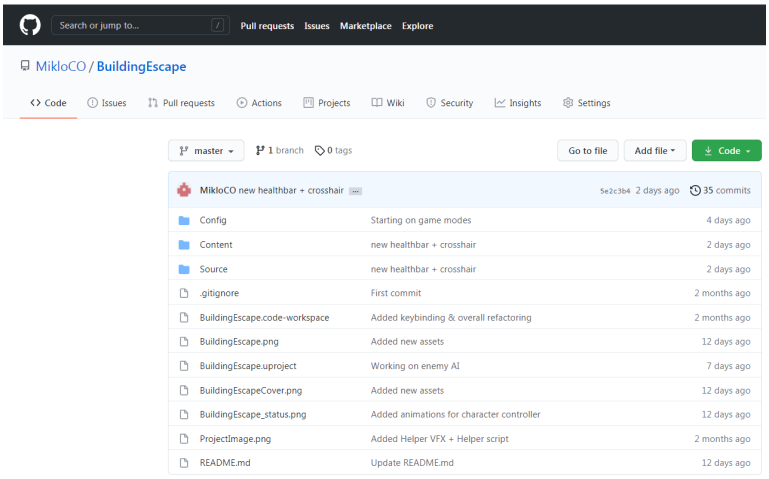
Project Post-mortem of worksample: UE4 V.22.3, C++, Maya & Substance Painter

I have created a game with 2 levels, using C++ where I have created some of the implementations in blueprints. The environment is modelled and textured by me using Maya and substance painter. I also used the UE4 particle system editor to create the portal. The characters, sound effects, animations and muzzle flash derive from the UE4 asset store. The first level consists of a single APawn, where the player can pick up and throw objects. The second level consists of a player ACharacter where the player can take and deal damage and has a health bar.

Video demonstration of both levels: Link. **Link to this projects repository:** Link

I have used Github through the entire project process where I learned how handy the tool is as well as github declines pushes exceeding 2GB worth of assets (teaching me to be cautious with how many assets I add to a level before each commit). And how useful it is when working from different computers and the possibility to revert bad commits using GIT bash.



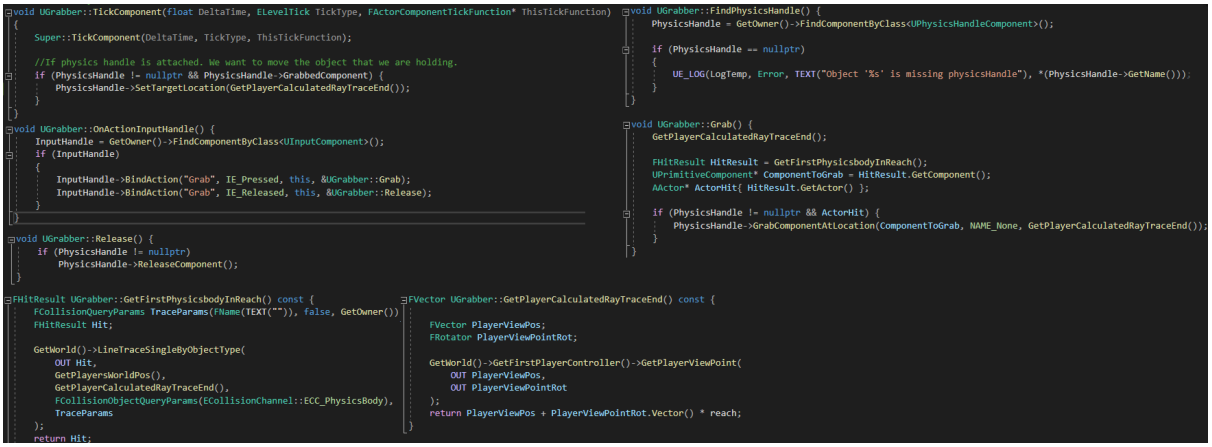
Level 1 : Using raytrace to pick up object (Scene TokyoNightScene)

(The original plan was to make a night scene, but the scene looked so good in the daylight that I left it be.)

In the first level I use DefaultPawn_BP as “default pawn class” in the BP_BuidlingEscapeMode to setup to spawn my pawn on play. Upon this pawn I have connected a Grabber (my own created C++ class) and physicshandle. In the code below, in **GetPlayerCalculatedRayTraceEnd**, “OUT” is merely a macro that we are borrowing from C# to make it clear that we are passing a reference (not const) inside a method call and not a function call.

