stock” of the weapons slightly clipping through the alien’s shoulder. Conclusion to this issue is simply that the weapons were made completely independently of the 3d model we used for human, so they were not very compatible.

Reflection

We feel like more time was spent developing a system that is well made to handle new additions of weapons, rather than creating unique abilities/weapons to increase the quality of the gameplay. The most important elements were implemented as both human and alien have a basic attack that is easy to change and add functionality further to, and in addition its easy to add more such abilities to the shop if we found more room to prioritize it. Regardless, the task had immense learning value as every part of it had to be dynamic in that human/alien

attributes had to be accessible for change during gameplay, and required a firm understanding both C++ and Blueprinting for the execution. If we had more time we would expand the functionality by creating more sockets on the human/alien skeleton and adding more upgrades in the shop that could be attached to those different types of sockets and provide different upgrades, thereby increasing the gameplay quality and replay value.

2.4 Networking and Multiplayer

Person responsible: Andreas Blakli

The main goal of the multiplayer aspect of the game were to have a running dedicated server which the players could connect to. Our final goal was to have the server running on Amazon's AWS service so anyone in the world could connect to the server, play and interact against/with other players.

Challenges

Setting up the dedicated server was "to put it nicely" an absolute pain. Personal note from Andreas Blakli: "I have never spent so much time on a programming project with so little return on time invested". The first challenge was Unreal itself, it requires you to pull the entire engine from Epic Games GitHub release version to be able to build and package a dedicated server executable file. Then run a series of scripts which downloads files required for the building of the engine. Then Visual Studio had to be configured with the correct settings, packages, and downloads so that the project would compile. When Unreal was fully built locally it was close to taking up 100Gb of diskspace. We then had to run Unreal from Visual Studio.

It required a lot of work implementing multiplayer into the game because the data of the clients needed to be replicated and synchronized with the dedicated server which required a lot of work. E.g. vector calculations for projectiles i.e. the direction, velocity, collision etc. What needed to be handled by server, what needed to be passed between the clients and the server. Figuring out how to do this in C++ was difficult as the documentation for this is poor. Examples on how to this in the blueprint system existed, but we wanted to actually program it ourselves to get a greater understanding of what was going on in the code and how Unreal itself worked.

The debugging process were extremely slow, each time we did a change, the project would need to be rebuilt in Visual Studio, then the dedicated server needed to be repacked which averaged out at 5 minutes just to test a new feature.

Due to time constraints, we were unable to deploy the server on Amazon's AWS service. Which means the server runs locally over LAN. So anybody connected on the same network would be able to connect to the server and play with each other. A VPN is a potential workaround to play with friends from a different network. We have setup the code so we can easily deploy the server later on, all that needs to be changed is the open level address to whatever IP address our server is running and change the ports used in the project settings.

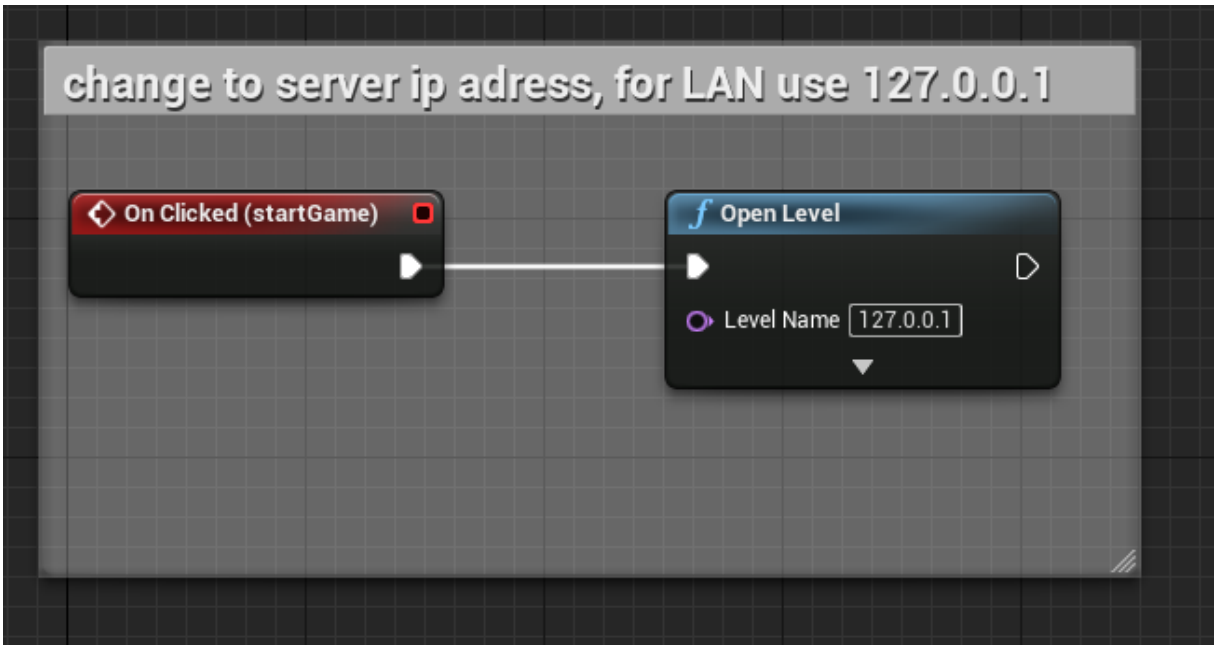


Figure 2

Reflection

Looking back on it now that we know how to do it is quite straightforward what's need to be done i.e. which properties and data that needs to be replicated and passed between the server and connected clients. Getting to this point though were we know what we need to do and which functions to use from Unreal's library was extremely tedious and frustrating due to lackluster documentation and bad tutorials where the presenter did not know the difference between local multiplayer and multiplayer with a dedicated server, because the logic is quite different between the two.

2.5 Game Mode

Person responsible: Andreas Blakli

Initially we wanted to create a team deathmatch game mode, but due to time constraints we opted to create a Free For All game mode instead since making the logic for this was easier. All of the game mode was written in C++, where a BP was created to bind the game mode to the map.

When all the players connected to the game have been killed (except the last player of course) the winner name is multicast to all the connected clients telling everyone who the boss of the server is using Unreal's DECLARE_DYNAMIC_MULTICAST_DELEGATE method. The winner name is set up, so it grabs the name of the connected client's pc. But if this game where released on Steam it would be able to grab the Steam username and use that instead.

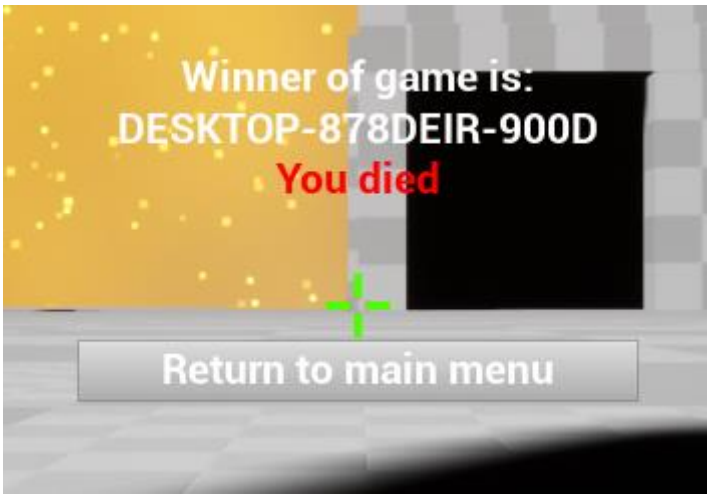


Figure 3

Figure 4

Challenges

Due to the way we did the multiplayer with seamless travel between the maps the logic for the game mode got a little difficult to write in a good modular way, because initially it had an array which would just add the player with the PostLogin() method from Unreal's library, but

this went out the window with seamless travel, so the game officially starts when the first player is killed. When a player is killed an array holding all the player states is updated to add all the connected players, this introduces a bug/problem where if a new player connects to the game after a player have died, he/she/it will never be able to win because the player state has not been added to the array. Otherwise, the only other big challenge we had was the multiplayer part with what needed to be replicated to the server as we discussed in section 2.4 Networking and Multiplayer. Also, an important note here is that the server needs to be restarted for the game mode to reset itself, we did not have time to implement a dynamic server reset.

