SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Naziv studija

Jerko Bućan

Z A V R Š N I R A D

Izrada 3D videoigre pomoću Unreal Engine 5

Split, rujan 2023

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE
Preddiplomski stručni studij Naziv studija

Predmet: Objektno programiranje

Z A V R Š N I R A D

Kandidat: Jerko Bućan

Naslov rada: Izrada 3D videoigre pomoću Unreal Enginea 5

Mentor: Ljiljana Despalatović

Split, rujan 2023.

SADRŽAJ

SAŽETAK

Sažetak završnog rada pod nazivom "Izrada 3D videoigre u Unreal Engineu 5" obuhvaća cjelokupno istraživanje, razvoj i implementaciju 3D puzzle igre koristeći Unreal Engine 5. U radu se detaljno opisuju ciljevi i svrha igre, osnovni pregled strukture rada te ključni aspekti istraživanja i razvojnog procesa.

Rad se temelji na primjeni objektno orijentiranog programiranja i vizualnog programiranja kroz nacrtni sustav, kao i korištenju C++ jezika za razvoj kompleksnih komponenti igre. Unreal Engine 5 je odabran kao glavni alat za razvoj igre zbog svoje moćne i napredne funkcionalnosti, podrške za različite platforme te mogućnosti integracije s raznim alatima.

U radu se detaljno opisuje kreiranje projekta, razvoj okoline i karaktera, implementacija igračkih mehanika i interakcija, te dodavanje novih komponenti i zagonetki na višim razinama igre. Također, opisuju se inovativni sustavi i rješenja koji su razvijeni unutar igre kako bi se pružilo jedinstveno i zanimljivo iskustvo igračima.

Istraživanje obuhvaća i testiranje igre kako bi se otkrile i ispravile greške te poboljšala ukupna igračka izvedba. Kroz iterativan proces razvoja, tim je uspio postići visoku kvalitetu igre koja pruža uzbudljivo iskustvo igračima.

Kroz ovaj završni rad, čitatelji će steći dublje razumijevanje razvojnog procesa 3D igre u Unreal Engineu 5 te otkriti ključne aspekte objektno orijentiranog programiranja, vizualnog programiranja i primjene C++ jezika u stvaranju igara. Također, rad će istaknuti važnost Unreal Enginea 5 kao snažnog i fleksibilnog alata za razvoj visokokvalitetnih 3D igara.

Ključne riječi: 3D puzzle igra, Unreal Engine 5, C++, vizualno skriptiranje.

SUMMARY

The summary of the final paper entitled " Creating a 3D video game in Unreal Engine 5" includes the entire research, development and implementation of a 3D puzzle game using Unreal Engine 5. The paper describes in detail the goals and purpose of the game, a basic overview of the work structure and key aspects of the research and development process.

The work is based on the application of object-oriented programming and visual programming through the Blueprint system, as well as the use of the C++ language for the development of complex game components. Unreal Engine 5 was chosen as the main game development tool due to its powerful and advanced functionality, cross-platform support and ability to integrate with various tools.

The paper describes in detail the creation of the project, the development of the environment and characters, the implementation of game mechanics and interactions, and the addition of new components and puzzles at higher levels of the game. It also describes innovative systems and solutions that have been developed within the game to provide a unique and interesting experience to players.

Research also includes game testing to detect and correct bugs and improve overall game performance. Through an iterative development process, the team was able to achieve a high quality game that provides an exciting experience for players.

Through this thesis, readers will gain a deeper understanding of the 3D game development process in Unreal Engine 5 and discover key aspects of object-oriented programming, visual programming, and the application of the C++ language in game creation. Also, the paper will highlight the importance of Unreal Engine 5 as a powerful and flexible tool for developing high-quality 3D games.

Keywords: 3D puzzle game, Unreal Engine 5, C++, visual scripting.

1. UVOD

Razvoj video igara predstavlja intrigantno polje koje se neprestano širi i evoluira zahvaljujući tehnološkom napretku i rastućem interesu igrača diljem svijeta. Unutar ovog kreativnog okruženja, Unreal Engine 5 se izdvaja kao jedan od najnaprednijih i najmoćnijih alata za stvaranje 3D igara. Ovaj završni rad pruža uvid u proces razvoja igre koristeći Unreal Engine 5, istražujući specifičnosti, metode i alate koji su korišteni u tom procesu.

Cilj ovog rada bio je istražiti mogućnosti Unreal Enginea 5 i stvoriti 3D igru koja će pružiti jedinstveno i zadovoljavajuće iskustvo igračima. Ova igra je razvijena s naglaskom na raznovrsnost razina i mehanika, vizualne efekte i pažljivo osmišljen zvučni dizajn kako bi se stvorila imerzivna igraća okolina. Istovremeno, kroz razvoj ove igre, nastojalo se razumjeti izazove s kojima se susreću razvojni timovi te pronaći optimalna rješenja za njihovo prevladavanje.

U ovom uvodu, definirat ćemo ciljeve i svrhu ove igre, prikazati osnovni pregled strukture rada te navesti ključne aspekte istraživanja i razvojnog procesa. Također, istaknut ćemo važnost Unreal Enginea 5 kao glavnog alata za razvoj igre i razloge zbog kojih smo ga odabrali.

Sljedeće poglavlje će se usredotočiti na specifikaciju igre, opisivanje detaljne vizije igre, njenih glavnih mehanika i ciljane publike. Zatim će slijediti poglavlje o metodama i alatima korištenima tijekom razvojnog procesa, gdje ćemo raspravljati o ključnim koracima u stvaranju igre i kako su odabrani alati doprinijeli uspješnom ostvarenju ciljeva.

Ovaj završni rad donosi cjelovit prikaz procesa razvoja 3D igre u Unreal Engineu 5, nadamo se da će pružiti korisne uvide i biti inspiracija za buduće istraživanje i razvoj u području razvoja video igara.

2. SPECIFIKACIJA IGRE

Žanr igre: 3D puzzle avantura

Ciljana platforma: PC

Ciljna publika: Igrači svih dobnih skupina koji uživaju u izazovnim mozgalicama i avanturama.

Opis igre

Puzzle igra koja stavlja igrača u šareni i čarobni svijet ispunjen zagonetkama. Igrač će preuzeti ulogu hrabrog istraživača koji se suočava s nizom izazova dok putuje kroz različite razine, svaka teža od prethodne. Svaka razina u igri predstavlja jedinstvenu zagonetku koju igrač mora riješiti kako bi napredovao na sljedeću razinu.

Glavne mehanike igre

1. **Sakupljanje predmeta:** Glavni cilj svake razine je sakupiti svo zlato kako bi se otključao put prema sljedećoj razini. Igrač mora pažljivo istraživati okolinu i koristiti logično razmišljanje.
2. **Puzzle mehanike:** Svaka druga razina donosi nove puzzle mehanike kojima igrač mora svladati kako bi riješio zagonetku i prikupio potrebne predmete. Mehanike mogu uključivati rotiranje platformi, gumbima za upravljanje platformi te lokoti s ključevima.
3. **Istraživanje i avantura:** Igrač će se suočiti s raznolikim izazovima u svakoj razini, što će zahtijevati istraživanje okoline i rješavanje logičkih enigmi.

Napredovanje kroz igru

Igrač će započeti na početnoj razini koja će služiti kako bi se upoznao s osnovnim mehanikama igre. Nakon toga, igrač će prelaziti kroz različite razine, od kojih će svaka biti teža od prethodne. Svaka druga razina donosi nove puzzle mehanike kako bi se osvježilo iskustvo i izazvalo igrača na nov način.

Vizualni stil

Grafički stil igre bit će šaren i privlačan, s prekrasnim 3D okolinama i detaljno dizajniranim karakterom. Svaka razina imat će jedinstvenu temu.

Zvučni dizajn i glazba

Zvučni dizajn igre bit će pažljivo osmišljen kako bi stvorio dojmljivu atmosferu. Zvukovi okoline, zvukovi puzzle mehanika i prateća glazba doprinijet će uranjanju igrača u čarobni svijet igre.

3. METODE I ALATI

3.1. C++

C++ je objektno orijentirani i višestruko primjenjiv programski jezik koji pruža programerima snažne alate za razvoj softvera na raznim platformama. Osim što je popularan izbor za razvoj računalnih igara, C++ se koristi i u područjima poput operativnih sustava, aplikacija u stvarnom vremenu, mobilnih aplikacija, mikro kontrolera, web preglednika i još mnogo toga.

Jedna od ključnih prednosti C++-a je mogućnost kombiniranja visokog i niskog nivoa programiranja. Na visokom nivou, programeri mogu koristiti objektno orijentiranu paradigmu za organizaciju koda i olakšavanje ponovnog korištenja. S druge strane, niski nivo omogućuje pristup hardverskim resursima i upravljanje memorijom, čime se postiže veća kontrola nad performansama i resursima sistema.

C++ također omogućuje direktan rad s memorijom pomoću pokazivača, što može biti korisno za optimizaciju i pristupanje hardverskim resursima. Međutim, ta fleksibilnost nosi i odgovornost programera da pravilno upravljaju memorijom kako bi izbjegli curenje memorije ili druge greške.

Dodatno, C++ podržava operator overloading, što omogućuje definiranje specifičnih operacija za korisnički definirane tipove podataka. To olakšava rad s kompleksnim objektima i omogućuje intuitivno korištenje operatora u skladu s očekivanjima programera.

C++ također nudi bogatu standardnu biblioteku, uključujući Standard Template Library (STL), koja sadrži algoritme, kontejnere i funkcije koje olakšavaju rad s podacima, stringovima, vektorima i drugim strukturama podataka.

3.2. Unreal Engine 5

Unreal Engine je jedan od najmoćnijih i najpopularnijih game enginea na tržištu, razvijen od strane tvrtke Epic Games. Ovaj game engine koristi se za izradu raznovrsnih interaktivnih iskustava, uključujući video igre, virtualnu stvarnost (VR) i proširenu stvarnost (AR), simulacije, treninge i mnoge druge aplikacije

Razvijen je prvi put od strane Tima Sweeneya 1995. godine. Tijekom godina, engine je doživio brojne nadogradnje i evolucije. Trenutna verzija, Unreal Engine 5, najavljena je 2020. godine i donosi revolucionarne tehnologije poput Nanite i Lumen za ultra-realističan vizualni doživljaj.

Pruža dvije glavne mogućnosti za programiranje igara - vizualno programiranje putem Blueprints sistema i programiranje u C++ jeziku. Blueprints omogućuju brzo i jednostavno stvaranje igračkih mehanika bez potrebe za kodiranjem, dok programiranje u C++ pruža veću fleksibilnost i snagu za razvoj kompleksnih igara.