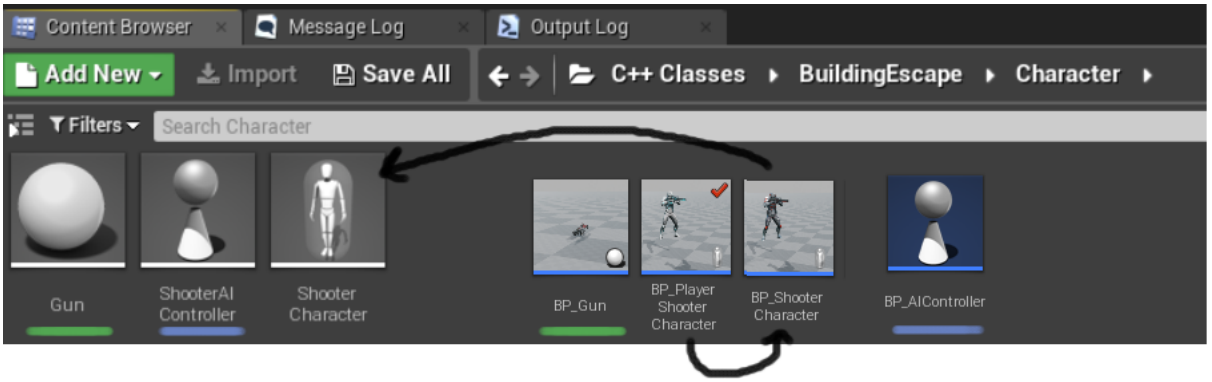
In BeginPlay() we are calling both FindPhysicsHandle & ActionInputHandle. Our PhysicsHandle is a nullptr precaution to make sure we do not end up in a nullpointer exception situation. By saying if(PhysicsHandle != nullptr &&) we make sure that we return (and do not execute the rest of the function) if PhysicsHandle is null. Because with && the rest of the condition won’t be evaluated. So in short we are checking if our pawn has a PhysicsHandle attached in the BP.

We use UInputComponent to keybind our Pawn (child to Actor). Now, moving on to our function Grab, we call our function “**GetPlayerCalculatedRayTraceEnd**” that returns a const FVector. Here we define parameters false and the actor (getOwner) to be passed in as a collision function. We also instantiate a FHitResult which we use in our method call to raytrace in the world. We will return the first blocking hit, (in our scene it will be either a cone or garbage can). We also have “**GetplayerWorldPos**” which returns a FVector containing the information of the actorLocation in the world. Then our “**GetPlayerCalculatedRayTraceEnd**” will return a FVector containing information about the pawns position in the world and rotation multiplied by the raytrace distance (reach), which right now is only accessible to adjust in the header file, but could be carved out as a property to the blueprint editor using *UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Raytrace settings", meta = (AllowPrivateAccess = "true"))*.



```
void UGrabber::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    //If physics handle is attached. We want to move the object that we are holding.
    if (PhysicsHandle != nullptr && PhysicsHandle->GrabbedComponent) {
        PhysicsHandle->SetTargetLocation(GetPlayerCalculatedRayTraceEnd());
    }
}
void UGrabber::OnActionInputHandle() {
    InputHandle = GetOwner()->FindComponentByClass<UInputComponent>();
    if (InputHandle)
    {
        InputHandle->BindAction("Grab", IE_Pressed, this, &UGrabber::Grab);
        InputHandle->BindAction("Grab", IE_Released, this, &UGrabber::Release);
    }
}
void UGrabber::Release() {
    if (PhysicsHandle != nullptr)
        PhysicsHandle->ReleaseComponent();
}
FHitResult UGrabber::GetFirstPhysicsbodyInReach() const {
    FCollisionQueryParams TraceParams(FName(TEXT("")), false, GetOwner());
    FHitResult Hit;

    GetWorld()->LineTraceSingleByObjectType(
        OUT Hit,
        GetPlayersWorldPos(),
        GetPlayerCalculatedRayTraceEnd(),
        FCollisionObjectQueryParams(ECollisionChannel::ECC_PhysicsBody),
        TraceParams
    );
    return Hit;
}
void UGrabber::FindPhysicsHandle() {
    PhysicsHandle = GetOwner()->FindComponentByClass<UPhysicsHandleComponent>();

    if (PhysicsHandle == nullptr)
    {
        UE_LOG(LogTemp, Error, TEXT("Object '%s' is missing physicsHandle"), *(PhysicsHandle->GetName()));
    }
}
void UGrabber::Grab() {
    GetPlayerCalculatedRayTraceEnd();

    FHitResult HitResult = GetFirstPhysicsbodyInReach();
    UPrimitiveComponent* ComponentToGrab = HitResult.GetComponent();
    AActor* ActorHit{ HitResult.GetActor() };

    if (PhysicsHandle != nullptr && ActorHit) {
        PhysicsHandle->GrabComponentAtLocation(ComponentToGrab, NAME_None, GetPlayerCalculatedRayTraceEnd());
    }
}
FVector UGrabber::GetPlayerCalculatedRayTraceEnd() const {

    FVector PlayerViewPos;
    FRotator PlayerViewPointRot;

    GetWorld()->GetFirstPlayerController()->GetPlayerViewPoint(
        OUT PlayerViewPos,
        OUT PlayerViewPointRot
    );
    return PlayerViewPos + PlayerViewPointRot.Vector() * reach;
}
```

