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

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 Theo her.

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.

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.

2.4 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.

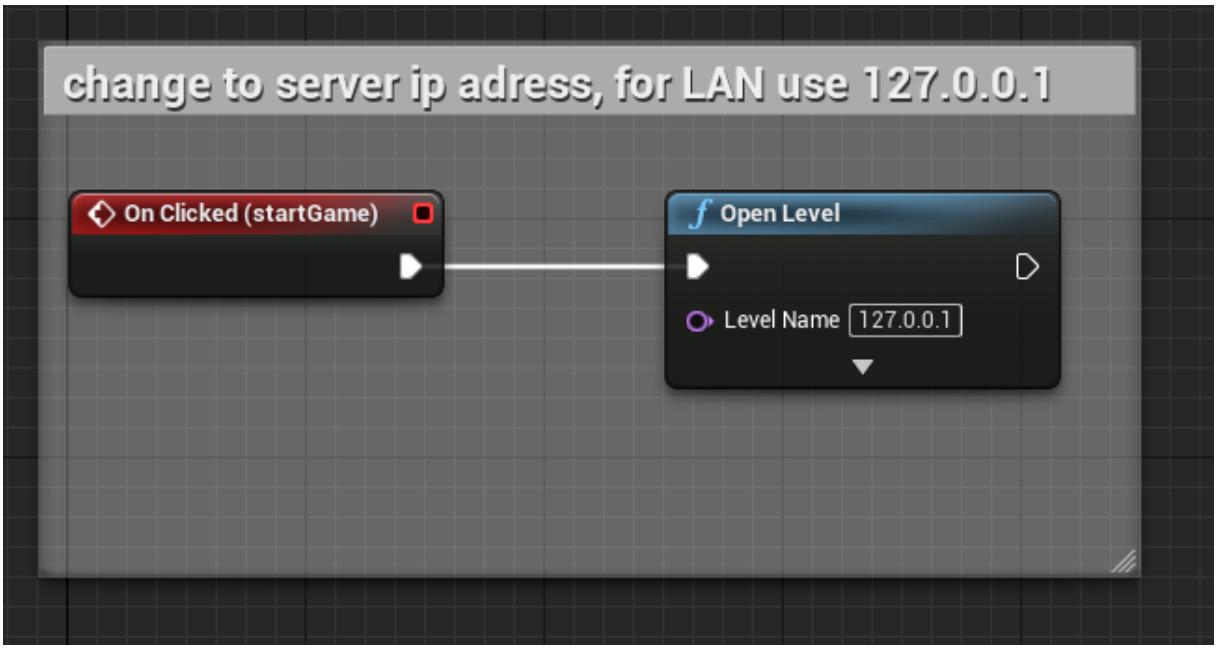

Figure 1

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.

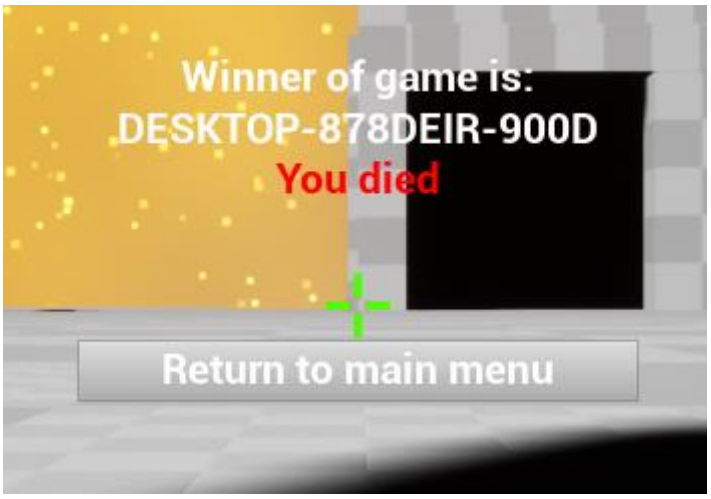

Figure 2

Figure 3

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, ????????

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).

