NATIONAL COLLEGE "INOCHENTIE MICU CLAIN"
BLAJ

PROFESSIONAL CERTIFICATE IN COMPUTER SCIENCE

Project Title: BOSS FIGHT

Coordinator:
Prof. Romanța Ghiță

Author:
Cristian Vintilă

BLAJ, 2021

TABLE OF CONTENTS

BOSS FIGHT

1. Motivation for Choosing the Theme

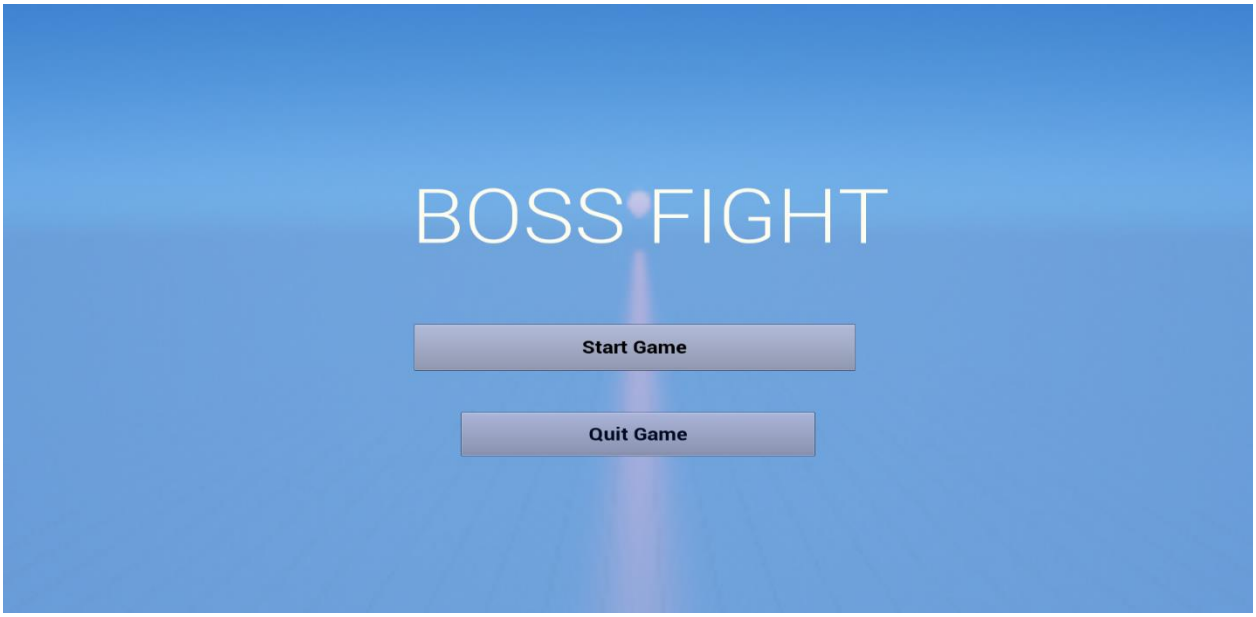
BOSS FIGHT is a single-player third-person shooter game where you have to defeat enemies encountered on your way to the top of the facility you are in. The game does not require an internet connection and is very accessible in terms of complexity and size.

The reason I chose to create a game is that since childhood, I was very attracted and, moreover, passionate about online games and very curious about how they are created. Reaching high school and having the opportunity to learn programming, with the help of my teacher, I went through the material and delved into numerous issues that helped me develop a certain algorithmic thinking, preparing me for this moment. As I documented myself more and more and realized what was needed to develop a game, I decided to go further and tackle the more complex part of programming, namely object-oriented programming (OOP), without which I could not complete the project.

The BOSS FIGHT game is created through two platforms: UNREAL ENGINE 4, a very advanced engine that works with C++, and VISUAL STUDIO 2019, where I programmed each entity in the game.

2. Application Structure. Organization of Informational Content

The first time you open the game, its menu appears, where, to start the game, you click the Start Game button on the left, and if you want to exit the game, you click the Quit Game button on the left as well.



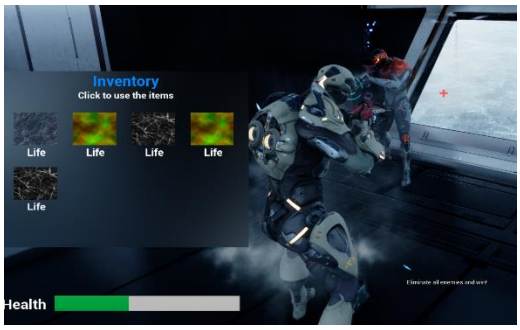
To continue playing, you need to know which keys to press to do certain things, but they are easy and specific to games in general. Thus, the player moves the character by pressing the W key - forward, A - left, S - back, D - right, and to jump, they press the Spacebar key. To open and close the inventory, press the Tab key, and to pause the game and return to it, press the Esc key. To shoot with the weapon, press the Left Click, and to ZoomIn/ZoomOut the camera, use ScrollUp/ScrollDown.

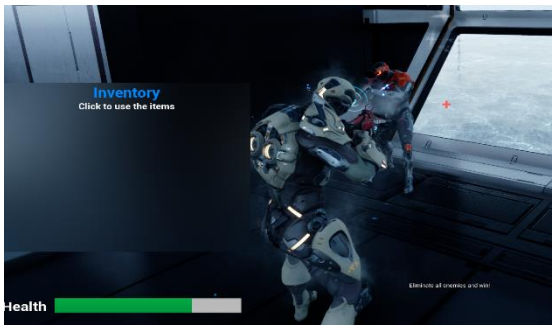
By clicking Start Game, you enter the actual game. Thus, the player's character appears in the middle of the screen, the weapon's target in the middle, and in the lower left part, a green bar representing the player's life. If all the green color disappears, the player dies and loses. When the player dies or wins, a 5-second timer appears, after which the game restarts, either to give you a chance to win if you lost, or simply to continue playing. Also, at the bottom center of the screen, there is a text indicating the mission received once you enter the game.