Reflection

We would really have liked to have implemented a team deathmatch mode, but with the given time constraints we are happy with how the FFA mode turned out. Also a dynamic server reset would have been nice to have implemented.

2.6 UI

Persons responsible: Andreas Blakli, Vegard Opkvitne Årnes

2.6.1 Main menu, pause menu, lobby, player HUD, game mode UI

Person responsible: Andreas Blakli

All the widgets where created in Unreal's blueprint system, since designing them there goes a lot faster than programming them in C++. The way the widgets get their data is from functions we created in C++ and events that activate in C++ which are then used in blueprints to create the widgets when the event gets called. I.e. as the screenshots show (this is from the humanCharacterBP).

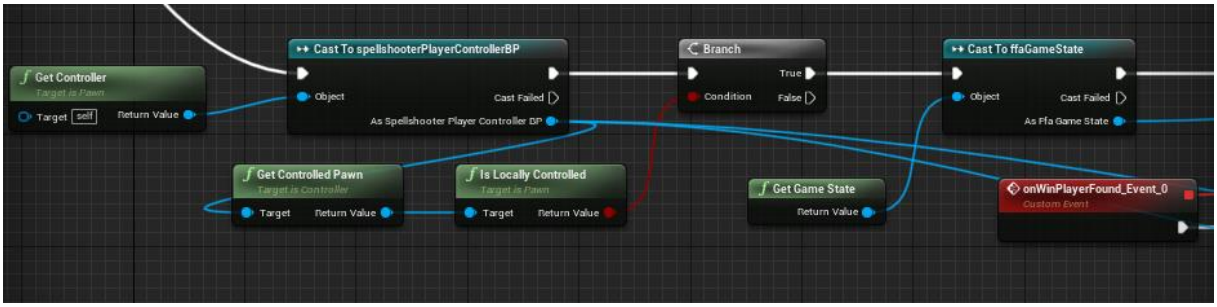


Figure 5

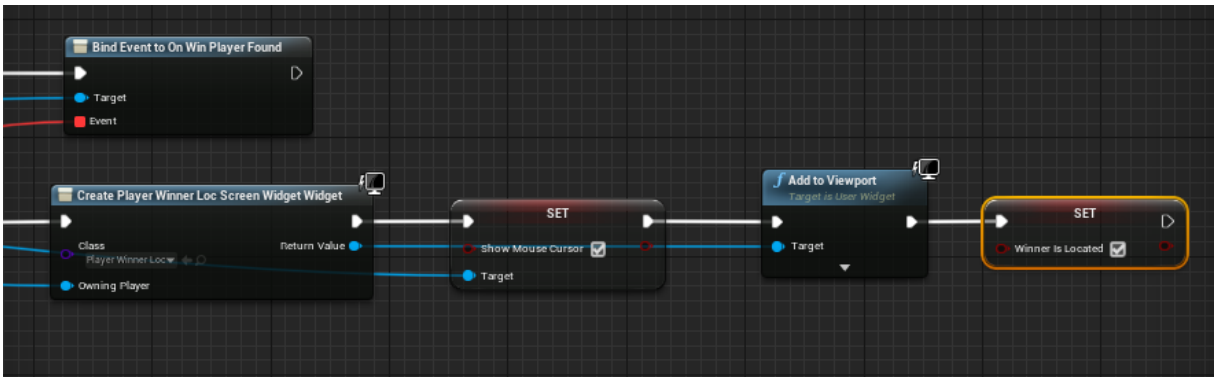


Figure 6

Challenges

The main challenge of all the UI elements where the player HUD because once we added multiplayer the HUD broke for all other clients except for the first client who connected to game. It took us a while to figure why this happened and find a fix for it. The main problem here was that as we mentioned in section 2.4 Networking and Multiplayer people who made tutorials for this did not know the difference between local multiplayer and multiplayer with a dedicated server. The solution was to get the pawn then get the controller of said pawn and cast it to the player controller, and then get the controller from the casted class.

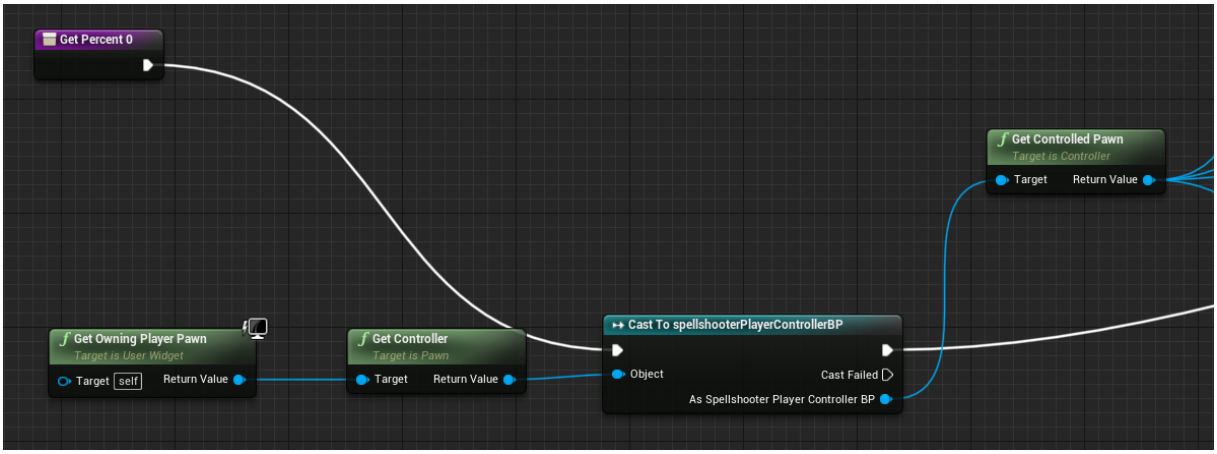


Figure 7

Since we had two different playable characters, we wanted to make the HUD universal so that we did not have to create a new HUD for each character. The solution was simple, we just got the class of actor who owned the player controller and did a quick comparison between the classes.

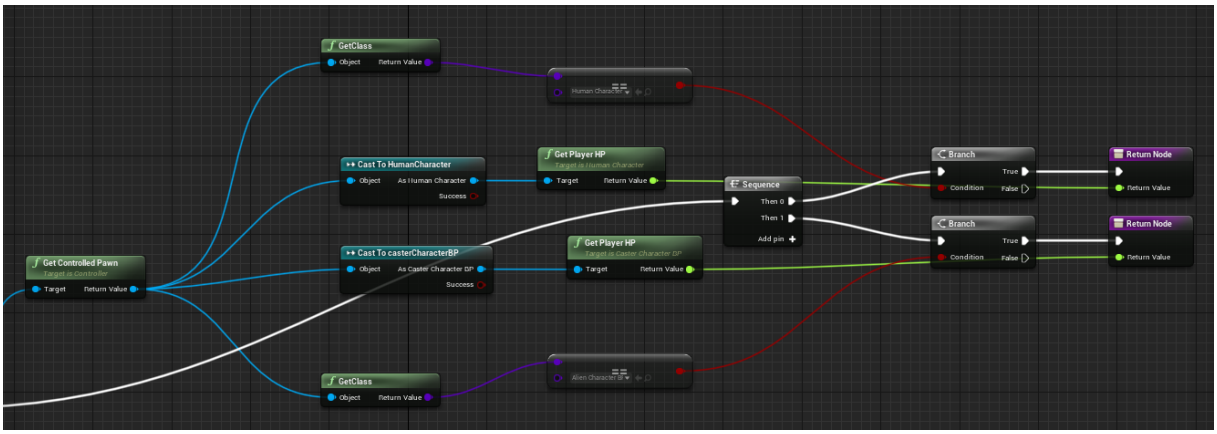


Figure 8

Reflection

Once we had figured out how to get and cast data from the right client all the other widget implementation went rather well. Better documentation on multiplayer with a dedicated server from Unreal's side would have helped here. Anything is easy when you know how to do it.