Unreal Engine je poznat po svojoj iznimnoj vizualnoj kvaliteti i podršci za foto realistične grafike. S tehnologijama poput fizički zasnovane renderinga (PBR), visokokvalitetnih materijala, globalnog osvjetljenja (Lumen) i naprednih efekata, Unreal Engine omogućuje stvaranje živopisnih i impresivnih svjetova.

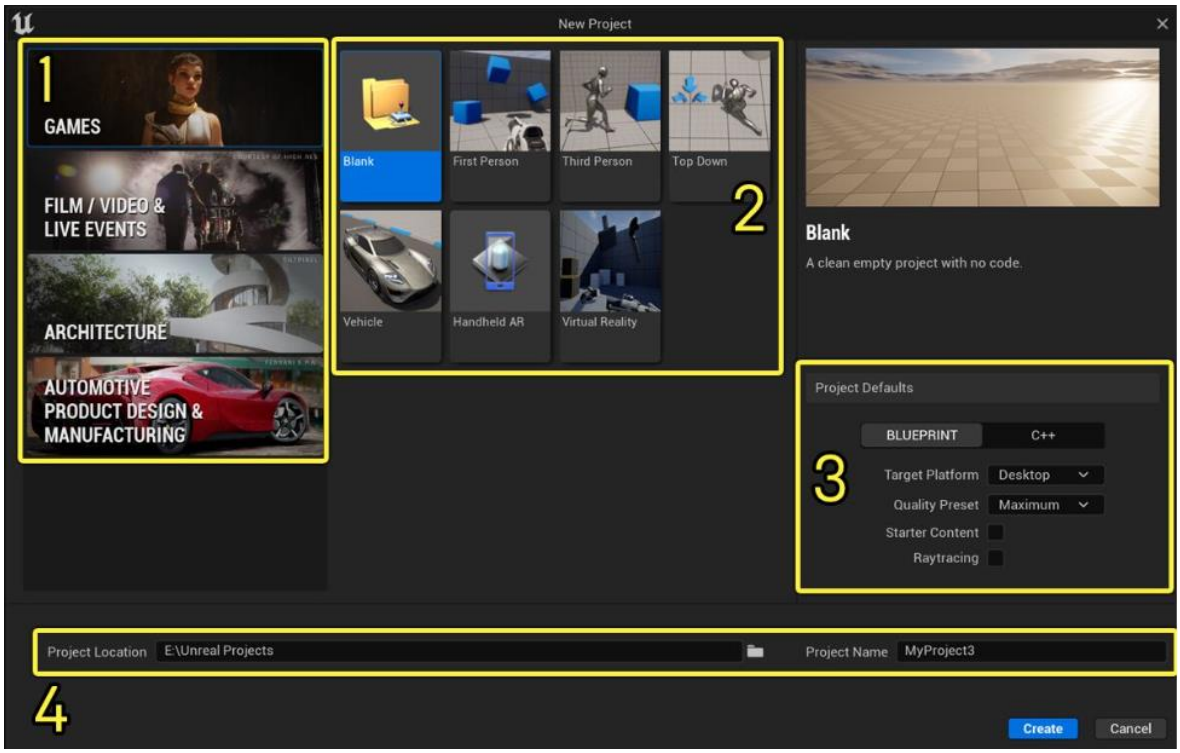
Podržava različite platforme, uključujući PC, konzole (PlayStation, Xbox), mobilne uređaje (iOS, Android), VR i AR uređaje. To omogućuje razvoj i distribuciju igara i aplikacija za široku publiku.

U ovom slučaju, koristilo se UE5 u kombinaciji s programskim jezikom C++ kako bi razvili različite klase i funkcionalnosti za igru. Evo detaljnijeg pregleda korištenjem C++ i različite klase u razvoju igre:

3.2.1. Kreiranje novog projekta

Kreiranje novog projekta u Unreal Engineu može biti jednostavan proces, a evo detaljnog opisa kako to učiniti. Ako ga već nemate instaliranog, možete ga preuzeti s službene web stranice Epic Gamesa.

Kada se program pokrene, otvara se preglednik (engl. *Unreal Project Browser*) (slika 1), možete odabrati željenu verziju Unreal Enginea (ako ih imate više instaliranih) i predložak za novi projekt. Unreal Engine nudi različite kategorije i predloške za različite vrste projekata.



Slika 1: Unreal projektni preglednik

Na lijevoj strani označeno brojem 1 se odabire kategorija, u našem slučaju igre (engl. *Games*), nakon toga prikažu se predlošci (broj 2 na slici 1) kao što su pogled iz prvog lica (engl. *First Person*), trećeg lica (engl. *Third Person*), prazan projekt (engl. *Blank*) i druge.

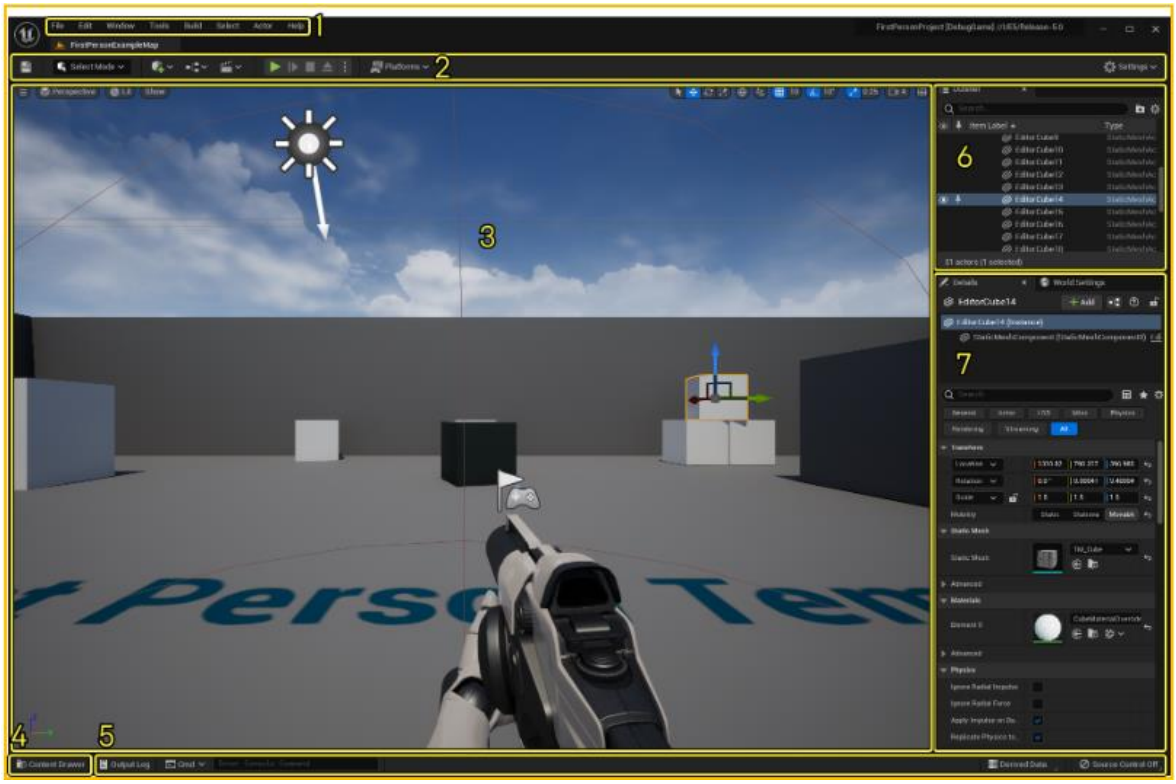
Ovdje odabiremo prazan projekt te se u zadanim postavkama projekta (engl. *Project Defaults*) (broj 3 na slici 1) omogućuje odabir ciljne platforme (engl. *Target*

Platform) za igru ili aplikaciju (tj. hardver na kojem će se izvoditi, poput računala ili mobilnog uređaja), konfiguriranje postavki kvalitete (engl. *Quality Preset*) i praćenja zraka (engl. *Ray tracing*) te pruža mnoge druge opcije za prilagodbu.

Dolje na dnu ovog preglednika (broj 4 na slici 1) se odabire gdje spremiti projekt te sam naziv projekta. Nakon što je sve odabrano od opcija, klikom na gumb stvaranje (engl. *Create*) kako započeti proces kreiranja projekta.

3.2.2. Upoznavanje sa sučeljem

Prije početka rada u Unreal Engineu, važno je upoznati se s razvojnim sučeljem. Glavni zaslon (slika 2) omogućuje prilagodbu sučelja prema vašim željama, što vam omogućuje optimalno organiziranje prozora za rad i povećava učinkovitost vašeg razvojnog procesa.



Slika 2: Glavni zaslon

Tablica 1: Objašnjenje brojeva na slici 2

Broj	Naziv	Opis
1	Traka izbornika	Izbornike za pristup naredbama i funkcionalnostima specifičnim za uređivač.
2	Alatna traka	Sadrži prečace za neke od najčešćih alata.
3	Prikaz scene	Prikazuje sadržaj razine, poput kamera, rotora, statičkih mreža i tako dalje.
4	Pregled sadržaja	Svi elementi koji se mogu koristiti unutar projekta
5	Donja traka za alate	Sadrži prečace za naredbenu konzolu, izlazni dnevnik i funkcionalnost izvedenih podataka. Također prikazuje status kontrole izvora.
6	Hijerarhijski prikaz	Prikazuje hijerarhijski prikaz stabala svih sadržaja na vašoj razini.
7	Detaljni prikaz	Prikazuje različite postavke ovisno o tome što ste odabrali u prikazu scene razine.

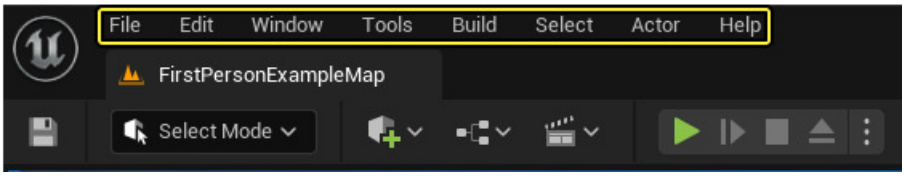
Prilagodba glavnog zaslona u Unreal Engine:

- Pomicanje i grupiranje prozora: Prozore možete pomicati i postavljati ih jedan pored drugog ili na različite strane zaslona kako biste stvorili raspored koji vam najbolje odgovara. Također, možete grupirati slične prozore zajedno koristeći tabove kako biste lakše prelazili između različitih konfiguracija.
- Dodavanje novih prozora: Unreal Engine 5 omogućuje vam dodavanje novih prozora koji odgovaraju vašim potrebama. Na primjer, možete dodati prozor za Blueprint Editor ako želite programirati igru kroz vizualno skriptiranje.
- Uklanjanje nepotrebnih prozora: Ako postoje prozori koji vam nisu potrebni, možete ih ukloniti kako biste oslobodili više prostora za prozore koji su vam važniji.
- Spremanje prilagođenih rasporeda: Kada postignete željeni raspored prozora, možete ga spremiti kao prilagođeni radni prostor kako biste ga mogli lako ponovno koristiti u budućnosti.

