# Korištenje C++ programskog jezika u stvaranju UE4 scene

**Cvjetković, Mitar**

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Mitar Cvjetković

# Korištenje C++ programskog jezika u stvaranju UE4 scene

Završni rad

Osijek, 2016.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Mitar Cvjetković

# Korištenje C++ programskog jezika u stvaranju UE4 scene

Završni rad
Mentor: doc. dr. sc. Domagoj Ševerdija

Osijek, 2016.

**Title: Using C++ for creating a scene in UE4**


## Sažetak:

Tema ovog rada je primjena znanja iz C++ programskog jezika kako bi se kreirala jednostavna scena u Unreal Engine 4 programu. Komponente koje čine takvu scenu mogu biti statičke ili dinamičke. Programiranjem u C++-u odnosno koristeći Visual Studio 2015, napravit ćemo osnove za dinamičke komponente. Te komponente će se sastojati od jednog kontrolabilnog lika odnosno igrača sa pogledom iz trećeg lica, nekoliko NPC-ova koji će ispisivati poruke na ekran kada im se približimo te nekoliko neprijatelja koji će pratiti igrača i pri kontaktu ga uništiti. Ipak, zbog jednostavnosti, njihova detaljnija obrada će se vršiti u Unreal Engineu koristeći opciju Blueprint, koja zapravo u potpunosti može zamijeniti programerski dio. Nakon što odredimo izgled i ponašanje dinamičkih komponenti scene, možemo prijeći na one statičke, odnosno na stvaranje reljefa i okruženja scene. Na kraju bit će potrebno po želji posložiti sve komponente i scena će biti spremna za interakciju s igračem preko odabranog kontrolera.



## Ključne riječi:

C++, programiranje, 3D scena, Unreal Engine 4, Blueprint



## Abstract:

The topic of this thesis is the usage of C++ programming skills in order to create a simple scene inside of Unreal Engine 4 program. The components that comprise a scene like that can be static or dynamic. By using C++, that is, by programming in Visual Studio 2015, we will make basics for those dynamic components which will be one controlable character/player with the third-person camera attached to it, a couple of NPCs which will "say" something once we get close enough to them, and a couple of enemies which will follow the player and destroy him upon contact. But, for keeping things simple, their in-detail editing will be done inside Unreal Engine using Blueprints which, in fact, can be used instead of C++ programming entirely. After we pick the look and behaviour of the dynamic components in the scene, we will start making the static ones, which will be the landscape and the surroundings. In the end we will just have to place on the scene the componets we want to and the scene will be ready for player interaction through desired controller device.



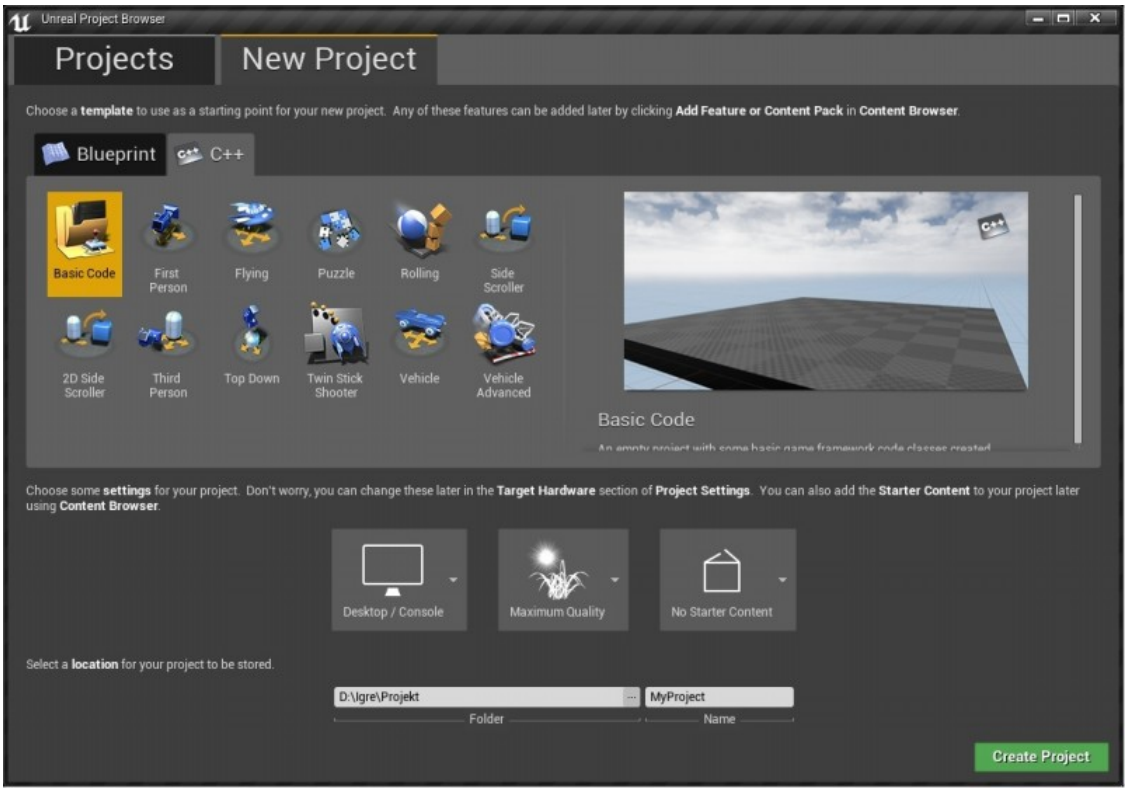
## Key words:

C++, programming, 3D scene, Unreal Engine 4, Blueprint

# Sadržaj

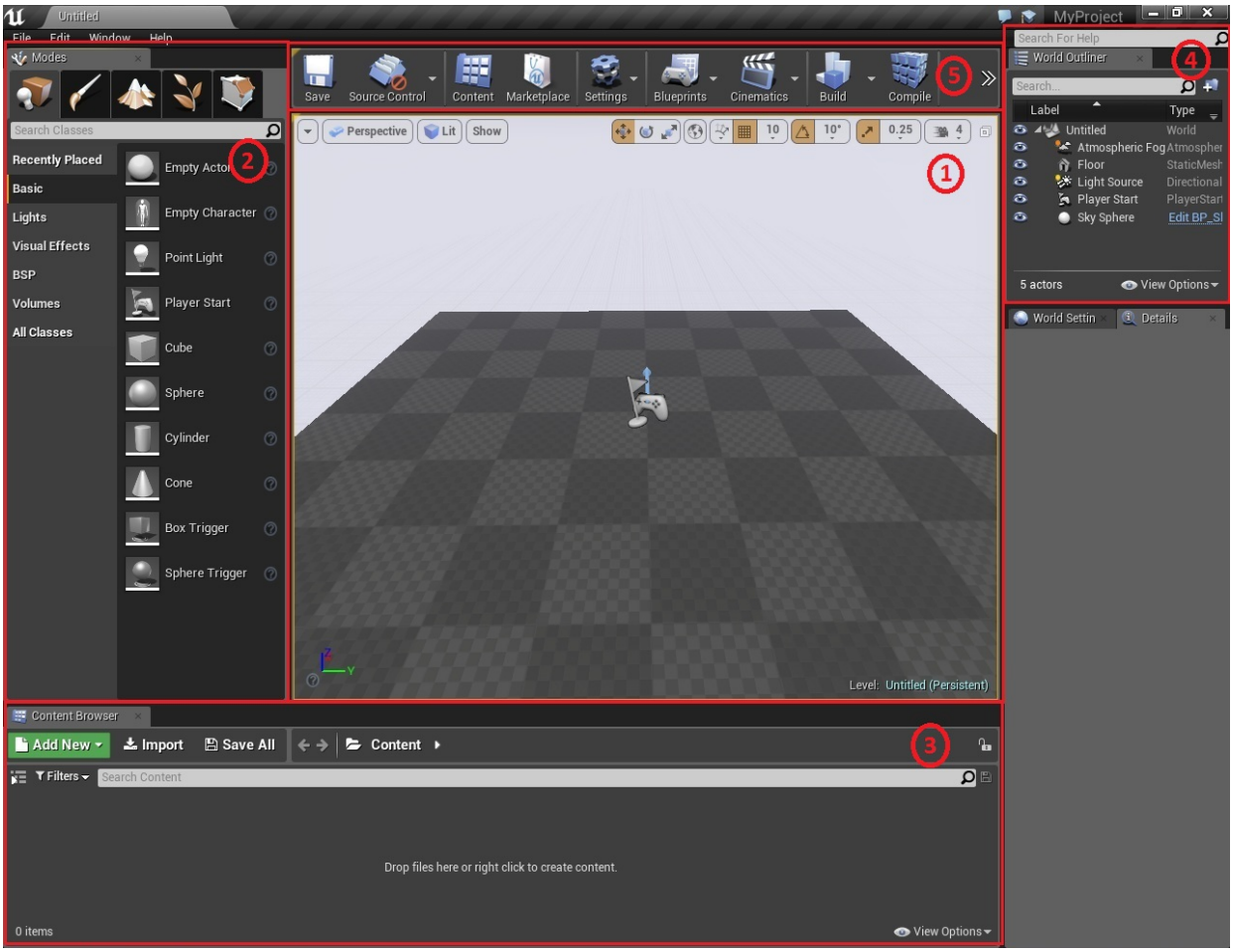
# 1 Uvod

Unreal Engine je alat za stvaranje igara i ujedno pokretač igara u realnom vremenu (engl. *game engine*) kojeg je razvila tvrtka Epic Games. On predstavlja osnovnu tehnologiju u razvoju igara i uveliko ga pojednostavljuje. Najnovija verzija Unreal Enginea, i verzija koja je korištena u ovom radu, je Unreal Engine 4 (u nastavku skraćeno UE4). UE4 nudi jedne od najmodernijih i najnaprednijih efekata i opcija koje trenutačno koriste najpoznatije svjetske kompanije [1]. Programski kod napisan za UE4 radit će kako na Windows i Mac operativnim sustavima, tako i na Android i iOS uređajima, a da bi smo taj kod znali ručno napisati, moramo poznavati programski jezik C++ [2].




Slika 1.1: Kreiranje novog projekta u UE4

Kako bi smo kreirali scenu u UE4, prvo moramo odabrati vrstu projekta. Čim otvorimo UE4 Editor, moći ćemo izabrati opciju novi projekt ili otvoriti neki postojeći ako smo prethodno radili na njemu. Pri stvaranju novog projekta ponuđena su nam dva načina, koristeći C++ i Visual Studio ili koristeći jedino UE4, radeći sa Blueprintovima (kao što je prikazano na Slici 1.1) koji predstavljaju vizualnu reprezentaciju programskog koda (o čemu će više biti rečeno u odjeljku 3). Nadalje, na Slici 1.1 vidimo da je moguće i odabrati želimo li automatski dodati početni sadržaj ili želimo praznu scenu (*Basic Code*), kakvu kvalitetu slike želimo i na kojoj platformi želimo da se naš projekt može pokrenuti. Na kraju odaberemo mjesto na kojem želimo spremiti projekt i njegovo ime, te pritiskom na dugme *Create Project* otvaramo UE4 Editor.




Slika 1.2: Izgled UE4 Editora

Na Slici 1.2 prikazan je izgled UE4 Editora koji se sastoji od nekoliko osnovnih cjelina:

1. *Viewport* - prozor kroz koji vidimo našu scenu, odnosno njeno trenutno stanje. Unutar njega se možemo kretati na isti način kao i u gotovoj igri.

2. *Modes* - odjeljak u kojem se nalaze alati za stvaranje osnovnih *Actora* (objekata koji se mogu postaviti u scenu [4]) kao što su osvjetljenje scene, početna pozicija u igri itd., te napredniji alati za instanciranje i oblikovanje reljefa.

3. *Content Browser* - osnovni preglednik preko kojeg možemo uključiti Actore iz neke druge scene u našu te im lako pristupiti kako bi ih po želji izmijenili.