The game restarts and continues when you win...

Another chance to win!

Pressing the Tab key, you open, on the left, a box called Inventory, in which there are some elements, "food" we could say, which, once used, give you back some of your life if you click on them after being injured by enemies.





Pause in the game is done with the help of the Esc key, and to return, press the Resume button. Two other options when you press Esc are to return to the main menu of the game - Main Menu and to exit the game - Quit Game.

3. Technical Implementation Details

Unreal Engine 4 is an engine created by Epic Games that allows you to create your own game, the only limits of this engine being imagination and knowledge at the time you want to implement a certain idea.

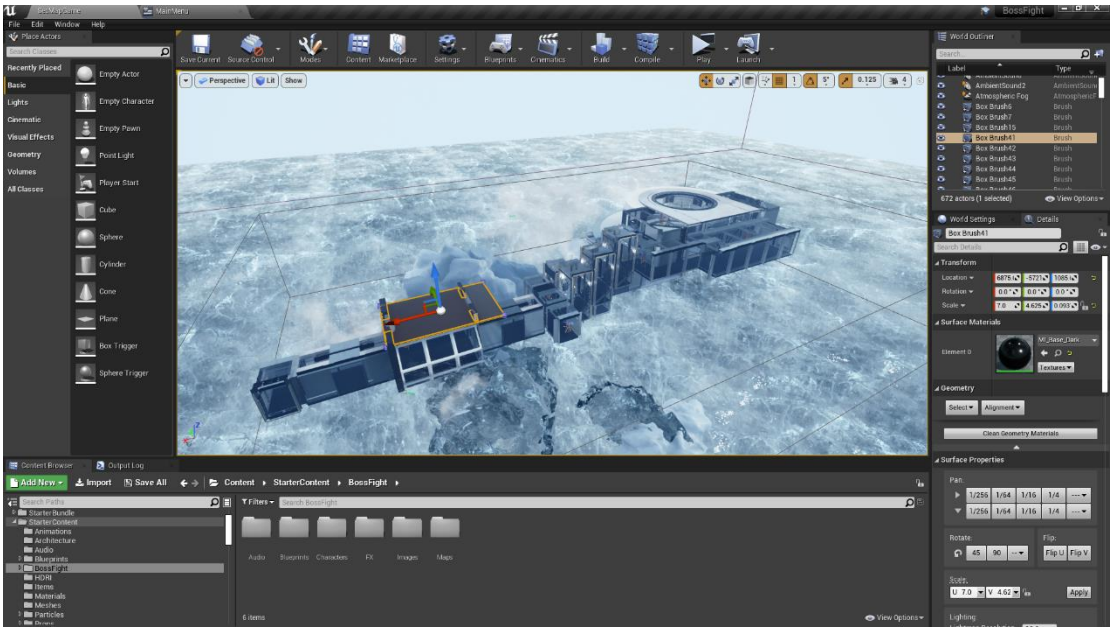
3.1 Graphics

When you enter the engine, it prompts you to create a new project or open an existing one. If you're new to it, everything seems quite confusing due to the numerous options available. However, after spending some time with it, it becomes less daunting. Each option is placed in a certain location for a reason, so you quickly start to understand what you can do.

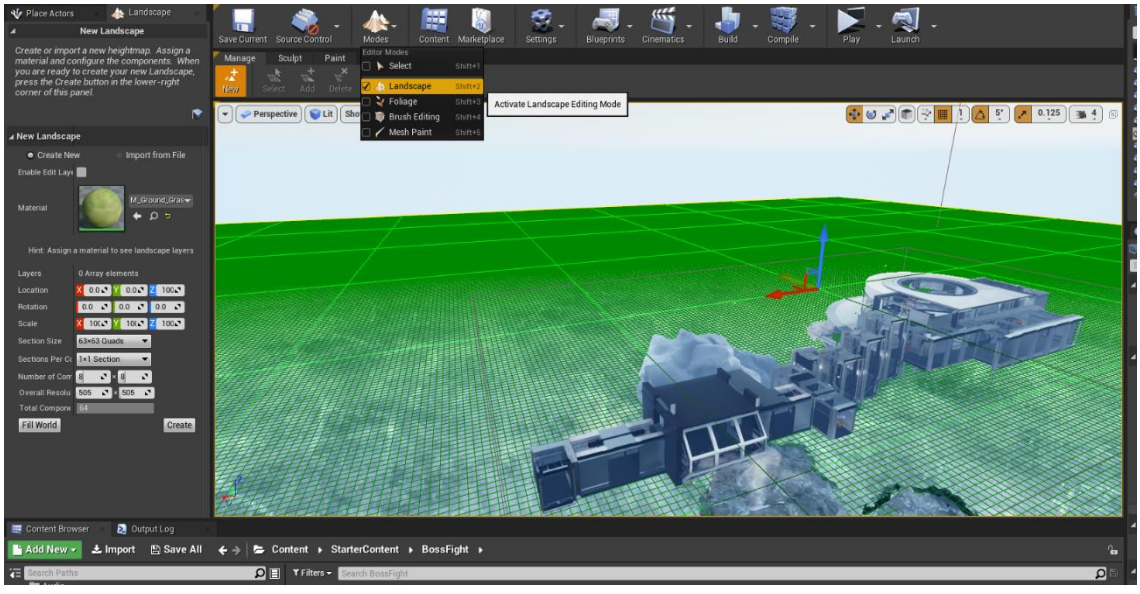
On the left side of the engine, various classes of objects are located, ranging from actual bodies like cylinders, cubes, planes, to visual effects, sounds, volumes, actor classes, characters, potions, etc. At the bottom is the Content Browser, which holds all the files of the respective project, where objects, textures, images, animations, shapes, particles, and much more can be added and found. On the right are generally panels showing the properties of the selected object, in World Settings, and all the objects present in that project that you've placed in that created world, in World Outliner.