Prilagodba glavnog zaslona u Unreal Engine 5 omogućuje vam da stvorite radno okruženje prilagođeno vašim radnim navikama i preferencijama, čime olakšava razvojni proces i omogućuje vam da se usredotočite na kreiranje izvrsnih igara ili virtualnih iskustava.

3.2.3. Glavni prozori

Svaki urednik ima traku izbornika (slika 3) koji se nalazi u gornjem desnom prozoru tog uređivača. Neki od izbornika, kao što su Datoteka, Prozor, i Pomoć, prisutni su u svim prozorima urednika, a ne samo u uređivaču razine. Ostali su specifični za urednika.



Slika 3: Traka izbornika

Alatna traka (slika 4) sadrži prečace za neke od najčešće korištenih alata i naredbi u ovom uređivaču. Podijeljen je na sljedeća područja:

1. Gumb za spremanje razine koja je trenutno otvorena,
2. Sadrži prečace za brzo prebacivanje između različitih načina uređivanja sadržaja unutar vaše razine.
3. Sadrži prečace za dodavanje i otvaranje uobičajenih vrsta sadržaja unutar uređivača razine(nacrti, kino sekvenca,…).
4. Sadrži gumbe za prečac (igranje, preskakanje, zaustavljanje i izbacivanje) za pokretanje vaše igre u uredniku.

5. Sadrži niz opcija koje možete koristiti za konfiguriranje, pripremu i raspoređivanje projekta na različite platforme, poput radne površine, mobilne ili konzole.

6. Sadrži različite postavke za uređivača, pregled razine i ponašanje u igri.

Slika 4: Alatna traka

Donja traka za alate (slika 5) sadrži prečace do naredbene konzole, izlaz dnevnik i funkcionalnost izvedenih podataka. Također prikazuje status kontrole izvora. Podijeljen je na sljedeća područja:

1. Izlazni dnevnik za uklanjanje pogrešaka koji ispisuje korisne informacije dok se aplikacija pokreće.

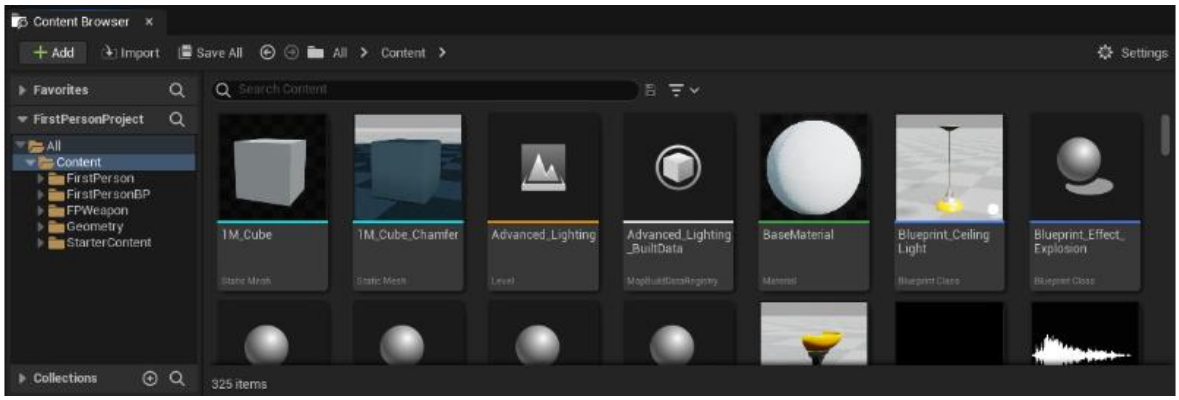
2. Naredbena konzola se ponaša kao bilo koje drugo sučelje naredbenog retka: unesite naredbe konzole kako biste pokrenuli određeno ponašanje urednika.

3. Izvedeni podaci koji pružaju funkcionalnost podataka.

4. Prikazuje status kontrole izvora ako je vaš projekt povezan s kontrolom izvora (na primjer, GitHub ili Perforce).

Slika 5: Donja traka za alate

Pregled sadržaja (slika 6) je prozor za istraživanje datoteka koji prikazuje sve elemente koji se mogu koristiti unutar projekta, uključujući modele, skripte, nacrti, audio datoteke, slike i druge resurse.



Slika 6: Prozor pregleda sadržaja

Modeli su 3D objekti koji čine sastavni dio igre. Unutar Unreal Enginea, modeli se često koriste u FBX formatu datoteka, i mogu predstavljati likove, objekte, okolinu i sve ostalo što vidimo u igri.

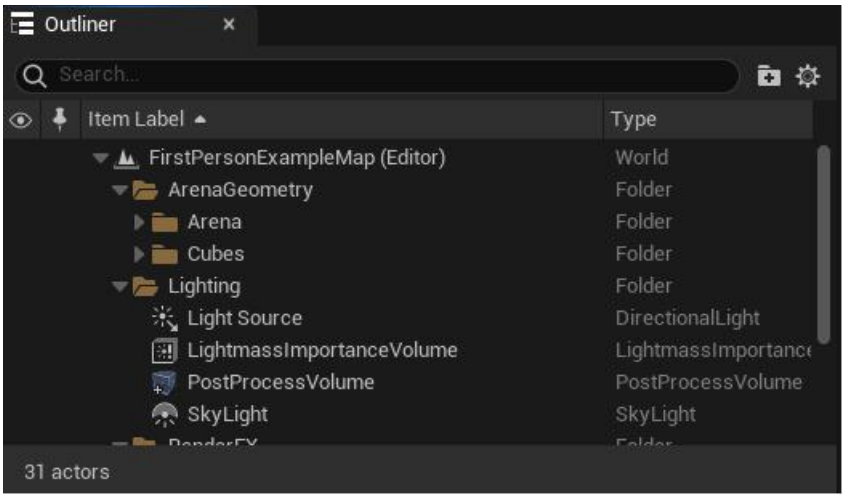
Podržava različite formate audio datoteka kao što su WAV, MP3 i OGG. Ove audio datoteke koriste se za zvukove u igri, uključujući glazbu, efekte zvuka i dijaloge likova. Slike u različitim formatima (BMP, TIF, JPG, PNG itd.) koriste se za teksture površina i grafiku u igri.

Animatori i dizajneri mogu stvarati kompleksne animacije za likove ili objekte unutar igre pomoću različitih animacijskih alata unutar Unreal Enginea. Materijali definiraju izgled površina i tekstura u igri. Koristeći grafički sučelje, moguće je stvoriti bogate i detaljne materijale koji će oplemeniti izgled igre. Također ima moćan sustav čestica koji se koristi za stvaranje impresivnih efekata, poput vatre, dima, eksplozija i magije. Nudi razne vrste svjetala i specijalnih efekata koji poboljšavaju atmosferu i vizualni dojam igre.

Nacrti su vizualni sustav skriptiranja unutar Unreal Enginea koji omogućuje izradu interaktivnosti i logike u igri bez potrebe za programiranjem. Pomoću nacrti klasa moguće je definirati nove tipove objekata i likova s unaprijed definiranim ponašanjem i karakteristikama.

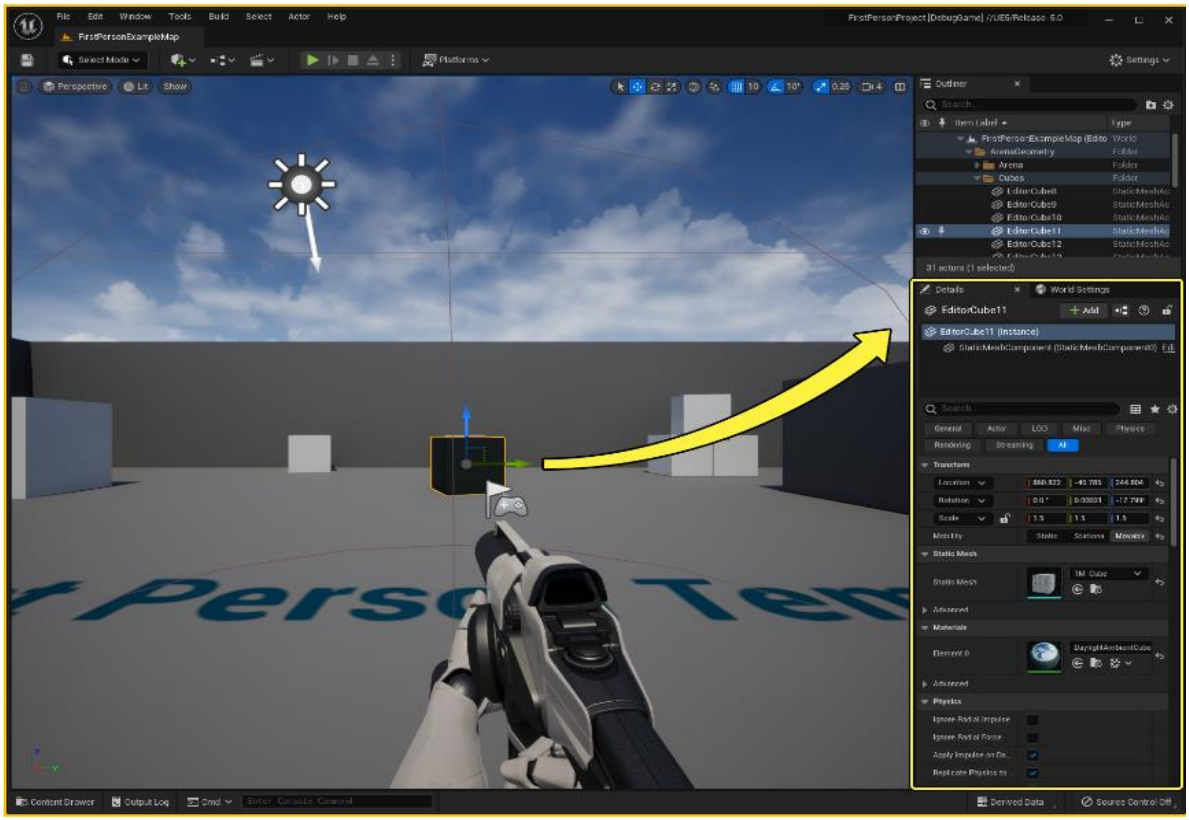
Omogućuje izradu kompleksnih razina i svjetova pomoću intuitivnog uređivača nivoa. Isto tako, na Epic Gamesovoj trgovini (Unreal Engine Marketplace) možete pronaći razne dodatke, poput gotovih modela, tekstura, materijala, kompletnih igara, alata i skripti koje mogu olakšati razvoj igre. Unutar ove igre isključivo su korišteni besplatni dodaci.