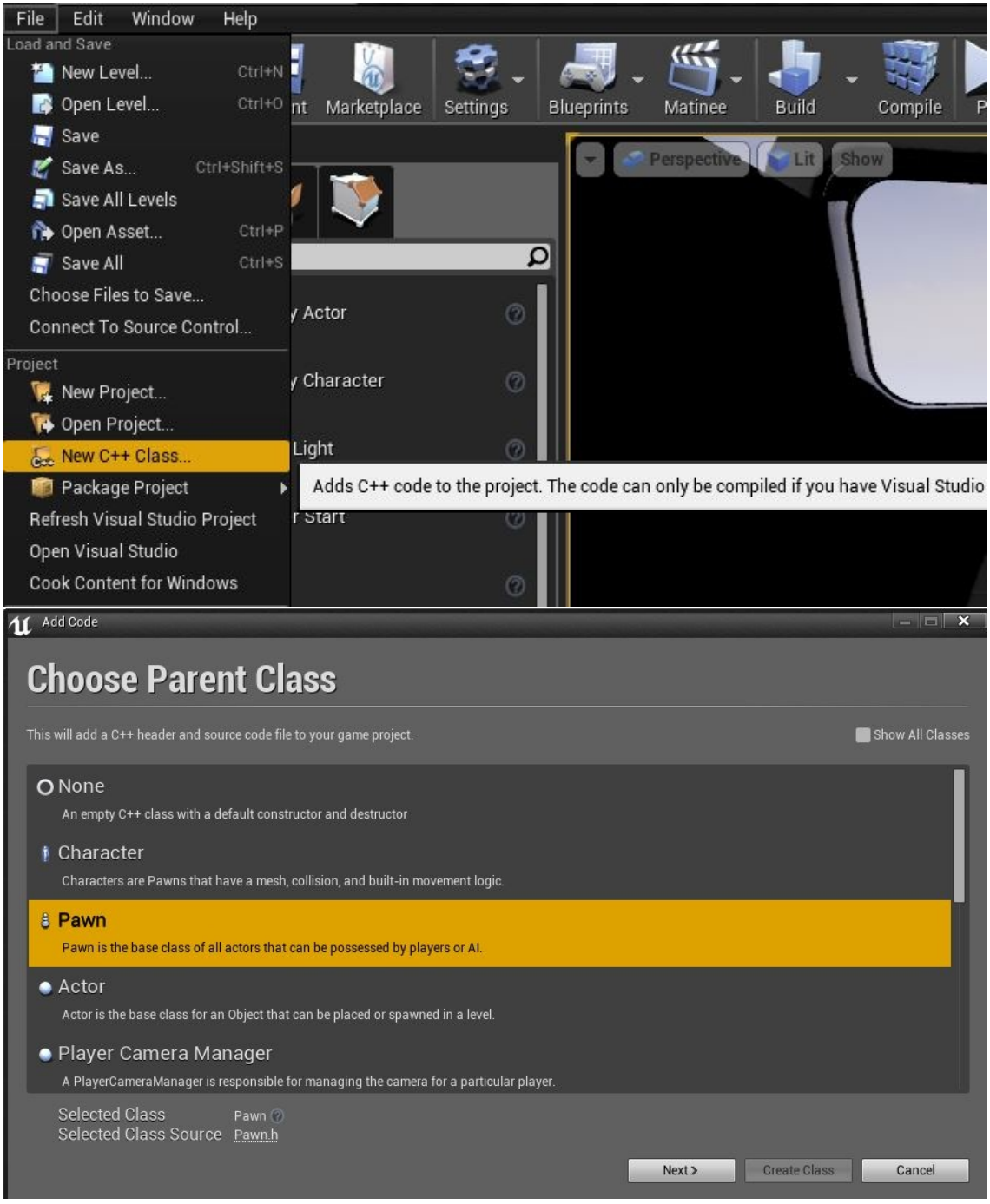
4. *World Outliner* - ima sličnu funkciju kao *Content Browser*. Primarno se koristi kako bi na najjednostavniji način mogli vidjeti sve Actore koji se nalaze u sceni. Nadalje, možemo ih sakriti iz scene ili prikazati njihove detalje.

5. *Toolbar* - na njemu se nalaze osnovni alati za pokretanje scene odnosno kompajliranje programskog koda, za uređenje Blueprintova te za formatiranje projekta u klasičnu video igru koja se onda pokreće otvaranjem *.exe* datoteke.

Programski kod korišten u ovom projektu dostupan je na GitHub platformi [5].

# 2 Stvaranje kontrolabilnog lika

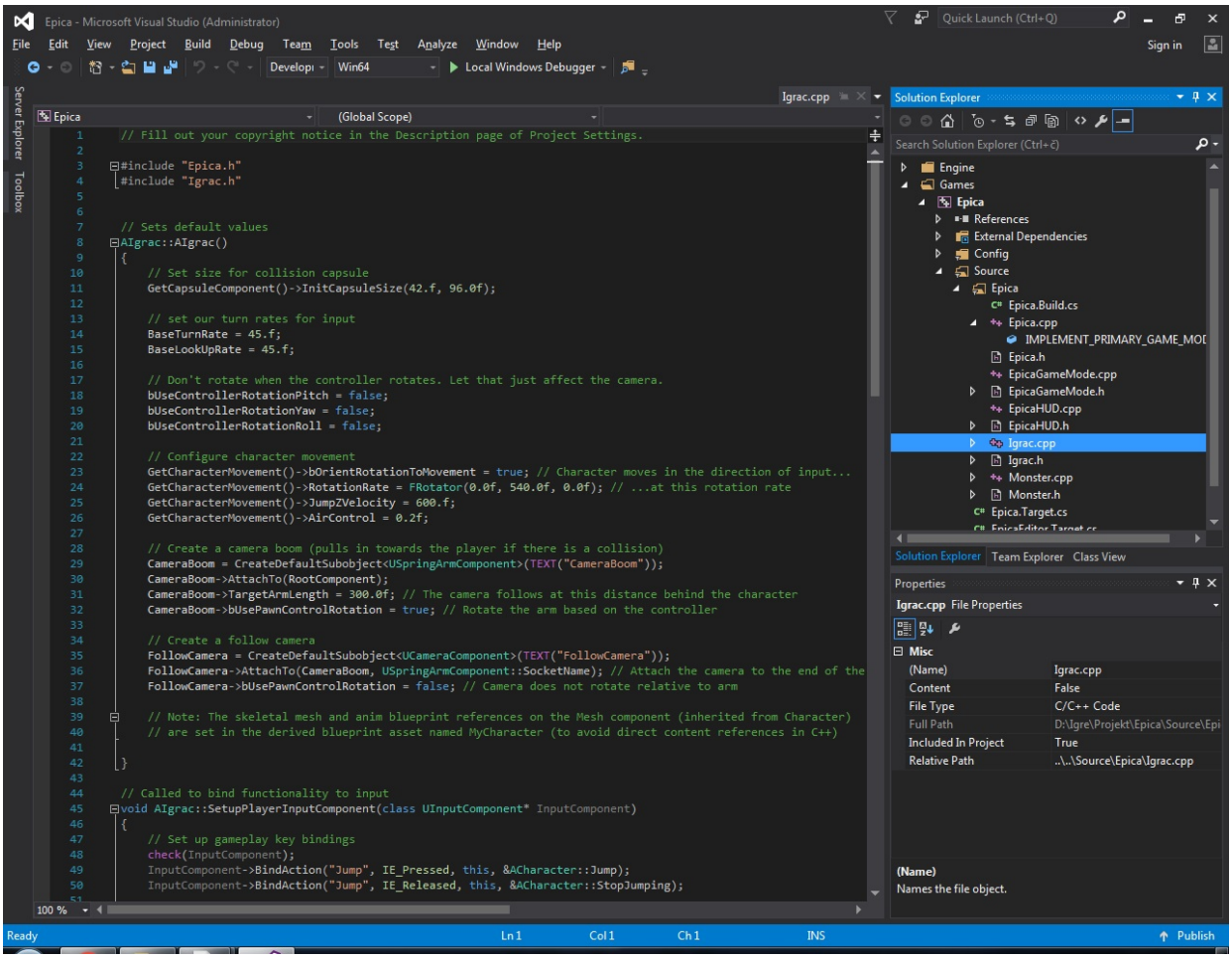
Prvi korak u uređivanju scene je stvaranje kontrolabilnog lika, odnosno pravljenje C++ klase na način koji je prikazan na Slici 2.1.




Slika 2.1: Stvaranje kontrolabilnog lika odnosno igrača

Pri kreiranju klase Igrac u UE4, za njenu roditeljsku klasu smo odabrali klasu Pawn koja predstavlja jednu od osnovnih klasa UE4 Enginea [3]. Zbog toga će nas igrač automatski dobiti opcije za implementiranje izgleda, mreže za detekciju sudara i osnovnog kretanja.

Osnovna svojstva i funkcije našeg lika implementirat ćemo direktno koristeći C++ programski jezik u programu Visual Studio.




Slika 2.2: Visual Studio 2015 sučelje

Nakon što kreiramo našu klasu Igrac, odmah možemo primijetiti dva osnovna macroa (nešto nalik funkcijama): UCLASS() i GENERATED_BODY(). Zbog njih naš C++ kod postaje dostupan za daljnju obradu u UE4, i to preko prethodno spomenutih Blueprintova.

Definicija klase Igrac i svih njenih metoda nalazi se u headeru Igrac.h, a implementacija pomenutih metoda nalazit će se u fileu Igrac.cpp. Metode u klasi Igrac koje reguliraju ponašanje kamere dolaze uz macro UPROPERTY() koji upravlja dostupnošću tih metoda, odnosno pitanjima da li i gdje se te metode mogu mijenjati. U ovom slučaju, odabrano je da su metode vidljive u UE4 Editoru, ali da se jedino mogu izmijeniti u Visual Studiu.

Da ne bismo morali pretraživati dokumentaciju UE4 kako bi našli osnovne metode za kontrolu kamere, koristimo gotov "kostur" koji možemo dobiti tako što pri kreiranju novog UE4 projekta odaberemo *Third Person* opciju. On će nam dati implementacije osnovnih metoda koje nakon toga prilagodimo potrebama svog projekta, kao i kratka objašnjenja.

```
GetCapsuleComponent()->InitCapsuleSize(42.f, 96.0f);

BaseTurnRate = 45.f;
BaseLookUpRate = 45.f;

bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationRoll = false;
```

Pozivom funkcije InitCapsuleSize() definirane su dimenzije mreže za detekciju sudara koja obavija našeg lika tj. instancu klase Igrac. Nakon toga definirane su vrijednosti varijabli koje upravljaju brzinom kretanja kamere, i tri varijable postavljene na vrijednost **false** koje sprječavaju da pomjeranje kamere utječe i na pomjeranje igrača. To znači da ćemo igrača i kameru moći pomjerati neovisno jednog o drugome što predstavlja jako bitno odličje igre iz trećeg lica.

```
CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));
CameraBoom->AttachTo(RootComponent);
CameraBoom->TargetArmLength = 300.0f;
CameraBoom->bUsePawnControlRotation = true;

FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));
FollowCamera->AttachTo(CameraBoom, USpringArmComponent::SocketName);
FollowCamera->bUsePawnControlRotation = false;
```

Sada je na redu definiranje osnovnih svojstava kamere preko CameraBoom koji je stvoren kao instanca klase USpringArmComponent. Upravo ta klasa predstavlja svojstvo približavanja kamere igraču u slučaju sudara kamere sa nekim predmetom. Ovime je definirana fiksna udaljenost na kojoj će kamera pratiti igrača, a postavljanjem varijable bUsePawnControlRotation na true, odabrana je opcija da kamera koristi igračeve kontrole za "razgledavanje". Nakon toga je instancirana sama kamera, kao instanca klase UCameraComponent, i bit će fiksirana na ranije odabranu udaljenost. Na kraju je još odabrana opcija da kontrole za rotaciju igrača ne utječu na rotaciju same kamere.