2.6.2 Weapon Selection UI

Person responsible: Vegard Opkvitne Årnes

The players need a source for the weapons of the game and naturally this was done using unreal engines' widget blueprint. The weapon selection screen is built up of two widget types, a general background interface, and a slot widget for each weapon the player can equip, loaded within the boundaries of the background interface. First, weapons are added to the individual players "Player Store" object, the "Player Store" is a custom UActorComponent class named storeComponent, each player controller is assigned one such storeComponent object and use it for all interaction with shop. The addition of weapons takes place in the spellshooterPlayerController blueprint as seen below:

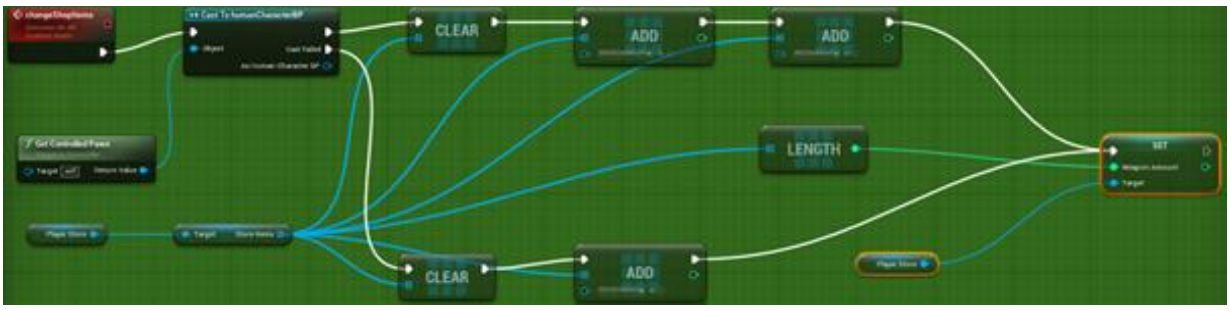


Figure 9

When function is call it adds specific items to the shop based on whether the player controller is a human or an alien, to do this it casts the pawn to humanCharacterBP, if it works then it executes the top node, if it fails it means the pawn is an Alien and executes the bottom row instead. This function updating the content of the shop is called every time a player changes the playable class. When player opens the Weapon selection UI, the begin construct default node runs and fills the shop with as many “Store Slot” buttons as “Player Store” has weapons, then updating their layout by adding the icon given to them in the “Data Asset”. The buttons in store slot are made clickable and calls an event dispatcher eithin the individual slot widget which fetches the spellshooterItem variable. The info is thereafter passed into the C++ “DECLAER_DYNAMIC_MULTICAST_DELEGATE_ThreeParams” function which makes the clicked item’s data accessible to the spellshooter player controller which handles the logic.

Challenges

The greatest challenge with the development was allowing for the contents of the shop to change based on what race/class the player decided to play and make consistently updates properly regardless of how often player changes sides. At first the logic that added weapons to player store as seen in figure above happened only on construct of player controller, meaning it couldn’t be changed. Therefore, we tried to add the same functionality directly at the end of the race change logic, but it refused to update content of runtime. The solution was to make it a function that was replicated on server regardless of it being a client-side feature,

Reflection

The weapon shop UI was a difficult, but rewarding element to optimise. The learning outcome was immense as there was a constant communication stream between Widgets, C++ class and the player controller blueprint. The task alone became a solid introduction in communication between C++ and blueprints and was a gateway to understanding how to make them work together with different parts of the logic.

3. Strengths and weaknesses of the Unreal Engine

There is no doubt that the Unreal Engine is an incredible powerful tool if you know how to use it, but it has its fair share of issues as well. The main problems we had during the development process of this game were lack of documentation especially for C++ implementation of multiplayer logic. If you want to do anything that is outside the normal A4 game projects in C++ you are going to have a bad time. We all dislike the blueprint system and the graphical interface Unreal uses, to put it simply it is annoying to use and if you are trying to make more than one short script it instantly turns into spaghetti. The readability of the logic is difficult once you start to get a lot of logic that is connected to a lot of functions and is casted between different classes, it is easy to get “lost” in the blueprints. Merging of these blueprint files was incredibly annoying to the point where we all agreed to not work on the same blueprint at the same time to avoid merging conflicts.

The nice things about the Unreal engine are that it is rather powerful, you have access to a ton of premade functions to handle different aspects of the program. You can easily design UI widgets, create maps and levels, animations, handle user input etc. You do not have to write your own game engine to handle rendering of the game, Unreal supports a ton of platforms, so packaging the project and converting it to work on other platforms is easy.

4. Final thoughts and reflection on the project

Looking back at our project, it was a little nightmare to work with in group. At the start I felt like we were being complacent and waiting for someone else to make a decision on what game genre we were going to make. Not much progress was being made due to other assignments taking priority. Once we started there was some miscommunication between group members where we would work on the same features without notifying each other about it. Once we all started to work on it, the problems began to arise. For starters we had issues getting in proper contact with each other due to COVID19 and the social distancing rules. This lack of communication played a part in early inactivity on this project. Several other problems were discovered when pushing the code to GitHub. It would sometimes break the build for someone else, and new features would break old ones. While some of these were due to unknown code inside unreal's compiler, a lot of the time it was due to poor communication between group members and overall poor playtest practices. Not only were there problems with communications, but the lack of documentation made it time-consuming to get some blueprint logic to work.

What we can take from this is that communication between different parties during game development is key to a good game development cycle. Setting things clear about where one should work so effort does not overlap and ruin blueprint features or code. Reading on documentations forwarding the information learned to group members to avoid future problems and work regularly on the project are also some practices to take from this experience.

5. Known bugs

Trying to set the resolution in the settings tab in the main menu only works when running the game in windowed mode or from the Unreal debugger. To make the game windowed when its fullscreen press f11.

6.1 Individual report by Andreas Blakli

Code I consider to be good 1:

In humanCharacter.cpp and casterCharacterBP.cpp (and the blueprint belonging to casterCharacterBP.cpp is called alienCharacterBP, why these confusing names? I honestly do not know, you'll have to ask my team members) the logic I created for health tracking, updating health calling different events e.g. displayDeathScreen(), spawning in projectiles on left mouse click (shooting) and replicating the data to the server and other connected clients is good. Beneath is some example snippets of code, for full code see the humanCharacter.cpp, casterCharacterBP.cpp, humanCharacter.h, casterCharacterBP.h, humanCharacterBP.uasset and alienCharacterBP.uasset (they are basically clones of each other).

updatePlayerHP(float HP) is a rather neat function which is clean and simple, each time the player is hit by a projectile playerTakeDamage(float damage) is called which then calls updatePlayerHP(float HP) with the passed damage value. The values are replicated to the server with Unreal's ReplicatedUsing method.

```
115  void AcasterCharacterBP::updatePlayerHP(float HP) {
116      currentPlayerHP += HP;
117      currentPlayerHP = FMath::Clamp(currentPlayerHP, 0.0f, maxPlayerHP);
118      tempPlayerHP = playerHPpercent;
119      playerHPpercent = currentPlayerHP / maxPlayerHP;
120      if (playerHPpercent <= 0) {
121          playerIsDead = true;
122      }
123      UE_LOG(LogTemp, Warning, TEXT("hp should update Alien"));
124  }
125
126  void AcasterCharacterBP::playerTakeDamage(float damage) {
127      updatePlayerHP(-damage);
128  }
129
130  void AcasterCharacterBP::onRep_currentPlayerHP() {
131      updatePlayerHP(0);
132  }
```