Hijerarhijski prikaz (slika 7) prikazuje sav sadržaja na vašoj razini. Prema zadanim postavkama nalazi se u gornjem desnom kutu prozora urednik. Služi da brzo sakrijte ili otkrijte objekte klikom na pridružene oko gumb, pristupite glumčevom izborniku desnom klikom na tog glumca tada možete iz tog izbornika izvesti dodatne, specifične glumice, stvoriti, pomicati i brisati sadržaja mape.



Slika 7: Hijerarhijski prikaz sadržaja na trenutnoj razini

Kada odaberete glumca na sceni razine detaljni prikaz (slika 8) će prikazati postavke i svojstva koja utječu na odabranog glumca. Prema zadanim postavkama nalazi se na desnoj strani prozora urednika, ispod hijerarhijskog prikaza.



Slika 8: Detaljni prikaz odabranog glumca na sceni razine

3.2.4. Terminologija

Projekt (engl. *Project*)

Sadrži sve sadržaj i elemente igre. To uključuje različite mape poput nacrta (engl. *Blueprints*) i materijala (engl. *Materials*) organizirane unutar projekta. Kroz sadržaj pregleda (engl. *Content Browser*) u Unreal Editoru, možete pregledavati istu strukturu direktorija koja se nalazi na vašem disku, unutar mape projekt (engl. *Project*).

Svaki projekt ima svoju „*.uproject“* datoteku koja služi za stvaranje, otvaranje ili spremanje projekta. Također je moguće raditi na više projekata istovremeno.

Nacrt (engl. *Blueprint*)

Nacrt je cjelovit sustav za skriptiranje igre koji koristi vizualno sučelje sa čvorovima kako biste kreirali igraće elemente unutar Unreal Editora. Objekti definirani pomoću nacrta često se nazivaju nacrti (engl. *Blueprints*).

Objekt (engl. *Object*)

Objekti predstavljaju osnovnu klasu u Unreal Engineu, pružajući temeljne funkcije za vaše resurse. Gotovo sve u Unreal Engineu nasljeđuje ili koristi funkcionalnosti od objekata. U C++-u, „UObject“ je osnovna klasa za sve objekte, koja podržava sakupljanje smeća (*engl. Garbage Collections*), metapodatke „UProperty“ za izlaganje varijabli u Unreal Editoru i serijalizaciju za učitavanje i spremanje.

Klasa (engl. *Class*)

Klasa definira ponašanja i svojstva određenog glumca ili objekta u Unreal Engineu. Klase imaju hijerarhijsku strukturu, nasljeđujući informacije od roditeljske klase i prosljeđujući ih svojoj djeci. Možete ih stvarati u C++ kodu ili kroz nacrte.

Glumac (engl. *Actor*)

Glumci su objekti koji se mogu postaviti na razinu igre i predstavljaju različite entitete u svijetu igre. Oni mogu biti različiti elementi kao što su igrači, neprijatelji, predmeti, kamere, svjetla, efekti i mnogi drugi. Glumci imaju sposobnost podržavati 3D transformacije poput premještanja, rotacije i skaliranja, što omogućuje da se postave na odgovarajuće pozicije i orijentacije na sceni.

Različiti tipovi glumaca određuju različite uloge u igri. Na primjer, igrački glumac predstavlja lik kojim igrač upravlja, dok neprijateljski glumci predstavljaju protivnike koje igrač mora poraziti. Glumci također mogu predstavljati statične ili dinamične objekte u igri, kao što su zgrade, vozila, oružje i slično.

Kroz kod igre, bilo u C++ programiranju ili kroz vizualno skriptiranje (*engl. Blueprints*), moguće je stvarati i uništavati glumce, postavljati njihove početne parametre, kontrolirati njihovo ponašanje, izvoditi animacije i interakcije s drugim glumcima te prilagođavati njihovu funkcionalnost prema potrebama igre.

Lijevanje (engl. *Casting*)

Lijevanje je postupak u kojem pokušavamo pretvoriti glumca određene klase u drugu klasu. Ovaj postupak može uspjeti ili propasti, ovisno o tome može li se glumac uspješno "pretvoriti" u ciljanu klasu. Ako lijevanje uspije, tada možemo pristupiti funkcionalnostima specifičnim za tu ciljanu klasu na glumcu kojem izvodimo lijevanje.

Lijevanje se razlikuje od jednostavne provjere pripada li glumac određenoj klasi, što bi rezultiralo binarnim odgovorom (da ili ne). Omogućuje interakciju s specifičnim funkcionalnostima ciljane klase, što može biti korisno za različite situacije u igri.

Komponenta (engl. *Component*)

Komponenta u Unreal Engineu predstavlja dio funkcionalnosti koji se može dodati glumcu. Kada dodate komponentu glumcu, on postaje sposoban koristiti funkcionalnost koju ta komponenta pruža. Primjerice komponenta reflektora će omogućiti glumcu da emitira svjetlost poput reflektora, komponenta rotirajućeg pokreta će natjerati vašeg glumca da se okreće ili rotira, audio komponenta će vašem glumcu omogućiti reprodukciju zvukova.

Komponente moraju biti pridružene glumcu kako bi mu dodale željene funkcionalnosti, i same po sebi ne mogu postojati. Unreal Engine omogućuje stvaranje raznih komponenti koje se mogu dodati glumcima kako bi se postigao raznovrstan i interaktivno igranje. Svaka komponenta ima svoje specifične mogućnosti i ulogu unutar igre.

Pijun (engl. *Pawn*)

Pijun je pod klasa glumca (engl. *Actor*) u Unreal Engineu koja predstavlja avatar ili lik u igri. Pijuni (Pawn) mogu služiti kao igračev ili NPC-jev karakter unutar igre.

Igrači mogu kontrolirati pijune direktno ili putem umjetne inteligencije. Kada igrač kontrolira pijuna, kažemo da je pijun "u posjedu" igrača. To znači da igrač može upravljati kretanjem i akcijama tog pijuna u igri. S druge strane, kada pijuna ne kontrolira igrač ili umjetna inteligencija, smatra se "ne posjedovanim". U tom slučaju, pijun može djelovati kao neovisni NPC, vođen unaprijed definiranim skriptama ili logikom.

Pijuni su ključni elementi za izgradnju igre jer predstavljaju likove s kojima igrači intrigiraju i čine igru živahnom i dinamičnom. Različiti pijuni mogu imati različite vještine, sposobnosti i ponašanja, što omogućuje raznolike izazove i iskustva unutar igre.

Lik (engl. *Character*)

Lik je pod klasa glumca pijuna koja je posebno dizajnirana za korištenje kao igračev lik unutar igre. Lik pruža već unaprijed definirane postavke sudara, ulazne kontrole za dvonožno kretanje te dodatni kod za upravljanje kretanjem kojim upravlja igrač.

Ukratko, likovi su ključni elementi igre koji omogućuju igračima da preuzmu kontrolu nad svojim avatarima i istražuju svijet igre. Unreal Engine 5 pruža moćne alate i fleksibilnost za izradu, animiranje i upravljanje likovima, što rezultira bogatim i autentičnim iskustvom igranja.

Upravljač igrača (engl. *Player Controller*)

Upravljač igrača je ključan element koji preuzima unos igrača i pretvara ga u interakcije unutar igre. Svaka igra u UE5 ima barem jedan upravljač igrača, koji obično posjeduje pijuna ili lika kao reprezentaciju igrača unutar igre.

Upravljač igrača također ima ključnu ulogu u mrežnim igrama s više igrača. Na poslužitelju, postoji jedna instanca Upravljača igrača za svakog igrača u igri kako bi se omogućila komunikacija putem mreže. Svaki klijent ima samo svoj upravljač igrača koji odgovara njihovom igraču i može koristiti taj kontroler za komunikaciju s poslužiteljem. Upravljač igrača u Unreal Engineu ima svoju povezanu C++ klasu koja omogućuje programerima prilagođavanje i proširivanje njegove funkcionalnosti.

AI kontroler (engl. *AI Controller*)

AI kontroler je element koji upravlja ponašanjem likova koji nisu igrači (NPC-jevi) unutar igre. Slično kao što kontroler igrača posjeduje pijuna kao reprezentaciju igrača, AI kontroler posjeduje pijuna koji predstavlja NPC lika u igri. Prema zadanim postavkama, pijuni i likovi u igri će imati osnovni AI kontroler, osim ako nisu izričito zaposjednuti od strane kontrolera igrača ili im nije rečeno da ne stvaraju AI kontroler za sebe.

AI kontroler je bitan za stvaranje vjerodostojnog i dinamičnog iskustva igre, jer omogućuje NPC-jevima da djeluju realistično i prilagode se različitim situacijama. Unreal Engine pruža moćan i prilagodljiv sustav AI kontrolera koji omogućuje programerima implementaciju složenog i inteligentnog ponašanja za NPC-jeve u njihovim igrama.

Stanje igrača (engl. *Player State*)

Stanje igrača predstavlja podatke i informacije o pojedinom igraču u igri, bilo da je riječ o stvarnom igraču (ljudskom igraču) ili botu koji simulira igrača (AI igraču). Ova klasa sadrži različite varijable koje definiraju trenutno stanje igrača i omogućuje praćenje njegovog napretka tijekom igranja.

Primjeri podataka koje stanje igrača može sadržavati uključuju ime ili nadimak igrača koji ga identificira unutar igre, informacija o trenutnoj razini ili stadiju u kojem se igrač nalazi, podatak o trenutnom zdravstvenom stanju igrača, podaci o postignućima ili uspjesima koje je igrač ostvario tijekom igre.