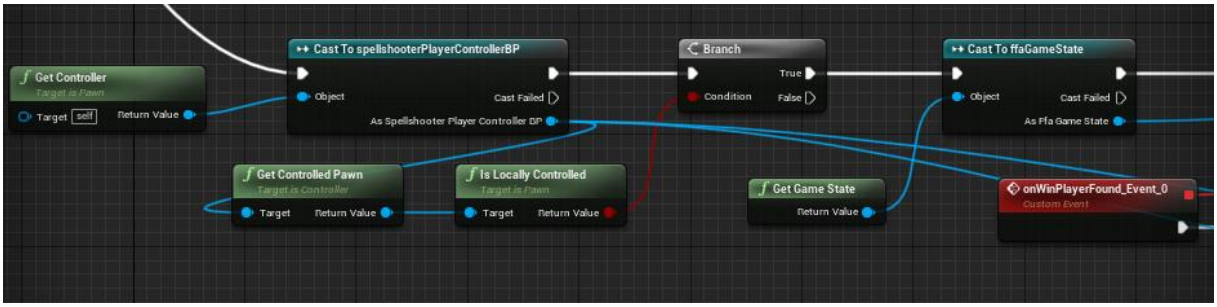

Figure 4

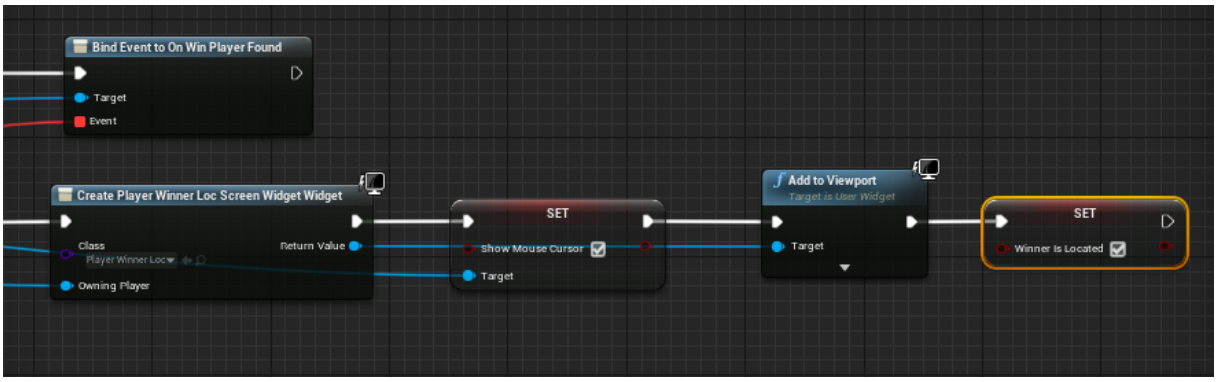

Figure 5

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.

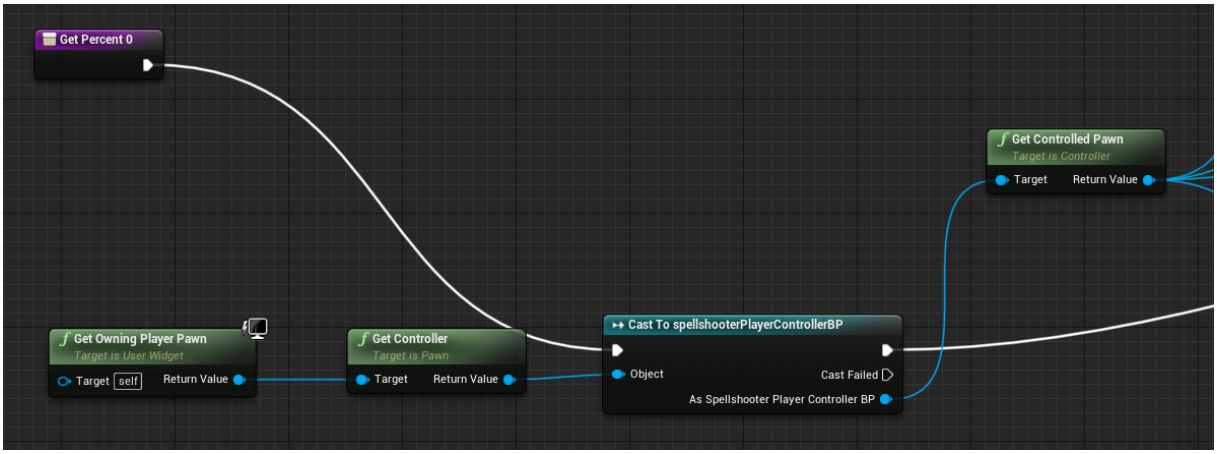

Figure 6

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.