Figure 10

The onRep_kill() and onRep_win() functions are rather nice, when onRep_kill() is called it checks if it is client or server, if it is the client then the displayDeathScreen() functions is called which basically is and empty function used as an event in the blueprints humanCharacterBP.uasset and alienCharacterBP.uasset. When the function activates the event

is called in the blueprint the widget is then created for the player.

```
216  void AHumanCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {
217      Super::GetLifetimeReplicatedProps(OutLifetimeProps);
218      DOREPLIFETIME(AHumanCharacter, killerHuman);
219      DOREPLIFETIME(AHumanCharacter, winnerP1);
220      DOREPLIFETIME(AHumanCharacter, currentPlayerHP);
221  }
222
223  void AHumanCharacter::onRep_kill() {
224      if (IsLocallyControlled()) {
225          displayDeathScreen();
226      }
227
228      GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);
229      GetMesh()->SetSimulatePhysics(true);
230      GetMesh()->SetCollisionEnabled(ECollisionEnabled::PhysicsOnly);
231      GetMesh()->SetCollisionResponseToAllChannels(ECR_Block);
232      SetLifeSpan(05.0f);
233  }
234
235  void AHumanCharacter::onRep_win() {
236      if (IsLocallyControlled() && playerIsDead == false) {
237          displayVictoryScreen();
238      }
239  }
```

Figure 11

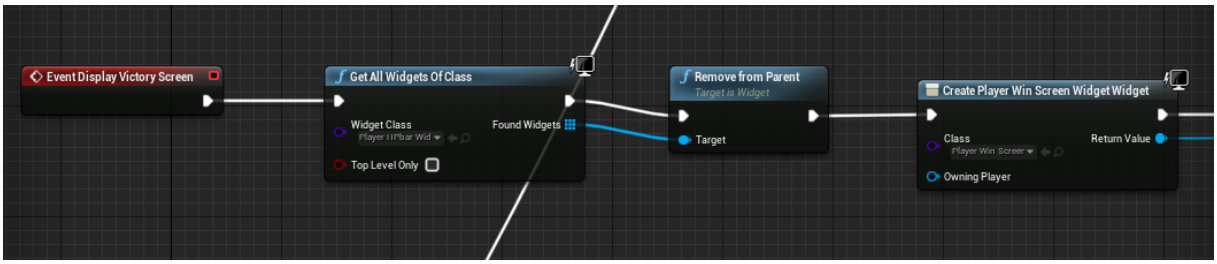


Figure 12

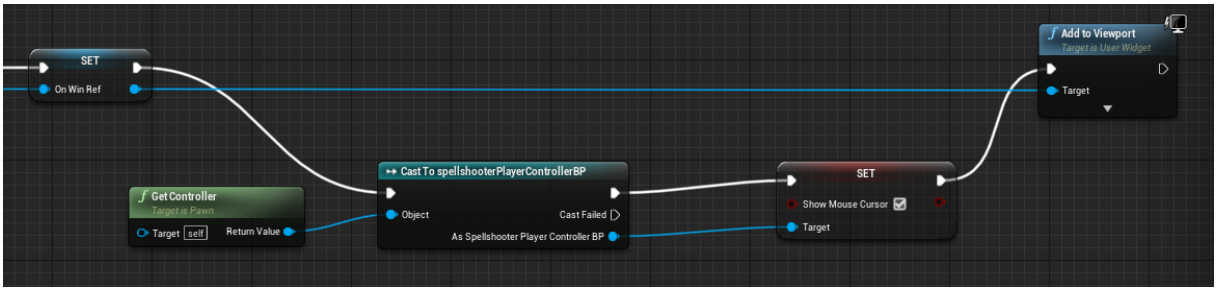


Figure 13

The way the projectiles is selected and handeled is rather good, it is set up in a modular way so all we need to later on when we wan't to change the projectile the player is using to represent different abilities and damgage values is to change a property in the blueprint.

```
206
207        Abullet* bullet = World->SpawnActor<Abullet>(BPbullet, MuzzleLocation, MuzzleRotation, SpawnParams);
```

Figure 14

```
37      UPROPERTY(EditAnywhere, Category = "shooting")
38          TSubclassOf<class Abullet> BPbullet;
```

Figure 15

Figure 16

Direct links to files in repository:

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Private/humanCharacter.cpp

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Private/casterCharacterBP.cpp

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Public/humanCharacter.h

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Public/casterCharacterBP.h

https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/Blueprints/humanCharacterBP.uasset

https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/Blueprints/alienCharacterBP.uasset

Code I consider to be good 2:

In bullet.cpp, bullet.h and BPbullet.uasset the main logic of the projectile creation and the handling of collision and give damage to players on hit is rather nice, it is set up in a modular way to allow us to easily make blueprint copies of bullet.cpp class to make new projectiles with different damage values, velocity, and mesh.

The onHit() function is rather big with a lot of if statements, this could have been set up in another way, but I felt that would severely reduce the readability of the code if I compressed the if statements too much, so I made the decision to stick with a bunch of if statements. The if statements are there to check which actor the projectile is spawned from and which class

that actor belongs to, so we can call the right functions with the right parameters since a lot of this data is replicated and needs to be passed to the server and game mode. And since we got multiple actors the players can choose between and play as the function got rather large.

```
void Abullet::onHit(UPrimitiveComponent* hitComp, AActor* otherActor, UPrimitiveComponent* otherComp, FVector normalImpulse, const FHitResult& hit)
    if (HasAuthority()) {
        if (AHumanCharacter* playerHit = Cast<AHumanCharacter>(otherActor)) {
            //playerHit->playerTakeDamage(25.0f);
            if (playerHit != Cast<AHumanCharacter>(GetOwner())) playerHit->playerTakeDamage(damageValue);
            if (playerHit->currentPlayerHP <= 0.0f) {
                if (AffaGameMode* mode = Cast<AffaGameMode>(GetWorld()->GetAuthGameMode())) {
                    UE_LOG(LogTemp, Warning, TEXT("you are dead1"));
                    if (GetOwner()->IsA(AHumanCharacter::StaticClass())) {
                        UE_LOG(LogTemp, Warning, TEXT("prob in 1"));
                        AHumanCharacter* killer = Cast<AHumanCharacter>(GetOwner());
                        mode->playerKilled(playerHit, killer, nullptr, nullptr);
                        playerHit->killerHuman = killer;
                        playerHit->onRep_kill();
                    }
                    else if (GetOwner()->IsA(AcasterCharacterBP::StaticClass())) {
                        UE_LOG(LogTemp, Warning, TEXT("prob in 2"));
                        AcasterCharacterBP* killer = Cast<AcasterCharacterBP>(GetOwner());
                        mode->playerKilled(playerHit, nullptr, nullptr, killer);
                        playerHit->killerHuman = killer;//this----------------------------------------------
                        playerHit->onRep_kill();
                    }
                }
            }
        }
        else if (AcasterCharacterBP* playerHitAli = Cast<AcasterCharacterBP>(otherActor)) {
            if (playerHitAli != Cast<AcasterCharacterBP>(GetOwner())) playerHitAli->playerTakeDamage(damageValue);
            if (playerHitAli->currentPlayerHP <= 0.0f) {
                if (AffaGameMode* mode = Cast<AffaGameMode>(GetWorld()->GetAuthGameMode())) {
                    UE_LOG(LogTemp, Warning, TEXT("you are dead2"));
                    if (GetOwner()->IsA(AHumanCharacter::StaticClass())) {
                        UE_LOG(LogTemp, Warning, TEXT("prob in 3"));
                        AHumanCharacter* killer = Cast<AHumanCharacter>(GetOwner());
                        mode->playerKilled(nullptr, killer, playerHitAli, nullptr);
                        playerHitAli->killerAlien = killer;//this----------------------------------------------
                        playerHitAli->onRep_kill();
                    }
                    else if (GetOwner()->IsA(AcasterCharacterBP::StaticClass())) {
                        UE_LOG(LogTemp, Warning, TEXT("prob in 4"));
                        AcasterCharacterBP* killer = Cast<AcasterCharacterBP>(GetOwner());
                        mode->playerKilled(nullptr, nullptr, playerHitAli, killer);
                        playerHitAli->killerAlien = killer;
                        playerHitAli->onRep_kill();
                    }
                }
            }
        }
    }
}
```