So, there are countless options, such as modifying various properties of objects, or tools to create the map you're on, the terrain, sculpting it in whichever way you want, changing the position of objects, size, angle, different perspectives to view the created level, and much more, all of which have helped in creating the environment in which the game unfolds.

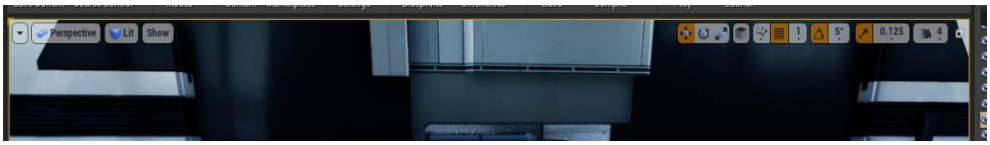
The map on which the game unfolds is created from various architectural objects, textures, and effects (for example, a texture of ice is added to the terrain).



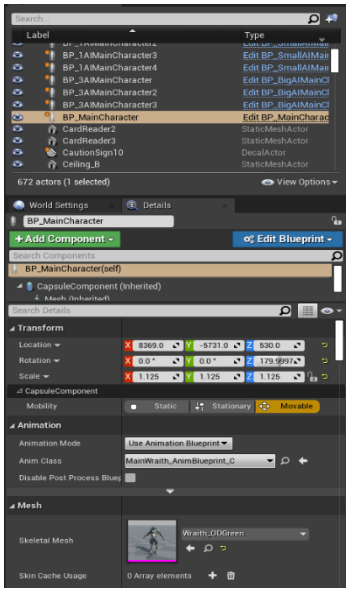
With the help of the Landscape tool, you create the terrain.



Options for changing perspective and various resizing options.



Multiple options that can be modified on a particular object/body.

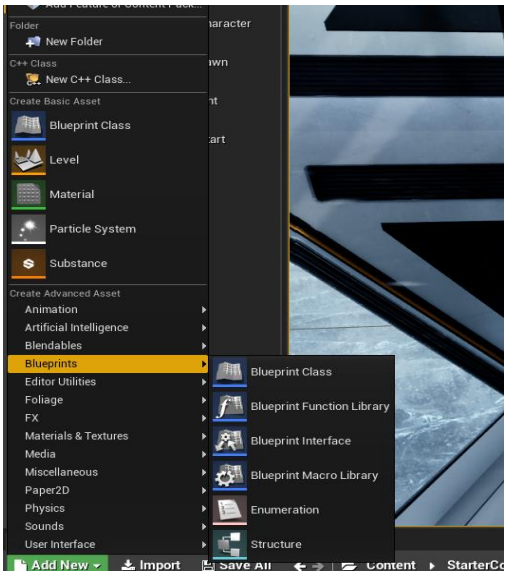


3.2 Programming

Programming in Unreal Engine 4 can be done in two ways: through writing code using the C++ language or with the help of so-called Blueprints.

Blueprints

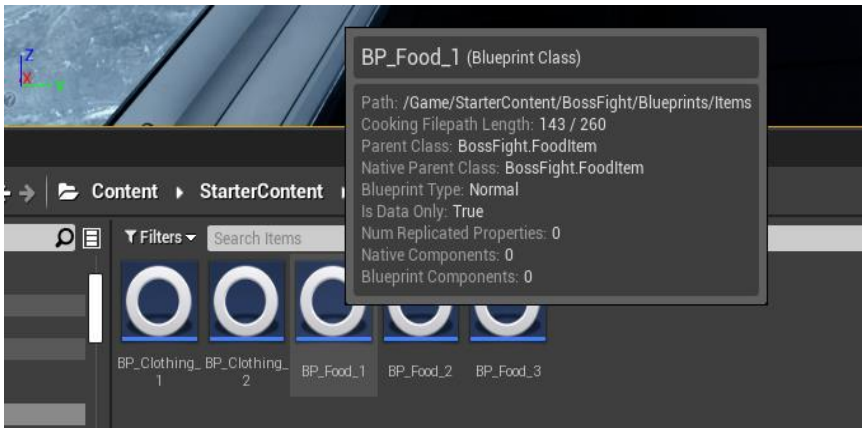
Blueprints are a system specific to this engine, providing a way to script by creating classes and structures much easier, as it's done visually using boxes that you connect to each other, each representing functions, variables, etc.



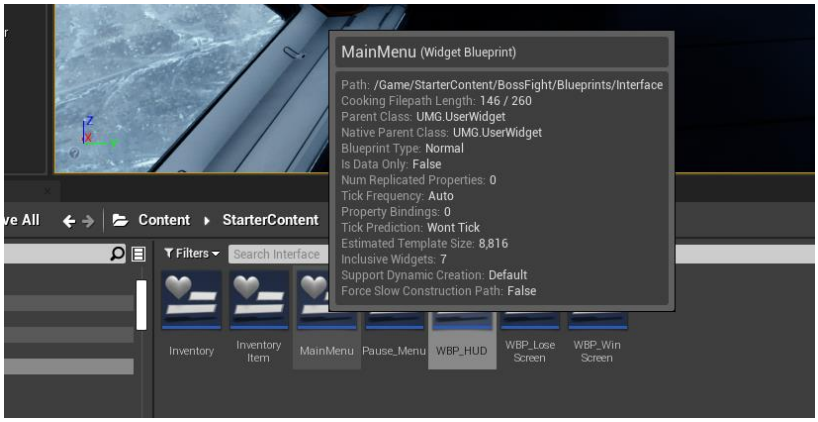
UMG (Unreal Motion Graphics UI Designer) is a visual tool for creating the user interface, which can be used to create interface elements such as in-game HUDs, menus, or other graphical interface-related elements. With these, various interfaces of the game were created, such as the main menu, pause menu, player's life (green bar), Lose Screen, Win Screen.