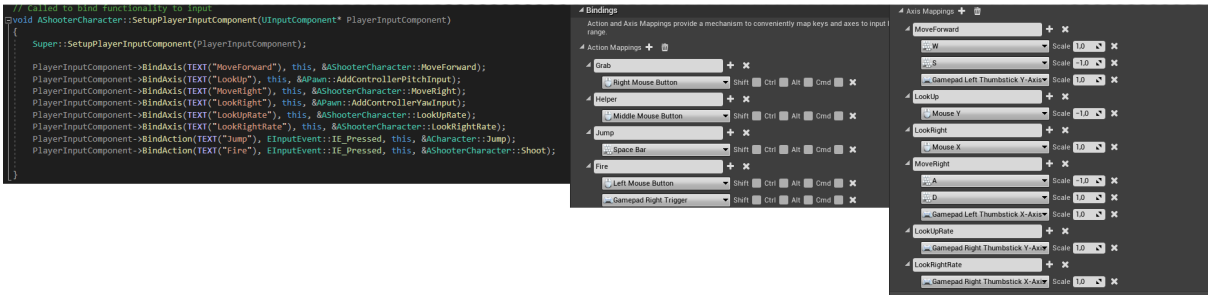
Parented blueprints : Level 2 (Scene Sandbox)



I have created the base classes “Shooter character”, “shooterAIController” and “Gun” which I use as parents in corresponding blueprints like “BP_ShooterCharacter” and “BP_Gun”. Why? Because Shooter Character is our parent to player and enemy, e.g. both of them have health, but a different amount, which is customizable because these properties are tweaked in the BP.

```
// Called when the game starts or when spawned
void AShooterCharacter::BeginPlay()
{
    Super::BeginPlay();

    Health = MaxHealth;

    gun = GetWorld()->SpawnActor<AGun>(GunClass);
    GetMesh()->HideBoneByName(TEXT("weapon_r"), EPhysBodyOp::PBO_None);
    gun->AttachToComponent(GetMesh(), FAttachmentTransformRules::KeepRelativeTransform, TEXT("weapon_rSocket"));
    gun->SetOwner(this);

}
```

On BeginPlay I spawn our gun class that through code removes the “weapon_r” socket. I also attach the gunMesh to the ACharacter (ShooterCharacter). Because both Enemy (BP) and Player (BP) inherit this class, I pass “this” as the class instance.



For pitch and yaw key bindings it was sufficient to use APawns ControllerPitch and controllerYaw. I also used ACharacters jump function in this project. The rest of the key bindings are however unique and therefore I have created my own functions to e.g. calculate “LookUpRate” or “MoveRight” etc.

```
void AShooterCharacter::MoveRight(float AxisValue) {
    AddMovementInput(GetActorRightVector() * AxisValue);
}
```

So in MoveRight, we pass in AxisValue which we use to calculate the “intensity” of the Y vector length in world space. Then I pass this as an argument in AddMovementInput to move the Character in the world. As you see there is no MoveLeft function created, as we don’t need this, because if AxisValue goes below 0 then the movement will be in the opposite direction. Also, since

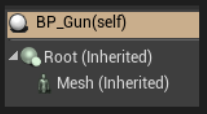
this is a character class we don't need to apply the movement in the Tick event as our subclass ACharacter does this automatically.

```
float AShooterCharacter::TakeDamage(float DamageAmount, struct FDamageEvent const& DamageEvent,
{
    float DamageApplied = Super::TakeDamage(DamageAmount, DamageEvent, EventInstigator, DamageC
    DamageApplied = FMath::Min(Health, DamageApplied);
    Health -= DamageApplied;

    if (IsDead()) {
        AShooterGameModeBase* GameMode = GetWorld()->GetAuthGameMode<AShooterGameModeBase>();

        if (GameMode != nullptr) {
            GameMode->PawnKilled(this);
        }

        DetachFromControllerPendingDestroy();
        GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);
    }
    return DamageApplied;
}
```

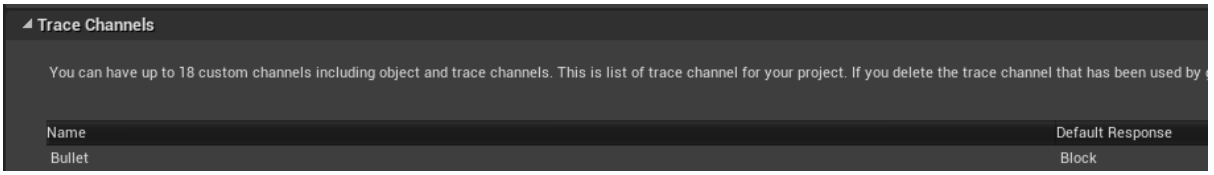
We have declared TakeDamage as a virtual function because we override this function from our derived class AActor. We check if Health of the current ACharacter is greater or equal to 0. We tell the current gamemode that the game has ended by triggering PawnKilled. Once dead we will detach our playercontroller (important, if we don't do this step, the eliminated enemy character will actually keep following /glitching after our player). And we also disable our collision channel on the capsulecomponent so that the eliminated character doesn't block our path etc.