Figure 17

As I mentioned earlier when creating a new blueprint based on the Abullet class we have the option to set the velocity and damage value, size and mesh used.

```
24      UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "damage")
25          float damageValue;
```

Figure 18

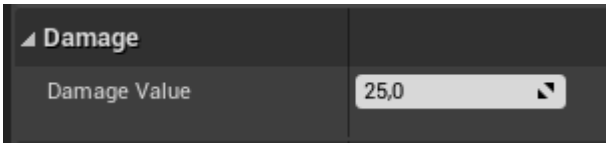


Figure 19

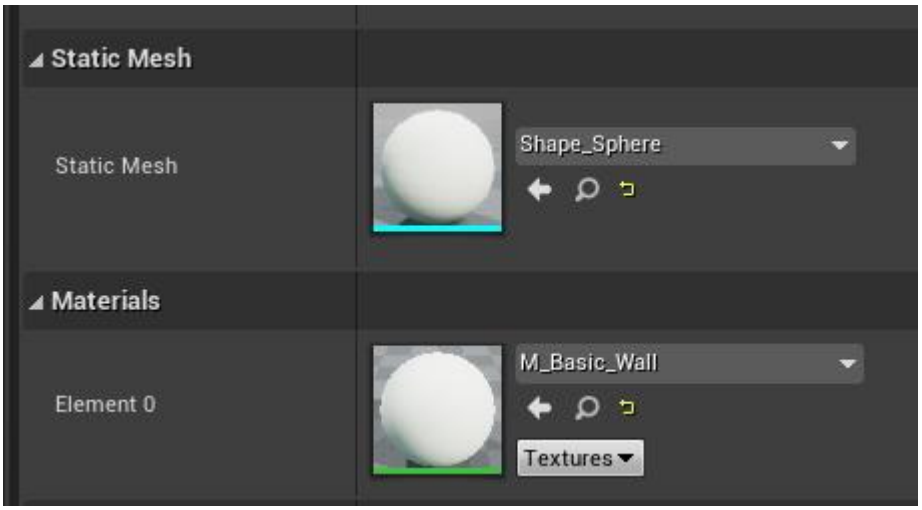


Figure 20

Direct links to files in repository:

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Private/bullet.cpp

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Public/bullet.h

https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Weapons/projectiles/BPbullet.uasset

Code I consider to be bad 1:

In playerHPbarWidget.uasset there lies a function called get percent, the logic itself is not too bad, but the way the function is used is. Because it updates the player HP bar based on tick rate, which is a waste of resources, in the games current state this is not too big of a problem since we do not have a lot of HUD elements rendered on the screen. But nonetheless it is bad programming practice to update an unchanged value unnecessarily as later on if we continued work on this game and got more HUD elements and other datapoints which is all updated on tick rate it could potentially introduce severe lag for the players. A solution for this is to make an event which is called when the player hp updates.

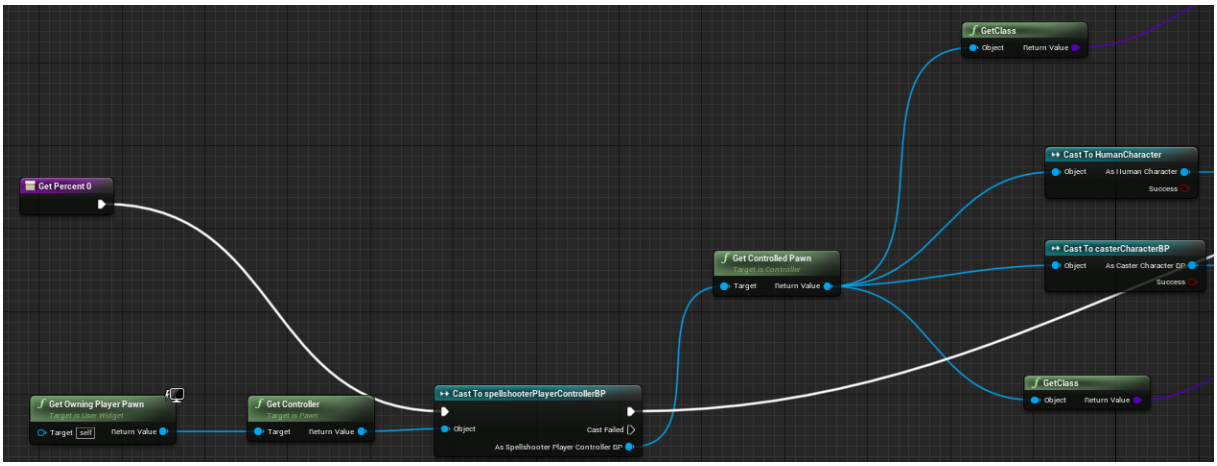


Figure 21

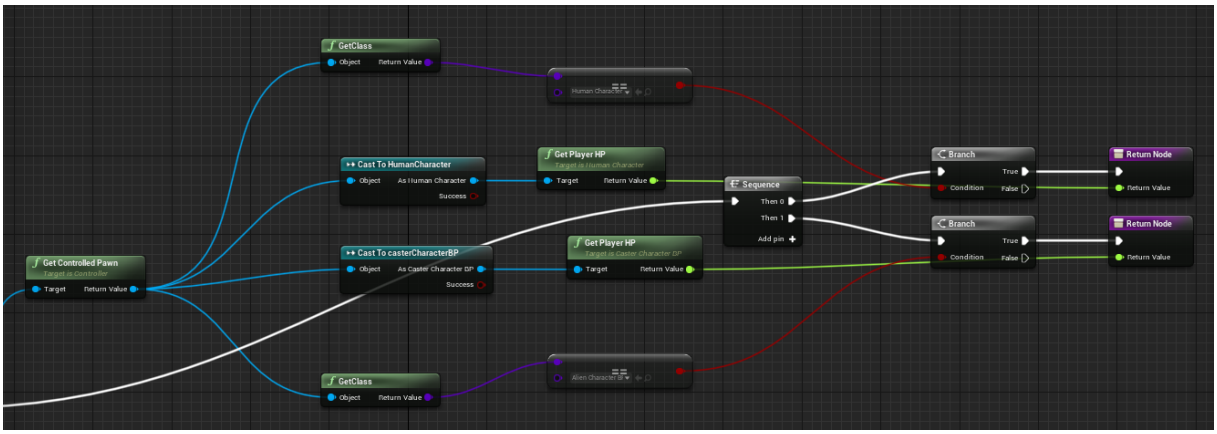


Figure 22

Direct link to the file in repository:
https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Multiplayer/playerHUD/playerHPbarWidget.uasset

Code I consider to be bad 2:

The code in ffaGameMode.cpp and ffaGameMode.h regarding the Free For All game mode is not a disaster, but it has some flaws. Before we used seamless travel on the server for map traversal all players that connected to the game were placed in an array using Unreal's PostLogin() method, this needed to be changed to work with seamless travel. So the way it is set up now is a compromise, the array is updated when a player is killed, which introduces a bug where if a new player connects to the game after a player have died they can't win because they are never added to the array. The function playerKilled(class AHumanCharacter* killed, class AHumanCharacter* killer, class AcasterCharacterBP* alienKilled, class AcasterCharacterBP* alienKiller) has rather a lot of if statements which can be compressed and shortened and optimized to remove some code duplication.

```
25  void AffaGameMode::playerKilled(class AHumanCharacter* killed, class AHumanCharacter* ki
26      if (HasAuthority() && doOnce == false) {
27          UWorld* World = GetWorld();
28          playerStArr = World->GetGameState()->PlayerArray;
29          onRep_updateArr();
30          numOfElements = 0;
31          numOfElements = playerStArr.Num();
32          UE_LOG(LogTemp, Warning, TEXT("player array:, %d"), numOfElements);
33          doOnce = true;
34      }
35      if (killed) {
36          if (APlayerState* player = Cast<APlayerState>(killed->GetPlayerState())) {
37              playerStArr.RemoveSingle(player);
38              onRep_updateArr();
39          }
40      }
41      if (alienKilled) {
42          if (APlayerState* player = Cast<APlayerState>(alienKilled->GetPlayerState())) {
43              playerStArr.RemoveSingle(player);
44              onRep_updateArr();
45          }
46      }
47      if (playerStArr.Num() == 1 && playerStArr.IsValidIndex(0) && killer) {
48          winPlayer(Cast<AffaPlayerState>(playerStArr[0]));
49          killer->winnerPl = killer;
50          killer->onRep_win();
51      }
52      if (playerStArr.Num() == 1 && playerStArr.IsValidIndex(0) && alienKiller) {
53          winPlayer(Cast<AffaPlayerState>(playerStArr[0]));
54          alienKiller->winnerPl = alienKiller;
55          alienKiller->onRep_win();
56      }
57  }
```

Figure 23

Direct links to files in repository:

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Private/ffaGameMode.cpp

https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Public/bullet.h

Personal reflection of this project and learning outcome

This project was a huge learning experience for me. I learned a lot about using the Unreal Engine as a tool and as a language and what I mean by saying as a language, is that Unreal C++ is not normal C++, so when writing in it you kind of have to learn Unreal's way of doing C++, which at times can be incredibly frustrating. My personal goal was to try to the best of my ability learn as much as possible of Unreal C++.