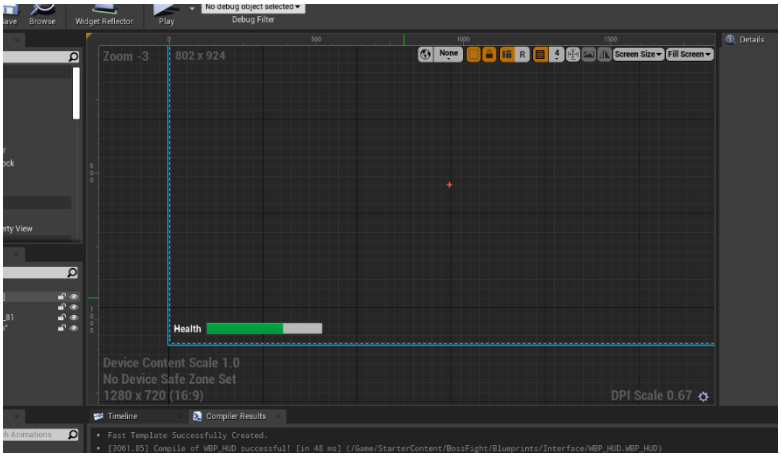
Widgets (GUI) are control elements in a graphical user interface, being an interactive element, such as a button or a scroll bar. With these, we created the inventory items that can be consumed to increase your life.

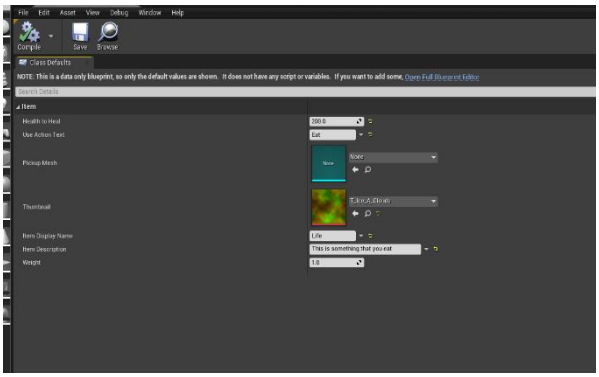
The "Food" from the inventory that increases life when used, class of objects.

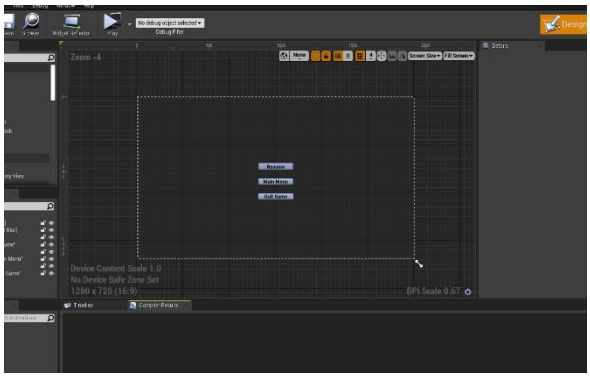


Interfaces present in the game: inventory, menus, life bar, weapon target.



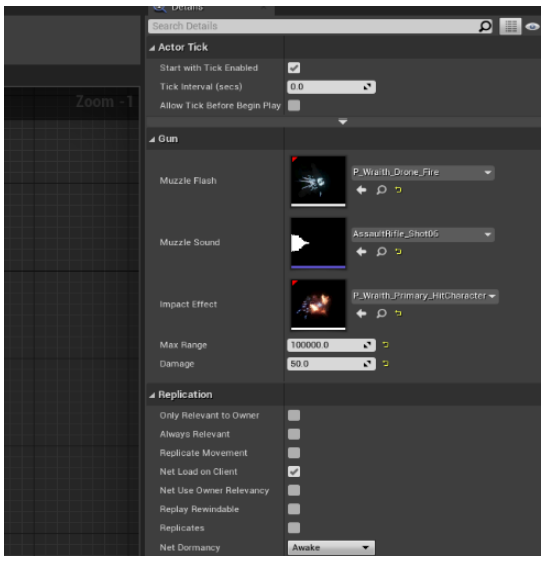




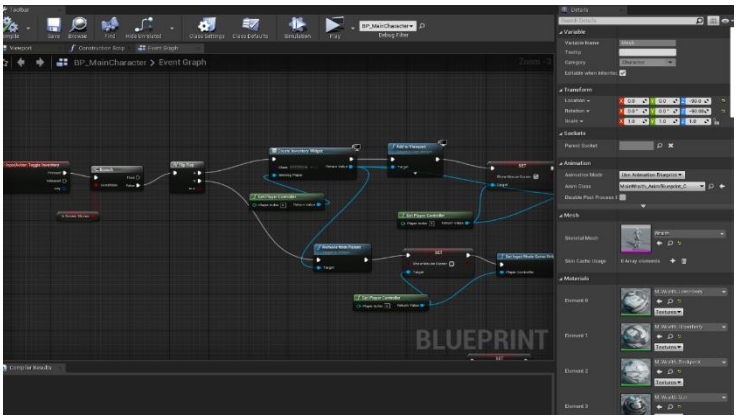


Objects such as the weapon you shoot with, the player character, the AIs, and those that appear in the level are Blueprint Classes derived from the Classes implemented in C++ to be used in the actual game world and to add different animations, textures, and other properties to them.