Stanje igrača posebno je važno u igrama s više igrača gdje svaki igrač ima svoj vlastiti objekt stanja igrača. Ova stanja igrača postoje na svim računalima uključenim u igru kako bi se osigurala sinkronizacija i ispravan prikaz podataka. Podaci o stanju igrača mogu se replicirati s poslužitelja na klijente kako bi se osiguralo da svi igrači imaju ažurirane informacije o drugim igračima u igri.

Način igre (engl. *GameMode*)

Način igre predstavlja skup pravila i postavki koje definiraju kako će se igra igrati. Ova pravila mogu uključivati različite aspekte igre, kao što je određuje kako se igrači mogu pridružiti igri, poput mogućnosti stvaranja novih likova, odabira timova ili povezivanja s postojećom igrom, može li se igra pauzirati tijekom igranja ili ne. Neki načini igre dopuštaju igračima pauziranje igre kako bi mogli pregledati postavke, izlaziti iz igre ili obaviti druge akcije, dok drugi načini mogu zabraniti pauziranje kako bi igra ostala dinamična i neometana. Specificira uvjete potrebne za pobjedu u igri. To može uključivati dosezanje određenih ciljeva, skupljanje predmeta, preživljavanje određeno vrijeme ili poraz protivnika.

Način igre omogućuje da definira jedinstvene načine igre prilagođene igri. Također, možete postaviti zadani način igre u postavkama projekta, a zatim nadjačati taj način za različite razine igre kako biste stvorili raznolike i različite načine igranja.

U igrama za više igrača, način igre postoji samo na poslužitelju, a pravila se repliciraju (šalju) svakom povezanom klijentu kako bi svi igrači imali ažurirane informacije o trenutnom načinu igre.

Stanje igre (engl. *Game State*)

Stanje igre je spremnik koji sadrži informacije o trenutnom stanju igre i koji se replicira (šalje) svakom klijentu u igri. Ova klasa služi za praćenje globalnih podataka igre koji su važni za sve igrače u igri, bez obzira na njihovu ulogu ili lokaciju u svijetu igre.

Primjeri podataka koje stanje igre može sadržavati uključuju rezultat igre, bodovi, pobjede i porazi. Informacije o tome je li utakmica već započela, traje ili je završila. Broj AI likova koji će biti stvoreni na temelju broja igrača u svijetu igre. Informacije o svjetskim događanjima ili promjenama koje se odvijaju u igri.

Stanje igre omogućuje sinkronizaciju podataka između svih igrača u igri kako bi svi imali ažurirane informacije o globalnom stanju igre. Ova klasa koristi se u igrama za više igrača kako bi se osigurala dosljednost i ispravnost podataka za sve sudionike.

Svaki igrač ima lokalnu instancu stanja igre na svom računalu, a te lokalne instance dobivaju ažurirane informacije od poslužiteljske instance stanja igre. Na taj način, svi igrači imaju ažurirane i usklađene informacije o stanju igre.

Stanje igre klasa u Unreal Engineu omogućuje praćenje i upravljanje globalnim podacima igre, čime se olakšava implementacija sustava bodovanja, praćenja postignuća, kontroliranje završetka igre i sličnih funkcionalnosti.

Kist (engl. *Brush*)

Kist je glumac koji se koristi za opisivanje 3D oblika, kao što su kocke, sfere i druge jednostavne geometrije. Kistovi su korisni za brzo blokiranje razine i definiranje osnovnih geometrijskih oblika u igri.

Možete postaviti kistove u razinu kako biste definirali geometriju razine. Ova tehnika poznata je kao binarna prostorna particija ili BSP kistovi. Kada koristite kistove za definiranje razine, oni se koriste za stvaranje prepreka, prostora i drugih osnovnih oblika koji tvore svijet igre.

Kistovi su posebno korisni u početnim fazama razvoja igre kada želite brzo izgraditi razine i testirati mehanike igre. Nakon što ste postavili kistove kako želite, možete dalje izgrađivati detaljniju geometriju i umjetnost u igri.

Važno je napomenuti da, iako su kistovi korisni za brzu izgradnju, njihova upotreba može biti ograničena u složenijim igrama s visokim zahtjevima za detaljima i performansama. U takvim slučajevima, razvojni tim često koristi naprednije modele i alate za izradu kompleksnije geometrije i okoline.

Volumen (engl. *Volume*)

Volumeni u Unreal Engineu su ograničeni 3D prostori koji imaju različite namjene na temelju učinaka koji su im pridruženi. Oni služe kao nevidljive zone s posebnim funkcionalnostima koje utječu na ponašanje glumaca u igri. Ovisno o vrsti volumena i pridruženim efektima, volumeni omogućuju različite interakcije i događaje u igri.

Ove vrste volumena omogućuju različite mehanike igre i interakcije s okolinom. Korištenjem volumena, može se stvoriti dinamično i raznoliko okruženje koje igračima pruža različite izazove i iskustva.

Razina (engl. *Level*)

Razina u Unreal Engineu predstavlja područje igranja koje definirate u igri. To područje sadrži sve što igrač može vidjeti, istraživati i s čime može interaktirati, uključujući geometriju svijeta, pijune (NPC-ove) i glumce (koji predstavljaju igračeve likove).

Unreal Engine koristi posebne datoteke za spremanje svake razine kao zasebnu „.umap“ datoteku. Ove datoteke često se nazivaju i "karte" jer predstavljaju prostor na kojem će se igra odvijati. Svaka razina može sadržavati različite elemente poput terena, zgrada, predmeta, likova, AI logike i drugih interaktivnih elemenata.

Razina omogućuje organiziranje i strukturiranje svijeta igre u logičke cjeline. U razini može se definirati različite scene, misije ili dijelove igre koje igrač može istraživati. Također, moguće je stvarati više razina kako bi se postigli različiti nivoi igre, različite zone, ili kako biste radili na različitim dijelovima igre paralelno.

Unutar Unreal Enginea postoji alat koji se naziva uređivač razina (*engl. Level Editor*) koji omogućuje stvaranje, uređivanje i organiziranje razina. Uz pomoć ovog alata, možete vizualno oblikovati svijet igre, postavljati objekte, teren, svjetla, kamere i ostale elemente kako biste kreirali željeni doživljaj igre.

Svijet (engl. *World*)

Svijet predstavlja spremnik koji sadrži sve razine koje čine vašu igru. To je konceptualni prostor u kojem se odvija cijela igra, a služi kao kontejner za organizaciju, upravljanje i strujanje različitih razina koje igrač može istraživati tijekom igranja.

Svijet obuhvaća sve različite razine koje su dizajnirane i kreirane kako bi igrači doživjeli različite dijelove igre. Na primjer, igra može imati više razina koje predstavljaju različite lokacije, razine težine, izazove ili misije. Svaki razina u svijetu može biti odvojena i samostalna .umap datoteka.

Unutar svijeta, igrači mogu putovati iz jedne razine u drugu kako bi napredovali u igri. Svijet također upravlja dinamičkim stvaranjem glumaca (pijuna i likova) u određenim razinama kako bi omogućio dinamičan i interaktivan doživljaj igre.

Uređivač svijeta u Unreal Engineu omogućuje vam da vizualno organizirate različite razine, povezujete ih, postavljate prijelaze između njih i postavljate pravila koja upravljaju prelaskom iz jedne razine u drugu. Svijet je centralno mjesto gdje se postavljaju temeljni elementi igre, kao što su glavne lokacije, postavke, stilovi, priče i interaktivni elementi.

3.3. Blender:

Blender je bio ključni alat za 3D modeliranje i animaciju u igri. Pomoću Blendera, uređeni su već postojeći 3D objekte kao i animacija za lika. Modeliranje i animacije izvedeni u Blenderu omogućili su kreativnost i realizam u dizajnu igre.

3.4. JetBrains Rider:

JetBrains Rider je integrirano razvojno okruženje (IDE) specijalizirano za razvoj aplikacija u programskom jeziku C# i .NET platformi. Razvijen od strane JetBrainsa, poznatog po svojim visokokvalitetnim alatima za programiranje, Rider nudi napredne značajke i intuitivan radni okvir za programere koji rade na .NET projektima.

Rider podržava programiranje u C++ jeziku, što je jedan od ključnih jezika korištenih za razvoj igara u Unreal Engineu. To omogućuje programerima da koriste sve mogućnosti C++ jezika i pruža im moćan alat za rad s Unreal Engine kodom. Sadrži sve potrebne alate za razvoj, uključujući tekstualni uređivač s naglašavanjem sintakse, alate za debugiranje, alat za upravljanje paketima, testiranje i još mnogo toga, što olakšava produktivnost i učinkovitost. Također, koristi snažne algoritme analize koda kako bi pružio inteligentne sugestije, ispravke i automatsko dovršavanje koda.

Rider pruža napredne alate za debugiranje koda u stvarnom vremenu, kao i profiliranje performansi kako bi se identificirale uske grlo i optimizirao rad aplikacije. Nudi bogate mogućnosti uključujući preimenovanje, ekstrakciju metoda, uklanjanje dupliciranog koda i mnoge druge, čime se olakšava održavanje i unapređenje koda.
Rider je dostupan za različite operativne sustave, uključujući Windows, macOS i Linux, što omogućuje programerima raditi na različitim platformama s istim IDE-om.

4. IMPLEMENTACIJA IGRE

U ovoj fazi razvoja, fokus je na opisu igre koja se sastoji od osam razina. Igrač preuzima ulogu ženskog lika te sakuplja zlatne predmete kako bi prešao na sljedeću razinu. Svako druga završena razina donosi novu puzzle mehaniku koja bi otežala prolaska kroz razinu. Igrač se suočava s raznim preprekama koje zahtijevaju logičko razmišljanje kako bi ih zaobišao, otključao, prešao preko te rotirao objekte. Cilj igre je pružiti igračima uzbudljivo iskustvo kroz rješavanje zagonetki i izazova.

Razvoj igre je podijeljen u nekoliko faza:

1. Klasa za lika,
2. Klasa za GameMode i Widgete,
3. Puzzle Mehanike

4.1. Skripta za glavnog lika

U ovom radu, razvoj funkcionalnosti lika za 3D puzzle igru izvršen je putem C++ klase `AGameProjectCharacter` i njene pripadajuće nacrt klase `BP_Character`. Glavni lik nasljeđuje osnovnu `ACharacter` klasu iz Unreal Enginea, što mu omogućuje korištenje standardnih mogućnosti za kontrolu lika u igri.