Moving on to our **Gun** class, we setup the Root and Mesh in the constructor.

```
AGun::AGun()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    Root = CreateDefaultSubobject<USceneComponent>(TEXT("Root Component"));
    RootComponent = Root;

    Mesh = CreateDefaultSubobject< USkeletalMeshComponent>(TEXT("Base Mesh"));
    Mesh->SetupAttachment(RootComponent);
}
```



We define our function **PullTrigger** in this class, but we actually call it in the ShooterCharacter class (accessing member pulltrigger through a gun pointer). On this function call, we will spawn a particle effect (muzzle flash) and sound. We will also execute functioncall **GunTrace** which takes 2 the two parameters that we use to calculate the shot direction and return a FHit on our custom trace channel "Bullet". Why do we do this? Yes, to avoid the player being shot through walls. You can see it working here.



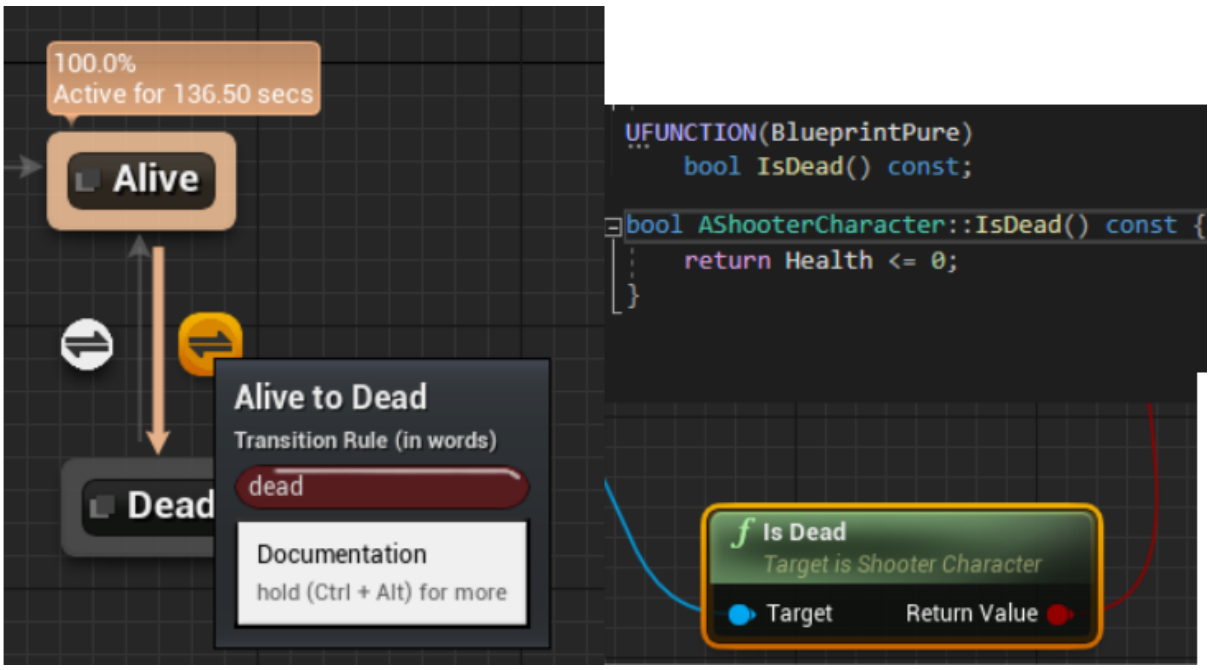
In **GunTrace**, we also calculate the position we fire from, shot range and direction (rotation). If we have hit something, we will spawn a particle effect at the hit location and a sound.