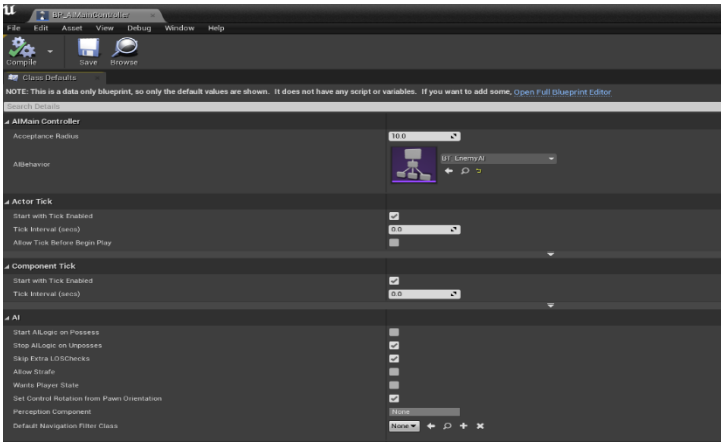
The particles and sounds for the weapon were added after implementing functions for various parts of the weapon in the Gun.h/Gun.cpp class.



Blueprint of the player character; those connected boxes are a sequence of code specific to the blueprints, involving linking the interfaces together and allowing the player to interact with them.



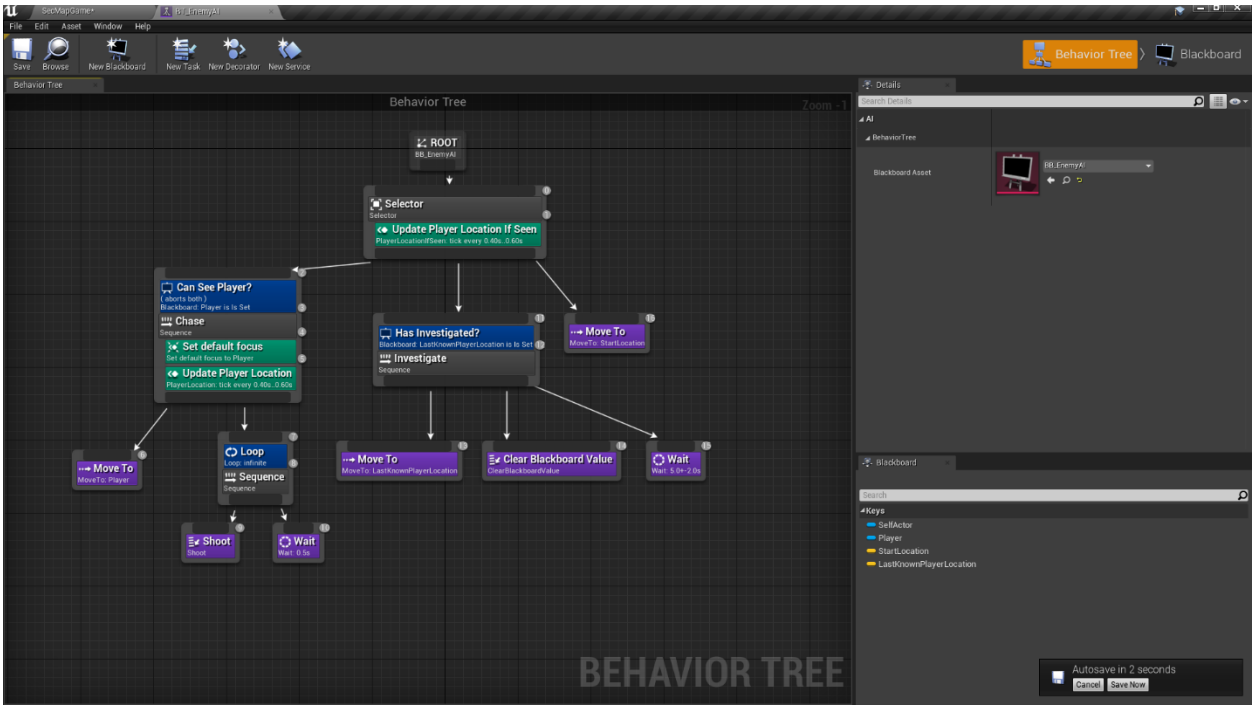
The Blueprint controlling the AI present in the game