Definiranjem metode SetupPlayerInputComponent klase Igrac, omogućavamo da unosom naredbe sa tipkovnice (najčešće samo jednu tipku) pokrenemo funkciju koja će pomjerati našeg lika ili njegovu kameru. Napravljene su funkcije za pomjeranje lika naprijed, natrag, lijevo i desno; "razgledavanje" kamerom u svim smjerovima, pri tom ne pomjerajući lika; zumiranje kamere u oba smjera, i skakanje.

```
void AIgrac::SetupPlayerInputComponent(class UInputComponent* InputComponent)
{
        check(InputComponent);
        InputComponent->BindAction("Jump", IE_Pressed, this, &ACharacter::Jump
            );
        InputComponent->BindAction("Jump", IE_Released, this, &ACharacter::
            StopJumping);

        InputComponent->BindAxis("MoveForward", this, &AIgrac::MoveForward);
        InputComponent->BindAxis("MoveRight", this, &AIgrac::MoveRight);
        InputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);
        InputComponent->BindAxis("LookUp", this, &APawn::
            AddControllerPitchInput);

        InputComponent->BindAxis("ZoomIn", this, &AIgrac::ZoomIn);
        InputComponent->BindAxis("ZoomOut", this, &AIgrac::ZoomOut);
}
```

```
void AIgrac::MoveForward(float Value)
{
        if ((Controller != NULL) && (Value != 0.0f))
        {
                const FRotator Rotation = Controller->GetControlRotation();
                const FRotator YawRotation(0, Rotation.Yaw, 0);

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

void AIgrac::MoveRight(float Value)
{
        if ((Controller != NULL) && (Value != 0.0f))
        {
                const FRotator Rotation = Controller->GetControlRotation();
                const FRotator YawRotation(0, Rotation.Yaw, 0);

                const FVector Direction = FRotationMatrix(YawRotation).
                    GetUnitAxis(EAxis::Y);
                AddMovementInput(Direction, Value);
        }
}
```

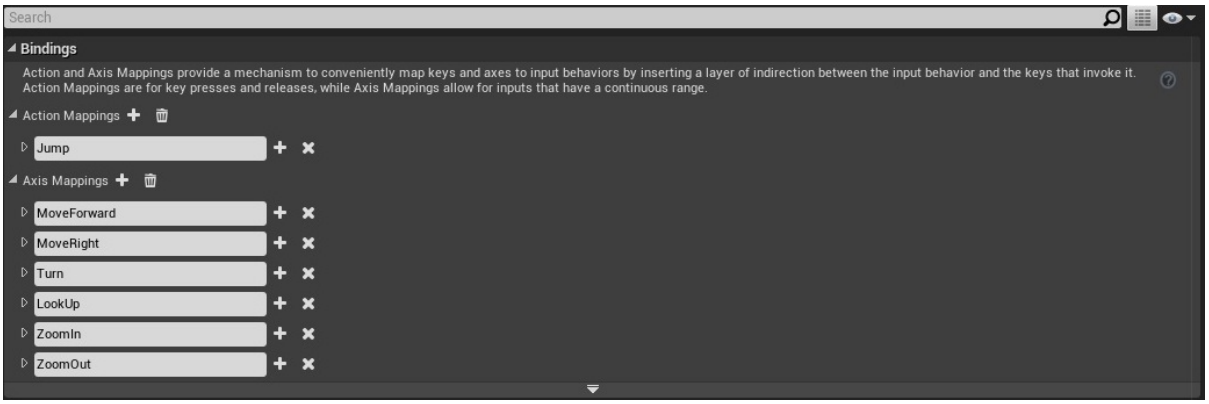
Kako bi smo mogli kretati igrača u sva četiri smjera, zapravo je dovoljno jedino napraviti funkcije za kretanje u naprijed i desno. Kretanje u ostalim smjerovima bit će analogno, samo sa suprotnim predznakom varijable. Funkcija MoveForward() regulira pomjeranje lika prema naprijed (i natrag, što će biti napravljeno kasnije u UE4 Editoru). Funkcija će se jedino izvršiti u slučaju kada damo naredbu za njeno izvršenje preko tipkovnice odnosno *Controllera* i kada je duljina pomjeranja različita od nule. Funkcija radi tako što će lika pomjeriti u smjeru naprijed za unesenu vrijednost (to predstavlja iznos vektora), ali prije toga mora preko funkcije GetControllerRotation() saznati za koliko stupnjeva je lik već bio rotiran (smjer vektora), jer zbog slobodne kamere lik neće uvijek gledati prema naprijed. Analogno je implementirana metoda MoveRight().

```
void AIgrac::ZoomIn(float Value)
{
        if ((Controller != NULL) && (Value != 0.0f)&& (CameraBoom->
            TargetArmLength > 100.0f))
        {
                CameraBoom->TargetArmLength -= Value;
        }
}

void AIgrac::ZoomOut(float Value)
{
        if ((Controller != NULL) && (Value != 0.0f) && (CameraBoom->
            TargetArmLength < 700.0f))
        {
                CameraBoom->TargetArmLength += Value;
        }
}
```

Na kraju su definirane metode ZoomIn i ZoomOut za zumiranje kamere. One jednostavno dodaju ili oduzimaju dobivenu vrijednost (od *Controllera*) od fiksne udaljenosti kamere od lika.

Bitno je primijetiti da smo za sada samo definirali funkcije za kretanje lika i kamere. Nigdje nismo rekli pritiskom na koji gumb tipkovnice ili miša će se one i pokrenuti. To moramo napraviti u UE4 Editoru preko opcije *Input* koja se nalazi u *Edit Project Settings* kako je prikazano na Slici 2.3. Ovdje možemo odabrati bilo kakav unos sa tipkovnice, povezati ga sa jednom od funkcjia za kretanje i unijeti osnovnu vrijednost za koju će se lik/kamera pomjeriti svaki put.

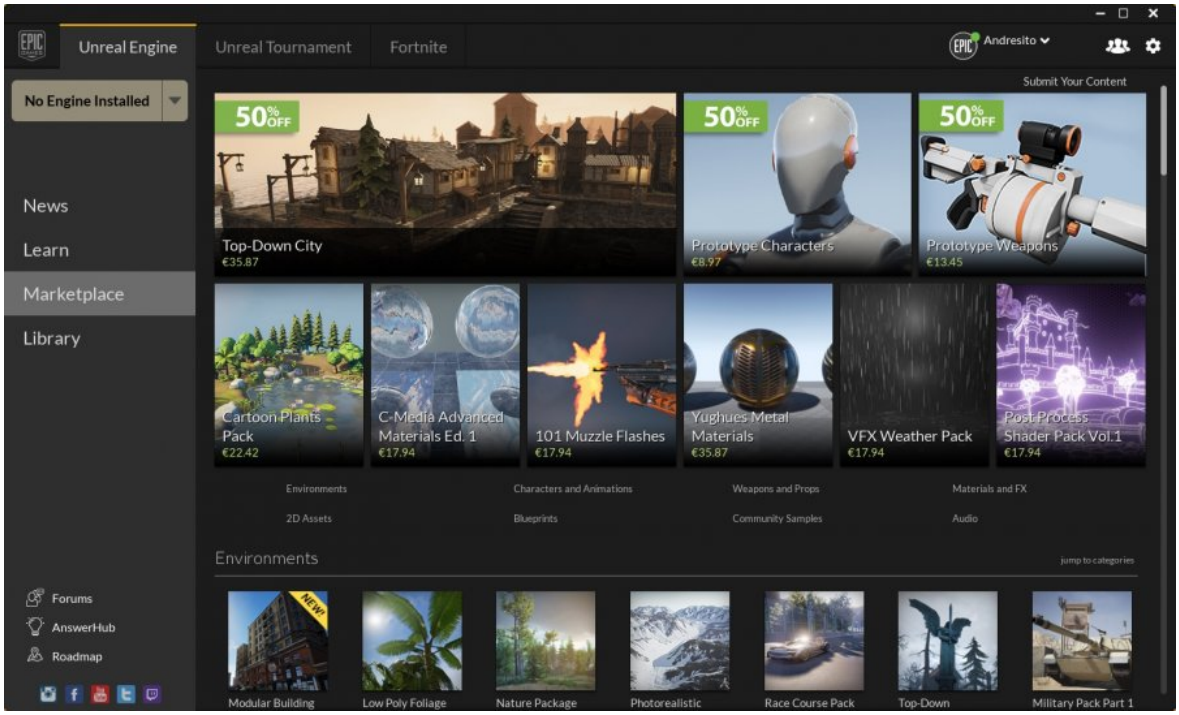



Slika 2.3: Postavljanje kontrola u UE4 sučelju

Ostala svojstva našeg lika kao što su zvučni efekti, njegov izgled i animacije, zbog jednostavnosti su napravljena u UE4 Blueprintu i bit će pojašnjena u sljedećem poglavlju.

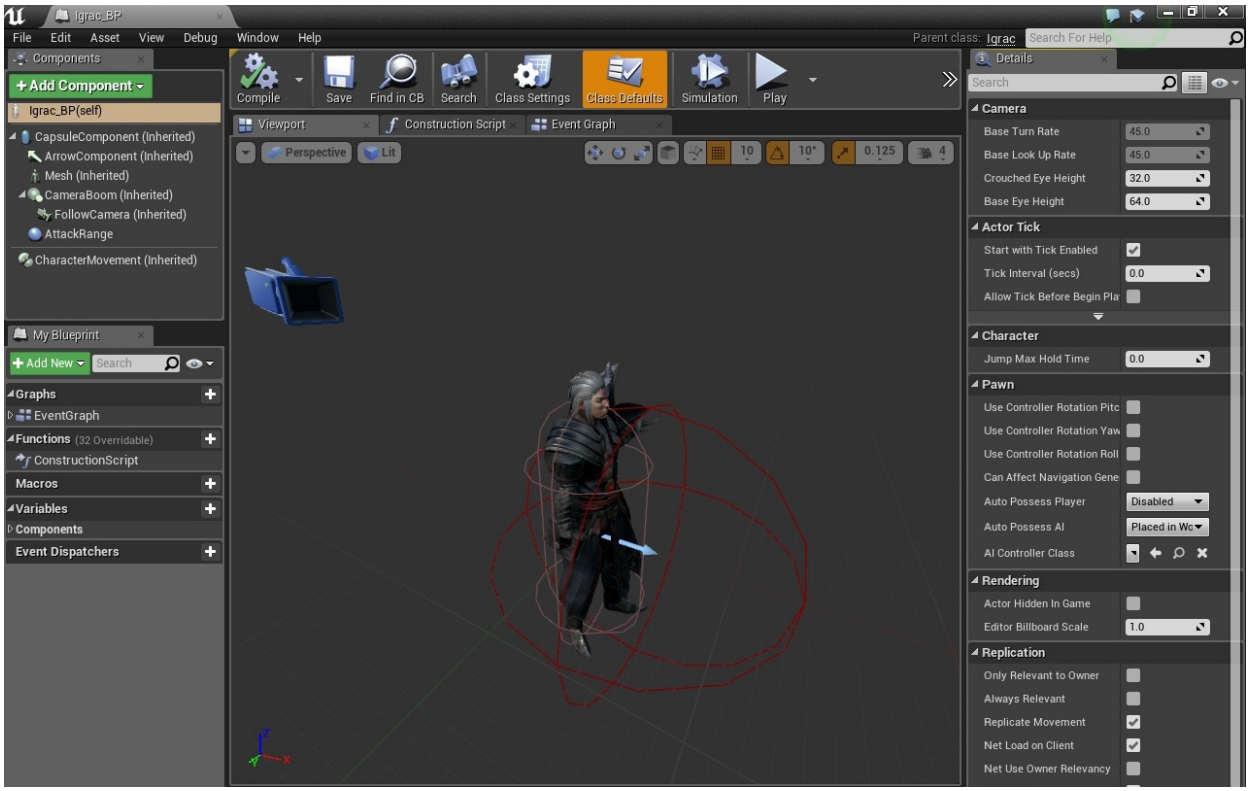
# 3 Blueprint kontrolabilnog lika

Završili smo sa pravljenjem "kostura" za naš kontrolabilni lik (tako što smo definirali osnovna svojstva klase u Visual Studiu), te je sad na redu da se definira izgled lika i njegovo mjesto u konačnoj sceni. Kako ne bismo morali sami crtati svog lika, možemo posjetiti UE4 Marketplace i pronaći neke gotove koji su besplatni.