Unutar konstruktor metode `AGameProjectCharacter` klase, izvršena je inicijalizacija ključnih član-varijabli lika, uključujući broj života, status različitih ključeva, vrijeme i maksimalnu brzinu kretanja. Također, implementirana je funkcionalnost kamere koja prati glavnog lika tijekom igranja. Za ovu svrhu koriste se klase `UCameraComponent`, koja predstavlja vidno polje kamere i omogućuje podešavanje postavki kamere. Dio konstruktora je prikazan ispisu 1.

```
PlayerCamera =
CreateDefaultSubobject<UCameraComponent>("PlayerCamera");
PlayerCamera->SetupAttachment(GetCapsuleComponent());


MoveSpeed = 1.0f;
bHasKeyRed = false;
bHasKeyBlue = false;
bHasKeyGrey = false;
bTimesUp = false;
```

Ispis 1: Dio konstruktora

Pored toga, ključne metode `MoveForward` i `MoveRight` (ispis 2) implementirane su za omogućavanje kontrole lika tijekom igre. Te metode koriste `AddMovementInput` funkciju kako bi omogućile glatko i precizno kretanje lika prema naprijed i bočno. `GetActorForwardVector` je funkcija koja vraća vektor koji predstavlja smjer naprijed za glavnog lika, odnosno vektor koji pokazuje u kojem smjeru gleda lik.

`GetInputAxisValue("MoveForward")` je funkcija koja dobavlja vrijednost unosa osi za os `MoveForward`. Os `MoveForward` predstavlja naprijed/straga kretanje, a vrijednost se dobiva iz korisničkog unosa. Ta vrijednost će odrediti koliko brzo će se objekt kretati prema naprijed.

`GetActorForwardVector()*GetInputAxisValue("MoveForward" +GetActorRightVector()*Value)` izračun je konačnog smjera kretanja objekta. Pomnožen je vektor smjera naprijed s vrijednošću unosa za naprijed/straga kretanje i dodan je vektor smjera udesno pomnožen s vrijednošću unosa za kretanje udesno. Na ovaj način se dobiva konačan vektor koji predstavlja ukupan smjer kretanja objekta, kombinirajući pomake u oba smjera.

Metoda `GetSafeNormal` normalizira vektor, što znači da mu postavlja duljinu na jedan. To je korisno jer će tako svi vektori smjera imati isti utjecaj na kretanje, bez obzira

na duljinu pojedinih vektora. Kada se pozove `AddMovementInput`, lik će se kretati prema naprijed ako je `Value` pozitivan ili unatrag ako je negativan.

`Value` je ulazna vrijednost koja određuje koliko brzo će lik ići prema naprijed ili unatrag. Ako je pozitivan, lik će se kretati prema naprijed s brzinom `MoveSpeed` (koja je član-varijabla klase `AGameProjectCharacter` i koja označava maksimalnu brzinu kretanja lika). Ako je negativan, lik će se kretati unatrag s istom brzinom.

```
void AGameProjectCharacter::MoveForward(float Value)
{
      FVector MovementDirection = (GetActorForwardVector() *
      Value + GetActorRightVector() *
      GetInputAxisValue("MoveRight")).GetSafeNormal();
      AddMovementInput(MovementDirection, MoveSpeed);
}
void AGameProjectCharacter::MoveRight(float Value)
{
      FVector MovementDirection = (GetActorForwardVector() *
      GetInputAxisValue("MoveForward") + GetActorRightVector() *
      Value).GetSafeNormal();
      AddMovementInput(MovementDirection, MoveSpeed);
}
```

Ispis 2: Metode kretanja lika

Metoda nazvana `SetupPlayerInputComponent` (ispis 3) koja je također dio klase `AGameProjectCharacter`. Ova metoda se koristi za postavljanje ulaznih kontrola (engl. *input*) koje će omogućiti igraču upravljanje glavnim likom (engl. *character*) tijekom igranja igre.

Unutar ove metode, koristi se `PlayerInputComponent` koji je objekt tipa `UInputComponent` i predstavlja komponentu koja omogućuje prikupljanje i obradu ulaznih događaja od igrača.

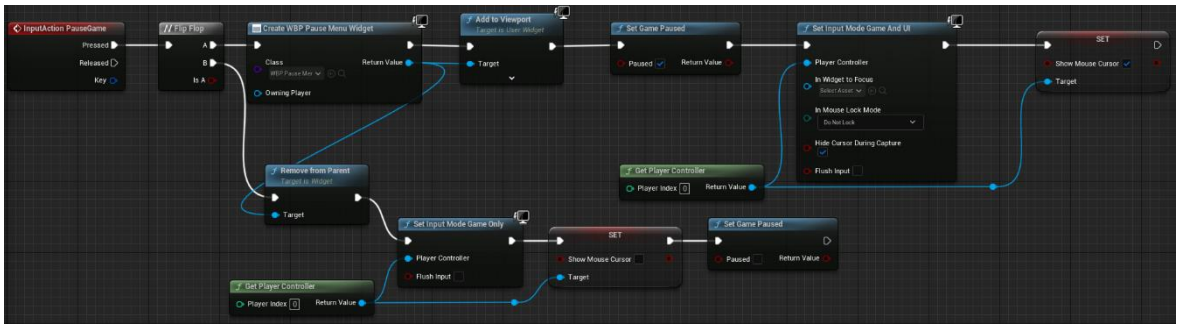
```
void
AGameProjectCharacter::SetupPlayerInputComponent(UInputComponent
* PlayerInputComponent)

{

      Super::SetupPlayerInputComponent(PlayerInputComponent);

      PlayerInputComponent->BindAxis("MoveForward", this,
&AGameProjectCharacter::MoveForward);

      PlayerInputComponent->BindAxis("MoveRight", this,
&AGameProjectCharacter::MoveRight);

      PlayerInputComponent->BindAction("PauseGame",
IE_Pressed,this, &AGameProjectCharacter::CheckPause);

}
```

Ispis 3: Metoda za postavljanje ulaznih kontrola

Pomoću funkcije `BindAxis` i `BindAction` postavljaju se različite akcije i osi koje će reagirati na ulazne kontrole. `BindAxis` linija povezuje ulaznu os `MoveForward` (koja je obično povezana s tipkama za kretanje naprijed i unatrag) s metodom `MoveForward` iz klase `AGameProjectCharacter`. Kada igrač pritisne odgovarajuće tipke za kretanje, metoda `MoveForward` će se aktivirati i glavni lik će se kretati naprijed ili unatrag ovisno o pritisnutim tipkama. `BindAxis` povezuje ulaznu os `MoveRight` (koja je obično povezana s tipkama za kretanje lijevo i desno) s metodom `MoveRight` iz klase `AGameProjectCharacter`. Kada igrač pritisne odgovarajuće tipke za kretanje lijevo i desno, metoda `MoveRight` će se aktivirati i glavni lik će se kretati lijevo ili desno ovisno o pritisnutim tipkama.

U klasi također se nalazi funkcionalnost za upravljanje stanjem igre. Na primjer, metoda `CheckPause` omogućuje igraču pauziranje igre kako bi provjerio postavke ili napravio kratku pauzu, nakon čega može nastaviti s igranjem. `BindAction` povezuje ulaznu akciju `PauseGame` (koja je povezana s tipkom za pauziranje igre) s metodom `CheckPause`. Kada igrač pritisne tipku za pauziranje igre, koja je u ovom slučaju tipka „P“ metoda `CheckPause` će se aktivirati i igra će se pauzirati ili nastaviti, ovisno o trenutnom stanju igre. To je definirano kroz nacrt (slika 9), u slučaju kada se tipka „P“ pritisne otvara se pauzirali zaslon koji je već prethodno kreiran pod nazivom

„WBP_PauseMenu“, igra se pauzira te se prikaze kursor s kojim se vrši interakcija s gumbima za nastavak igre, ponovno pokretanje ili povratak na glavni izbornik. Kroz ove veze (engl. *bindings*), igrač će moći upravljati likom u igri pomoću ulaznih kontrola i tako sudjelovati u igranju igre.



Slika 9: Nacrt pauziranje igre

Metoda "Tick" (ispis 4) se poziva svaki okvir igre i koristi se za ažuriranje logike lika ili igre tijekom igranja.

```
void AGameProjectCharacter::Tick(float DeltaTime)

{

      Super::Tick(DeltaTime);

      float ForwardValue = GetInputAxisValue("MoveForward");
      float RightValue = GetInputAxisValue("MoveRight");

      if (ForwardValue != 0.0f || RightValue != 0.0f)
      {
          if (ForwardValue > 0.0f && RightValue > 0.0f)
          {
             LastMovementDirection =
      EMovementDirection::ForwardRight;
          }
          else if (ForwardValue > 0.0f && RightValue < 0.0f)
          {
             LastMovementDirection =
      EMovementDirection::ForwardLeft;
```

```
    }
    else if (ForwardValue < 0.0f && RightValue > 0.0f)
    {
       LastMovementDirection =
EMovementDirection::BackwardRight;
    }
    else if (ForwardValue < 0.0f && RightValue < 0.0f)
    {
       LastMovementDirection =
EMovementDirection::BackwardLeft;
    }
    else if (ForwardValue > 0.0f)
    {
       LastMovementDirection =
EMovementDirection::Forward;
    }
    else if (ForwardValue < 0.0f)
    {
       LastMovementDirection =
EMovementDirection::Backward;
    }
    else if (RightValue > 0.0f)
    {
       LastMovementDirection = EMovementDirection::Right;
    }
    else if (RightValue < 0.0f)
    {
       LastMovementDirection = EMovementDirection::Left;
    }
}

switch (LastMovementDirection)
{
case EMovementDirection::Forward:
    CharacterMesh->SetRelativeRotation(FRotator(0.0f,
270.0f, 0.0f));
    break;
case EMovementDirection::ForwardRight:
    CharacterMesh->SetRelativeRotation(FRotator(0.0f,
315.0f, 0.0f));
    break;
case EMovementDirection::ForwardLeft:
    CharacterMesh->SetRelativeRotation(FRotator(0.0f,
225.0f, 0.0f));
    break;
case EMovementDirection::Backward:
    CharacterMesh->SetRelativeRotation(FRotator(0.0f,
90.0f, 0.0f));
    break;
```

```
        case EMovementDirection::BackwardRight:
            CharacterMesh->SetRelativeRotation(FRotator(0.0f,
        45.0f, 0.0f));
            break;
        case EMovementDirection::BackwardLeft:
            CharacterMesh->SetRelativeRotation(FRotator(0.0f,
        135.0f, 0.0f));
            break;
        case EMovementDirection::Left:
            CharacterMesh->SetRelativeRotation(FRotator(0.0f,
        180.0f, 0.0f));
            break;
        case EMovementDirection::Right:
            CharacterMesh->SetRelativeRotation(FRotator(0.0f,
        0.0f, 0.0f));
            break;
        default:
            break;
        }
}
```

Ispis 4: Dio `Tick` metode koji rotira lika

`Tick` je poziv metode iz nad klase (roditeljske klase), što osigurava da se i dalje izvršava osnovna logika `Tick` metode u roditeljskoj klasi. Dok `switch` provjerava vrijednost varijable `LastMovementDirection`. Varijabla predstavlja smjer posljednjeg kretanja lika. Ovisno o vrijednosti varijable, postavit će se odgovarajuća rotacija lika, što će ga vizualno orijentirati prema naprijed, unatrag, lijevo, desno ili dijagonalno.