The 2 lines “Params.AddIgnoredActor(this); & Params.AddIgnoredActor(GetOwner());” prevents our NPC character from shooting themselves, so this is good to note. Pulltrigger will check if the bullet has hit. Now our bullet can hit regular objects (like a building- in this particular scene I actually removed the collisions and just put boxes with collisions on them. In a simple scene like this that works just fine, but if I had had a more complex environment with corners and shapes I would have opt-ed for a 3D tool to wrap the object with a collision there). No matter if we hit a wall or the character, the particle and sound will spawn on the Gun. However, if we hit our enemy characters, then we will do damage on the hit actor. Because there are several actors in the scene, we will instantiate a pointer that fetches the “hits owner controller”, and we will call the Actors default TakeDamage function (that we have overridden and defined in the **ShooterCharacter** class, hence the access to the takeDamage through our **ShooterCharacter** class).

```cpp
bool AGun::GunTrace(FHitResult& Hit, FVector& ShotDirection)
{
    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(
        OUT Hit,
        Location,
        End,
        ECollisionChannel::ECC_GameTraceChannel1,
        Params
    );
}
```

```cpp
void AGun::PullTrigger() {
    UGameplayStatics::SpawnEmitterAttached(MuzzleFlare, Mesh, TEXT("MuzzleFlashSocket"));
    UGameplayStatics::SpawnSoundAttached(MuzzleSound, Mesh, TEXT("MuzzleFlashSocket"));

    FHitResult Hit;
    FVector ShotDirection;

    bool bBulletHit = GunTrace(Hit, ShotDirection);

    if (bBulletHit) {
        UGameplayStatics::SpawnEmitterAtLocation(
            GetWorld(),
            BulletHit,
            Hit.Location,
            ShotDirection.Rotation()
        );

        UGameplayStatics::PlaySoundAtLocation(
            GetWorld(),
            ImpactSound,
            Hit.Location
        );

        AActor* HitCharacter = Hit.GetActor();

        if (HitCharacter != nullptr) {
            FPointDamageEvent DamageEvent(damage, Hit, ShotDirection, nullptr);
            AController* OwnerController = GetOwnerController();
            HitCharacter->TakeDamage(
                damage,
                DamageEvent,
                OwnerController,
                this
            );
        }
    }
}
```

Animation blueprints



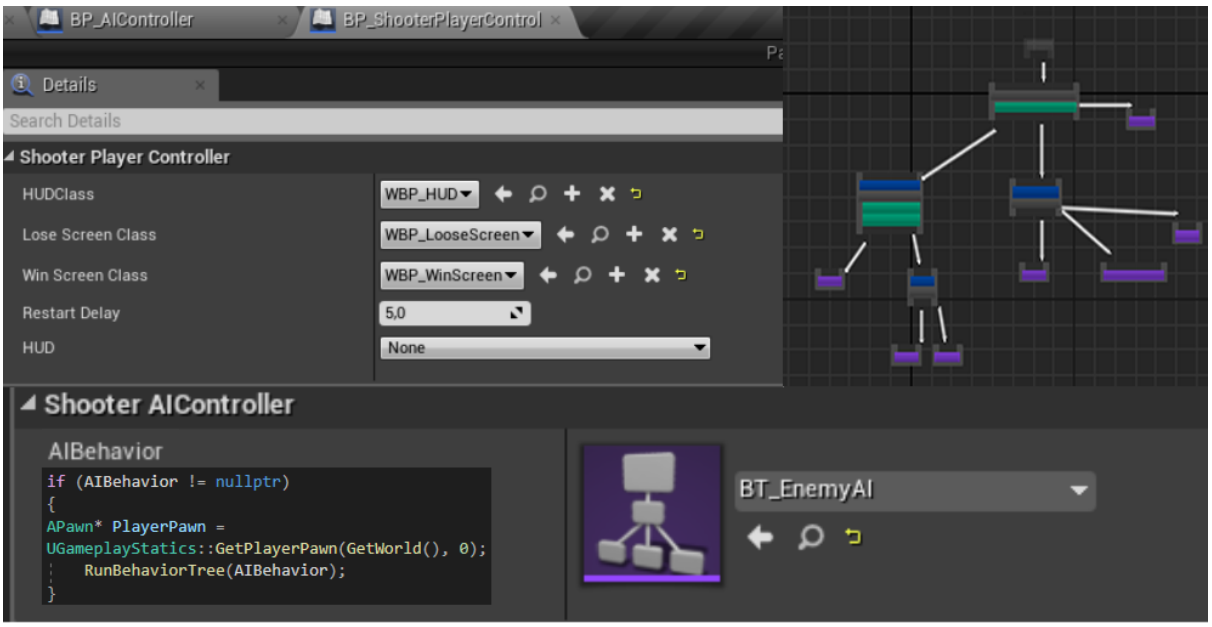
If our character has a health less or equal to 0, then we will set IsDead as true. We declared IsDead as a UFUNCTION(BlueprintPure), which means we can use it as a node in BP as above (because we have created this new BP node in C++).

Widget blueprints

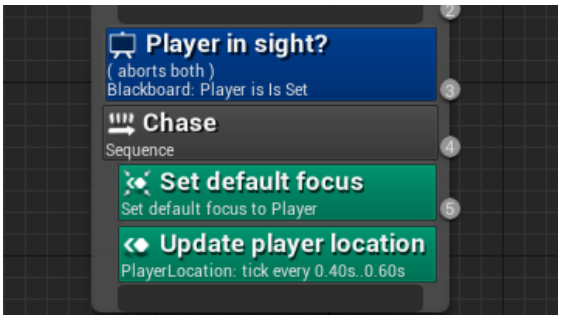
When the game has ended, we will display either the win screen or loose screen depending on bIsWinner (true = Player, false = enemy).