Slika 3.1: UE4 Marketplace

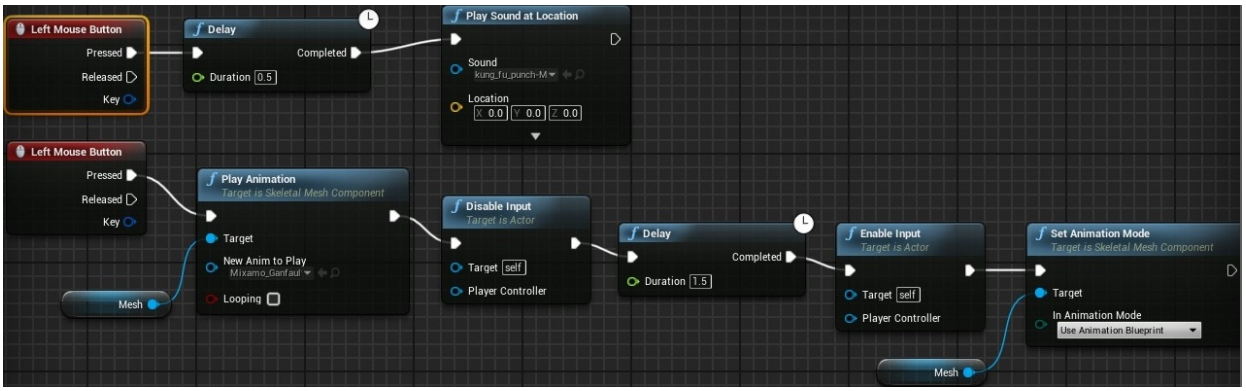
Nakon što smo pronašli odgovarajući izgled, vrijeme je da ga preko blueprinta povežemo sa našom klasom Igrac i nastavimo sa podešavanjem daljnjih opcija. Kao što je navedeno ranije, blueprint klase predstavlja njenu vizualnu reprezentaciju što uveliko pojednostavljuje njeno uređenje. Naravno, sve to se može odraditi direktno u Visual Studiu koristeći C++ programski jezik ali na ovaj način naš projekt postaje dostupniji dizajnerima koji uopće ne moraju znati programirati.




Slika 3.2: Blueprint klase Igrac

Na Slici 3.2 prikazan je izgled blueprinta klase Igrac. Na lijevoj strani vidimo već definirana svojstva ove klase kao što su mreža za detekciju sudara i prateća kamera. Na srednjem dijelu slike nalazi se trodimenzionalna vizualna reprezentacija cijele klase i svih njenih elemenata. Dimenzije mreže se ovdje mogu jednostavno podešavati korištenjem miša, razvlačenjem do željene veličine. Isto vrijedi i za kameru koja se na sličan način može postaviti na željenu poziciju. Na desnoj strani možemo primijetiti da su neke opcije zatamnjene te im se vrijednosti ne mogu promijeniti. To je upravo zbog opcije koju smo koristili u samom C++ "kosturu" naše klase kada smo rekli da će neke opcije biti samo vidljive u blueprintu, ali da će se moći podešavati jedino u izvornom kodu.

Ukoliko našem liku želimo dodati još neke dodatne kontrole koje nisu ugrađene u "kostur" klase, moramo proširiti mogućnosti klase i to preko opcije *Event Graph* koja predstavlja jednu od mnogih mogućnosti blueprinta.




Slika 3.3: Event Graph klase Igrac

Kao što vidimo na Slici 3.3, *Event Graph* predstavlja grafički prikaz pozivanja funkcija koje upravljaju našim likom u željenim situacijama [4]. Vidljiva su dva grafa odvojena zbog jednostavnosti, inače su se mogla spojiti u jedan. Gornji regulira reprodukciju zvuka pri pritisku lijeve tipke miša, dok je donji kompliciraniji i regulira reprodukciju animacije (pokreta) pri pritisku lijeve tipke miša. Pri davanju kontrole za napad, želimo da naš lik uradi animaciju udarca i da se čuje odgovarajući zvučni efekt.

Dakle, kao što vidimo na gornjem grafu, kada se dogodi situacija da igrač pritisne lijevu tipku miša (Left Mouse Button čvor na grafu), nakon kratke pauze od pola sekunde (Delay čvor na grafu) pozvat će se funkcija za reprodukciju zvuka (Play Sound at Location čvor na grafu). Ona prima dvije varijable. Varijabla Sound predstavlja lokaciju zvučnog efekta kojeg želimo reproducirati a varijabla Location predstavlja točnu lokaciju odnosno udaljenost od lika na kojoj želimo reproducirati zvuk.

Donji graf regulira animaciju udarca. Kada igrač pritisne lijevu tipku miša, funkcija Play Animation će pokrenuti izvršavanje animacije, a varijabla nad kojom će se funkcija izvršiti je Mesh odnosno skeletalna mreža našeg kontrolabilnog lika. Kako ne bi došlo do situacije u kojoj igrač stalno pritišće lijevu tipku miša te se sljedeća animacija aktivira za vrijeme izvršavanja prethodne, uvedene su funkcije Enable Input i Disable Input. Dakle, kada započne animacija udarca, funkcijom Disable Input na varijabli Controller ćemo onemogućiti unos nove kontrole u trajanju od 1.5 sekundi koliko traje animacija udarca (regulirano pozivom funkcije Delay), nakon čega pozivamo funkciju Enable Input na istoj varijabli kako bi se vratilo početno stanje.

Ovime smo završili sa podešavanjem svojstava našeg kontrolabilnog lika. Ukoliko želimo dodati još novih funkcija, to možemo uraditi dodavanjem novih metoda u klasu Igrac u Visual Studiu, ili dodavanjem novih grafova ponašanja i drugih opcija u blueprintu.

# 4 Stvaranje protivnika

Na isti način kao što smo napravili klasu Igrac, sada ćemo napraviti klasu Monster koja će također naslijediti osnovnu klasu Character. Njena uloga bit će postavljanje "kostura" za stvaranje neprijateljski nastrojenih dinamičkih objekata u sceni koji će imati određene interakcije sa našim kontrolabilnim likom.

```
UCLASS()
class EPICA_API AMonster : public ACharacter
{
        GENERATED_UCLASS_BODY()

        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
            MonsterProperties)
        float Speed;
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
            MonsterProperties)
        float HitPoints;
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
            MonsterProperties)
        float BaseAttackDamage;
        UPROPERTY(EditAnywhere, BlueprintReadWrite, Category =
            MonsterProperties)
        float AttackTimeout;
        UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category =
            MonsterProperties)
        float TimeSinceLastStrike;
        UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Collision
            )
        USphereComponent* SightSphere;
        UPROPERTY(VisibleDefaultsOnly, BlueprintReadOnly, Category = Collision
            )
        USphereComponent* AttackRangeSphere;
        AMonster();

        virtual void BeginPlay() override;

        virtual void Tick( float DeltaSeconds ) override;

        virtual void SetupPlayerInputComponent(class UInputComponent*
            InputComponent) override;
};
```

Ponovno, macroima UCLASS() i GENERATED_UCLASS_BODY() omogućavamo UE4 Editoru da učita ovu klasu zbog daljnje obrade, dok macro UPROPERTY() regulira mogućnosti podešavanja svakog elementa klase.

Klasa Monster može sadžavati razne varijable i metode koje definiraju svojstva i ponašanje njenih instanci u našoj sceni. Varijabla Speed predstavlja brzinu kretanja neprijatelja, varijabla HitPoints predstavlja količinu štete koju on može primiti prije nego što ga uništimo itd. Posljednje dvije varijable su instance klase USphereComponent i one reguliraju detekcijske sfere koje obavijaju instance klase Monster. Funkcije Tick i SetupPlayerInputComponent su dvije virtualne funkcije. Funkcija Tick ima svojstvo da se poziva svakog *framea* što inače iznosi 0.16 sekundi, a druga funkcija će ostati neimplementirana jer ne želimo da igrač kontrolerom bude u mogućnosti kontrolirati neprijatelje.

```
AMonster::AMonster(const class FObjectInitializer& PCIP):Super(PCIP)
{
        PrimaryActorTick.bCanEverTick = true;

        Speed = 20;
        HitPoints = 20;
        BaseAttackDamage = 1;
        AttackTimeout = 1.5f;
        TimeSinceLastStrike = 0;

        SightSphere = PCIP.CreateDefaultSubobject<USphereComponent>(this,TEXT(
            "SightSphere"));

        SightSphere->AttachTo(RootComponent);

        AttackRangeSphere = PCIP.CreateDefaultSubobject <USphereComponent>(
            this,TEXT("AttackRangeSphere"));

        AttackRangeSphere->AttachTo(RootComponent);
}
```

Na red dolazi implementacija konstruktora klase Monster. Postavljanjem varijable bCanEverTick na **true** smo zapravo rekli da ova klasa može koristiti funkciju Tick što će nam trebati u nastavku. Zbog definiranja osnovnih vrijednosti varijabli klase Monster u njenom konstruktoru, po defaultu će svaka njena instanca (igračev protivnik) imati brzinu kretanja i životne bodove jednake 20, osnovnu jačinu napada 1 i pauzu od najmanje 1.5 sekundi između svaka dva napada. Početna vrijednost proteklog vremena od posljednjeg napada je 0 jer se prvi napad još nije ni dogodio. Na kraju u konstruktoru još stvaramo i dvije detekcijske sfere (varijable SightSphere i AttackRangeSphere) koje će biti "zakačene" za protivnika.

```
void AMonster::Tick( float DeltaSeconds )
{
        Super::Tick( DeltaSeconds );

        AIgrac *avatar = Cast<AIgrac>(UGameplayStatics::GetPlayerPawn(GetWorld
            (), 0));

        if (!avatar) return;

        FVector toPlayer = avatar->GetActorLocation() - GetActorLocation();
        float distanceToPlayer = toPlayer.Size();

        if (distanceToPlayer > SightSphere->GetScaledSphereRadius())
        {
                return;
        }
        toPlayer /= distanceToPlayer;

        AddMovementInput(toPlayer, Speed*DeltaSeconds);

        FRotator toPlayerRotation = toPlayer.Rotation();
        toPlayerRotation.Pitch = 0;
        RootComponent->SetWorldRotation(toPlayerRotation);
}
```

