IZRADA 3D IGRE U UNREAL ENGINEU

Copić, Matej

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Split / Sveučilište u Splitu**

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-12-04**

Repository / Repozitorij:

Repository of University Department of Professional Studies

zir.nsk.hr

UNIVERSITY OF SPLIT

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni prijediplomski studij Računarstvo

MATEJ COPIĆ

Z A V R Š N I R A D

IZRADA 3D IGRE U UNREAL ENGINEU

Split, studeni 2024.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Stručni prijediplomski studij Računarstvo

Predmet: Objektno programiranje

Z A V R Š N I R A D

Kandidat: Matej Copić

Naslov rada: Izrada 3D igre u Unreal Engineu

Mentor: Nikola Grgić, viši predavač

Split, studeni 2024.

Sadržaj

Sažetak

Cilj ovog završnog rada je prikazati tehničke aspekte razvoja videoigre unutar okruženja Unreal Engine s naglaskom na žanr akcijskog *roguelikea*. Ova vrsta igre kombinira brzu akciju i nasumično generirane izazove, što igračima pruža dinamično iskustvo prepuno nepredvidivih elemenata. Razvijajući igru programer koristi alate kao što su *Blueprints* i programski jezik C++, koji su ključni za izradu složenih sustava unutar Unreal Enginea. Izrada ovakve igre zahtijeva poznavanje različitih aspekata razvoja, uključujući dizajn nivoa, rad s 3D modelima, animacijama te programiranje AI (engl. *Artificial Intelligence*) neprijatelja i ponašanja. U procesu razvoja posebna pažnja posvećena je balansiranju igračkog iskustva, osiguravajući da svaki prolaz kroz igru nudi jedinstveno iskustvo. To se postiže proceduralnim generiranjem nivoa te promjenama u izgledu i sposobnostima neprijatelja, što održava svježinu igre čak i nakon višestrukih prolazaka.

Ključne riječi: 3D igra, *Roguelike*, Unreal Engine

Summary

3D Game Development using Unreal Engine

The goal of this final thesis is to present the technical aspects of video game development within the Unreal Engine environment, with an emphasis on the action *roguelike* genre. This type of game combines fast-paced action and randomly generated challenges, giving players a dynamic experience full of unpredictable elements. Throughout game development, the developer uses tools such as *Blueprints* (visual scripting) and the C++ programming language, which are key to creating complex systems within the Unreal Engine. Making these kinds of games requires knowledge of various aspects of development, including level design, working with 3D models, animations, and programming AI enemies and their behaviors. During the development process, special attention was paid to balancing the gameplay, ensuring that each pass through the game offers a unique experience. This is achieved by random generation of levels, and changes in the appearance and abilities of enemies, which keeps the game fresh even after multiple playthroughs.

Keywords: 3D game, *Roguelike*, Unreal Engine

1. Uvod

U ovom radu prikazana je izrada akcijske *roguelike* igre u Unreal Engine okruženju. Glavni motiv za razvoj ovog projekta bio je istražiti mogućnosti koje nudi Unreal kao jedan od vodećih alata za razvoj videoigara. Sa svojim moćnim vizualnim alatima i podrškom za visoku razinu grafičkih detalja, okruženje omogućuje izradu složenih i tehnički zahtjevnih igara koje zadovoljavaju suvremene standarde industrije igara. Pristup izradi igre temelji se na korištenju jezika C++ i sustava *blueprint*, koji omogućuje vizualno skriptiranje bez potrebe za dubokim programerskim znanjem.

U radu su analizirane ključne komponente potrebne za razvoj *roguelike* igre, kao što su:

- Proceduralno generiranje nivoa koje osigurava da svaki prolazak kroz igru bude jedinstven.
- Borbeni sustav i kontrola igrača s naglaskom na brze i dinamične akcije.
- Različite vrste neprijatelja sa sustavom AI ponašanja.
- Sustav napredovanja i nadogradnje sposobnosti lika.
- Skupljanje resursa kroz razine.

Svi modeli i ostali resursi korišteni u igri preuzeti su iz Unreal Marketplacea ili kreirani koristeći dostupne alate za izradu sadržaja. Fokus je bio na optimizaciji performansi igre, koristeći Unrealove alate kao što su LOD (engl. *Level of Detail*) sustavi i dinamičko osvjetljenje kako bi igra bila fluidna i vizualno privlačna.

U idućim poglavljima ovog rada detaljno se obrađuju svi ključni aspekti izrade igre u Unreal Engine okruženju. Drugo poglavlje započinje pregledom osnovnih alata Unreal Enginea, uključujući njegovo radno sučelje, renderiranje i grafiku, te sustave za optimizaciju performansi. Razmatra se i fizika u igri te sustav *blueprint* koji omogućuje vizualno programiranje. Treće poglavlje fokusira se na samu igru Arqana, predstavljajući klase igrača i neprijatelja, njihove karakteristike, kretanje i borbeni sustav, kao i sustav menadžmenta igre, neprijatelja, razina i poboljšanja. Posebna pažnja posvećena je proceduralnom generiranju nivoa te sustavu oružja i projektila.

2. Alat Unreal Engine

Unreal Engine jedan je od najnaprednijih razvojnih alata za stvaranje interaktivnih 3D sadržaja koji nudi širok spektar funkcionalnosti za različite industrije s posebnim naglaskom na razvoj videoigara. Produkt Epic Gamesa koji omogućuje programerima i dizajnerima da kreiraju složene virtualne svjetove koristeći snažan grafički sustav i moćne alate za skriptiranje i optimizaciju. Jedna od ključnih prednosti Unreala je njegova sposobnost renderiranja u stvarnom vremenu, koja omogućuje kreiranje vizualno impresivnih scena s dinamičkim osvjetljenjem, sjajnim materijalima i realistično prikazanim efektima.

Funkcionalnosti Unreala obuhvaćaju sve aspekte razvoja igara, od dizajna okoliša do programiranja mehanike igre (engl. *gameplay*). Sustav materijala omogućuje korisnicima da stvaraju složene i fotorealistične teksture, dok napredni alati za postavljanje svjetla i sjena pružaju potpunu kontrolu nad atmosferom unutar igre. Okruženje podržava fizikalno bazirano renderiranje, što doprinosi stvaranju realističnih materijala i simuliranju refleksije svjetla.

Kada je riječ o kreiranju logike igre, Unreal nudi jedinstveni sustav *blueprinta*, koji omogućuje vizualno skriptiranje bez potrebe za pisanjem kôda. Ovaj sustav omogućuje jednostavnu implementaciju interaktivnih elemenata unutar igre, a napredniji korisnici mogu koristiti C++ za preciznu kontrolu nad svakim aspektom projekta. Mogućnost kombiniranja vizualnog skriptiranja i tradicionalnog programiranja čini okruženje iznimno prilagodljivim za različite vrste korisnika, od početnika do profesionalnih razvojnih timova.

Okruženje također nudi integrirane alate za animaciju, koji omogućuju kreiranje složenih animacija likova, kao i kontrolu nad njihovim pokretima i interakcijom s okolinom. Osim toga, ugrađeni alati za umjetnu inteligenciju (AI) omogućuju programerima da stvore inteligentne likove koji mogu donositi odluke, kretati se kroz razine igre i reagirati na akcije igrača.

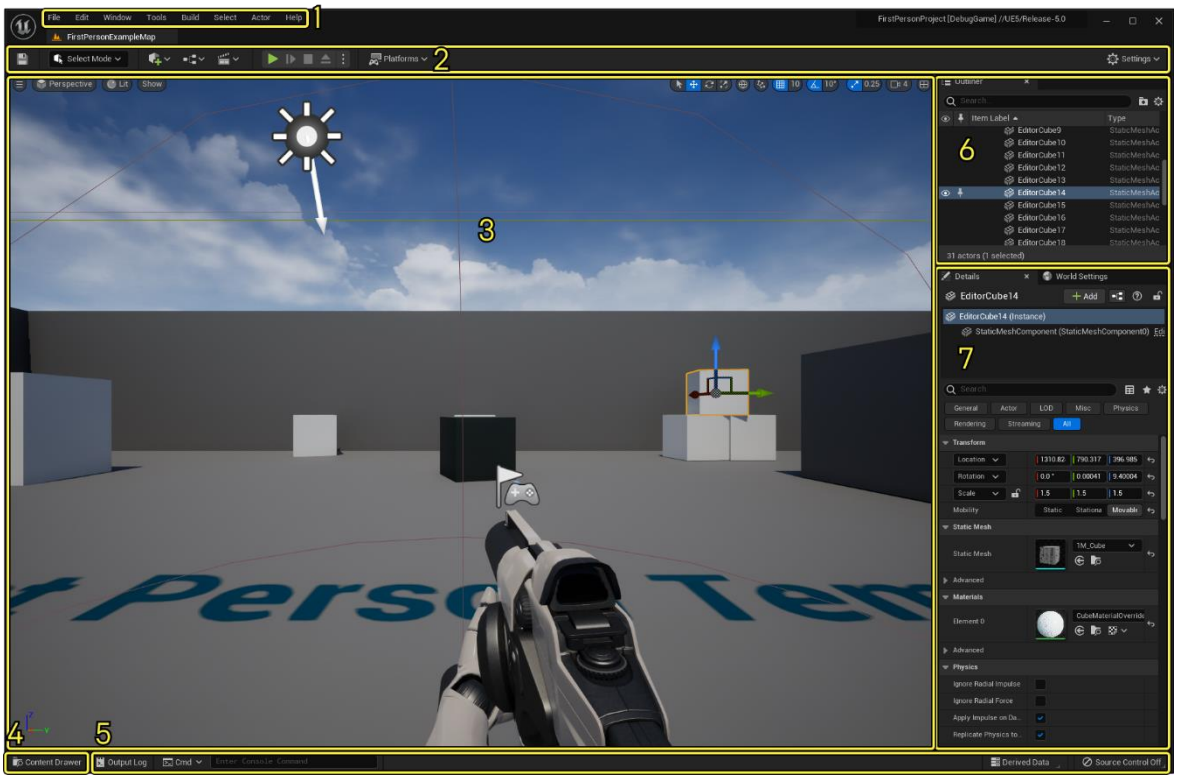
Za razvoj multiplayer igara, Unreal Engine nudi podršku za mrežne igre i replikaciju objekata, omogućujući sinkronizaciju igre između više igrača. Mrežni sustav omogućuje razvoj kompleksnih mrežnih okruženja s minimalnim utjecajem na performanse, što ga čini idealnim alatom za kreiranje multiplayer iskustava.

Osim svih ovih mogućnosti, Unreal nudi snažne alate za optimizaciju performansi, poput Profilera i sustava za upravljanje resursima, što omogućuje programerima da optimiziraju svoje projekte za različite platforme, uključujući PC, konzole i mobilne uređaje. Završni korak svakog projekta – izvoz i distribucija – također je podržan, s opcijama za izvoz igre na više platformi i jednostavno postavljanje finalnog proizvoda.