Kroz `AGameProjectCharacter` klasu, implementirana je i metoda za detekciju smrti lika `Die` (ispis 5).

```
void AGameProjectCharacter::Die()

{

      AGameProjectGameModeBase* GameMode =
      Cast<AGameProjectGameModeBase>(GetWorld()-
      >GetAuthGameMode());

      if (GameMode)

      {

            GameMode->GameOver();

      }

}
```

Ispis 5: Metoda `Die`

U metodi `GameOver`, prvo se pokušava dohvatiti referenca na upravljač igrača (engl.*PlayerController*) koji kontrolira igrača pomoću funkcije `GetPlayerController`. Ako je `PlayerController` valjan, dalje se nastavlja s izvođenjem koda. Zatim se stvara widget za prikazivanje GameOver poruke pomoću funkcije `CreateWidget`, pri čemu se provjerava je li stvoreni widget valjan pomoću `Cast<UGameProjectGameOverWidget>`. Ako je widget valjan, dodaje se na prikazani zaslon pomoću `AddToViewport`, te se postavlja pauza na igri i mijenja način rada unosa kako bi igrač mogao interaktirati samo s prikazanim widgetom (ispis 6).

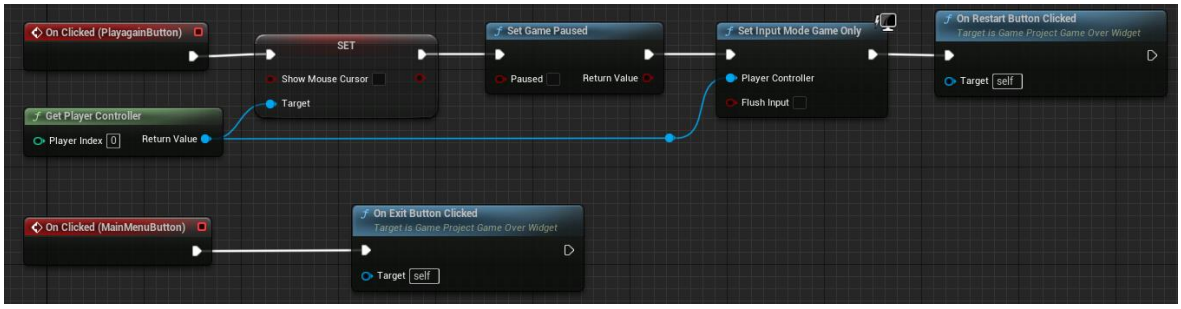
```
APlayerController* PlayerController =
UGameplayStatics::GetPlayerController(this, 0);
    if (PlayerController)
    {
       GameOverWidget =
Cast<UGameProjectGameOverWidget>(CreateWidget(GetWorld(),GameOve
rWidgetClass));
       if (GameOverWidget)
       {
          GameOverWidget->AddToViewport();
          PlayerController->SetPause(true);
          PlayerController->SetInputMode(FInputModeUIOnly());
          PlayerController->bShowMouseCursor = true;
       }
    }
```

Ispis 6: Metoda `GameOver`

Widget sadrži dva gumba: izlaz (engl. *Exit*) i ponovno pokretanje (engl. *Restart)*, koji omogućuju igraču da napusti igru ili ponovno pokrene trenutnu razinu (slika 9 i ispis 7).

```
void UGameProjectGameOverWidget::OnExitButtonClicked()
{
    UGameplayStatics::OpenLevel(GetWorld(), FName("MenuLevel"),
true);
}

void UGameProjectGameOverWidget::OnRestartButtonClicked()
{
    FString LevelName =
UGameplayStatics::GetCurrentLevelName(GetWorld(), true);
    UGameplayStatics::OpenLevel(GetWorld(), FName(LevelName),
true);
}
```

Ispis 7: Metode u `UGameProjectGameOverWidget` klasi



Slika 10: Pozivanje metoda kroz nacrt

`OnExitButtonClicked` metoda se poziva kada igrač klikne na gumb „Exit“. U metodi se koristi funkcija `OpenLevel` kako bi se igra prebacila na razinu s imenom „MenuLevel“. Argument `true` u funkciji znači da se stvaraju nove instancije razina, što znači da će se prethodna razina istog imena zatvoriti i otvoriti nova instanca razina „MenuLevel“. Ovo se obično koristi za prelazak iz jedne razine u drugu ili za povratak na glavni izbornik igre.

Klik na gumb „Restart“ poziva se metoda `OnRestartButtonClicked`. Prvo se dohvaća ime trenutne razine pomoću funkcije `GetCurrentLevelName`. Zatim se koristi funkcija `OpenLevel` kako bi se igra ponovno pokrenula na trenutnoj razini, tako što se prebacuje na razinu s istim imenom kao trenutna razina. Također, argument `true` znači da će se stvarati nove instancije razina.

Kombinacijom koda iz klasa `AGameProjectCharacter` i `AGameProjectGameModeBase`, igra ima implementiranu logiku završetka igre kad lik igrača umre. Kad lik umre, prikazuje se „GameOver“ poruka na ekranu, igra se zaustavlja, i igraču se omogućuje interakcija s porukom. Ova funkcionalnost doprinosi boljem iskustvu igrača i omogućuje mu da zna kada je igra završena.

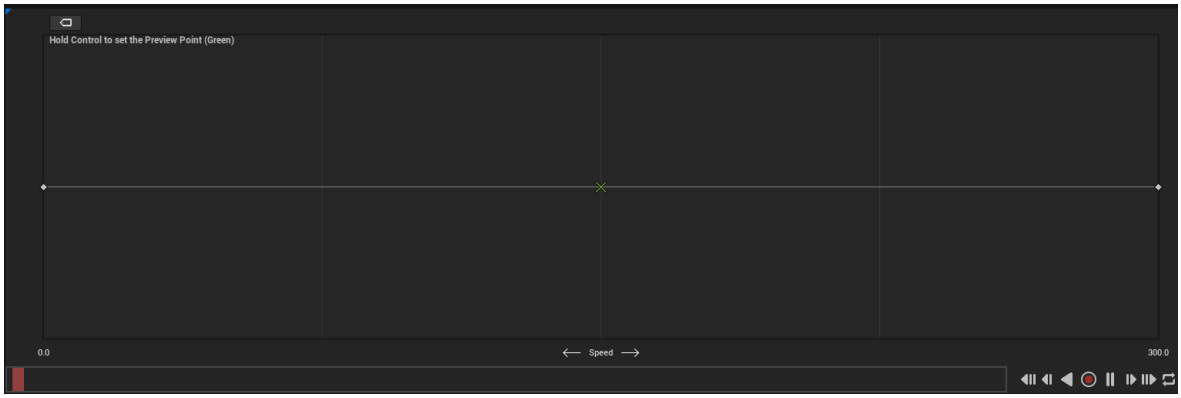
Animacija lika u Unreal Engine-u omogućava da likovi u igri ožive i dobiju realističan izgled i pokrete. Ovaj proces započinje uvozom 3D modela lika u Unreal Engine, zajedno s odgovarajućim kosturom koji će omogućiti animiranje lika. Nakon što je 3D model uvezen, koristi se alat za izradu animacija, poput Blend Spacea, za kreiranje različitih pokreta i animacija lika.

Blend Space je alat koji omogućava kombiniranje više animacija kako bi se stvorili glatki prijelazi između njih. Primjerice, može se koristiti za stvaranje trčanja, hodanja i stajanja, a zatim se međusobno kombiniraju kako bi se omogućilo glatko prelazak iz jednog u drugi.

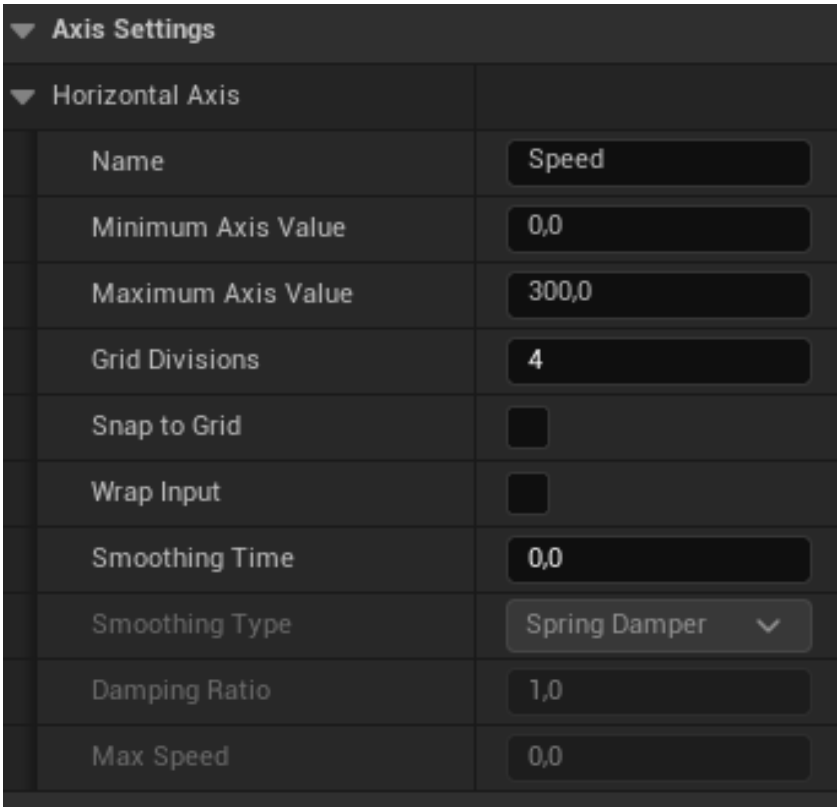
Kostur lika, također poznat kao rigging, je postupak postavljanja kostiju i zglobova unutar 3D modela kako bi se omogućilo animiranje lika. Svaka kost predstavlja dio lika (noga, ruka, glava itd.), dok su zglobovi mjesta gdje se kosti povezuju i omogućavaju pokrete.