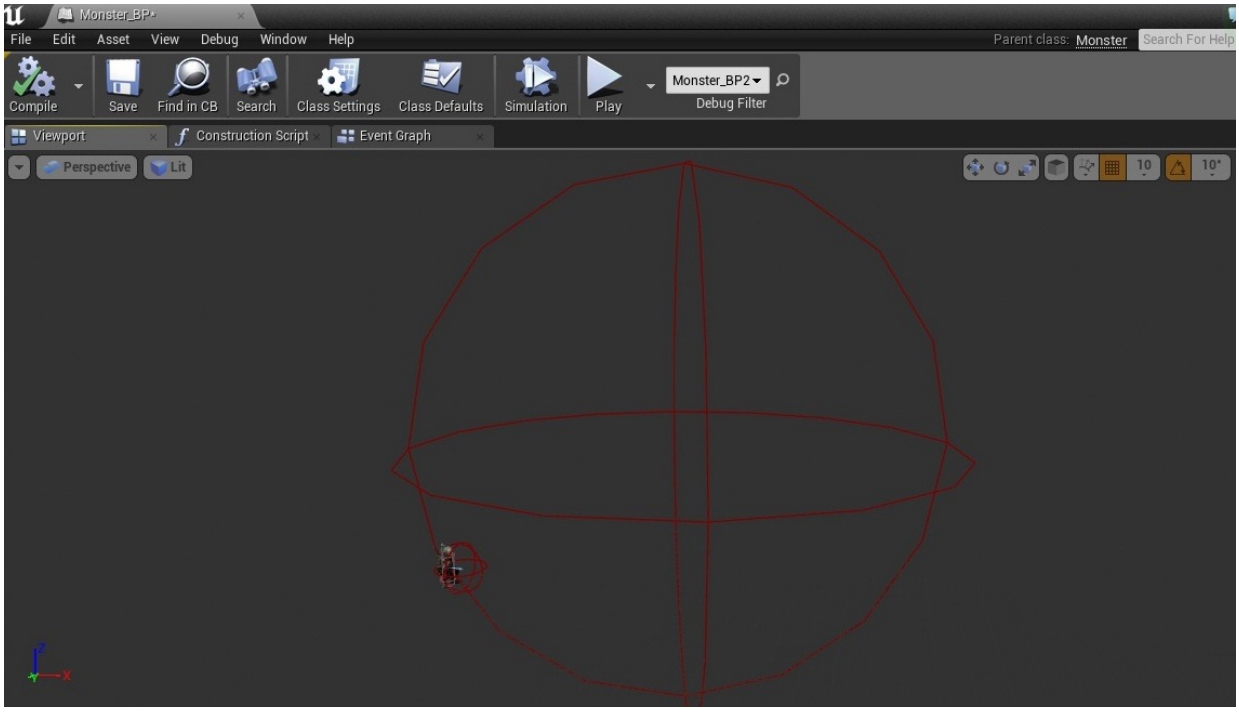
Još preostaje implementirati funkciju Tick. Ovu funkciju koristimo kako bi smo isprogramirali osnovno ponašanje protivnika prema igraču, a to je da čim igrač uđe u područje SightSphere tj. vidno polje neprijatelja, on će se odmah početi kretati prema njemu sve dok ga ne dodirne. Za sada nećemo definirati ništa poslije toga, već će to biti urađeno u blueprintu klase Monster u UE4 Editoru.

Dakle, funkcija Tick prima vrijeme koje predstavlja pauzu između njena svaka dva pokretanja, kako bi kretanje protivnika bilo fluidno i prirodno a ne isprekidano. Prvo se stvara pokazivač na igrača koji će dobiti vrijednost funckije GetPlayerPawn() i on će pokazivati na trenutnu lokaciju igrača u sceni. Ako igrač ne postoji, funkcija se zaustavlja (i neće raditi ništa dok igrač ne bude stvoren), u suprotnom nastavlja dalje. Stvara se vektor čiji će početak biti na poziciji igrača, a kraj na poziciji neprijatelja, a nakon toga i varijabla koja predstavlja duljinu ovog vektora. Sada na red dolazi računanje je li igrač unutar SightSphere neprijatelja tj. da li ga ovaj vidi, odnosno provjerava se je li udaljenost igrača do neprijatelja veća od radijusa sfere vidnog polja. U slučaju da jeste, trenutni poziv funkcije završava. Tek sada se protivnik pomjera u smjeru vektora udaljenosti zadanom brzinom, i nakon toga se funkcijom SetWorldRotation() rotira prema igraču. Varijabla Pitch je po defaultu postavljena na nulu ali se može mijenjati u slučaju da trebamo korigirati početnu rotaciju neprijatelja.

Na kraju, osnovni sistem napada protivnika bit će odrađen u UE4 blueprintu. U našoj sceni on je jako jednostavan jer se svodi na to da čim funkcija za približavanje prema igraču (unutar funkcije Tick) završi sa radom, odnosno čim se protivnik i igrač dodirnu, igrač će biti uništen i ubrzo nakon toga vraćen na početnu poziciju u sceni.

# 5 Blueprint protivnika

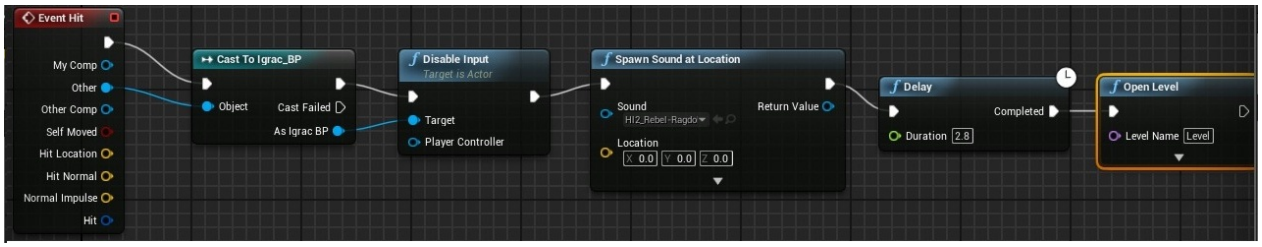
Završili smo sa definiranjem osnovnih svojstava klase Monster u Visual Studiu. Za sve ostalo koristit ćemo blueprint i *Event Graph* klase Monster unutar UE4 Editora.




Slika 5.1: Vizualna reprezentacija detekcijskih sfera klase Monster

Kao što je vidljivo na Slici 5.1, velika sfera predstavlja trodimenzionalnu reprezentaciju varijable SightSphere klase Monster, odnosno vidno polje protivnika. Mala sfera predstavlja varijablu AttackRangeSphere. Ona je napravljena tako da ugrubo aproksimira oblik trodimenzionalnog protivnika i služi za detekciju dodira između njega i kontrolabilnog lika.

Koristeći *Event Graph* (Slika 5.2) isprogramirat ćemo "neprijateljsko ponašanje" instanci klase Monster prema instanci klase Igrac odnosno prema kontrolabilnom liku. Želimo simulirati uništenje igrača pri samom dodiru sa neprijateljem. Prvi dio je napravljen u programskom "kosturu" klase. Kada se igrač nađe unutar vidnog polja protivnika, on će ga početi pratiti sve dok ga ne stigne. Njihov dodir će aktivirati situaciju Event Hit u *Event Graphu* odnosno poziv funkcije Disable Input zbog koje će igrač izgubiti kontrolu nad svojim likom. Ovo stanje će potrajati 2.8 sekundi (regulirano pozivom funkcije Delay) za vrijeme kojega će se reproducirati željeni zvučni efekti koji će dati do znanja igraču da je njegov lik uništen. Nakon što je funkcija Delay završila sa radom, poziva se funkcija Open Level koja će restartovati igru odnosno vratiti lika na početnu poziciju u nivou, a igraču vratiti kontrolu nad likom kako bi mogao nastaviti igru.

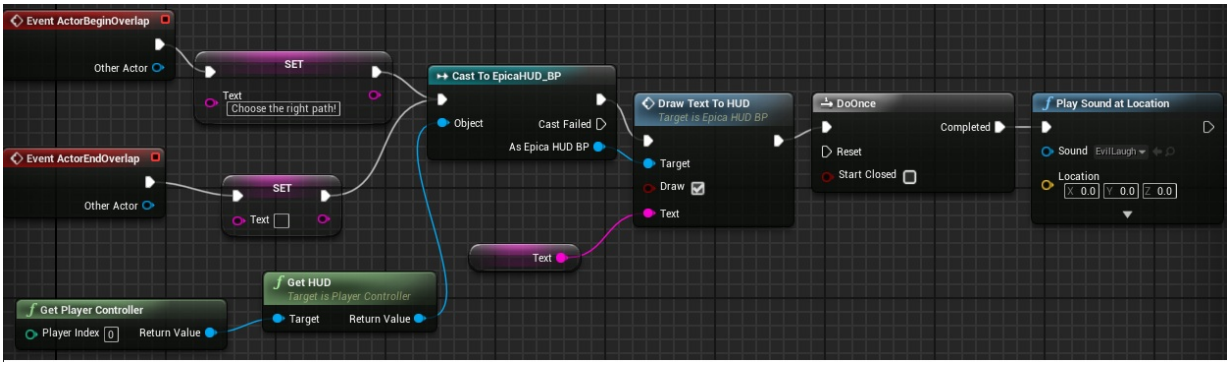



Slika 5.2: *Event Graph* klase Monster

# 6 Stvaranje NPC-ova

NPC (skraćeno od engl. *non-player character*) najčešće predstavlja prijateljski nastrojenog lika u igri čija je funkcija da daje zadatke igraču, pojasni određene mehanizme igre itd. U našoj sceni će se nalaziti dva takva lika. Ponovno ćemo otići na UE4 Marketplace i odabrati besplatne 3D modele po želji, ali prije toga za njih moramo napraviti programski "kostur" u Visual Studiu. Zbog jednostavnosti ćemo koristiti istu klasu kao za našeg kontrolabilnog lika, s tim što ćemo izbaciti sve funkcije zadužene za kontroliranje lika i kamere.

Funkcija koju će imati naši NPC-ovi će biti da nakon što im pridemo ispišu pozdravnu poruku na ekran i reproduciraju određeni zvučni efekt. To će biti urađeno u *Event Graphu* klase NPC kao što je prikazano na Slici 6.1.




Slika 6.1: *Event Graph* klase NPC

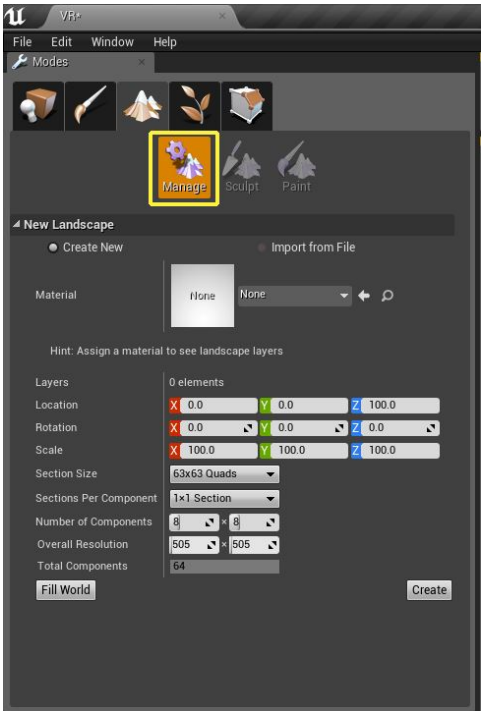
Dakle, kada se nađemo u vidnom polju NPC-a (unutar detekcijske sfere istog tipa kao kod klase Monster), ispunjeni su uvjeti aktiviranja ActorBeginOverlap. Postavimo varijablu Text na "Welcome!" što predstavlja pozdravnu poruku koju će NPC ispisati na ekran. Ekran u ovom slučaju predstavlja HUD (skraćeno od engl. *head-up display*), metoda kojom se informacije vizualno prikazuju igraču kao dio korisničkog interfejsa [4]. Najjednostavniji način implementacije ispisivanja poruke na ekran bio bi pozivom funkcije PrintString [3], ali njome ne bismo mogli podešavati veličinu, font niti položaj teksta. Umjesto nje, koristimo funkciju Draw Text To HUD. Problem na koji ovdje nalazimo je taj što je HUD zasebna klasa, odvojena od naše klase NPC unutar koje se sve ovo implementira. Zbog toga nam treba funkcija Cast To koja će nam omogućiti pristup funkciji Draw Text to HUD. Kako tekst ne bi ostao zauvijek na ekranu, u *Event Graphu* treba stvoriti i situaciju ActorEndOverlap, odnosno situaciju kada igrač napusti vidno polje NPC-a. U tom slučaju varijablu Text postavljamo na prazan string i na isti način šaljemo funkciji Draw Text To HUD.