Unreal Engine, svojim širokim rasponom funkcionalnosti i prilagodljivošću, ističe se kao moćan alat za razvoj kompleksnih, visokokvalitetnih igara i interaktivnih sadržaja.

2.1 Radno sučelje

Prije početka rada, važno je upoznati se s korisničkim sučeljem alata. Glavni zaslon je prilagodljiv i omogućuje korisnicima da organiziraju radne prozore prema svojim preferencijama. To uključuje mogućnost prilagođavanja rasporeda prozora i kartica. Glavno sučelje je prikazano na slici 1.



Slika 1: Unreal Engine sučelje

Glavno sučelje sastoji se od (1) izborničke trake (engl. *Menu Bar*), (2) glavne alatne trake (engl. *Main Toolbar*), (3) prikaza razine (engl. *Level Viewport*), (4) ladice sadržaja

(engl. *Content Drawer*), (5) donje alatne trake (engl. *Bottom Toolbar*), (6) hijerarhijskog prikaza (engl. *Outliner*) i (7) ploče s detaljima (engl. *Details Panel*).

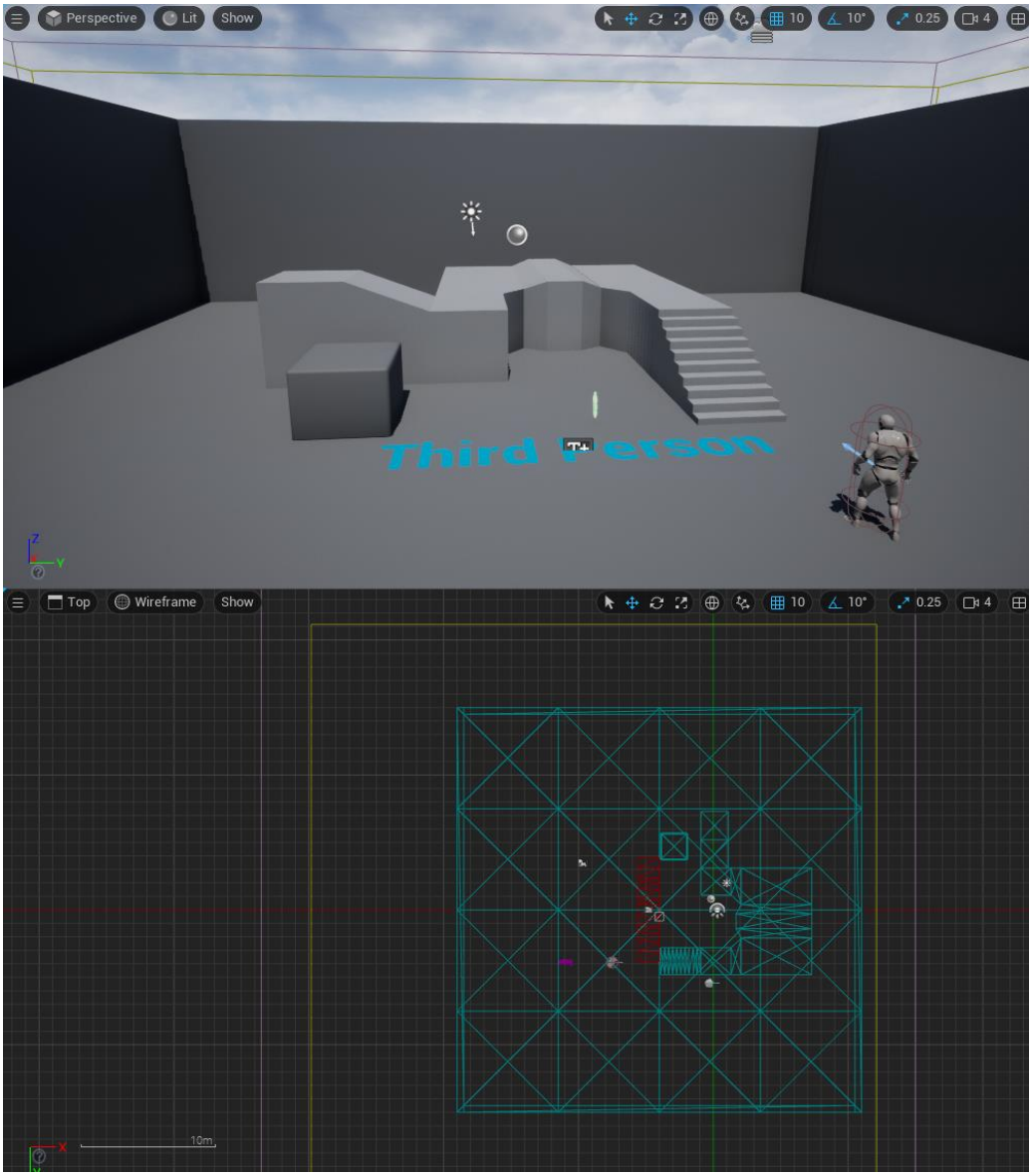
Izbornička traka sadrži naredbe, funkcije i postavke specifične za uređivač. Glavna alatna traka (slika 2) sadrži prečace do nekih od najčešće korištenih alata i naredbi u Unreal Editoru. Podijeljena je na sljedeća područja: (1) dugme za spremanje, (2) odabir načina rada (engl. *Mode Selection*), (3) prečaci sadržaja (engl. *Content Shortcuts*), (4) kontrole načina reprodukcije (engl. *Play Mode Controls*), (5) izbornik platformi, (6) postavke.

Slika 2: Glavna alatna traka

Okvir za prikaz razine prikazuje sadržaj razine koja je trenutno otvorena. Ovdje se može pregledavati i uređivati sadržaj vaše aktivne razine, što je moguće vidjeti na slici 3. Okvir za prikaz razine općenito može prikazati sadržaj razine na dva različita načina:

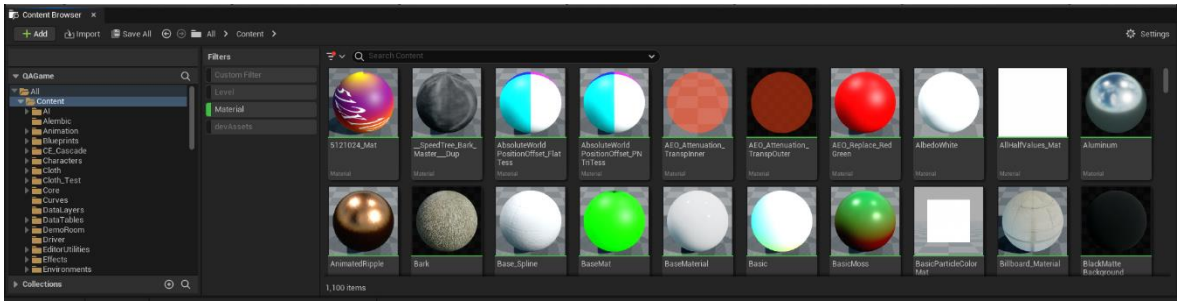
- Perspektiva ili 3D prikaz kojim se može kretati kako bi se vidio sadržaj okvira za prikaz iz različitih kutova.

- Ortografski ili 2D prikaz koji gleda niz jednu od glavnih osi (X, Y ili Z).



Slika 3: Prikaz razine

Ladica sadržaja je prozor za pregled datoteka koji prikazuje sva sredstva (engl. *assets*), *blueprinte* i druge datoteke sadržane u vašem projektu. Može se koristiti za pregledavanje sadržaja, povlačenje sredstava u razinu, premještanje sredstava između projekata i više. Izgled Ladice sadržaja je moguće vidjeti na slici 4.

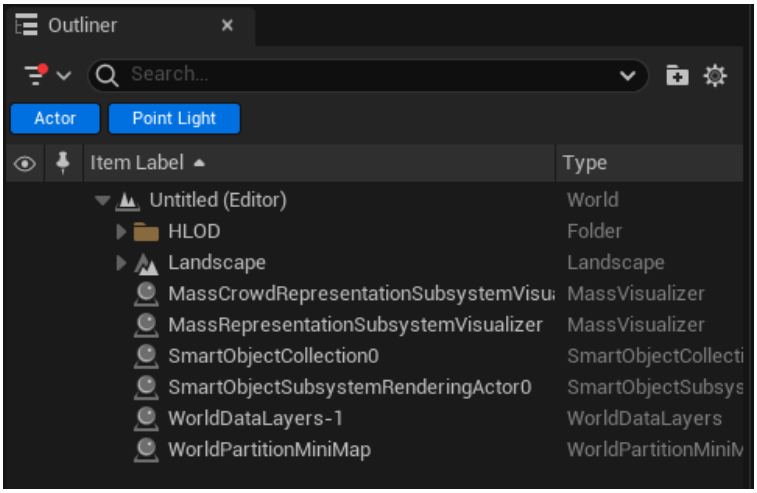


Slika 4: Ladica sadržaja

Donja alatna traka (slika 5) sadrži prečace do izlaznog dnevnika (engl. *Output Log*), naredbene konzole (engl. *Command Console*) i funkcionalnosti izvedenih podataka (engl. *Derived Data*). Također prikazuje status kontrole izvora (engl. *Source Control*).

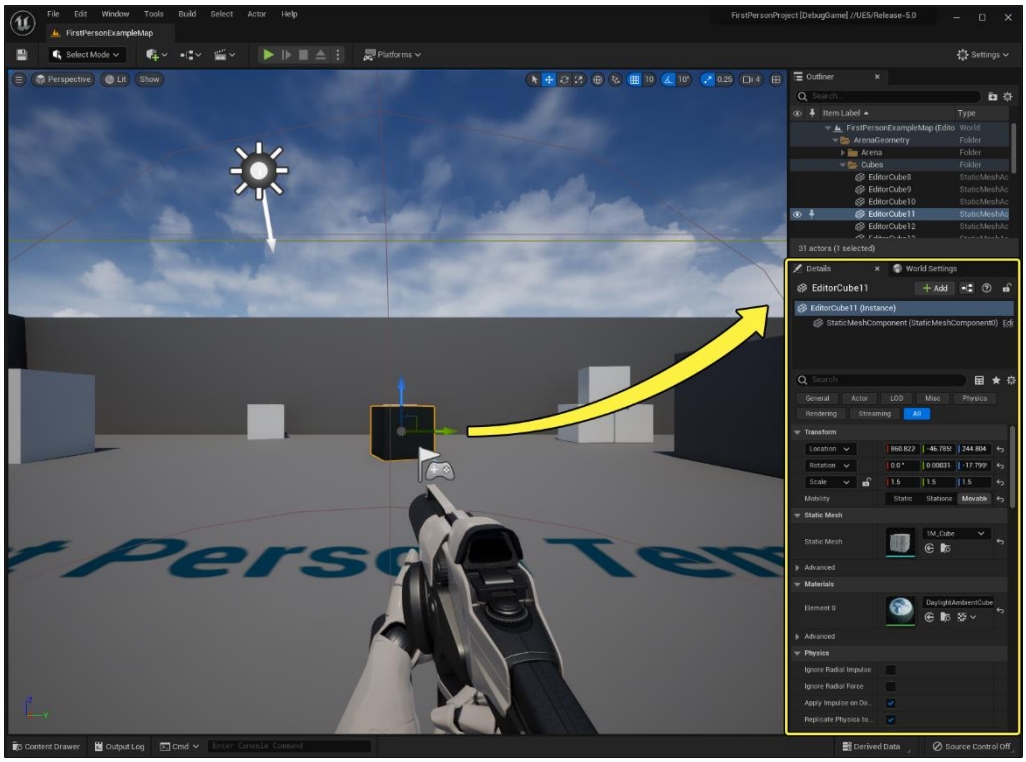
Slika 5: Donja alatna traka

Na slici 6 prikazana je Hijerarhijska ploča. Kao što naziv sugerira, prikazuje sadržaj unutar razine hijerarhijskim poretkom. Osim toga, može se jako lako dodavati i brisati sadržaj unutar razine.