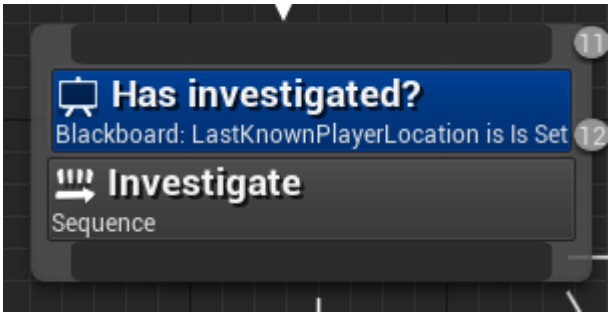
```cpp
void AShooterPlayerController::GameHasEnded(class AActor* EndGameFocus, bool bIsWinner)
{
    Super::GameHasEnded(EndGameFocus, bIsWinner);
    HUD->RemoveFromViewport();

    if (bIsWinner)
    {
        UUserWidget* WinScreen = CreateWidget(this, WinScreenClass);
        if (WinScreen != nullptr)
        {
            WinScreen->AddToViewport();
        }
    }

    else
    {
        UUserWidget* LooseScreen = CreateWidget(this, LoseScreenClass);
        if (LooseScreen != nullptr)
        {
            LooseScreen->AddToViewport();
        }
    }
    GetWorldTimerManager().SetTimer(RestartTimer, this, &APlayerController::RestartLevel, RestartDelay);
}
```

In the blueprint adaptation of playercontroller, I set the Widgets that should pop up when the game has ended. Now let's talk about BP AIController. In the C++ class we say "run bBehavior tree" if there is a behaviour tree selected in the BP "AIBehaviour" slot. (See below)

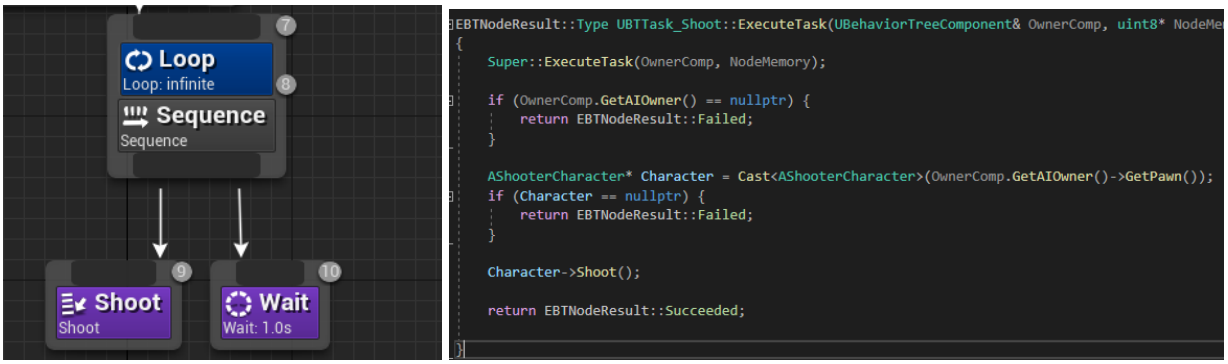


The enemies that are placed in the world will act according to the rules that we have setup in the Behaviour Tree. The behaviour tree consists of simple rules that will make the AI behave according to the given situation: Player in sight? & Has investigated?





In the below image we have two tasks that we will loop through as long as the blackboard condition “player in sight?” is true. When the task “Shoot” is active, we will run our custom created task defined in C++. We will check if GetAIOwner() on the **shootercharacter** will return an AIOwner. We will return “abort/Fail” on the task if we return nullptr on GetAIOwner. If these are not nullpointers, then we will call the characters shoot function. (Because ShooterCharacter is the parent of the **ShooterCharacter**). Here is a link to the full Behaviour Tree: Link.



Hand Drawn UI

I have used photoshop to create a killscreen that I use as an overlay in the blueprint widget, I will call this widget when the character has been eliminated. The same process is valid for then the player wins the game (shoots all the NPC:s).

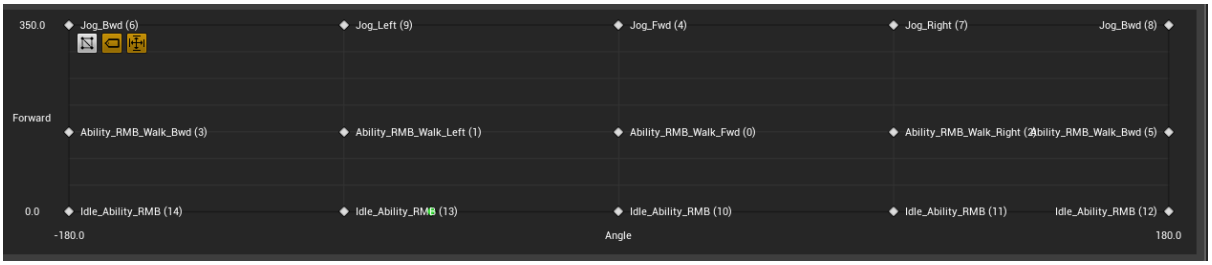
I have also created a crosshair and healthbar for the HUD widget.