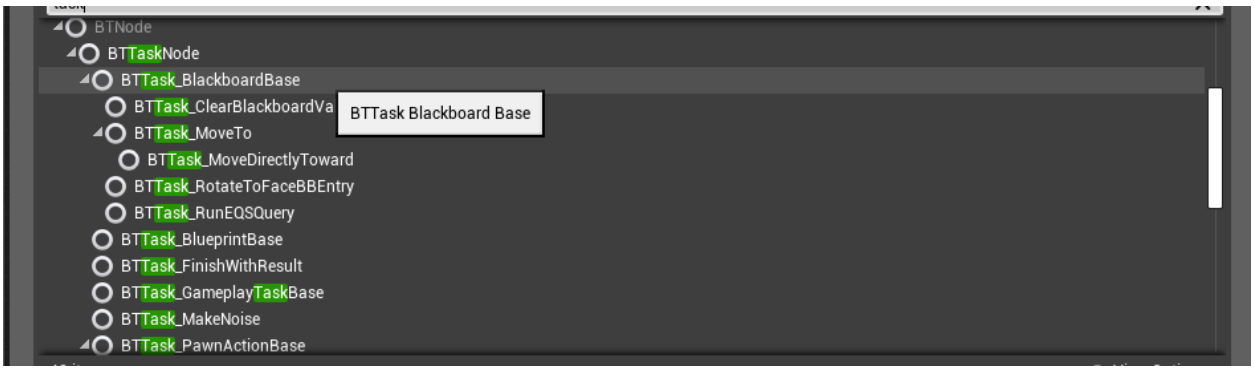


The HUD (interface) and the Win and Lose screens are declared within a C++ class.

```
UCLASS()
class BOSSFIGHT_API ABossFightController : public APlayerController
{
    GENERATED_BODY()

public:
    virtual void GameHasEnded(class AActor* EndGameFocus = nullptr, bool bIsWinner = false) override;

protected:
    virtual void BeginPlay() override;

private:
    // Win/Lose screen widgets
    UPROPERTY(EditAnywhere)
    TSubclassOf<class UUserWidget> HUDClass;
    UPROPERTY(EditAnywhere)
    TSubclassOf<class UUserWidget> WinScreenClass;
    UPROPERTY(EditAnywhere)
    TSubclassOf<class UUserWidget> LoseScreenClass;

    UPROPERTY(EditAnywhere)
    float RestartDelay = 5;

    FTimerHandle RestartTime;

    UUserWidget* HUD;

};
```

The AI (Artificial Intelligence) in the game is based on a Behavior Tree where different components are added in the game, such as tanks, for example, if the AI can see the player, if not to investigate from one side to another until it sees the player, and when it sees it, to start shooting. Each task has so-called decorators, which help in detecting the player or the player's last location, and depending on what it detected, it performs an action (Move to, Shoot, Wait, etc.).





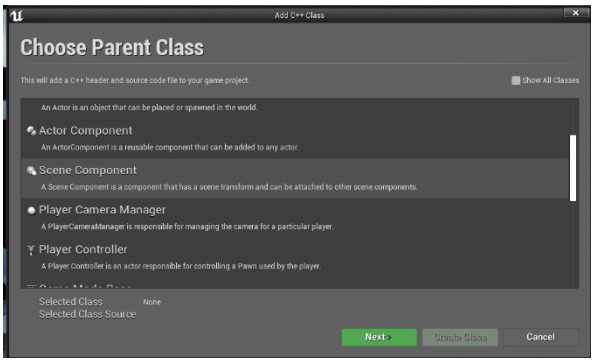
Programming in C++

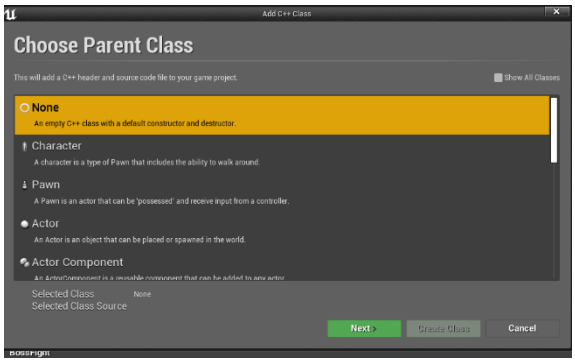
Visual Studio 2019 is an integrated development environment by Microsoft that helps in developing various applications on any platform, such as Unreal Engine 4.

Everything on the map, such as the player, AIs, weapon you shoot, widgets used to create the game interface, has a supporting C++ code that helps them function.

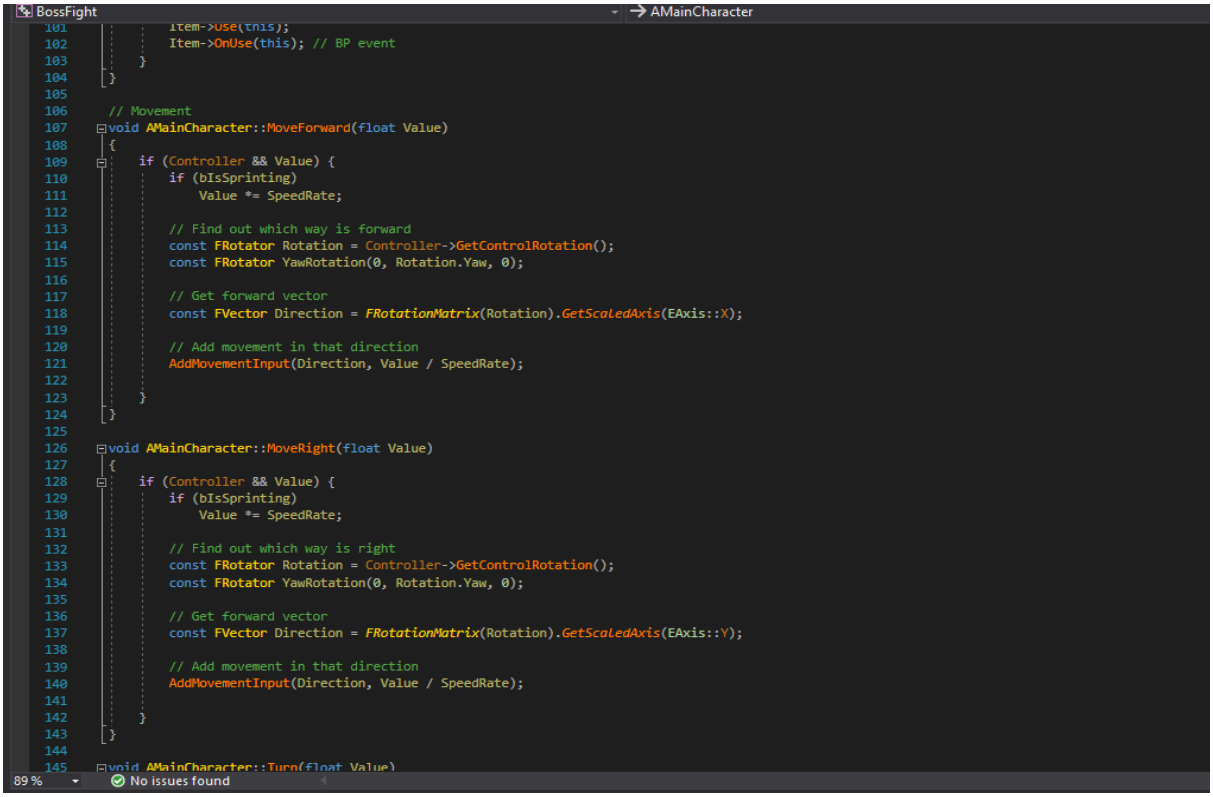
There are classes like Actors, Pawns, Characters with the ability to move, Player Controller, interface, HUD, Widgets, etc.