Slika 6: Hijerarhijska ploča

Kada se odabere entitet (engl. *actor*) u prikazu razine, ploča s detaljima prikazuje postavke i svojstva koja utječu na odabrani entitet, što je vidljivo na slici 7.



Slika 7: Ploča s detaljima

2.2 Renderiranje i grafika

Unreal Engine nudi impresivne grafičke mogućnosti, posebno u domeni renderiranja u stvarnom vremenu (engl. *Real-time Rendering*), što omogućuje stvaranje visokokvalitetnih vizualnih sadržaja. Alat pruža napredne tehnike osvjetljenja, detaljnu kontrolu nad materijalima i teksturama te integrirane alate za optimizaciju grafike kako bi projekti funkcionirali na različitim platformama. Renderiranje u stvarnom vremenu omogućuje razvoj sadržaja s dinamičnim promjenama osvjetljenja i sjena dajući realistično iskustvo korisnicima.

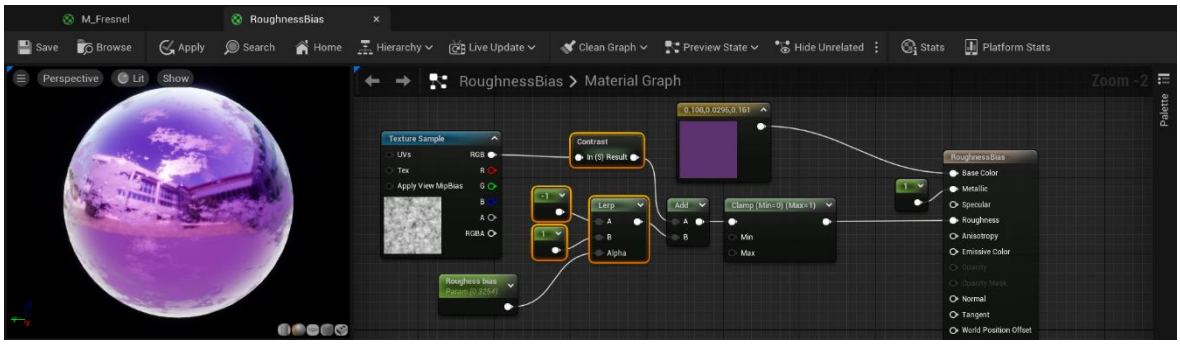
2.2.1 Materijali, shaderi i teksture

Unreal Engine nudi iznimno moćan i fleksibilan sustav za rad s materijalima. Korisnici mogu kreirati materijale koristeći urednik materijala (engl. *Material Editor*), gdje se koriste čvorovi (engl. *nodes*) za kombiniranje različitih tekstura i efekata. Sustav

materijala podržava fotorealistične prikaze, ali i stilizirane grafičke efekte, ovisno o potrebama projekta. Materijali mogu biti napravljeni tako da reagiraju na svjetlost na realističan način koristeći materijale temeljene na fizikalnim svojstvima. Urednik materijala se može vidjeti na slici 8.

Primjena tekstura unutar materijala omogućuje još veću kontrolu nad izgledom objekata. Teksture se mogu koristiti za definiranje površinskih detalja kao što su boja, hrapavost, mape za simulaciju reljefa, refleksije i druge osobine materijala. Ovi elementi pridonose stvaranju vizualno bogatih scena koje izgledaju prirodno i uvjerljivo.

Jedna od ključnih karakteristika sustava materijala je prilagodljivost. Unreal Engine podržava dinamičke materijale koji mogu mijenjati svojstva tijekom izvođenja igre, primjerice boju ili emisiju svjetla, ovisno o uvjetima u igri. Takva fleksibilnost omogućuje stvaranje dinamičnih i interaktivnih elemenata unutar scene.



Slika 8: Urednik materijala

2.2.2 Optimizacija grafike

Kako bi se osigurale visoke performanse uz održavanje vrhunske grafike, Unreal Engine nudi alate za optimizaciju poput sustava razine detalja (engl. *Level of Detail*), koji automatski smanjuje detalje objekata na velikim udaljenostima od kamere.

Osim toga, uklanjanje okluzije (engl. *Occlusion Culling*) sprječava renderiranje objekata koje igrač ne vidi, čime se štedi na performansama i resursima.Optimizacija je ključna za igre koje se razvijaju za različite platforme, od PC-a i konzola do mobilnih uređaja

i Unreal Engine osigurava alate kako bi igre funkcionirale besprijekorno bez gubitka kvalitete slike.

2.3 Fizika u Unreal Engineu

Unreal Engine posjeduje snažan fizikalni sustav koji omogućuje simulaciju stvarnog svijeta unutar igara i aplikacija. Koristeći integrirane module fizike, programeri mogu simulirati različite fizičke fenomene poput kretanja objekata, sudara, sile gravitacije i interakcije s okolinom. Fizika je ključni element u mnogim igrama, osobito onima koje zahtijevaju realizam u kretanju objekata i likova.

Simulacija krutih tijela omogućuje realistično kretanje i sudare između objekata. Kada dva objekta dođu u kontakt, Unreal Engine izračunava njihove interakcije, što uključuje odskok, trenje i gubitak energije prilikom sudara. Ova simulacija je ključna za mnoge igre, osobito one koje uključuju akcijske scene ili destrukciju okoliša. Osim krutih tijela, Unreal Engine omogućuje simulaciju mekih tijela i tkanina. Simulacija tkanina koristi se za realističan prikaz odjeće, zavjesa, zastava i sličnih objekata. Unreal Engine omogućuje precizno simuliranje načina na koji tkanina reagira na sile poput vjetra, gravitacije i kontakta s drugim objektima. Tkanina se može pričvrstiti za likove i tako omogućuje njeno prirodno kretanje u skladu s pokretima lika.

Unreal Engine pruža mogućnost modificiranja fizike kroz sustav *blueprint*a i C++ kôd. Programeri mogu dodavati prilagođene fizičke efekte ili prilagoditi postojeće simulacije kako bi stvorili jedinstveno iskustvo unutar igre. Na primjer, programer može koristiti *blueprinte* za definiranje ponašanja objekata u specifičnim uvjetima, poput pada objekta s visine ili reakcije na eksploziju.

Skripte također omogućuju integraciju fizike s drugim sustavima unutar igre, poput animacija, zvuka ili vizualnih efekata. Na ovaj način fizikalni sustav može postati sastavni dio iskustva igranja, poboljšavajući interakciju igrača s virtualnim svijetom.

2.4 Blueprinti

Blueprinti su jedan od najistaknutijih alata u Unreal Engineu, pružajući korisnicima mogućnost vizualnog skriptiranja bez potrebe za pisanjem kôda. Korištenje *blueprinta* omogućuje izradu kompleksne logike putem vizualnog sučelja, što značajno ubrzava razvojni proces i čini ga pristupačnijim osobama koje nemaju iskustva s programiranjem. Ovaj sustav koristi čvorove koji se povezuju kako bi definirali ponašanje objekata, interakcije, animacije i druge aspekte igre.

Blueprinti se temelje na vizualnom skriptiranju pomoću čvorova koji predstavljaju različite funkcije, varijable, događaje i akcije. Ovi čvorovi se povezuju kako bi stvorili logičke tokove koji definiraju što će se dogoditi u igri kao rezultat određenih akcija ili uvjeta. Svaki *blueprint* se može promatrati kao skripta koji kontrolira određeni aspekt igre, bilo da je riječ o ponašanju pojedinačnog objekta ili o kompleksnom sustavu.

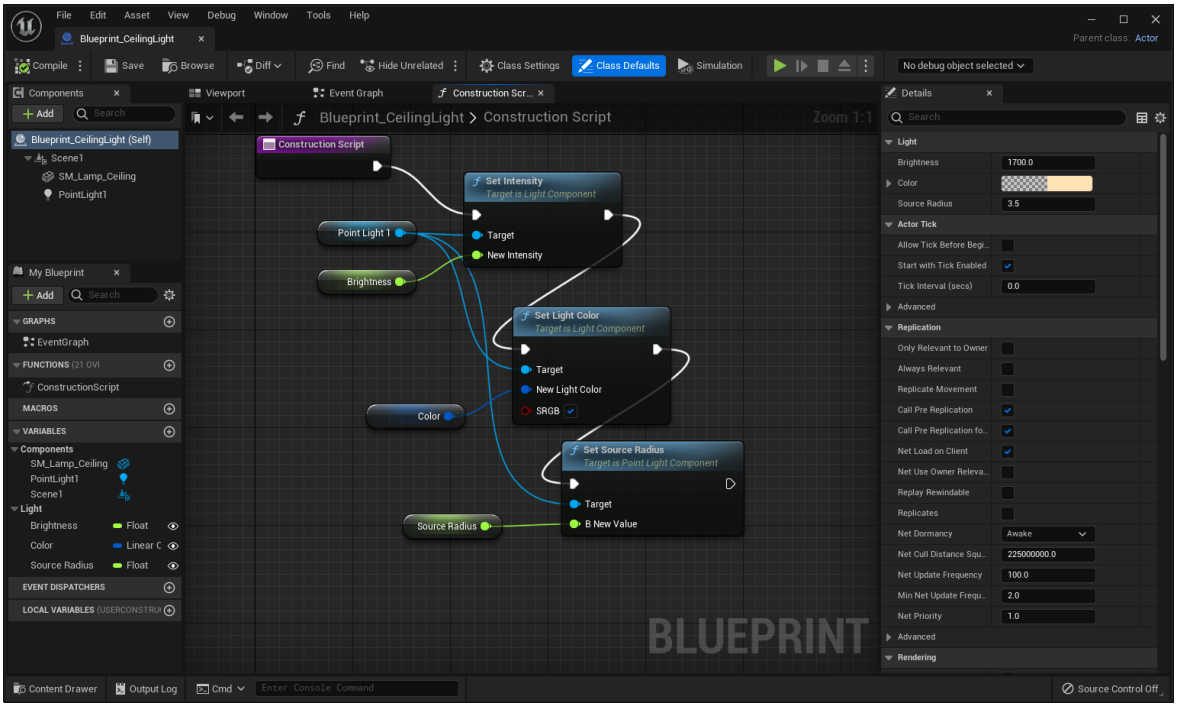
Postoje različite vrste *blueprinta*:

- *Blueprinti* razine: kontrolira događaje i logiku na razini igre.
- Klasni *blueprinti*: omogućuje stvaranje prilagođenih klasa objekata s jedinstvenim ponašanjem.
- *Blueprinti* sučelja: definira skup funkcija koje klase mogu implementirati.