Since I basically worked on almost all of the aspects of the game, I got well versed in Unreal's level editor (map creation), user interface widgets (the best comparison I can make for the widget creator and editor is Android Studio GUI), actor creation, blueprint

programming, C++ programming and how to mix and communicate between the widgets, blueprints and C++ code. Some of the most frustrating aspects of this project was setting up the dedicated server and implementing the logic to make the data replicate, be passed to the server and the other connected clients. As I wanted to learn Unreal C++ this complicated thing as to finding the right methods to use from Unreal's library (and documentation) and how to make good code implementations. There where examples on how to set the game up for multiplayer with a dedicated sever in blueprints, but this was not my goal with this project. So, learning and figuring out how to do this in C++ was extremely tedious and time consuming. I think I spent close to 150-200 hours total on this project and I do not feel like the results I got in this project reflects the time I spent. Now that I know what to do and which Unreal methods to use, expanding on the multiplayer logic is not easy, but neither difficult. Other insanely annoying problems I ran into were when people did not do proper playtesting with what they had created and checked if it worked with the existing code and pushed broken code to the repo. This took up unnecessary time to fix when I would pull and try to merge and suddenly nothing would work. This could easily have been avoided by using branches in GitHub or doing proper playtesting before pushing. Incase things like this happened is why we had local backups of the project so we could revert it back and continue to work on the project. Unreal corrupting its own intermediary files were not fun, it did this on multiple occasions, Unreal suddenly crashing for no reason, figuring out how to fix these issues took up time.

I would have liked to have fixed my two bad code examples, but due to time constraints I simply did not have time to do this.

My final thoughts on this project are rather conflicted in one way it has been a very powerful learning experience, it's fun creating a game and being able to play and interact with what you have created, and on the other hand Unreal have been an absolute pain to work with.

6.2 Individual report by Vegard Opkvitne Årnes

Code I consider to be good 1:

The code I consider to be the best are the different building blocks that make up the item selection/weapon equipment system. A surface level description of the system has been provided previously in the document, see paragraph 2.3.3. Gameplay Abilities/Weapons and 2.6.1. Weapon Selection UI, but I wish to point out how specific parts worked.

The entire Weapon Selection logic is within the aforementioned “Player Store” variable each player controller has. The “Player Store” is an object of the storeComponent class made, and the key methods behind the functionality is as follows:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_ThreeParams(FItemAdded, AActor*, Owner, UStoreComponent*, Store, USpellshooterItem*, Item);
```

Figure 24

```
UPROPERTY(BlueprintAssignable, Category = "Store")
    FItemAdded OnItemAdded {};
```

Figure 25