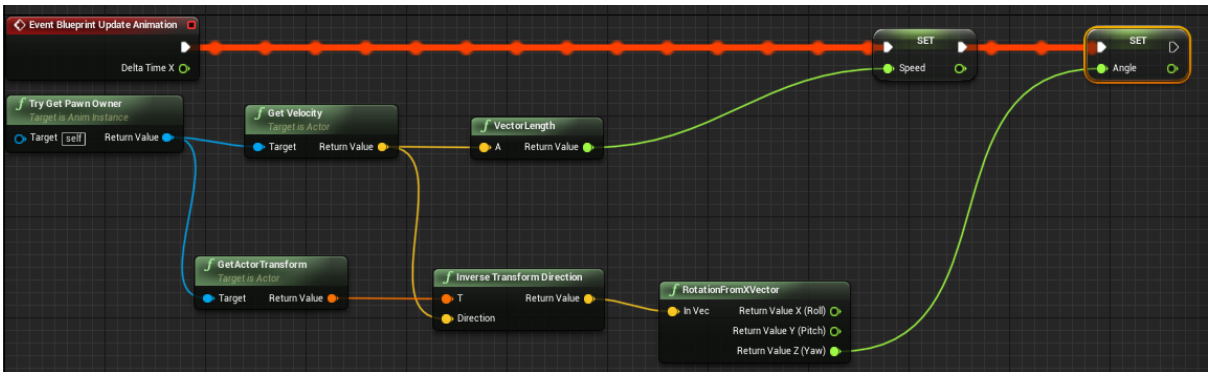
Link to gif displaying deathScreen event

Link to a cool shot of the enemies running out of the portal.

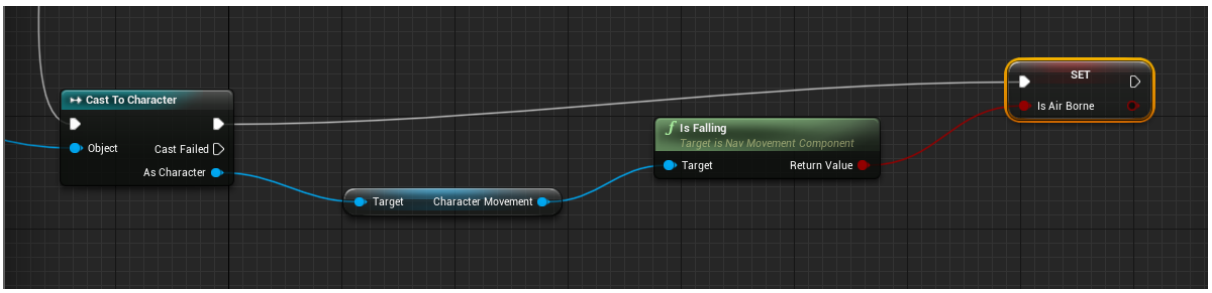
Animation Blendspace & Event graph

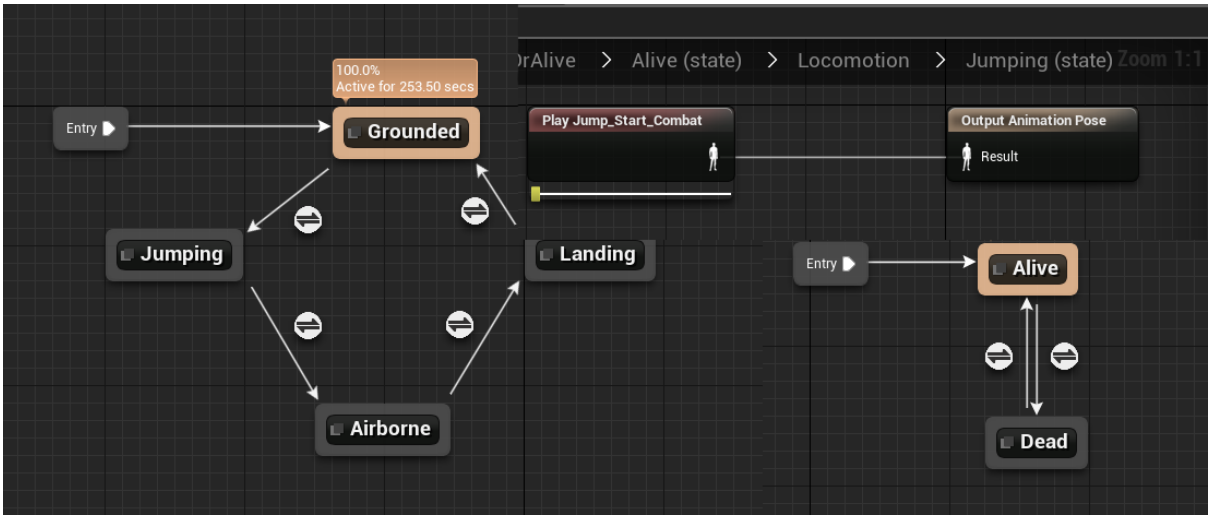
I have used Blendspace to seamlessly go between the character animations cycle of Jog, walk, turning and idle.