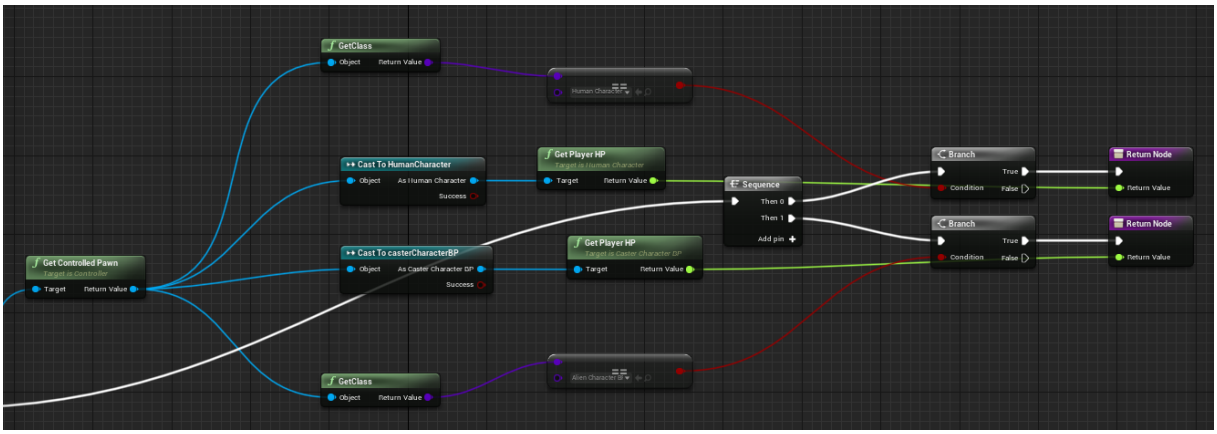

Figure 7

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.

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 from the Unreal debugger.