2.4.1 Prednosti i mane blueprinta

Jedna od glavnih prednosti *blueprinta* je njihova jednostavnost i intuitivnost. Umjesto pisanja kôda, korisnici jednostavno povlače i spajaju čvorove unutar uređivača *blueprinta* (slika 9), čime se smanjuje mogućnost pogrešaka koje su često prisutne u tradicionalnom kodiranju. *Blueprinti* omogućuju brzu iteraciju i testiranje jer se promjene mogu odmah primijeniti i testirati unutar enginea, što ubrzava razvojni ciklus.

Iako *blueprinti* u Unreal Engineu pružaju jednostavnost i pristupačnost, imaju nekoliko mana. Prvo, u složenim projektima mogu postati nepregledni i teški za održavanje. Drugo, performanse *blueprinta* su često slabije od C++ kôda, što može utjecati na efikasnost u velikim igrama. Konačno, suradnja u timovima i kontrola verzija mogu biti kompliciraniji zbog prirode vizualnog sučelja *blueprinta*.



Slika 9: Uređivač *blueprinta*

2.4.2 Interakcije i događaji

Blueprinti omogućuju definiranje interakcija između različitih objekata unutar igre. Na primjer, jednostavnim povlačenjem čvorova, programer može definirati što će se dogoditi kada igrač pritisne dugme, kada objekt uđe u određeno područje ili kada dva objekta dođu u kontakt. Ovi događaji mogu biti povezani s različitim akcijama, poput pokretanja animacija, igranja zvukova ili promjene stanja igre.

Jedan od najčešćih primjera je korištenje *blueprinta* za kontroliranje ponašanja vrata u igri. Programer može definirati događaj koji se pokreće kada igrač dođe u blizinu vrata i pritisne određeni dugme nakon čega se vrata automatski otvaraju animacijom.

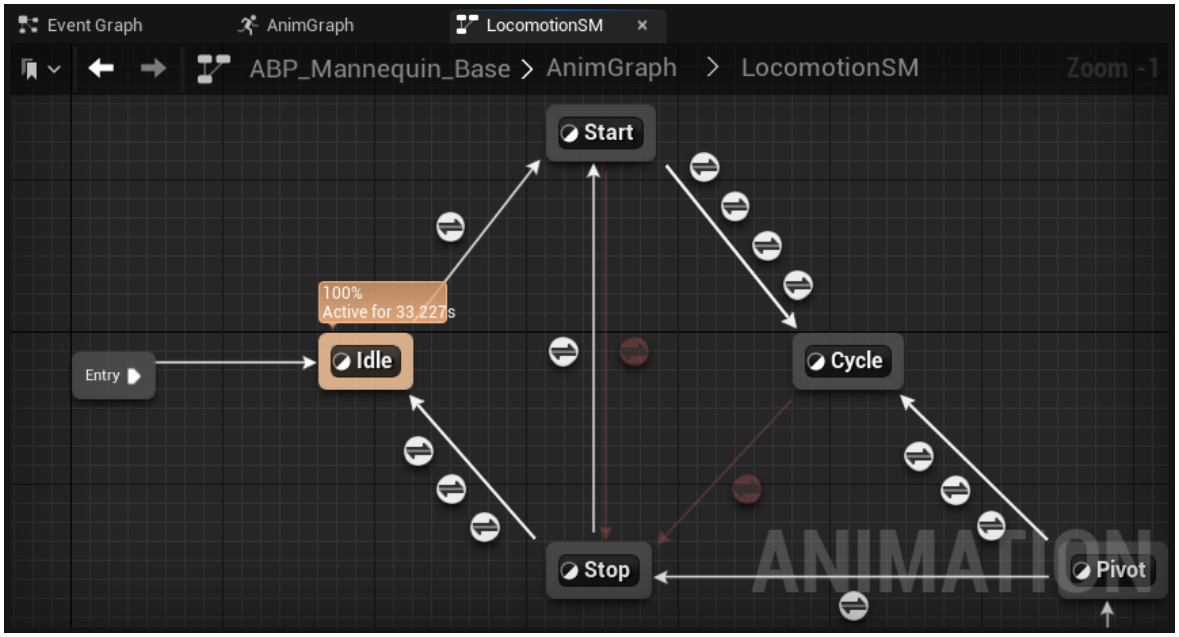
2.4.3 Varijable i funkcije

Unutar *blueprinta* moguće je definirati varijable koje mogu pohraniti informacije o stanju igre, primjerice zdravlje igrača, pozicija objekata ili trenutno oružje koje igrač koristi. Ove varijable mogu se koristiti za uvjetno izvršavanje različitih akcija unutar igre.

Primjera radi, ako zdravlje igrača padne ispod određene razine, *blueprint* može pokrenuti događaj koji aktivira animaciju gubitka života ili prebacivanje na ekran za kraj igre.

2.4.4 Animacije i uređaj stanja

Korištenje *blueprinta* nije ograničeno samo na logiku igre; može se koristiti i za kontroliranje animacija, vizualnih efekata te mnogih drugih aspekata vizualnog prikaza igre. Programer primjerice može koristiti *blueprinte* i uređaj stanja (engl. *State Machine*) kako bi pokrenuo animacije likova na temelju određenih uvjeta, poput trčanja, skakanja ili napada. Animacije se mogu povezati s logikom unutar *blueprinta* omogućavajući stvaranje kompleksnih interakcija između likova i okoline. Primjer uređaja stanja se može vidjeti na slici 10.



Slika 10: Uređaj stanja

2.4.5 Predlošci i gotova rješenja

Unreal Engine dolazi s brojnim predlošcima i gotovim rješenjima unutar *blueprinta*, što omogućuje programerima brz početak rada. Na primjer, postoji veliki broj gotovih *blueprinta* za osnovne *gameplay* mehanike poput kretanja, skakanja, sakupljanja predmeta te interakcije s okolišem. Ovi se predlošci mogu prilagoditi i proširiti kako bi odgovarali specifičnim potrebama projekta čime se značajno smanjuje vrijeme potrebno za izradu osnovnih funkcionalnosti.

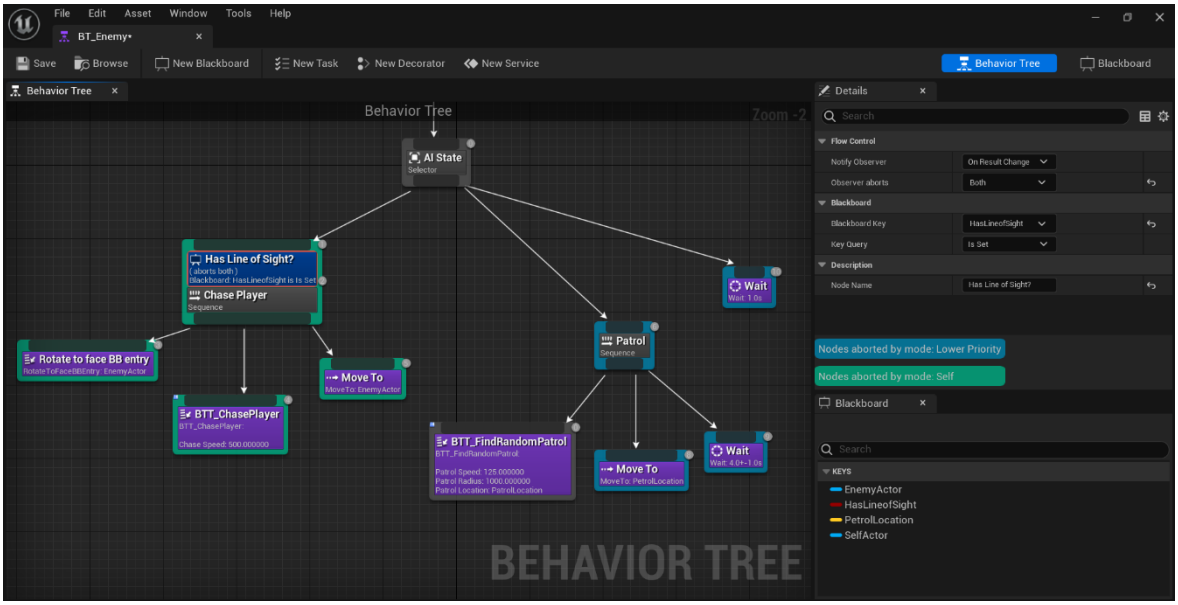
2.5 Alati umjetne inteligencije

Unreal Engine nudi širok spektar alata za razvoj AI-a (engl. *Artificial Intelligence*) unutar igara, omogućujući stvaranje složenih, dinamičnih neprijatelja, NPC-ova (engl. *Non-player characters*) i drugih elemenata igre. Ovi alati su dizajnirani kako bi olakšali izradu AI ponašanja bez potrebe za dubokim poznavanjem programiranja, iako iskusniji programeri mogu iskoristiti dodatne funkcionalnosti kroz C++ kodiranje.

2.5.1 Stablo ponašanja

Jedan od ključnih AI alata u Unreal Engineu su stabla ponašanja (engl. *Behavior Trees*), koja omogućuju modeliranje složenog ponašanja likova. Primjer stabla ponašanja može se vidjeti na slici 11. To je grafički alat za definiranje hijerarhije zadataka i odluka koje AI likovi donose tijekom igre. Pomoću ovog sustava programeri mogu jednostavno definirati kako će se likovi ponašati u različitim situacijama, od osnovnih zadataka poput patroliranja do složenijih interakcija poput borbe i reagiranja na okolinu.

Stabla ponašanja funkcioniraju na principu izvršavanja čvorova unutar stabla zadataka. Na vrhu stabla je korijen čvor od kojeg se granaju različiti zadaci i uvjeti koji definiraju ponašanje AI lika. Na primjer, AI može prvo provjeriti postoje li neprijatelji u blizini, a zatim odlučiti hoće li napasti, pobjeći ili se sakriti. Ova struktura omogućuje fleksibilnost u dizajniranju ponašanja i lako proširivanje dodavanjem novih čvorova i uvjeta.



Slika 11: Stablo ponašanja

2.5.2 Navigacijska mreža

Navigacija je ključni aspekt AI sustava u igrama, a Unreal Engine koristi sustav navigacijske mreže (engl. *Navigation Mesh*) za rješavanje ovog problema. Navigacijska mreža je mreža koja definira prohodna područja u igri te omogućuje AI likovima da se kreću realističnim putem unutar okoline. Kada je mreža postavljena, AI može automatski pronaći put između različitih točaka na karti, izbjegavati prepreke i dinamično reagirati na promjene u okolišu.