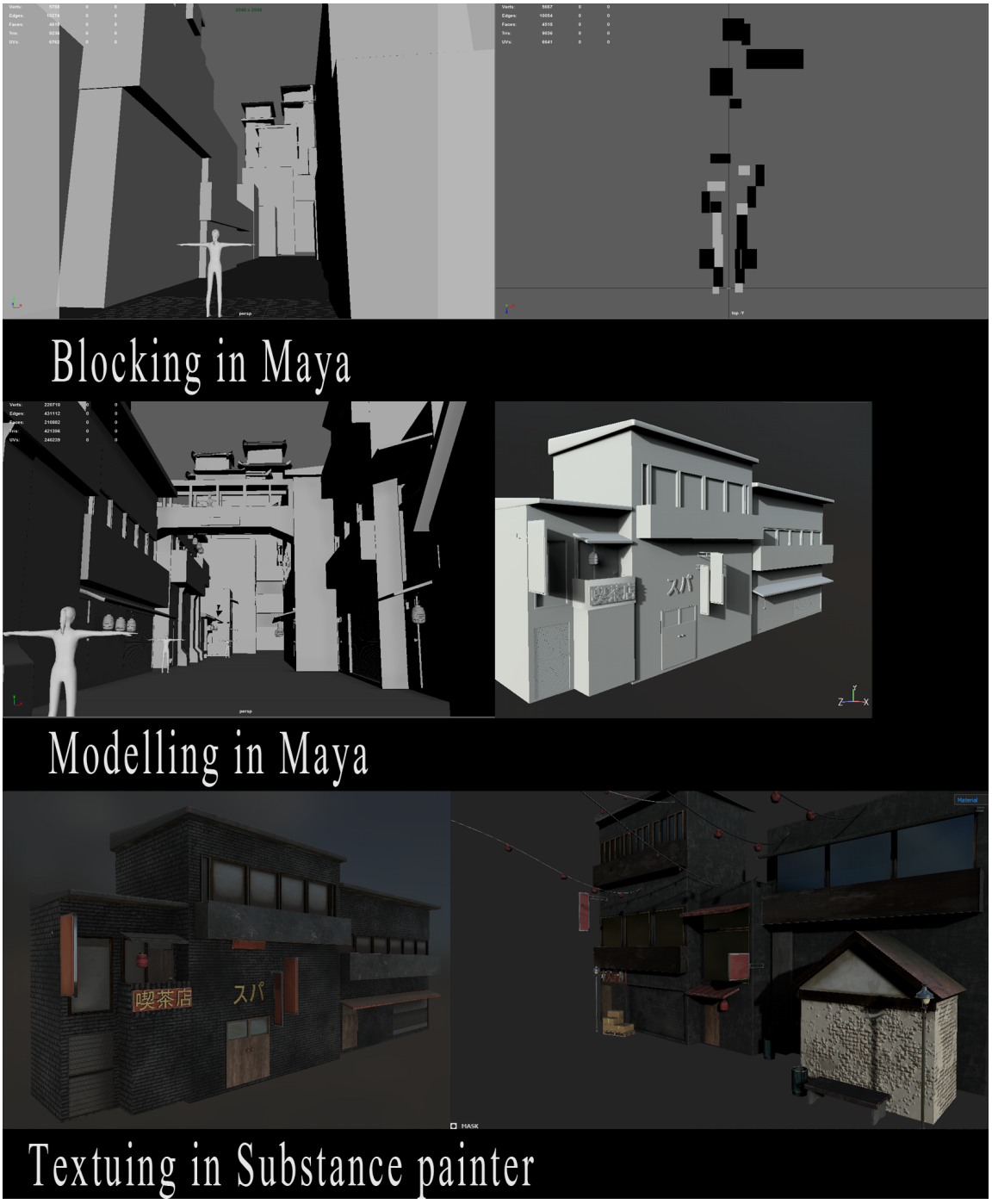
Using an Animation blueprint, I have had the support to use statemachines that will play different animations depending on the characters “state”, meaning, is character alive? Yes, then is character grounded, jumping, landing or airborne? Depending on the automatic rules set on each connection in the nodes, for example, the condition “Jumping to Airborne”, if the var IsAirBorne returns true, then this will enable the state airborn, and the jump animation will play.





Modelling process

I have created the scene using Maya and substance painter (SP) together with UDIM technique to UV-map and texture the scene. In the fourth picture, the mesh consists of 4 tiles, with 4 different mesh groups. Which enabled me to work with mesh layers in SP. In SP I worked with smart materials to quickly texture my environment as I was alone to texture this massive scene.



Below you can find links to my portfolio and github:
https://mikloco.github.io/portfolio/home.html
https://www.artstation.com/oliviamikler
https://github.com/MikloCO/BuildingEscape