Osim teksta želimo reproducirati i određeni zvučni efekt što radimo pozivom funkcije Spawn Sound at Location čiji je način rada objašnjen u trećem poglavlju. Na kraju još samo moramo reprodukciju zvuka staviti pod kontrolu funkcije DoOnce koja će spriječiti konstantnu reprodukciju zvuka u slučaju da igrač ponavlja ulazak u vidno polje NPC-a.

# 7 Konačno stvaranje scene

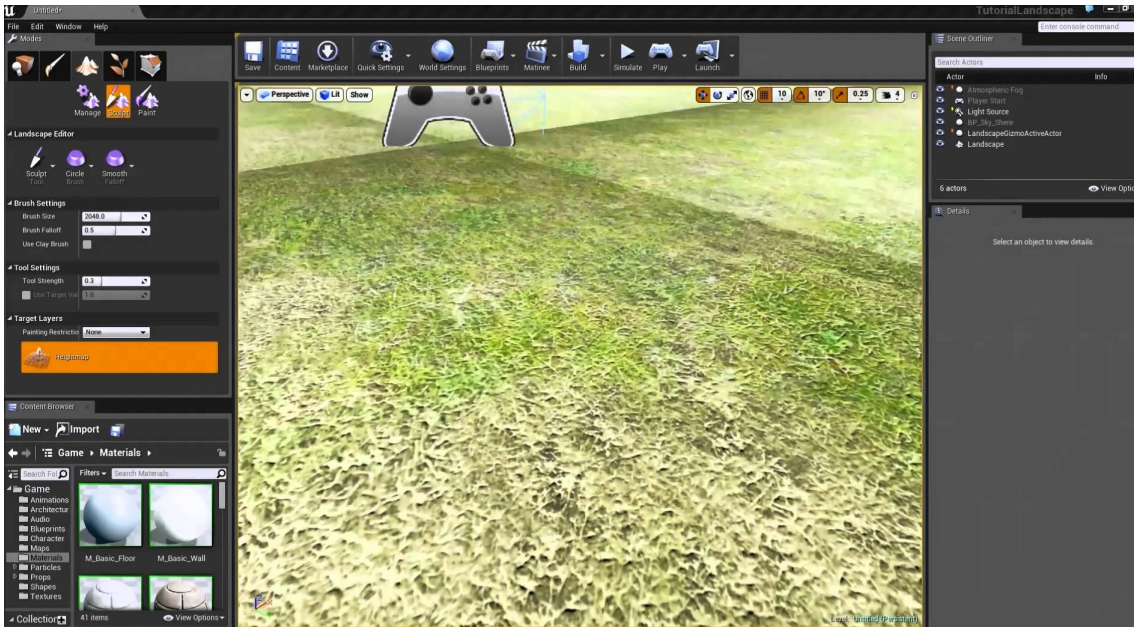
Kao posljednji korak dolazi stvaranje same scene koje se u potpunosti vrši u UE4. Do sada smo napravili sve dinamičke komponente naše scene (kontrolabilni lik, neprijatelji i NPC-ovi). Sada je na redu stvaranje statičkih komponenti (reljef, mrtva priroda), njihovo pozicioniranje te postavljanje dinamičkih komponenti na željena mjesta u sceni.

Najprije ćemo napraviti osnovni reljef scene odnosno običnu ravnu podlogu određenih dimenzija i teksture. Za to koristimo opciju *Landscape* (Slika 7.1).



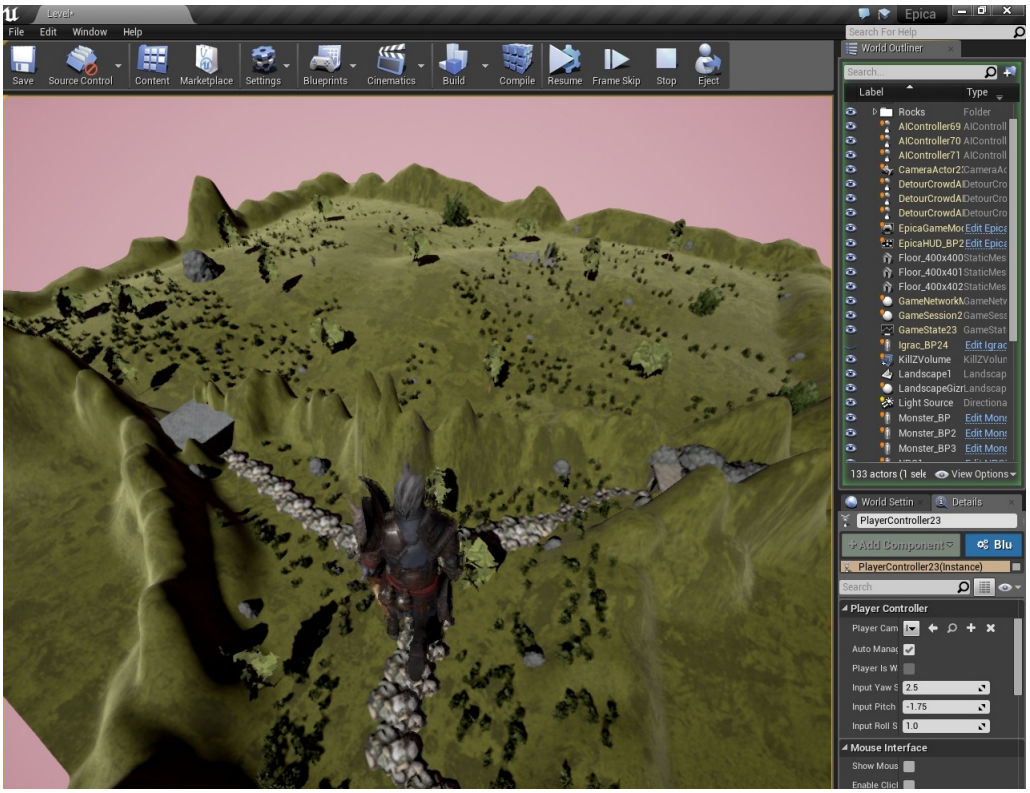

Slika 7.1: Opcija *Landscape* UE4 Editora

Kao rezultat, u glavnom editoru će postati vidljivo trenutačno stanje scene. Prazna je i njen jedini element je zelena podloga kao što je i vidljivo na Slici 7.2. Ako nismo zadovoljni osnovnim osvjetljenjem scene, možemo ga prilagoditi korištenjem opcija komponente *Light Source* te na taj način promijeniti boju, jakost, postojanje sjenki i sličnog. Možemo promijeniti i lokaciju na kojoj se igrač stvara na početku igre, odnosno pri stvaranju nivoa, modifikacijom komponente *PlayerStart*.



Slika 7.2: Trenutačno stanje scene posmatrano kroz glavni editor

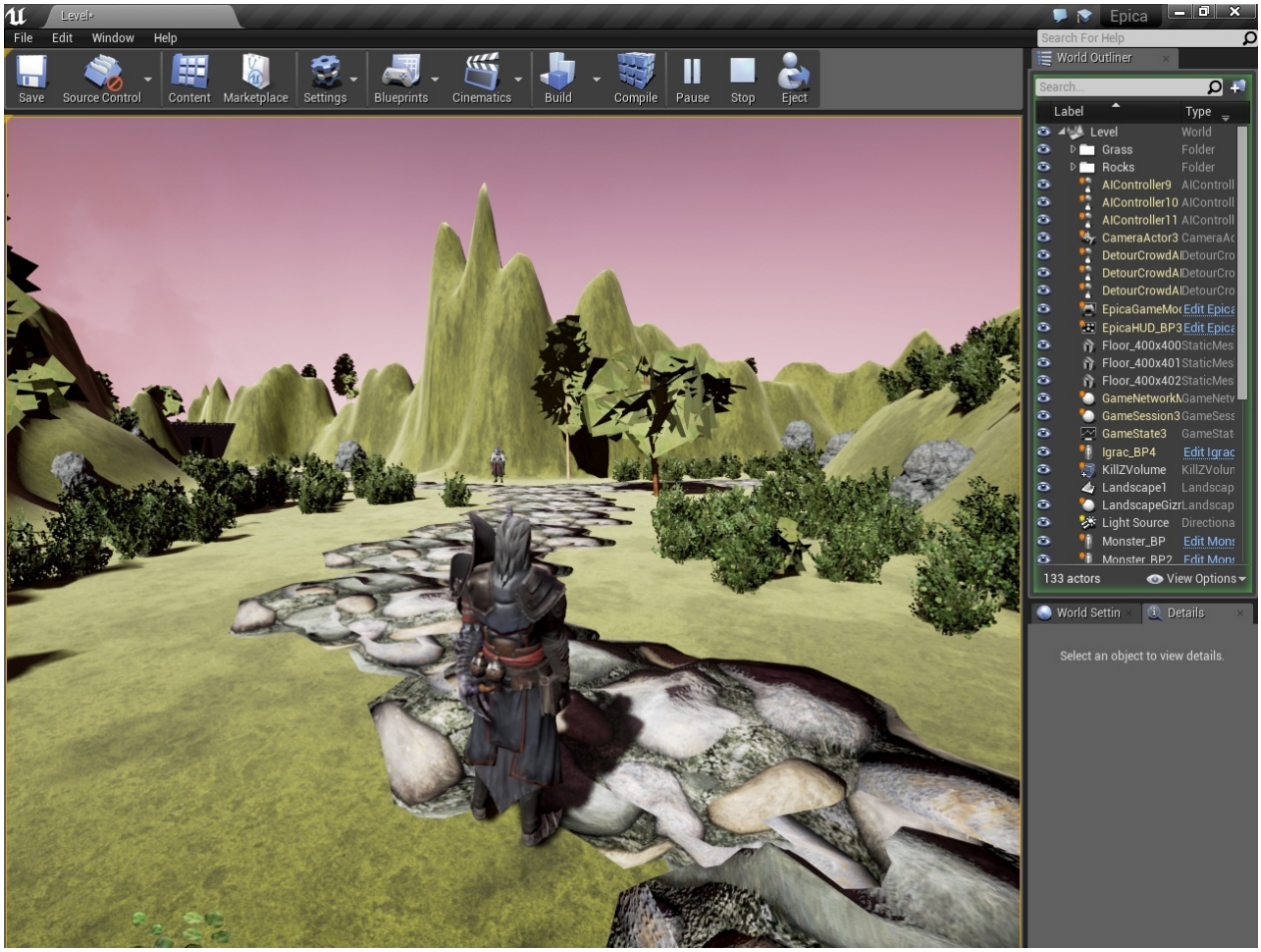
Koristeći opciju *Sculpt* sada po želji možemo oblikovati tu podlogu. Najprije ćemo napraviti planine, dodati drveće, kamenje i ostale 3D objekte čije strukture već postoje u UE4 biblioteci. Na nama je samo da odaberemo koje želimo, mišem ih prevučemo na željena mjesta na sceni i time završimo sa postavljanjem statičkih komponenti naše scene. Na Slici 7.3 možemo vidjeti izgled čitave scene odnosno reljefa iz ptičije perspektive.