Navigacijska se mreža automatski generira prema geometriji levela i može se prilagoditi kako bi se kontroliralo gdje AI likovi mogu i ne mogu hodati. Na primjer, može se definirati da AI likovi ne mogu hodati po strmim padinama ili da trebaju izbjegavati određene opasne zone. Navigacijska mreža također podržava složene strukture poput mostova i različitih visina što omogućuje AI-u da se inteligentno kreće u kompleksnim 3D okruženjima.

2.5.3 AI kontroleri

AI kontroleri su klase unutar Unreal Enginea koje upravljaju ponašanjem AI likova. Svaki AI lik u igri povezan je s AI kontrolerom koji kontrolira njegovu logiku, kretanje i interakciju s drugim objektima unutar igre. AI kontroleri se mogu koristiti zajedno sa

stablima ponašanja i navigacijskim mrežama kako bi se upravljalo ponašanjem NPC-ova. Korištenje AI kontrolera omogućuje jednostavno odvajanje logike AI ponašanja od samih likova što olakšava upravljanje i ponovno korištenje istih kontrolera na različitim likovima u igri. Na primjer, jedan AI kontroler može se koristiti za kontroliranje svih neprijateljskih vojnika u igri, dok drugi kontroler može biti zadužen za upravljanje ponašanjem civilnih NPC-ova.

3. Arqana

Igra nosi ime inspirirano tarot arkanama, s klasama poput *World*, *Fool* i *Emperor* koje igraču nude različite stilove igre. Ključna mehanika uključuje aktivaciju specijalnih sposobnosti preko dugmeta "Q", što je simbolično uklopljeno i u ime igre.

3.1 Klasa igrača

Klasa `APlayerCharacter` u projektu implementira ključne funkcionalnosti igrača. Ova klasa nasljeđuje od `ACharacter` klase koja nudi osnovne mogućnosti kontrole lika, uključujući kretanje, interakciju s okolinom i upravljanje oružjem. Detaljnija analiza njezinih glavnih funkcionalnosti može se razdijeliti u nekoliko ključnih dijelova: kretanje, sustav oružja, borba, specijalne sposobnosti te integracija s drugim elementima igre.

3.1.1 Kretanje i kamera

Kretanje lika implementirano je pomoću funkcija `MoveForward` i `MoveRight` koje omogućuju igraču kontrolu nad smjerom kretanja putem ulaza (`Input`). Kamera se postavlja pomoću funkcije `SetupCamera` gdje je kamera prilagođena da prati lik iz perspektive prvog lica, što je često korišteno u akcijskim igrama. Ova funkcionalnost omogućuje igraču jednostavno i intuitivno upravljanje likom u 3D prostoru, kao što je prikazano u ispisu 1.

```
void APlayerCharacter::MoveForward(float InputValue)
{
	FVector ForwardDirection = GetActorForwardVector();
	AddMovementInput(ForwardDirection, InputValue);
}

void APlayerCharacter::MoveRight(float InputValue)
{
	FVector RightDirection = GetActorRightVector();
	AddMovementInput(RightDirection, InputValue);
}

void APlayerCharacter::SetupCamera()
{
	Camera = CreateDefaultSubobject<UCameraComponent>(TEXT("Player
Camera"));
	Camera->SetupAttachment(RootComponent);
	Camera->bUsePawnControlRotation = true;
	Camera->SetRelativeLocation(FVector(0.0f, 0.0f, 50.0f));
}
```

Ispis 1: Kretanje i kamera

3.1.2 Oružje igrača

Sustav oružja igra važnu ulogu u igri, a dodjeljivanje oružja implementirano je metodom `AttachWeapon` koja omogućuje dodavanje oružja liku. Ovo je posebno važno u igrama koje se fokusiraju na borbu, točnije gdje različita oružja mogu imati različite učinke na neprijatelje. Funkcija koristi klasu `ABaseWeapon` za kreiranje i pridruživanje oružja liku, što je vidljivo u ispisu 2.

```
void APlayerCharacter::AttachWeapon(TSubclassOf<ABaseWeapon>
WeaponClassParam)
{
	if (WeaponClass && GetWorld())
	{
		FActorSpawnParameters SpawnParams;
		SpawnParams.Owner = this;
		SpawnParams.Instigator = GetInstigator();
		Weapon = GetWorld()-
>SpawnActor<ABaseWeapon>(WeaponClassParam,	SpawnParams);

		if (Weapon && CharacterMesh)
		{
			Weapon->AttachToComponent(CharacterMesh,
FAttachmentTransformRules::SnapToTargetIncludingScale,
			TEXT("WeaponSocket"));

		}
	}
}
```

Ispis 2: Dodavanje oružja

3.1.3 Karakteristike i specijalne sposobnosti igrača

Konstruktor klase `APlayerCharacter` inicijalizira osnovne vrijednosti statistika igrača, poput zdravlja (`health`), snage (`strength`), kritičnog udara (`crit`) i agilnosti (`dexterity`). Te vrijednosti definiraju osnovne sposobnosti lika u borbi i interakciji s okolinom. Konstruktor je prikazan u ispisu 3.

```
APlayerCharacter::APlayerCharacter()
{

	health = 100.0f;
	strength = 1.0f;
	crit = 2.0f;
	critChance = 0.1f;
	dexterity = 1.1f;
	abilityCooldown = 15;

	isAbilityCooldown = false;
	DeathDefiance = false;

	SetupCamera();
	SetupCharacterMesh();
	SetupCharacterMovement();

}
```

Ispis 3: Konstruktor igrača

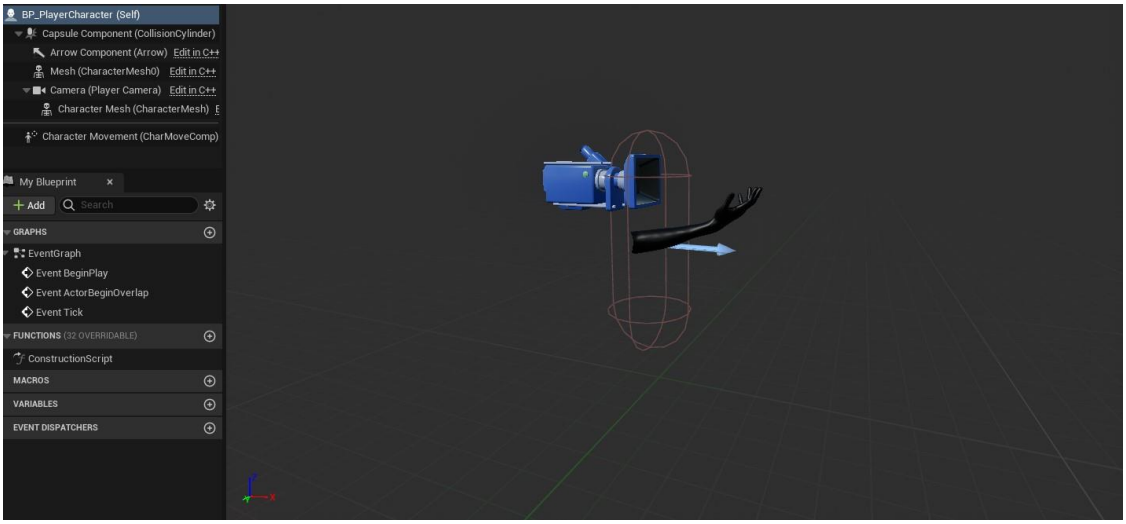
Kao i mnogi moderni likovi u igrama, klasa `APlayerCharacter` ima specijalne sposobnosti definirane kroz funkcije kao što su `ResetAbilityCooldown` i `SpecialAbility`, koju je moguće vidjeti u ispisu 4. Ove funkcije omogućuju igraču korištenje posebnih moći s vremenskim ograničenjem (`abilityCooldown`). Implementacija specijalnih sposobnosti omogućuje stvaranje unikatnih mehanika.

```
void AWorldPlayerCharacter::SpecialAbility()
{
	if (isAbilityCooldown)
	{
		return;
	}

	UWorld* World = GetWorld();
	if (World)
	{
		World->GetWorldSettings()->SetTimeDilation(0.1f);

		CustomTimeDilation = 10.0f;

		GetWorldTimerManager().SetTimer(TimeHandle, this,
&AWorldPlayerCharacter::ResetTime, 0.5f, false);
	}

	isAbilityCooldown = true;
	GetWorld()->GetTimerManager().SetTimer(AbilityCooldownTimerHandle,
this, &APlayerCharacter::ResetAbilityCooldown, abilityCooldown, false);
}
```

Ispis 4: Specijalna sposobnost igrača

3.1.4 Izgled igrača

Izgled igrača je riješen pomoću C++ kôda i *blueprinta*, što je prikazano na slici 12. Oblik igrača (engl. *Mesh*) je učitan u funkciji `SetupCharacterMesh` (ispis 5), dok su pozicija i ostali detalji namješteni u *blueprintu*. Koristeći skeletalni oblik (engl. *Skeletal Mesh*), komponentu smjera (engl. *Arrow Component*) i kamere dobijemo FPS (engl. *First Person Shooter*) izgled igre.



Slika 12: *Blueprint* igrača

```
void APlayerCharacter::SetupCharacterMesh()
{
	CharacterMesh = 
CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("CharacterMesh"));
	CharacterMesh->SetupAttachment(Camera);

	static ConstructorHelpers::FObjectFinder<USkeletalMesh>
CharacterMeshAsset(TEXT("SkeletalMesh'/Game/Meshes/arm.arm'"));
	if (CharacterMeshAsset.Succeeded())
	{
		CharacterMesh->SetSkeletalMesh(CharacterMeshAsset.Object);
	}
}
```

Ispis 5: Namještanje oblika igrača

3.2 Klasa neprijatelja

Klasa `ABaseEnemy` nasljeđuje klasu `ACharacter`, što joj omogućuje korištenje osnovnih funkcionalnosti likova unutar igre, uključujući pokrete i interakcije s okolinom. Osim standardnih varijabli `health`, `attackDamage` i `attackRange` klasa sadrži i varijablu klase `AAIController` s pomoću kojeg se neprijatelji orijentiranju.