Different classes can be created based on the need.





The character's movement is based on assigning specific variables, vectors, and rotation, and adding them as parameters to a predefined function representing the character's movement, which is based on these variables to calculate the correct direction. For camera movement up/down, left/right, predefined functions are used because they are very optimal movement functions, calculating as a multiplication between the value given in the project settings (1 or -1 for different directions), a value representing the angle of rotation, and to have continuous rotation, we also multiply by a delta time, leading to smooth rotation. Each function has a suggestive name that seems to tell you what it does.



To shoot with the weapon, you need to obtain the correct controller of the owner, as otherwise, you cannot track the bullet trajectory, and it won't hit. If it's correct, we calculate the location where the bullet will hit using the direction, location, and rotation, and the bullet trajectory and hit location are simulated.

```cpp
bool AGun::GunTrace(FHitResult& Hit, FVector& ShotDirection)
{
    // If we cant get owner controller then we cant do a gun trace and hit anything
    AController* OwnerController = GetOwnerController();
    if (OwnerController == nullptr)
        return false;

    FVector Location;
    FRotator Rotation;
    OwnerController->GetPlayerViewPoint(Location, Rotation);
    ShotDirection = -Rotation.Vector();

    FVector End = Location + Rotation.Vector() * MaxRange;
    FCollisionQueryParams Params;
    Params.AddIgnoredActor(this);
    Params.AddIgnoredActor(GetOwner());
    return GetWorld()->LineTraceSingleByChannel(Hit, Location, End, ECollisionChannel::ECC_GameTraceChannel1, Params);
}

AController* AGun::GetOwnerController() const
{
    APawn* OwnerPawn = Cast<APawn>(GetOwner());
    if (OwnerPawn == nullptr)
        return nullptr;
    return OwnerPawn->GetController();

}
```

```cpp
void AGun::PullTrigger()
{
    UGameplayStatics::SpawnEmitterAttached(MuzzleFlash, GunMesh, "MuzzleFlashSocket");
    UGameplayStatics::SpawnSoundAttached(MuzzleSound, GunMesh, "MuzzleFlashSocket");

    FHitResult Hit;
    FVector ShotDirection;
    bool bSuccess = GunTrace(Hit, ShotDirection);
    if (bSuccess)
    {
        // Shot direction and damage done
        UGameplayStatics::SpawnEmitterAtLocation(GetWorld(), ImpactEffect, Hit.Location, ShotDirection.Rotation());

        AActor* HitActor = Hit.GetActor();
        if (HitActor)
        {
            FPointDamageEvent DamageEvent(Damage, Hit, ShotDirection, nullptr);
            AController* OwnerController = GetOwnerController();
            HitActor->TakeDamage(Damage, DamageEvent, OwnerController, this);
        }
    }

}

// Called when the game starts or when spawned
```

When you hit the target, damage is applied to that target. Thus, the life of the one hit decreases. The taken/caused damage is checked using a damage variable, calculated with a special function, as the minimum between the current life and the damage taken, after which we subtract the damage from the character's/enemy's life. When the life variable reaches 0, an IsDead() function is called, which returns null, and with the help of blueprints, it produces an death animation in the game.

```cpp
void AMainCharacter::Shoot()
{
    Gun->PullTrigger();
}

float AMainCharacter::TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent, class AController* EventInstigator, AActor* DamageCauser)
{
    float DamageToApply = Super::TakeDamage(DamageAmount, DamageEvent, EventInstigator, DamageCauser);
    DamageToApply = FMath::Min(Health, DamageToApply);
    Health -= DamageToApply;
    UE_LOG(LogTemp, Warning, TEXT("Health left %f"), Health); // Just Log output

    if (IsDead())
    {
        AMainBossFightGameMode* GameMode = GetWorld()->GetAuthGameMode<AMainBossFightGameMode>();
        if (GameMode != nullptr)
        {
            GameMode->PawnKilled(this);
        }
        DetachFromControllerPendingDestroy();
        GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    }

    return DamageToApply;

}
```

```cpp
bool AMainCharacter::IsDead() const
{
    return Health <= 0;
}

float AMainCharacter::GetHealthPercent() const
{
    return Health / MaxHealth;
}
```