Slika 7.3: Kompletna scena

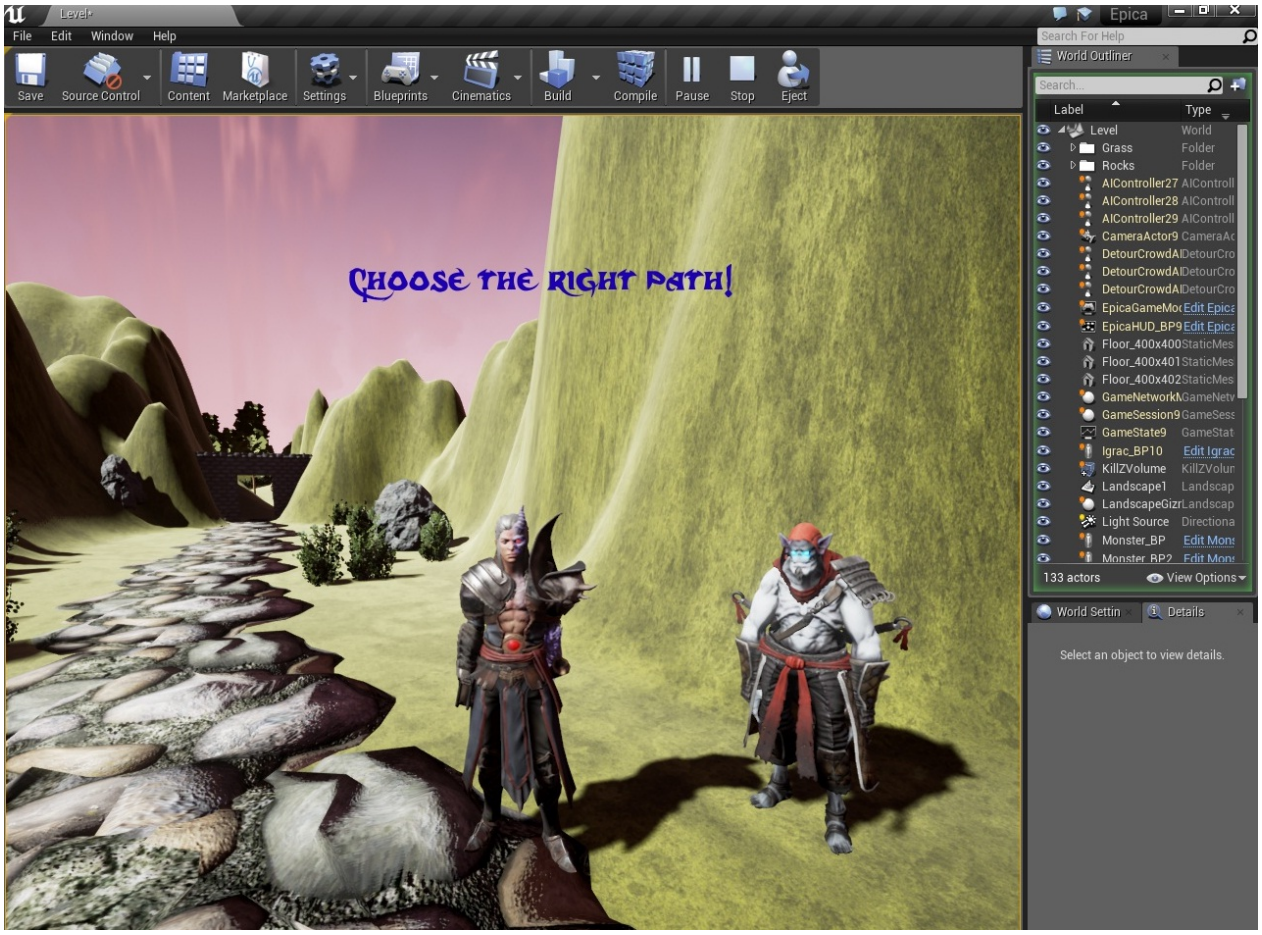
Još je preostalo postaviti dva NPC-a i nekoliko neprijatelja na željene lokacije i otići na opciju *Simulate* kako bi započela simulacija scene. Unutar simulacije bit ćemo stavljeni u poziciju igrača, za razliku od pozicije developera u kojoj smo bili do sada, te ćemo moći isprobati scenu iz prve ruke kao da se radi o završenom produktu.

Dakle, pri pokretanju simulacije scene, naći ćemo se na početnoj poziciji kao što je vidljivo na Slici 7.4.



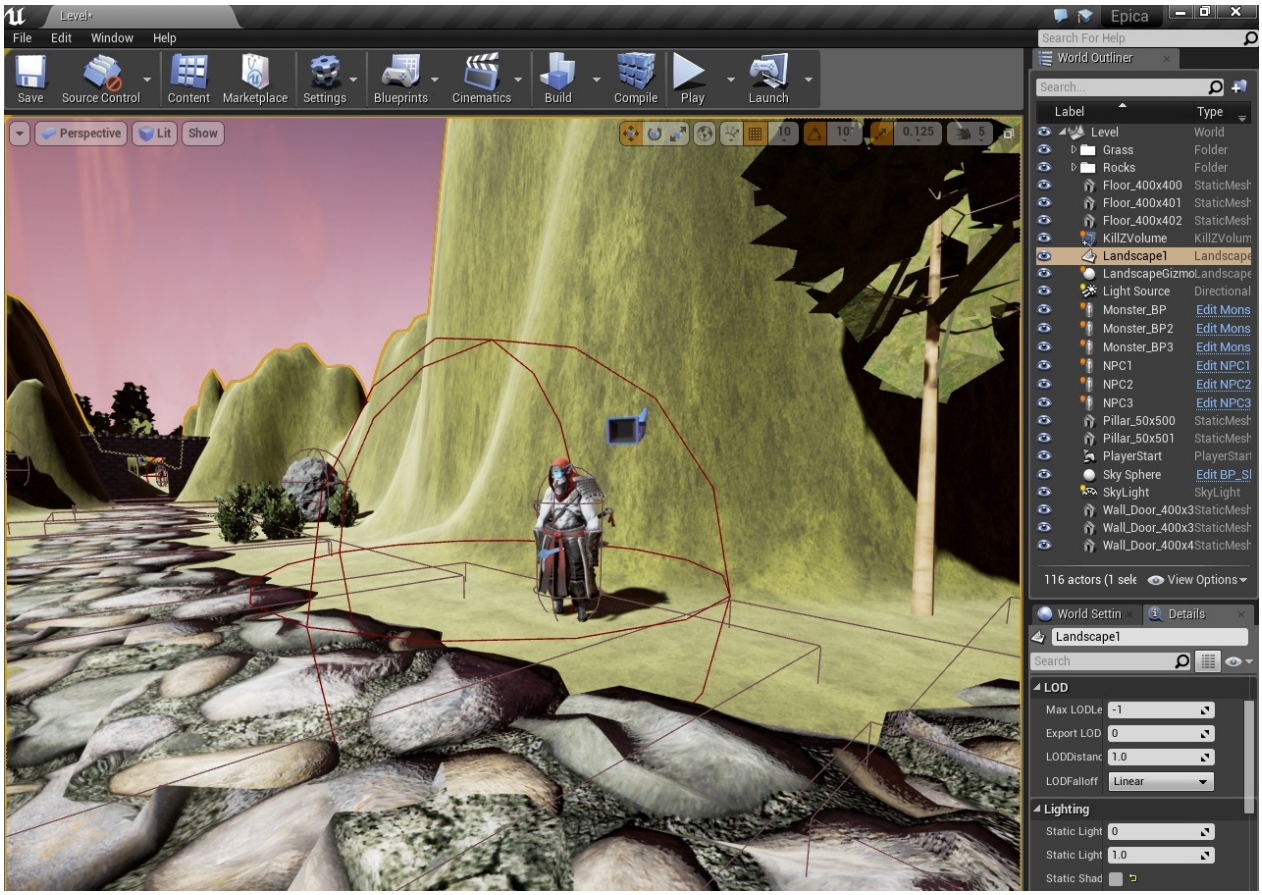

Slika 7.4: Stvaranje lika na početnoj poziciji

Ispred nas se nalazi staza koja vodi do prvog NPC-a i onda se dijeli na dvije strane, lijevu i desnu (Slika 7.5). Kada se dovoljno približimo NPC-u ,ispisat će se poruka na ekran i reproducirati zvučni efekt odabran u *Event Graphu*. Ovaj NPC nam savjetuje da odaberemo pravi put, jedan od njih može biti koban.



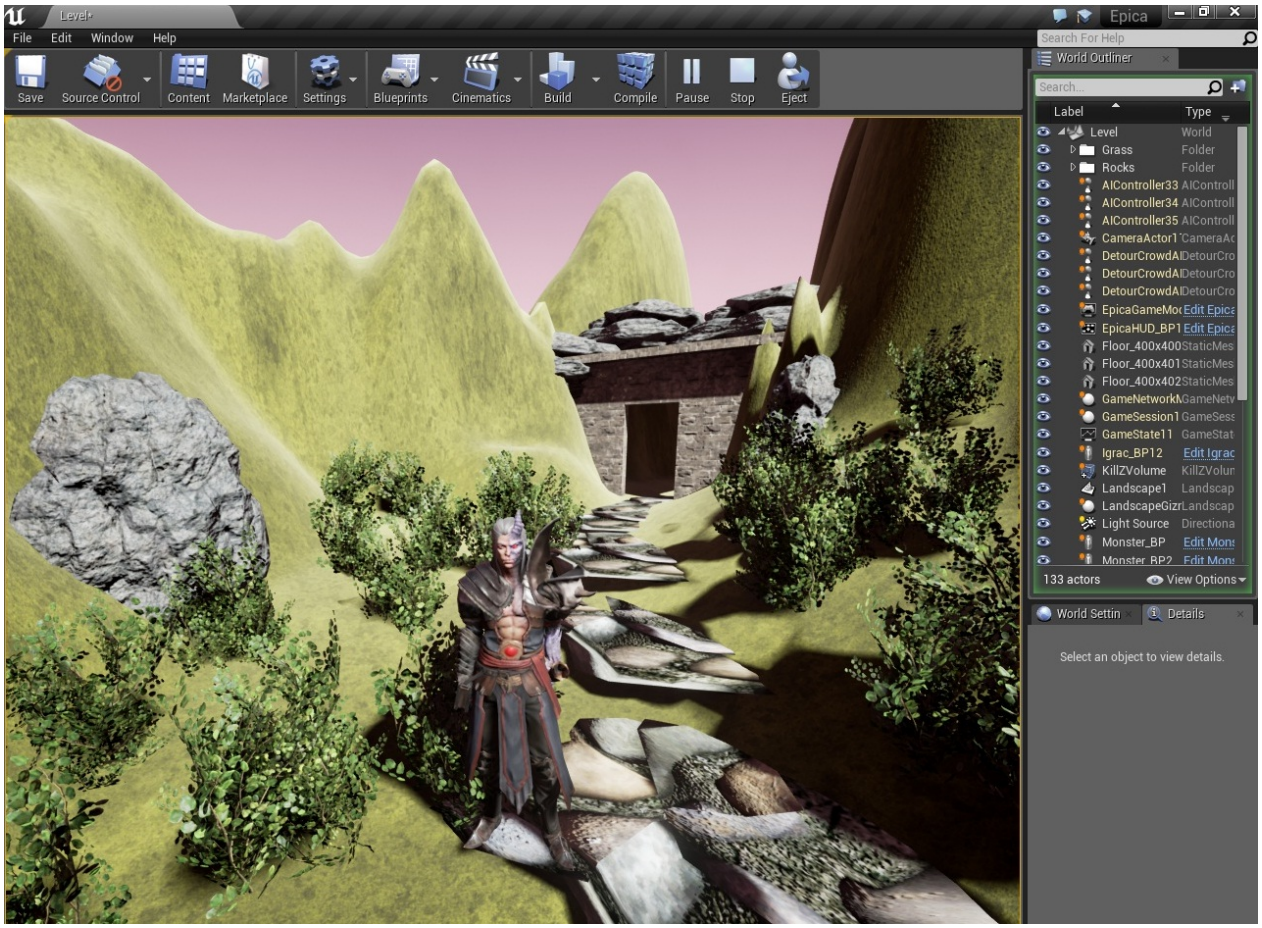

Slika 7.5: Prilazak prvom NPC-u i pogled na lijevu stazu

Reakcija NPC-a je izazvana našim ulaskom u područje njegove detekcijske sfere. Njene stvarne dimenzije nakon implementacije unutar scene možemo vidjeti ako izađemo iz simulacije scene i promatramo kroz editor (Slika 7.6).