3.2.1 Kretanje i borba

Za funkcionalnost pronalaženja puta (engl. *Pathfinding*) i napadanja neprijatelja koriste se funkcije `Tick`, `MoveToTarget` i `NormalAttack`, koje su prikazane u ispisu 6. Funkcija `Tick` se poziva svakog okvira (engl. *Frame*) igre, što omogućuje stalno ažuriranje ponašanja neprijatelja. U svakom ciklusu neprijatelj provjerava udaljenost do igrača i odlučuje hoće li se približiti ili napasti. Unutar funkcije `Tick` koristi se funkcija `FVector::Dist` kako bi se izračunala udaljenost između neprijatelja i igrača. Ako je ta udaljenost unutar dometa napada, neprijatelj pokreće napad (`NormalAttack`). Ako je igrač izvan dometa napada, neprijatelj koristi funkciju `MoveToTarget` da se približi igraču. Kada je neprijatelj dovoljno blizu igraču (`attackRange`), aktivira se funkcija `NormalAttack` koja omogućuje neprijatelju da napadne igrača. Nakon što neprijatelj izvrši napad, funkcija `NormalAttack` postavlja varijable `isAttacking` na *true* i `canAttack` na *false* kako bi se spriječilo stalno napadanje. Zatim, pomoću tajmera, ponovno omogućuje napad nakon određenog vremenskog intervala, resetirajući animaciju napada. Također, oduzima `health` igrača proporcionalno zadanoj vrijednosti štete (`attackDamage`), što je osnovna mehanika borbe između igrača i neprijatelja.

```
void AMeleeEnemy::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    APlayerCharacter* PlayerCharacter = Cast<APlayerCharacter>(UGamep-
layStatics::GetPlayerCharacter(GetWorld(), 0));
    if (PlayerCharacter)
    {
        float DistanceToPlayer = FVector::Dist(GetActorLocation(),
PlayerCharacter->GetActorLocation());
        if (DistanceToPlayer <= attackRange && canAttack)
        {
            NormalAttack();
        }
        else
        {
            MoveToTarget(PlayerCharacter);
        }
    }
}

void ABaseEnemy::MoveToTarget(APlayerCharacter* Target)
{
      if (AIController && Target)
      {
            AIController->MoveToActor(Cast<AActor>(Target));
      }
}

void AMeleeEnemy::NormalAttack()
{
    isAttacking = true;
    canAttack = false;

    APlayerCharacter* PlayerCharacter = Cast<APlayerCharacter>(UGamep-
layStatics::GetPlayerCharacter(GetWorld(), 0));
    if (PlayerCharacter)
    {
        PlayerCharacter->TakeDamage(attackDamage);
    }

    GetWorld()->GetTimerManager().SetTimer(AttackTimerHandle, this, &AMe-
leeEnemy::ResetAttackAnim, 0.75f, false);
}
```

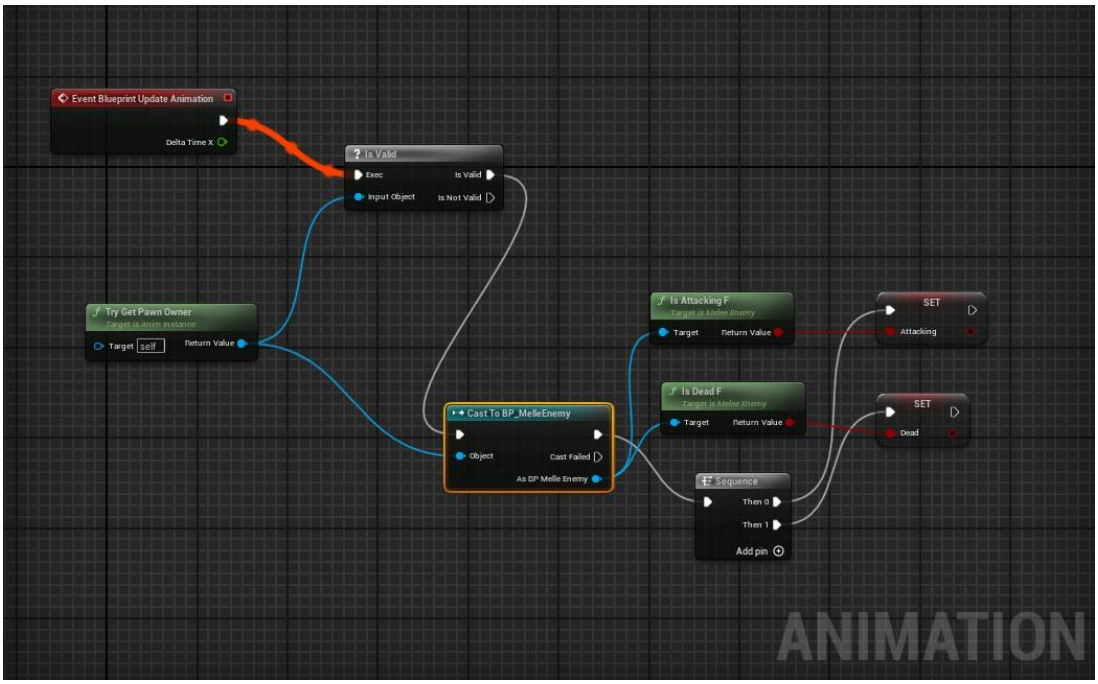
Ispis 6: Kretanje i napad neprijatelja

3.2.2 Animacije neprijatelja

Animacije neprijatelja koriste *blueprint* animacije, prikazane na slici 13, i uređaj stanja, prikazan na slici 14, u kombinaciji s varijablom `isAttacking` u C++ kôdu kako bi se

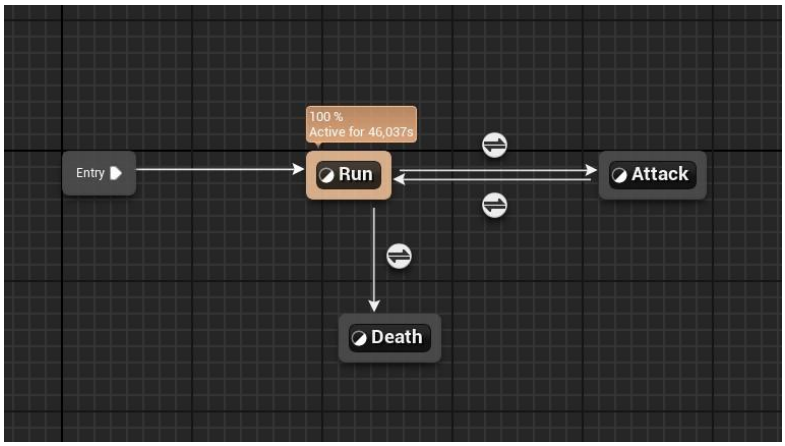
animacije napada sinkronizirale s ponašanjem neprijatelja. Ova integracija omogućuje fluidne i realistično sinkronizirane animacije koje se pokreću na temelju logike iz C++ kôda.

Blueprint stalno provjerava vrijednosti C++ varijabli `isAttacking` i `isDead` i prema njima mijenja vrijednosti varijabli `Attacking` i `Dead` unutar *blueprinta*.



Slika 13: *Blueprint* animacije neprijatelja

Uređaj stanja prikazuje osnovna stanja kroz koja neprijatelj prolazi u igri i tranzicije između tih stanja. Tri glavna stanja su `Run`, `Attack` i `Death`. Stanje `Run` je bazno stanje pješaka neprijatelja (engl. *Melee Enemy*).



Slika 14: Uređaj stanja neprijatelja

3.2.3 Vrste neprijatelja

Osim neprijatelja pješaka postoje i neprijatelji na daljinu (engl. *Ranged Enemy*) i glavni neprijatelji (engl. *Boss*). Glavna funkcionalnost neprijatelja na daljinu je ispaljivanje projektila prema igraču. Funkcija `ShootProjectile` je prikazana u ispisu 7 i ona koristi Unrealov sustav projektila za stvaranje i ispaljivanje projektila prema igraču s većih distanci.

```
void ARangedEnemy::ShootProjectile()
{
	if (ProjectileClass)
	{
		FVector MuzzleLocation = 
EnemyMesh>GetSocketLocation(FName("Muzzle"));
		FRotator MuzzleRotation = GetActorRotation();

		FActorSpawnParameters SpawnParams;
		SpawnParams.Owner = this;
		SpawnParams.Instigator = GetInstigator();

		ARangedEnemyProjectile* Arrow = GetWorld()-
>SpawnActor<ARangedEnemyProjectile>(ProjectileClass, MuzzleLocation, 
MuzzleRotation, SpawnParams);
		if (Arrow)
		{
			Arrow->SetDamage(attackDamage);
		APlayerController* PlayerController = 
UGameplayStatics::GetPlayerController(GetWorld(), 0);
			if (PlayerController)
			{
				AActor* PlayerActor = PlayerController-
>GetPawn();
				if (PlayerActor)
				{
					FVector LaunchDirection = (PlayerActor-
>GetActorLocation() - MuzzleLocation).GetSafeNormal();
					Arrow->ProjectileMovement->Velocity = 
LaunchDirection * Arrow->ProjectileMovement->InitialSpeed;
				}
			}
		}
	}
}
```

Ispis 7: Pucanje neprijatelja

Glavni neprijatelji u igrama predstavljaju izazovnije protivnike s većim brojem napada i specijalnim sposobnostima. Klasa `ABossEnemy` dodaje nekoliko složenih mehanika koje se razlikuju od standardnih neprijatelja. Osim što ima snažnije predispozicije, ima i dvije specijalne sposobnosti. Prava je AOE napad (engl. *Area of Effect*), odnosno napad šireg područja. Regularni napad ošteti igrača kada je igrač unutra neprijateljevog `attackRange`-

a, no AOE napad se aktivira nakon što prođe određen broj sekundi i ošteti igrača ako je unutar radijusa koji je predefiniran. Implementacija AOE napada je izvedena u funkciji `PerformAOEAttack`, koja je vidljiva u ispisu 8.

```
void ABossEnemy::PerformAOEAttack()
{
	isDoingAOE = true;

	APlayerCharacter* PlayerCharacter =
Cast<APlayerCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(),
0));
	if (PlayerCharacter)
	{
		float Distance = FVector::Dist(GetActorLocation(),
PlayerCharacter->GetActorLocation());
		if (Distance <= AOEAttackRadius && canAttack)
		{
			PlayerCharacter->TakeDamage(AOEDamage);
		}
	}

	GetWorld()->GetTimerManager().SetTimer(AOETimerHandle, this,
&ABossEnemy::ResetAOEAnim, 2.25f, false);
}
```

Ispis 8: AOE napad glavnog neprijatelja

U ispisu 9 prikazana je druga specijalna sposobnost glavnog neprijatelja, mehanika bijesa (engl. *Enrage Mechanic*). Nakon što glavni neprijatelj padne ispod 50% `health`-a, `movementSpeed` i `attackDamage` karakteristike su unaprijeđene kako bi otežale borbu.

```
void ABossEnemy::Enrage()
{
	isEnraged = true;
	attackDamge *= 1.5f;
	movementSpeed *= 1.5f;
	EnemyMovement->MaxWalkSpeed = movementSpeed;
}
```

Ispis 9: Mehanika bijesa

3.3 Menadžer igre

Klasa `AGameManager` predstavlja centralnu točku upravljanja logikom igre te uključuje kreiranje, upravljanje i uništavanje elemenata igre. Ova klasa nadgleda napredak kroz razine, upravlja efektima igrača, rukovodi generiranjem svijeta te nadgleda neprijatelje, poboljšanja i ekonomiju igre.