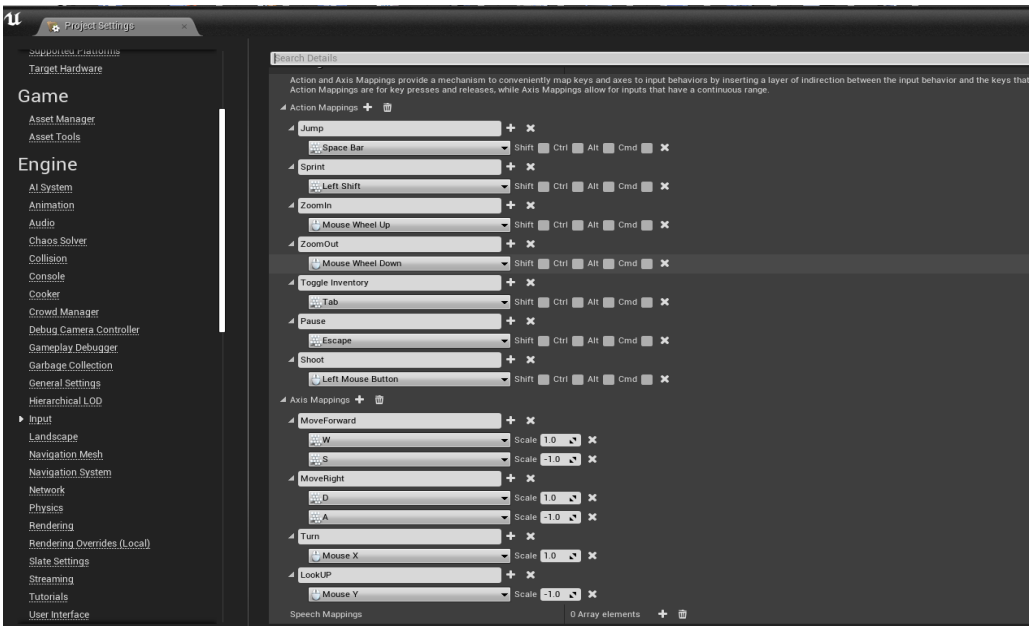
Camera Zoom In/Out and character sprint are the easiest to implement. For camera zoom in/out, we only need to increment/decrement the arm between the camera and the character if the size is at a minimum/maximum value. For the character's sprint, we need a variable that "keeps track" if we press the key or not to sprint, if the variable is true, meaning we press Shift, the initial walking speed is doubled.

```
// Sprint
void AMainCharacter::BeginSprint()
{
    bIsSprinting = true;
}

void AMainCharacter::StopSprint()
{
    bIsSprinting = false;
}

// Camera zoom
void AMainCharacter::CameraZoomIn()
{
    float var = 10.f;
    if (CameraZoom >= 250.f) {
        CameraBoom->TargetArmLength = CameraZoom;
        CameraZoom -= var;
    }
    else {
        CameraBoom->TargetArmLength = CameraZoom;
    }
}

void AMainCharacter::CameraZoomOut()
{
    float var = 10.f;
    if (CameraZoom <= 350.f) {
        CameraBoom->TargetArmLength = CameraZoom;
        CameraZoom += var;
    }
    else {
        CameraBoom->TargetArmLength = CameraZoom;
    }
}
```

The character is assigned the functions created to do what we created in functions, for example, to move in any direction, to jump, to shoot with the weapon, to rotate the camera and look around, depending on the utility, either through the action of a key or through direction and rotation, movement on axes.

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

    // Gameplay key bindings
    PlayerInputComponent->BindAxis("MoveForward", this, &AMainCharacter::MoveForward);
    PlayerInputComponent->BindAxis("MoveRight", this, &AMainCharacter::MoveRight);

    // "Turn" handles devices that provide an absolute delta, like a mouse
    // "TurnAtRate" handles devices that we choose to treat as a rate of change, like a joystick
    PlayerInputComponent->BindAxis("Turn", this, &AMainCharacter::Turn);
    PlayerInputComponent->BindAxis("TurnAtRate", this, &AMainCharacter::TurnAtRate);
    PlayerInputComponent->BindAxis("LookUp", this, &AMainCharacter::LookUp);
    PlayerInputComponent->BindAxis("LookUpAtRate", this, &AMainCharacter::LookUpAtRate);

    // Sprint
    PlayerInputComponent->BindAction("Sprint", EInputEvent::IE_Pressed, this, &AMainCharacter::BeginSprint);
    PlayerInputComponent->BindAction("Sprint", EInputEvent::IE_Released, this, &AMainCharacter::StopSprint);

    // Jump
    PlayerInputComponent->BindAction("Jump", EInputEvent::IE_Pressed, this, &ACharacter::Jump);
    // Handle touch inputs, like jump
    PlayerInputComponent->BindTouch(IE_Pressed, this, &AMainCharacter::TouchStarted);

    // Zoom
    PlayerInputComponent->BindAction("ZoomIn", EInputEvent::IE_Pressed, this, &AMainCharacter::CameraZoomIn);
    PlayerInputComponent->BindAction("ZoomOut", EInputEvent::IE_Released, this, &AMainCharacter::CameraZoomOut);

    // Shoot
    PlayerInputComponent->BindAction("Shoot", EInputEvent::IE_Pressed, this, &AMainCharacter::Shoot);
}
```

As I said above, each function for character movement direction, camera, which considers a certain direction on the axis, calculates a value based on the project settings to give forward/backward, left/right movement. This value can be 1 or -1. Also, each function, not just the directional ones that consider an axis but also those interactive with the help of keys, for example, shooting with the weapon, sprinting, jumping, are assigned the specific keys for each command.



4. Necessary Hardware and Software Resources

Since the game is still in its early stages, not yet optimized, the hardware resources are quite high. Thus, it would be necessary to have:

- Windows 10
- Mouse and keyboard
- A powerful processor, at least i5 8th gen
- Dedicated video card, GTX 1050
- At least 4GB of RAM
- HDD/SSD with 5GB of available space

5. Development Possibilities

Being in the early stage, many things can be added to what is in the game so far, such as:

- New levels
- Difficulty levels
- Terrain, map, new textures
- New enemies, different ones
- Abilities
- AI improvement
- Game optimization
- Etc.

6. Bibliography

Official documentation website of Unreal Engine 4: https://docs.unrealengine.com/en-US/ProgrammingAndScripting/ProgrammingWithCPP/index.html

All existing functions: https://docs.unrealengine.com/en-US/API/QuickStart/index.html

Architecture elements, animation, environment, materials, textures, weapons, visual effects, characters: https://www.unrealengine.com/marketplace/en-US/store

Other Unreal Engine 4 tutorials: https://unrealcpp.com/

https://www.youtube.com/c/ReubenWardTutorials/playlists