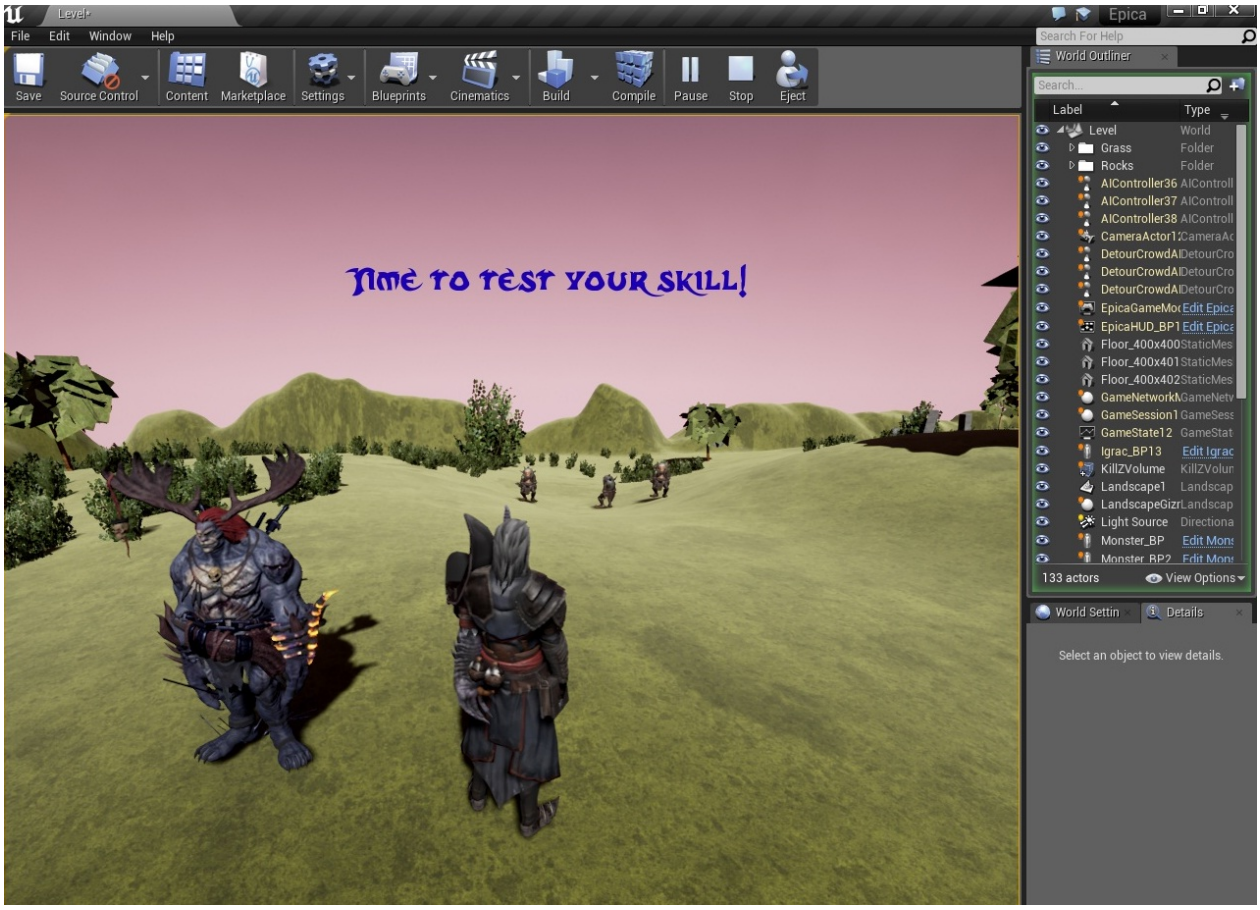

Slika 7.6: Detekcijska sfera prvog NPC-a

Vraćamo se natrag u simulaciju scene, ispred prvog NPC-a, na mjesto na kojem imamo izbor: lijeva ili desna staza. Ukoliko se odlučimo za desnu, nakon prolaska kroz mračnu prostoriju naići ćemo na iznenadan kraj puta nakon koga je provalija. Ako smo neoprezni i upadnemo, naš lik će "ispasti" iz granica scene, bit će uništen i scena će se restartovati odnosno vratiti će lika na početnu poziciju. Izgled te staze vidljiv je na Slici 7.7.



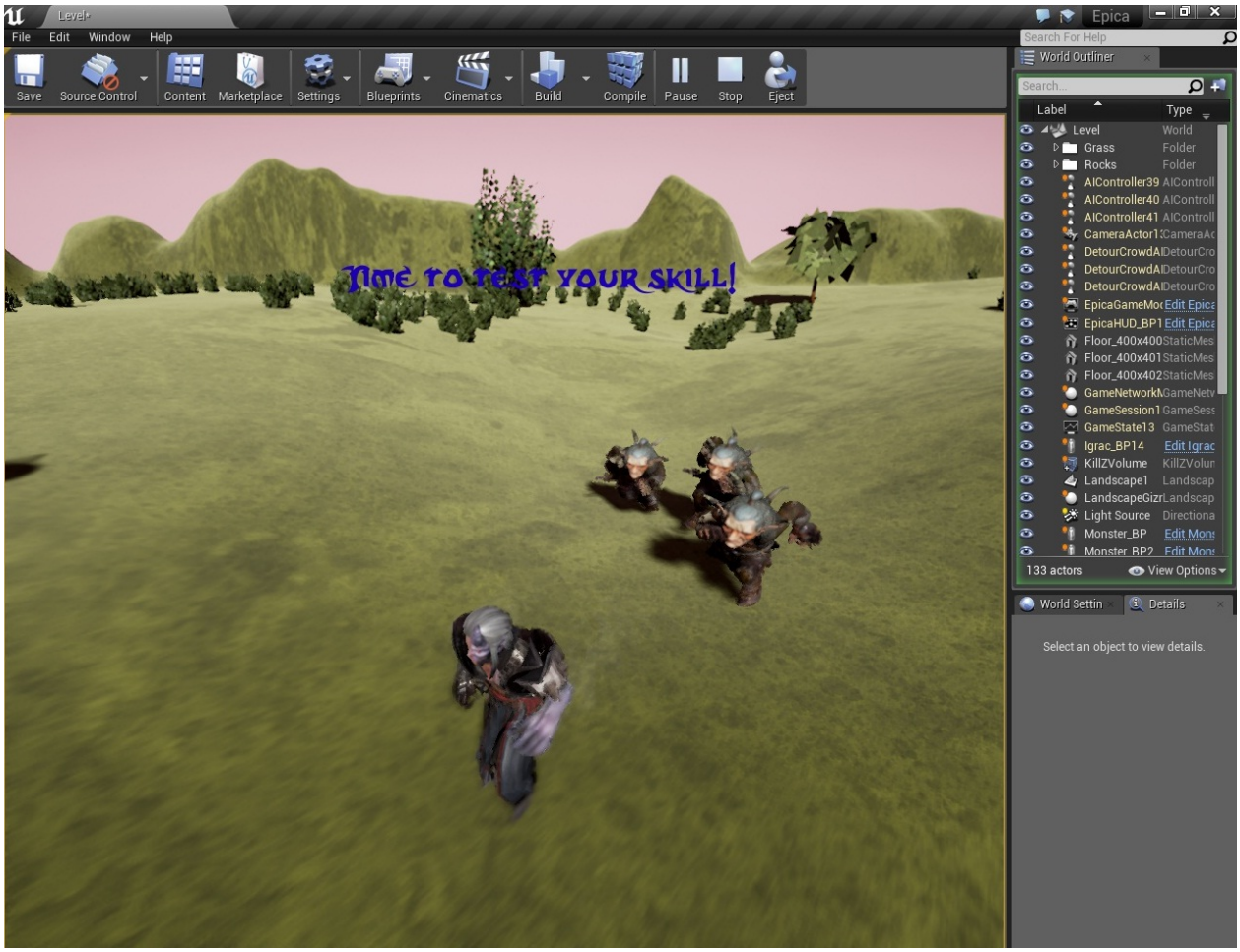

Slika 7.7: Staza koja vodi do provalije

Nakon restartovanja scene i vraćanja našeg lika na početnu poziciju, ponovno ćemo se uputiti do prvog NPC-a, ali ćemo ovaj put otići na lijevu granu staze. Prateći ju dolazimo do drugog NPC-a u sceni. On nam "kaže" da je vrijeme da pokažemo svoje vještine, a u njegovoj pozadini vidimo nekoliko neprijateljski nastrojenih malih goblina (Slika 7.8).




Slika 7.8: Drugi NPC i neprijatelji u pozadini

Zbog ponašanja kojeg smo im isprogramirali u "kosturu" klase i kasnije proširili u *Event Graphu*, čim uđemo unutar njihovog vidnog polja oni će početi trčati prema nama kao što je vidljivo na Slici 7.9. Njihovu brzinu kretanja smo postavili da bude nešto manja od igračeve tako da ako smo oprezni, iako ih je više, možemo pobjeći izvan njihovog vidnog polja zbog čega će nas oni prestati pratiti. Ukoliko to ne uspijemo, pri prvom kontaktu sa bilo kojim od njih naš lik bit će uništen, reproducirat će se postavljeni zvuk i nakon toga bit ćemo vraćeni na početnu poziciju u sceni.




Slika 7.9: Igrač pokušava pobjeći od neprijatelja

Ovime smo završili sa pregledavanjem scene iz pozicije igrača. Ukoliko smo gotovi sa uređivanjem scene i želimo da ona bude igriva na drugim računalima bez upotrebe UE4 Editora, moramo ju eksportirati u obliku *.exe* filea. To radimo preko opcije *Package Project*. U njoj dodatno možemo uređivati ime i opis našeg projekta, platforme na kojima želimo da se on pokreće itd. Nakon što je eksportiranje završeno i imamo *.exe* file našeg projekta, možemo ga postaviti na CD ili neki drugi prenosivi uređaj, i naš je projekt spreman za distribuciju.

# 8 Zaključak

Tema ovog rada bila je stvaranje jednostavne scene unutar Unreal Engine 4 programa koristeći se prethodno stečenim znanjem iz C++ programskog jezika. Unreal Engine 4 ima ugrađenu podršku za Visual Studio što smo i koristili kako bi napravili "kostur" svih elemenata scene. Tijekom ovog procesa pokazalo se da nije idealno kodirati sve na ovaj način. Za pravljenje složenijih struktura bilo bi potrebno poprilično dobro se snalaziti u nekim naprednijim aspektima C++ programskog jezika. Kako bi bio dostupan i početnicima, sam Unreal Engine Editor developerima nudi alternativu preko blueprintova i grafova ponašanja tako da svatko sam može odabrati u kojoj mjeri želi ručno programirati u C++ okruženju a koliko koristeći se drugim metodama. Scena stvorena u ovom radu predviđena je da se pokreće na osobnim računalima koji imaju Windows operativni sustav, no to ne znači da se igre napravljene koristeći Unreal Engine ne mogu igrati i na ostalim platformama. Štoviše, proces pravljenja igre za bilo koju platformu ostaje potpuno isti, jedino se po završetku, odnosno kada želimo eksportirati projekt za distribuciju, bira željena platforma. To znači da promjenom samo jedne opcije, naša scena može biti igriva na bilo kojoj platformi. Zbog ovoga, kao i zbog jednostavnosti i dostupnosti početnicima, Unreal Engine predstavlja jedan od najboljih alata za pravljenje svih vrsta video igara.

# Literatura

[1] B. Carnall *Unreal Engine 4.X By Example* Packt Publishing Ltd. Birmingham UK, 2015

[2] N. Valcasara *Unreal Engine Game Development Blueprints* Packt Publishing Ltd. Birmingham UK, 2016

[3] W. Sherif *Learning C++ by Creating Games with UE4* Packt Publishing Ltd. Birmingham UK, 2015

[4] *Unreal Engine 4 Documentation*

[5] M. Cvjetković, GitHub repository https://github.com/IceVortex/Epica_UE4