3.3.1 Menadžment neprijatelja

Klasa `AGameManager` je zadužena za dodavanje neprijatelja pješaka putem funkcije `SpawnEnemies` i neprijatelja na daljinu putem funkcije `SpawnRangedEnemies`. Implementacije funkcija je moguće vidjeti na ispisima 10 i 11. Broj neprijatelja koji se dodaju varira, ovisno o trenutnoj razini, što igru čini težom svaku novu razinu.

```
void AGameManager::SpawnEnemies()
{
	if (EnemyClass)
	{
		int32 NumberOfEnemies = FMath::RandRange(CurrentLevel + 1,
CurrentLevel * 2);

		for (int32 i = 0; i < NumberOfEnemies; i++)
		{
			FVector SpawnLocation = GetRandomSpawnLocation();
			GetWorld()->SpawnActor<ABaseEnemy>(EnemyClass,
SpawnLocation, FRotator::ZeroRotator);
		}

		EnemiesRemaining = NumberOfEnemies;
	}
}
```

Ispis 10: Funkcija za dodavanje neprijatelja pješaka

```
void AGameManager::SpawnRangedEnemies()
{
	if (WorldGenerator && RangedEnemyBlueprint)
	{
		for (AActor* TowerTile : WorldGenerator-
>GetSpawnedTowerTiles())
		{
			if (TowerTile)
			{
				UStaticMeshComponent* MeshComponent = TowerTile-
>FindComponentByClass<UStaticMeshComponent>();
				if (MeshComponent)
				{
					FVector SocketLocation = MeshComponent-
>GetSocketLocation(FName("RangedEnemy"));
					GetWorld()-
>SpawnActor<ABaseEnemy>(RangedEnemyBlueprint, SocketLocation,
FRotator::ZeroRotator);
					EnemiesRemaining++;
				}
			}
		}
	}
}
```

Ispis 11: Funkcija za dodavanje neprijatelja na daljinu

3.3.2 Menadžment razina

Na početku igre, u funkciji `BeginPlay`, koristi se objekt `WorldGenerator` za generiranje razine (ispis 12). Funkcija `StartLevel` (ispis 13) provjerava mogućnost primjene slučajnih efekata na igrača putem funkcije `EffectCheck`. Ako igrač pobijedi sve neprijatelje, trenutna razina se uništava i nova se stvara.

```
void AGameManager::BeginPlay()
{
	Super::BeginPlay();

	WorldGenerator =
Cast<AWorldGenerator>(UGameplayStatics::GetActorOfClass(GetWorld(),
AWorldGenerator::StaticClass()));
	if (WorldGenerator)
	{
		WorldGenerator->GenerateLevel();
	}

	StartLevel();
}
```

Ispis 12: Funkcija generiranja razine

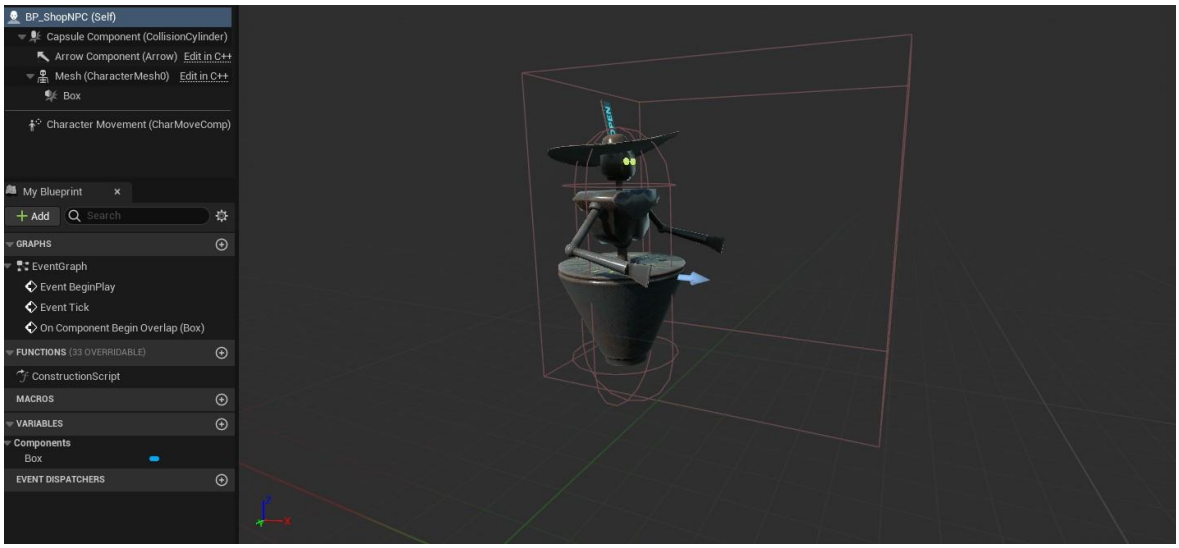
Sustav efekata omogućuje različite pozitivne (`buff`) ili negativne (`debuff`) efekte na igrača kao što su `StrengthBuff` ili `HealthDebuff`. Ovi efekti se mogu nasumično aktivirati pri početku razine pomoću funkcije `ApplyEffect`.

```
void AGameManager::StartLevel()
{
	if (EffectCheck())
	{
		EEffect RandomEffect =
static_cast<EEffect>(FMath::RandRange(0,
static_cast<int32>(EEffect::StrengthDebuff)));

		switch (RandomEffect)
		{
		case EEffect::StrengthBuff:
			isStrengthBuff = true;
			break;
		case EEffect::StrengthDebuff:
			isStrengthDebuff = true;
			break;
		case EEffect::DexterityBuff:
			isDexterityBuff = true;
			break;
		case EEffect::DexterityDebuff:
			isDexterityDebuff = true;
			break;
		case EEffect::HealthBuff:
			isHealthBuff = true;
			break;
		case EEffect::HealthDebuff:
			isHealthDebuff = true;
			break;
		default:
			break;
		}

		ApplyEffect(RandomEffect);
	}

	if (CurrentLevel % 3 == 0)
	{
		SpawnBoss();
		SpawnEnemies();
		SpawnRangedEnemies();
	}
	else
	{
		SpawnEnemies();
		SpawnRangedEnemies();
	}
}
```

Ispis 13: Funkcija početka razine

3.3.3 Menadžment poboljšanja igrača, ekonomije i NPC-a

Nakon svake dovršene razine postoji šansa da će se stvoriti NPC za dućan ili NPC za poboljšanja. Oboje rade na isti princip da se na ekranu prikaže HUD (engl. *Heads Up Display*) koji prikazuje tri opcije poboljšanja. Ključna razlika je u tome što NPC za dućan traži novac za kupnju poboljšanja, dok NPC za poboljšanja nudi opcije besplatno. Novac se zarađuje za svakog ubijenog neprijatelja. Kako bi se interakcija s NPC-em dogodila potrebno je ući u NPC-evu sudarnu kutiju (engl. *Box Collision*). Izgled NPC-a i njegove kutije sudara može se vidjeti na slici 15.



Slika 15: NPC dućana i njegova kutija sudara

Klasa `AGameManager` upravlja sustavom poboljšanja igrača uz pomoć funkcije `RandomizeUpgradeOptions` koja nasumično odabire poboljšanja iz enum-a `EUpgrade`. Funkcija i enum su prikazani u ispisu 14. Kad igrač odabere poboljšanje, ono se primjenjuje pomoću funkcije `ApplyUpgrade`, koja povećava statuse poput zdravlja, snage ili kritičnog udarca.

```
UENUM(BlueprintType)
enum class EUpgrade : uint8
{
	HealthUpgrade,
	DexterityUpgrade,
	StrengthUpgrade,
	CritDamageUpgrade,
	CritChanceUpgrade,
	AbilityCooldownUpgrade,

	MAX
};

TArray<EUpgrade> AGameManager::RandomizeUpgradeOptions()
{
	isActive = true;
	const int32 NumOptions = 3;
	TArray<EUpgrade> ChosenOptions;

	for (int32 i = 0; i < NumOptions; ++i)
	{
		EUpgrade Option = SelectRandomUpgrade();
		ChosenOptions.Add(Option);
	}

	return ChosenOptions;
}
```

Ispis 14: Poboljšanja igrača

3.4 Razine

Klasa `AWorldGenerator` u Unreal Engineu odgovorna je za proceduralnu generaciju razina u igri. Osnovna funkcionalnost ove klase obuhvaća stvaranje terena, kula i zamki na heksagonalnoj mreži korištenjem Perlinovog šuma (engl. *Perlin Nois*) za varijacije u visini terena i proceduralno generiranje položaja objekata.

Unutar funkcije `BeginPlay` varijabla `Seed` se nasumično generira pomoću funkcije `FMath::Rand`, čime se osigurava da svaka generirana razina bude unikatna. Ova sjemenska vrijednost koristi se pri generiranju Perlinovog šuma za pozicioniranje ploča u svijetu.

U ispisu 15 je prikazana funkcija `GenerateLevel` koja je odgovorna za generiranje cijele razine. Razina se generira na heksagonalnoj mreži pomoću dvostruke petlje koja prolazi kroz sve moguće koordinate unutar radijusa razine (`LevelRadius`). Funkcija `GetHexWorldPosition` koristi Perlinov šum za određivanje Z-pozicije svake ploče, što stvara varijacije u visini terena, dok X i Y koordinate određuju položaj svake ploče unutar

mreže. Na svakoj generiranoj ploči može se nalaziti zamka ili kula, ovisno o proceduralno generiranim vrijednostima i postavljenim ograničenjima (`MaxTraps`, `MaxTowers`).

```
void AWorldGenerator::GenerateLevel()
{
    SpawnedGroundTiles.Empty();
    SpawnedTowerTiles.Empty();
    SpawnedGroundTrapTiles.Empty();
    SpawnableLocations.Empty();

    int32 TrapCount = 0;

    for (int32 Q = -LevelRadius; Q <= LevelRadius; ++Q)
    {
        int32 R1 = FMath::Max(-LevelRadius, -Q - LevelRadius);
        int32 R2 = FMath::Min(LevelRadius, -Q + LevelRadius);
        for (int32 R = R1; R <= R2; ++R)
        {
            FVector TileLocation = GetHexWorldPosition(Q, R);

            PlaceGroundTile(TileLocation);
            SpawnableLocations.Add(TileLocation);

            float NoiseValue = GetPerlinNoiseValue(Q, R);

            if (TrapCount < MaxTraps && FMath::FRand() <= 0.1f)
            {
                PlaceGroundTrapTile(TileLocation);
                TrapCount++;
            }
        }
    }

    for (int32 i = 0; i < MaxTowers; ++i)
    {
        int32 Q = FMath::RandRange(-LevelRadius, LevelRadius);
        int32 R = FMath::RandRange(FMath::Max(-LevelRadius, -Q -
LevelRadius), FMath::Min(LevelRadius, -Q + LevelRadius));
        FVector TowerLocation = GetHexWorldPosition(Q, R);
        PlaceTowerTile(TowerLocation);
    }
}
```

Ispis 15: Generiranje razine

Funkcija `GetHexWorldPosition` koristi matematičku formulu za pozicioniranje šesterokutnih ploča na mreži. X i Y koordinate određuju položaj ploče u ravnini koristeći geometriju šesterokuta. Visina Z se generira pomoću Perlinovog šuma koji stvara glatke varijacije u terenu, a one se skaliraju pomoću faktora `HeightScale` kako bi se postigla željena visinska raznolikost terena. Implementacija funkcije `GetHexWorldPosition` je vidljiva u ispisu 16.