```
bool UStoreComponent::getItemFromStore(USpellshooterItem* Item){
    OnItemAdded.Broadcast(GetOwner(), this, Item);
    return true;
}
```

Figure 26

The UStoreComponent::getItemFromStore(USpellshooterItem* item) function is triggered in the storeInterface widget (What I refer to as Weapon Selection UI) once the player clicks on it. Each weapon button in the Weapon Selection UI has the provided data asset containing all weapon info, and it is that Data Asset which is passed into the getItemFromStore function. In addition, the Data Asset is of type USpellshooterItemWeapon, but the function parameters take USpellshooterItem object pointers. This is because the function works for all child classes of USpellshooterItem, so if we were to implement for example USpellshooterItemConsumable or USpellshooterItemArmor, this logic would still be perfectly functional. The usage of getItemFromStore can be seen in image below:

Figure 27

The getItemFromStore function runs the onItemAdded.broadcast() which is a dynamic multicast delegate, meaning that it’s an event we can access in blueprints, that will trigger

with every broadcast. The broadcast triggers the “on Item Added” event in spellshooterPlayerController that gives them the weapon. The event in question can be seen in the image below.

Figure 28

The blueprint above is what actually provides the gameplay ability/weapon. First it spawns the relevant actor, then checks whether the controlled pawn is alien or human, gives the relevant bullet blueprint, deletes previously held weapon, and attaches the spawned actor to the provided socket. The red boxes highlight that the same things happen, but the top one is for human and the bottom one is for alien.

To prove why I think its good, I want to illustrate steps taken to for example new magical weapon for caster:
First make an actor blueprint containing the graphical elements you want the magic to have when holding it. Then make a bullet blueprint with the graphical elements determining how the attack will look, and add the on collision logic you wish inside the same blueprint if you wish to add unique effects. Third, make a 256x256 Icon, and put them into a spellshooterItemWeapon as such:
Create FireAsset with the aforementioned arrtibutes.

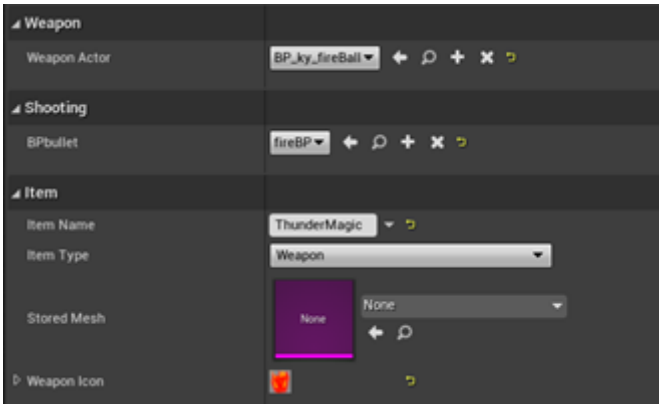


Figure 29

Add it to the shop if player is not human.

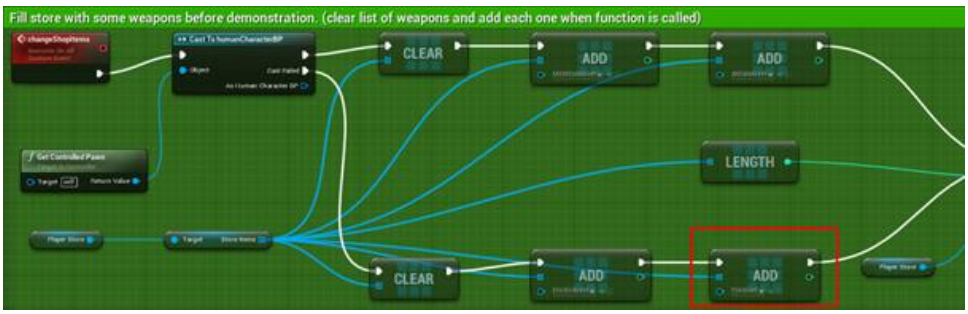


Figure 30

And its now a fully functional magic weapon in the game. In this case the Weapon Actor is just the particle system component with the fire animation and the BPbullet is the standard bullet with the same fire effect added on top of it.

Direct links to files in repository:
https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Public/StoreComponent.h
https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Private/StoreComponent.cpp
https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/Blueprints/spellshooterPlayerControllerBP.uasset
https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/UI/storeInterface.uasset
https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/UI/storeSlots.uasset

Figure 31

Code I consider to be bad 1:

We made a big point out of avoiding bad code because more often than not, bad code would only work when playing in Unreals "Offline Mode" meaning that an abundance of issues would become apparent when attempting to run it in multiplayer, so a shameful amount of time was spent on refactoring code. There are some harmless but bad code still though.
In the Weapon Equipping logic, when the "on Item Added" event takes place, game deletes the previous held weapon, and creates a new one for the player to hold. However, at the start of the game there is no weapon for the event to delete before creating the new weapon. The solution is to create an AK47 actor blueprint that the players are "holding", even if this bypasses the error of trying to delete something non-existent, the AK47 actor blueprint stays on the map and is not actually deleted. This means that for every a new player connects, or chooses to play human, an unnecessary actor is created and stays in the world, which is a waste of resources, albeit a small one. The blueprinting guilty for this is the following:

Figure 32

As can be seen here, the location is -100 on the Y axis, meaning that they spawn where we cant see them. Attempts were made to have them deleted properly on weapon change, but no solution was found. In hindsight its likely due to the deletion attempts I made not being replicated properly to the server, meaning that while it gets deleted locally, it doesn't disappear from the multiplayer source and is thus technically still claiming resources.

Direct links to files in repository:
https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/Blueprints/humanCharacterBP.uasset

Personal reflection of this project and learning outcome

I started the project with zero experience with Unreal Engine, while simultaneously having plenty of expectations towards how the workflow would be. Seeing how popular the engine is for large companies, I first of all expected all of the features to either be intuitive or well documented, in addition I thought it was given that the engine was well designed for

cooperation between different developers, but things ended up more complicated than expected. I believe it has been mentioned by others, but the blueprint system is not merge friendly, so we had to coordinate our workflow manually to a larger degree than what we are used to. Also because of the great amount of both C++ files and blueprint files that are generated, its difficult to keep track of what each person is doing where, therefore we had multiple instances where we either deleted others work through merging, or were overwriting the functionality they were implementing by working on things loosely tied to it. The most valuable learning outcome from the project has been to learn to share more information about what we are implementing ourselves in a comprehensive manner, because its far from intuitive whether or not you will affect other peoples work negatively. Communication has been a larger factor than ever before as we have previously only worked with large text documents of code where merging is automatic and its visible if you're affecting other people's functionality. Another unexpected issue was the problem of finding documentation or help with solving issues especially when they are C++ related. My assumption is that a large amount of the guides, documentation, and issue threads assume that the developer is someone who aims to use blueprint only and who has little knowledge in coding. Our goal was to learn Unreal as best as possible, and that meant using the more powerful C++ options in development, so finding resources that treat you like more than just a hobbyist game developer was difficult. I feel that at the end now, the game is like a nice ground architecture that is ready to be modelled into something with actual gameplay value, and I wish we had the time to do so.

If I had more time I would have added sound, because there is definitely a dimension of immersion lacking. Second to that I would add abilities which can attach to other sockets than weapon, for example shoes that provide speed, flight, or jumping abilities. As has been stated earlier in the report, the groundwork for items and abilities is very well in place and arranging equipping and socketing of specific blueprints is done within a matter of minutes at this point.

To conclude my part on this project I will say that my relationship with unreal has become rather love/hate. I think in the end its about the developers communicative abilities as well as ability to organise development that makes or breaks the project, and obviously one could say that for any group project, but in this case I mean it in a technical way as Unreal Engine is genuinely difficult to get into for new people. Had we known from the start how we had to adjust out workflow, I believe we would have come much farther, but the end result is still one I enjoyed making quite a lot.

6.3 Individual report by Theo Camille Gascogne

Code I consider to be good:

Within the casterCharacterBP.cpp aswell as the blueprint which belongs to this cpp file, called 'alienCharacterBP' (yea that's my weird naming pattern, I excuse for any confusing that might have given you dear reader), the movement and the actor's turning is what I concider good code. The movement code is rather efficient as it will only run if there is a controller on the actor and if it is moving at all (the float value). Because these functions will first find the vectors for the directions the actor is going to, it can also be used for backwards and left movement since all it needs are negative values to go in the opposite direction. And the turning code, the code that controls the directional rotation of the actor, is simply smoother turning for the actor based on how many fps the game is currently running on. This was implemented to help mitigate speed problems should the fps increase or decrease between different platforms (one pc might run with more fps than another pc).

The way how changing between characters is done is quite good. How it works is when choosing your character/class, the player controller unpossesses the current actor and that actor is then removed from the game. A word is registered when the character/class has been selected, that word is then forwarded to a branch check where depending on the word, a new actor is spawned which is then possessed by the player controller and can then be played. The way this done trough the hurdles networking places is that widgets are client only, so the trigger is only in client side (potentially cause bugs if ran on a server) but the word being registered can be forwarded trough a function placed inside the player controller blueprint which is server side, and it is then used to spawn the correct actor and possess them.

```
void AcasterCharacterBP::MoveForward(float Value)
{
    if ((Controller != NULL) && (Value != 0.0f))
    {
        // find out which way is forward
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation(0, Rotation.Yaw, 0);

        // get forward vector
        const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);
        AddMovementInput(Direction, Value);
    }
}

void AcasterCharacterBP::MoveRight(float Value)
{
    if ((Controller != NULL) && (Value != 0.0f))
    {
        // find out which way is right
        const FRotator Rotation = Controller->GetControlRotation();
        const FRotator YawRotation(0, Rotation.Yaw, 0);

        // get right vector
        const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);
        // add movement in that direction
        AddMovementInput(Direction, Value);
    }
}
```

```
void AcasterCharacterBP::TurnAtRate(float Value)
{
    AddControllerPitchInput(Value * BaseTurnRate * GetWorld()->GetDeltaSeconds());
}

void AcasterCharacterBP::LookUpAtRate(float Value)
{
    AddControllerPitchInput(Value * BaseLookUpAtRate * GetWorld()->GetDeltaSeconds());
}
```

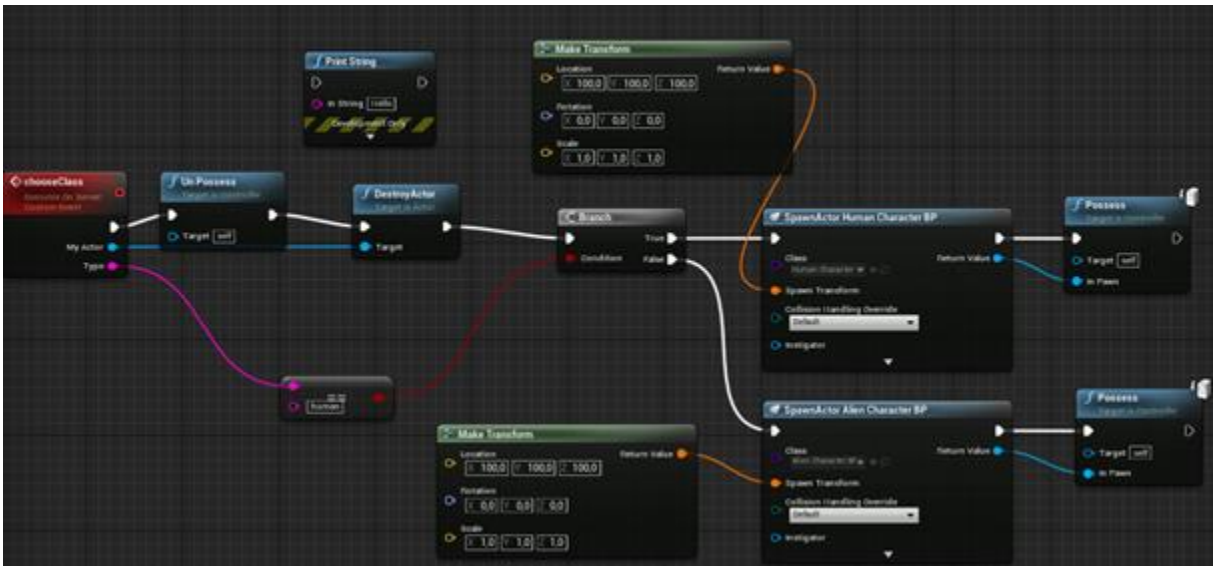
```
// Called to bind functionality to input
void AcasterCharacterBP::SetupPlayerInputComponent(UInputComponent* PlayerInputComponent)
{
    Super::SetupPlayerInputComponent(PlayerInputComponent);


    PlayerInputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);
    PlayerInputComponent->BindAxis("TurnRate", this, &AcasterCharacterBP::TurnAtRate);
    PlayerInputComponent->BindAxis("LookUp", this, &APawn::AddControllerPitchInput);
    PlayerInputComponent->BindAxis("LookUpRate", this, &AcasterCharacterBP::LookUpAtRate);

    PlayerInputComponent->BindAction("Jump", IE_Pressed, this, &ACharacter::Jump);
    PlayerInputComponent->BindAction("Jump", IE_Released, this, &ACharacter::StopJumping);

    PlayerInputComponent->BindAxis("MoveForward", this, &AcasterCharacterBP::MoveForward);
    PlayerInputComponent->BindAxis("MoveRight", this, &AcasterCharacterBP::MoveRight);

}
```



https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Private/casterCharacterBP.cpp

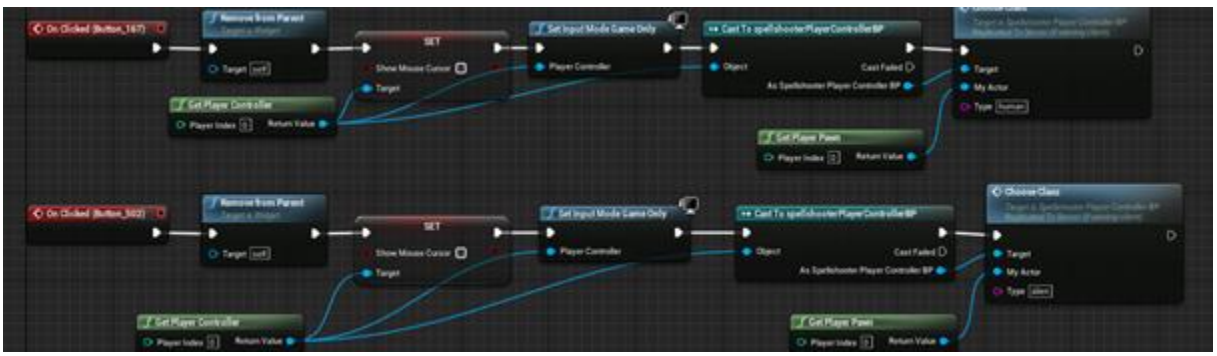
https://github.com/VitriolicTurtle/kcnGame/blob/main/Source/Spellshooter/Public/casterCharacterBP.h

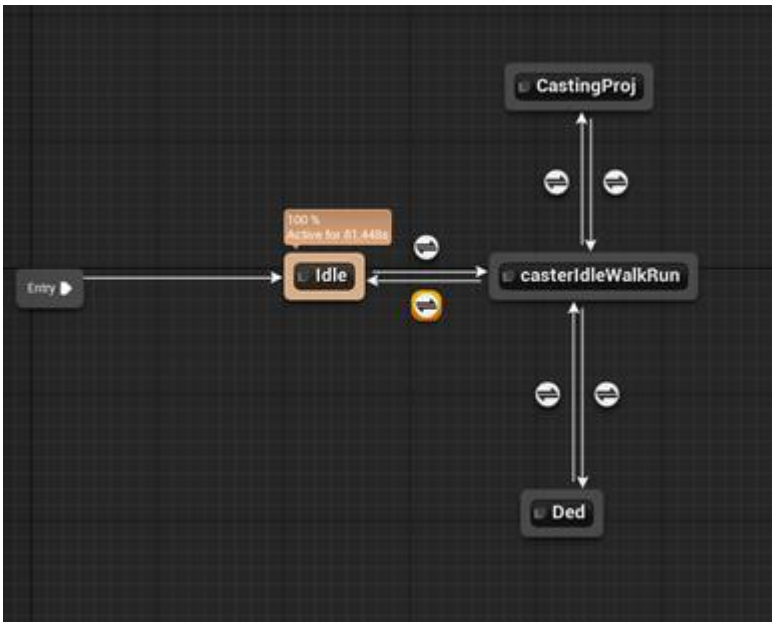
https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/Blueprints/spellshooterPlayerControllerBP.uasset

Code I consider to be bad:

I weren't as code heavy as my group members, but here is what I consider bad/inefficient BP logic.

Upon choosing what class the player wants, there is a duplicated code for each button. This works fine but how the player controller and pawn is gotten is not optimal. The issue is that this was designed to be only used in client, which is why the player index is kept as 0, but becomes a problem when used on a server where clients on their side have a player index 0 themselves, so it can create a bug where the server would spawn more entities than wanted due to the activation of code triggered by choose class in multiple clients. What should have been done is a function which gives a unique player index to clients so this flaw wont potentially produce more entities than wanted. Another issue is that animations do not have a good way of properly transition between extremes (i.e left and right), so if directions are changed immediately, the animations wont transition smoothy, but they will jump from one to the other. The reason is because the condition to transition between the animations are set to only one float value and the acceleration of the speed is done instantly, which skips a few animations such as the walk and run animations. And because there is no proper acceleration of the speed value, the vectors from the casters.cpp mentioned in the previous section will suddenly change as the directional values changes with the player inputs in an instant.





https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Human/UI/chooseClassGame.uasset

https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Glork/BlueprintsGlork/Glork/Glor0kAnimBP.uasset

https://github.com/VitriolicTurtle/kcnGame/blob/main/Content/Glork/BlueprintsGlork/Glork/casterIdleWalkRun.uasset

Personal reflection of this project and learning outcome

This project has been a lot to take in. I did not have any experience with UE4 and my expectations of the workflow was quite wrong to put it mildly. All I knew about unreal engine is that it was the most powerful amongst the game engines used by hobbyist game developers but also the most difficult to use, and at the time I just thought that it was considered hard due to how complex a game can get, not that it was hard to use in general. With such a large community of game developers and uses inside professional companies you'd think that there would be extensive documentation on how the engine and it's blueprints work, as well as guides on how to use them to achieve specific results. The biggest problem that struck us is how both intuitive and unintuitive blueprints work. In one way they are rather straight forward with which entities to cast to and interact with, but when applied in a different setting with the same idea the engine will refuse to work. And the lack of proper documentation really did not help there, since we opted to learn as much as possible from UE4 but finding guides or help for anything beyond 'blueprints for beginners' was a very time consuming. Another glaring problem we had was how everything we implement broke for each code we pushed to the build on Github. This forced us to incorporate coordination of our contributions and play test everything at least twice before pushing codes into our workflow and make sure everybody knew what everyone were doing so overlapping work could be avoided.

Personally what I took from this experience is that communication between team members is very important as well as play testing everything to avoid as many errors as possible. These errors could be small overrides to entire features being erased/replaced with new code because two or more team members would work on the same blueprint or CPP file. Given more time I would put in effort to smooth out and optimize code to be as flawless and bug free as possible, as UE4 often forces the use of blueprint logic that is flawed and may give birth of unexpected bugs. This is especially the case in networked projects where you must throw caution on whether your code is executed client-side or server-side. But in the end, this experience has taught me how important communication is in professional development and that building new features should be done similarly to most other features already existing to avoid conflicts between old and new features, and while that goes for every project out there it is especially important when every error you make ends up with hours upon hours of work and money down the drain.

7. Sources

AK-47 Asset -
https://drive.google.com/file//1TEHLeQG094UkoPOUgxlPx52ObJysXx23/view

M4-A1 Asset - https://free3d.com/3d-model/csgo-m4a4--91075.html

Magic effects - https://www.unrealengine.com/marketplace/en-US/product/a36bac8b05004e999dd4b1d332501f49

Alien and Human model and animations - https://www.mixamo.com/#/