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 8

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 9

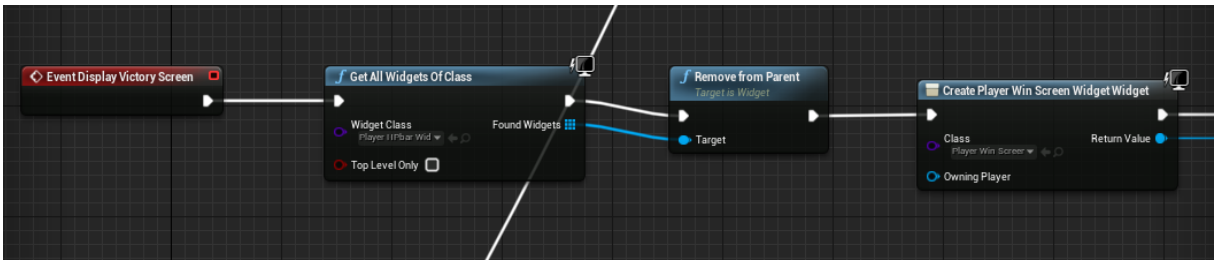

Figure 10

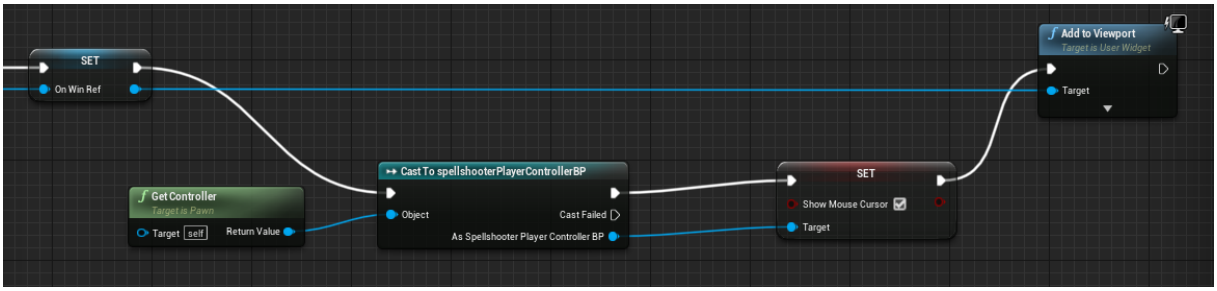

Figure 11

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 12

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

Figure 13

Figure 14

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 15

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 16

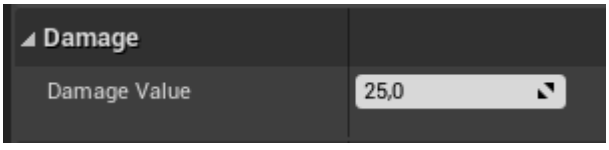

Figure 17

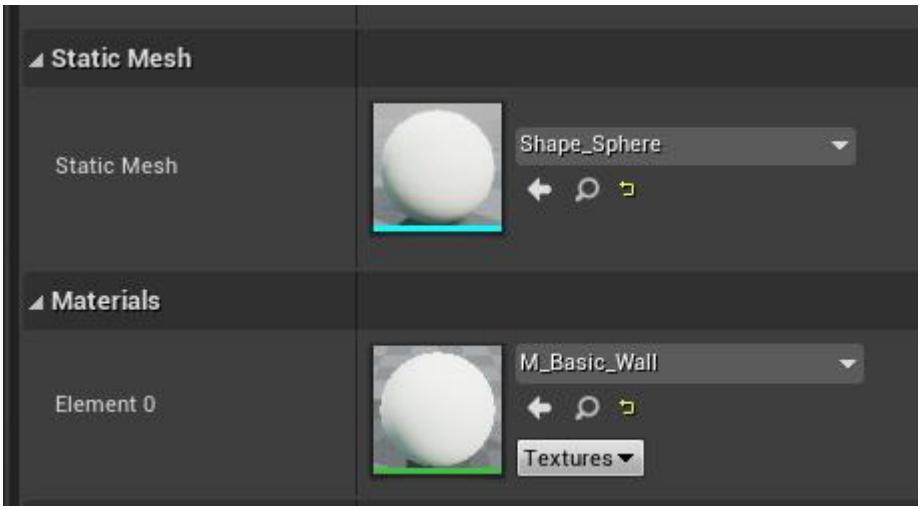

Figure 18

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.

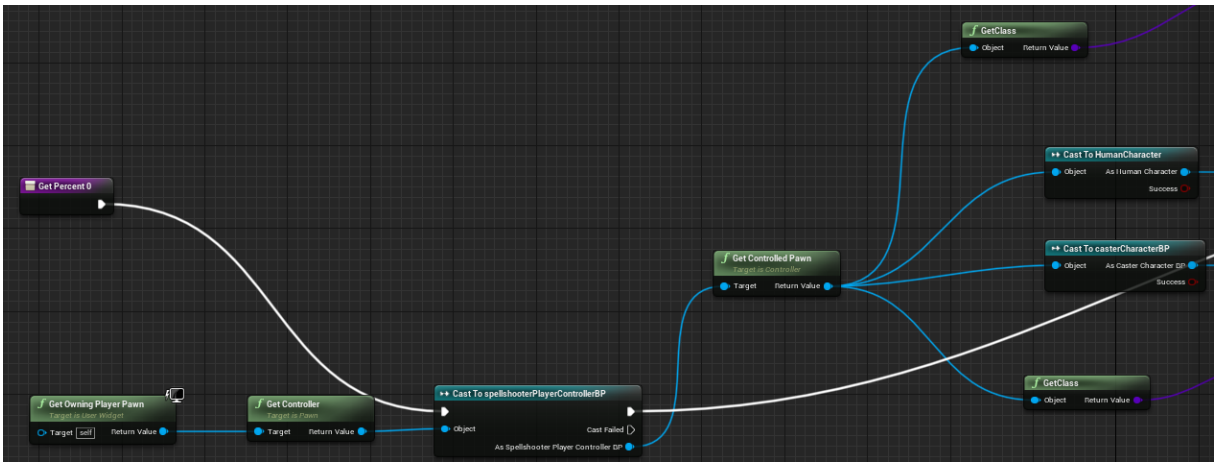

Figure 19

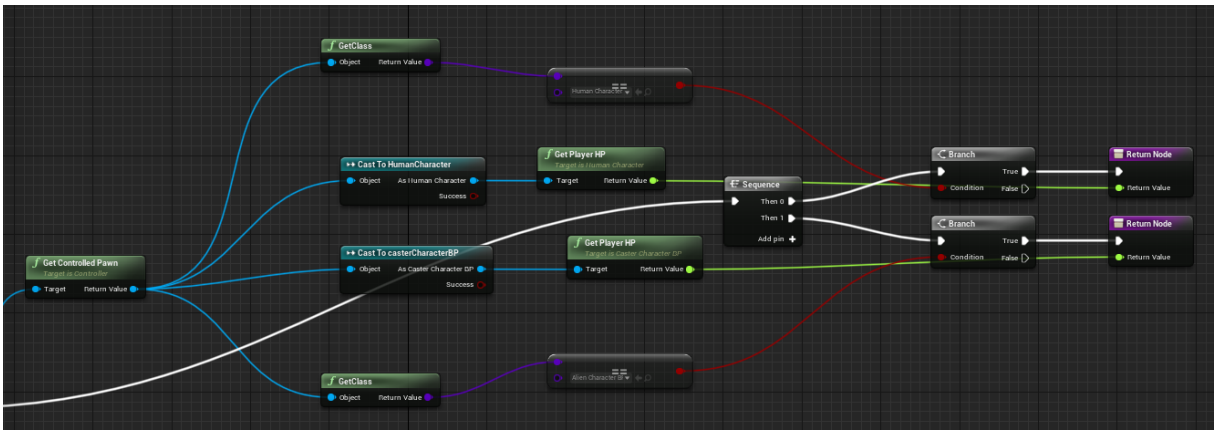

Figure 20

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 21

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.

7. Sources