Perlinov šum je algoritam koji generira glatke, prirodne varijacije koje nalikuju teksturama pronađenim u stvarnom svijetu, poput oblaka, terena i vodenih površina. Posebno je koristan u računalnoj grafici, proceduralnom generiranju terena, animacijama i simulacijama prirodnih fenomena.

```
FVector AWorldGenerator::GetHexWorldPosition(int32 Q, int32 R)
{
    float X = HexSize * (3.0f / 2.0f) * Q;
    float Y = HexSize * FMath::Sqrt(3.0f) * (R + Q / 2.0f);
    float NoiseValue = FMath::PerlinNoise2D(FVector2D((Q + Seed) *
NoiseScale, (R + Seed) * NoiseScale));
    float Z = NoiseValue * HeightScale;

    return FVector(X, Y, Z);
}
```

Ispis 16: Određivanje koordinata heksagonalnih ploča

3.5 Oružja i projektili

Oružja i projektili u videoigrama često igraju ključnu ulogu u oblikovanju iskustva na način da igračima nude različite mogućnosti za interakciju s okolinom i neprijateljima. U ovom kontekstu, klasa `ABaseWeapon` predstavlja osnovni okvir za sve vrste oružja, dok klasa `AWeaponProjectile` opisuje specifično ponašanje projektila koji su ispaljeni iz tih oružja.

Klasa `ABaseWeapon` definirana je kao osnovna klasa za sva oružja u igri koja nasljeđuje svojstva iz klase `AActor`, što znači da se može pojaviti u svijetu igre i može komunicirati s ostalim objektima u igri.

Funkcija `UseWeapon` (ispis 17) je virtualna funkcija koja mora biti implementirana u naslijeđenim klasama. Ova funkcija definira konkretno ponašanje oružja, primjerice kako oružje puca ili napada. Budući da je deklarirana kao `PURE_VIRTUAL`, to znači da se ne može direktno koristiti u ovoj osnovnoj klasi, već ju treba nadjačati u izvedenim klasama specifičnih oružja.

```
void ASpellbookWeapon::UseWeapon()
{
	FVector CameraLocation;
	FRotator CameraRotation;
	GetActorEyesViewPoint(CameraLocation, CameraRotation);

	ProjectileSpawnOffset.Set(100.0f, 0.0f, 0.0f);

	FVector ProjectileSpawnLocation = CameraLocation +
FTransform(CameraRotation).TransformVector(ProjectileSpawnOffset);
	FRotator ProjectileSpawnRotation = CameraRotation;

	UWorld* World = GetWorld();
	if (World)
	{
		APlayerCharacter* PlayerCharacter =
Cast<APlayerCharacter>(GetOwner());
		if (PlayerCharacter)
		{
			FActorSpawnParameters SpawnParams;
			SpawnParams.Owner = this;
			SpawnParams.Instigator = GetInstigator();

			AWeaponProjectile* Projectile = World-
>SpawnActor<AWeaponProjectile>(ProjectileClass, ProjectileSpawnLocation,
ProjectileSpawnRotation, SpawnParams);
			if (Projectile)
			{
				Projectile->SetDamage(power * PlayerCharacter-
>GetAttackDamage());
				FVector LaunchDirection =
ProjectileSpawnRotation.Vector();
				Projectile->ProjectileMovement->Velocity =
LaunchDirection * Projectile->ProjectileMovement->InitialSpeed;
			}
		}
	}
}
```

Ispis 17: Pucanje oružja

Klasa `AWeaponProjectile` predstavlja projektil koji oružje može ispaliti. Kao i oružje i projektil je naslijeđen iz klase `AActor`, što znači da je i on fizički objekt u igri. Ova klasa definira specifično ponašanje projektila, uključujući njegovu brzinu, kretanje i interakciju s drugim objektima.

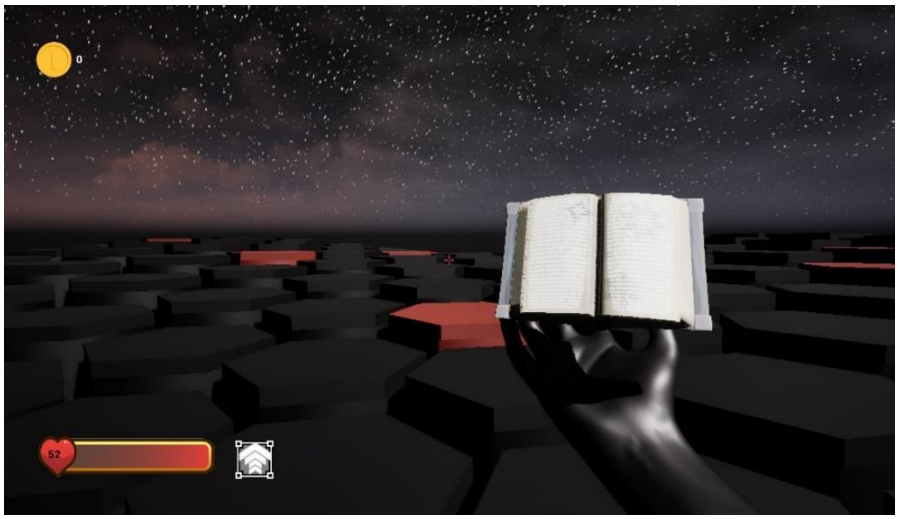
Kada projektil udari u neki objekt u igri, poziva se funkcija `OnHit`. Ova funkcija detektira koji je objekt pogođen i ako je pogođen neprijateljski objekt (`ABaseEnemy`), nanosi štetu tom objektu koristeći izračun iz varijable `damage`. Također, funkcija uključuje provjeru za kritični udarac, gdje ako igrač ima određenu šansu za kritični pogodak (`critChance`), šteta može biti povećana. Funkcija je vidljiva u ispisu 18.

```
void AWeaponProjectile::OnHit(UPrimitiveComponent* HitComp, AActor*
OtherActor, UPrimitiveComponent* OtherComp, FVector NormalImpulse, const
FHitResult& Hit)
{
	if (OtherActor && OtherActor != this && OtherComp)
	{
		ABaseEnemy* HitEnemy = Cast<ABaseEnemy>(OtherActor);
		if (HitEnemy)
		{
			if (FMath::FRand() <= PlayerCharacter->critChance)
			{
				damage *= PlayerCharacter->crit;
			}
			HitEnemy->TakeDamage(damage);
		}

		Destroy();
	}
}
```

Ispis 18: Detekcija hitca

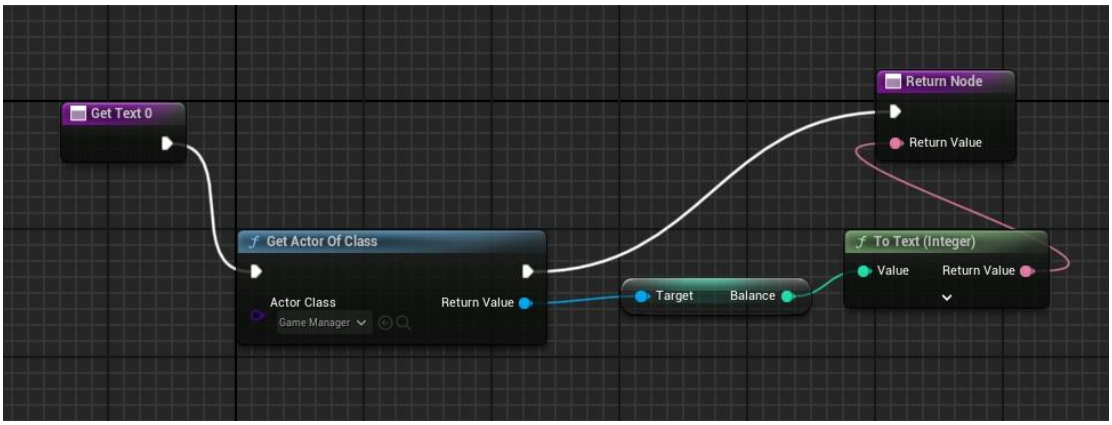
3.6 HUD

HUD (engl. Head-Up Display) je vrsta korisničkog sučelja koje prikazuje važne informacije igračima tijekom igranja. Tipično uključuje elemente kao što su brojači novca, životne trake, indikatori sposobnosti ili oružja te druge informacije koje pomažu igraču. Glavni HUD, prikazan na slici 16, je napravljen pomoću *blueprinta* grafičkih elemenata (engl. *Widget Blueprint*). On omogućuje igračima interakciju s različitim informacijama i elementima korisničkog sučelja na intuitivan način. Sastoji se od od brojača novca, životne trake (engl. *Health Bar*) i indikatora specijalne sposobnosti.



Slika 16: Glavni HUD

Vrijednosti i statusi potrebni za ažuriranje HUD-a se prate unutar *blueprinta* grafičkog elementa (slika 17). Svaki element koji zahtijeva konstantno ažuriranje ima jedan takav *blueprint* kako bi precizno i efikasno pratili vrijednosti i statuse varijabli.



Slika 17: *Blueprint* grafičkog elementa

4. Zaključak

Izradom ovog rada prikazana je snaga Unreal Enginea kao moćnog alata za razvoj igara, posebno u kontekstu akcijskih *roguelike* igara. Unreal Engine omogućuje izradu složenih mehanika i visokokvalitetne grafike kroz jednostavan rad sa sustavom *blueprinta* i integraciju s C++ kôdom. Projektom se uspješno demonstriralo kako se temelji Unreal Enginea mogu koristiti za kreiranje dinamičnog igračkog iskustva, uz fokus na proceduralno generirane nivoe, napredne AI protivnike te sustave borbe koji su prilagođeni žanru.

Iako je ovaj projekt predstavio osnovne koncepte igre, postoji velik potencijal za proširenje i dodatne optimizacije. To uključuje razvoj složenijih sustava napredovanja, dodavanje različitih tipova neprijatelja te prilagodbu težine igre prema napretku igrača. Unreal Engine pruža mnoge alate za daljnje unapređenje projekta, uključujući sustave za optimizaciju performansi, fiziku i animacije.

Valja naglasiti kako je projekt osmišljen tako da omogućuje jednostavno ponovno korištenje većine komponenti u budućim projektima, što je ključna prednost korištenja modularnog pristupa u Unreal Engineu.

Zaključno, projektom se demonstrirala primjena modernih tehnologija za stvaranje zabavnog i izazovnog igračkog iskustva u okviru akcijskih *roguelike* igara.

5. Literatura

[1] „Unreal Engine 5.3 Documentation“

`https://dev.epicgames.com/documentation/en-us/unreal-engine` (posjećeno 15.8.2024.).

[2] „Perlinov šum“

`https://en.wikipedia.org/wiki/Perlin_noise` (posjećeno 16.8.2024.).