Alat za izradu animacija u Unreal Engine-u omogućava postavljanje ključnih okvira (engl. *keyframes*) koji definiraju položaj i rotaciju kostiju tijekom vremena. Ovi ključni okviri čine glatke prijelaze između različitih stanja i pokreta lika.

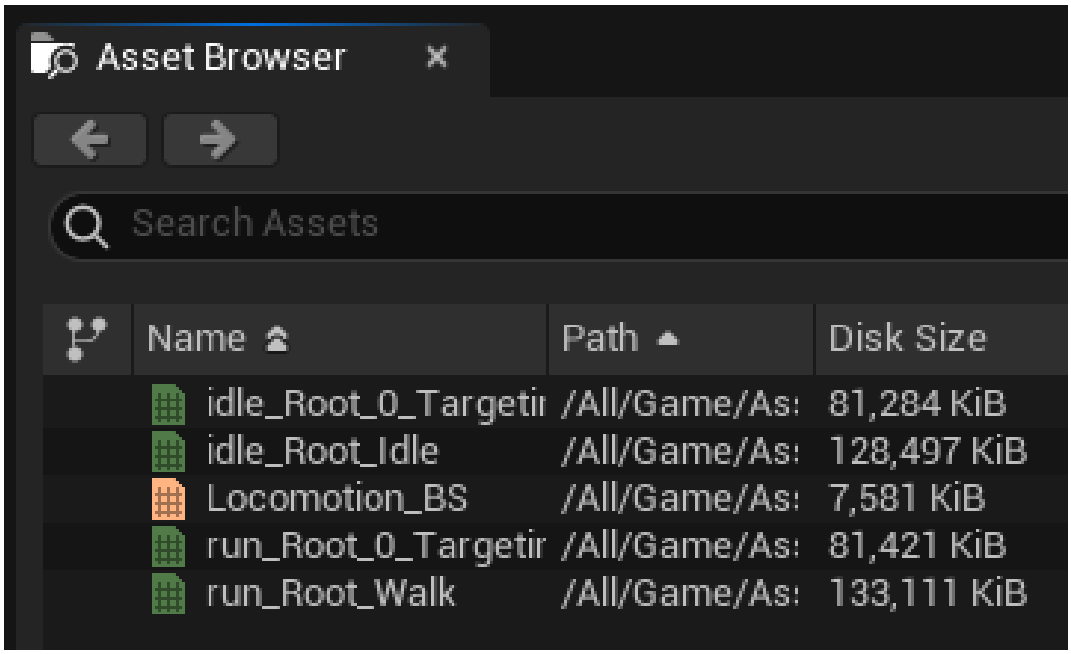
Na slici 11 možete vidjeti horizontalne os koordinatne mreže. U detaljima koordinatne mreže (slika 12) horizontalna os je nazvana brzina (engl. speed) s maksimalnom vrijednosti osi (engl. *Maximum Axis Value*) 300, dok je minimalna 0. Horizontalan os je podijeljena na 4 dijela. U donjem dijelu desne strane je popis svih animacija koje su uvedene u projekt (slika 13). Za prikaz tih animacija ostvaruje se dvoklikom na pojedinu animaciju gdje se otvara novi prozor u kojem se prikazuje animacija.



Slika 11: Koordinatna mreža



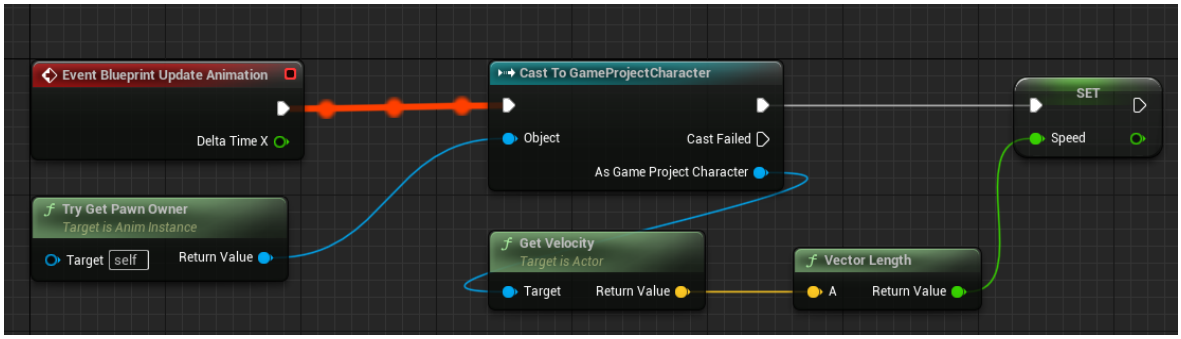
Slika 12: Detalji koordinatna mreže



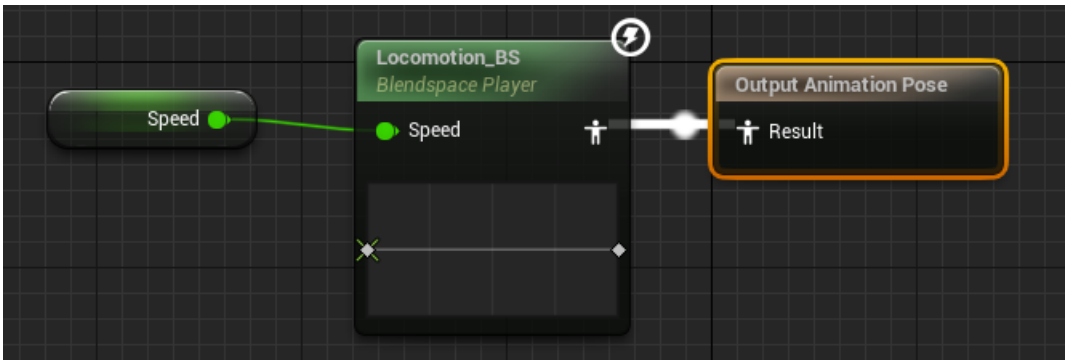
Slika 13: Lista animacija

Na horizontalnoj osi gdje je minimalna vrijedno 0, što predstavlja mirovanje odnosno brzinu kretanja lika. U listi animacija se nađe ona koja je *Idle* i postavi se na tu vrijednost. Nakon toga na maksimalnu vrijednost u ovom slučaju 300 se odabire vrsta animacije *Walk*. Lik sada ima animacije od šetanja do stajanja na mjestu.

Sada se odabire nacrt (engl. *blueprint*) na vrhu desne strane prozora gdje je potrebno napraviti događaj koji se osvježavati brzinu lika (slika 14) te kasnije poslati stanju koji se osvježavati animaciju između stajanja i hodanja (slika 15).



Slika 14: Nacrt brzine



Slika 15: Primjer stanja

4.2. Implementacija GameMode klase u igri

Postoji GameMode klasa, `AGameProjectGameModeBase`, koja je odgovorna za upravljanje logikom igre. Ova klasa ima niz metoda koje se koriste za različite funkcionalnosti igre, uključujući ažuriranje stanja igre, praćenje prikupljenih predmeta, rad s vremenom i prikazivanje odgovarajućih widgeta.

Metoda `BeginPlay` je dio početnog životnog ciklusa GameMode klase i poziva se kada igra započne (ispis 8). Prvo se poziva `Super::BeginPlay` kako bi se izvršile osnovne inicijalizacije iz nadređene klase. Zatim se dohvaća niz svih instanci objekata klase `AGameProjectItemBase` u igri pomoću `GetAllActorsOfClass`.

Broj prikupljenih predmeta `ItemsCollected` postavlja se na broj elemenata u nizu, što predstavlja ukupan broj predmeta u razini `ItemsInLevel`. Također, u ovoj metodi se stvara i prikazuje widget klase `UGameProjectWidget`, koji služi za prikaz informacija o igri, uključujući prikupljene predmete. Ako je stvaranje widgeta uspješno, dodaje se na viewport i poziva se metoda `UpdateItemText` za ažuriranje broja prikupljenih predmeta u widgetu.

Nakon toga, stvara se instanca klase `AGameProjectTimer` naziva `TimerManager` i pokreće se mjerenje vremena pomoću `TimerManager-`

>StartTimer. Na kraju, ažuriraju se svi gumbi klase AGameProjectARotatingButton u razini kako bi se postavio njihov trenutni položaj na temelju stanja bIsBtnPressed.

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

    TArray<AActor*> Items;
    UGameplayStatics::GetAllActorsOfClass(GetWorld(),
AGameProjectItemBase::StaticClass(), Items);
    ItemsInLevel = Items.Num();

    if (GameWidgetClass)
    {
       GameWidget =
Cast<UGameProjectWidget>(CreateWidget(GetWorld(),GameWidgetClass
));
       if (GameWidget)
       {
          GameWidget->AddToViewport();
          UpdateItemText();
       }
    }

    TimerManager = GetWorld()->SpawnActor<AGameProjectTimer>();
    if (TimerManager)
    {
       TimerManager->StartTimer();
    }

    TArray<AActor*> Buttons;
    UGameplayStatics::GetAllActorsOfClass(GetWorld(),
AGameProjectARotatingButton::StaticClass(), Buttons);
```

```
    for (AActor* button : Buttons)
    {
       AGameProjectARotatingButton* RotatingButton =
Cast<AGameProjectARotatingButton>(button);
       if (RotatingButton)
       {
          RotatingButton->UpdateButtonLocation(RotatingButton-
>bIsBtnPressed);
       }
    }
}
```

Ispis 8: Metoda "BeginPlay"

Metoda `UpdateItemText` ažurira prikaz broja prikupljenih predmeta u widgetu `UGameProjectWidget`. Dok metoda `ItemCollected` poziva kada igrač prikupi novi predmet. Povećava se broj prikupljenih predmeta `ItemsCollected` i poziva se `UpdateItemText` kako bi se ažurirao prikaz u widgetu (ispis 9).

```
void AGameProjectGameModeBase::UpdateItemText()
{
    GameWidget->SetItemText(ItemsCollected, ItemsInLevel);
}

void AGameProjectGameModeBase::ItemCollected()
{
    ItemsCollected++;
    UpdateItemText();
}
```

Ispis 9: Metode za prikaz prikupljenih predmeta

Tu je i metoda `StopTimer` gdje zaustavlja mjerenje vremena koje je pokrenuto u metodi `BeginPlay`. Poziva se `TimerManager->StopTimer` kako bi se zaustavilo mjerenje vremena. U `UpdateTimer` metodi se ažurira prikaz preostalog vremena u widgetu `UGameProjectWidget`. Poziva se `GameWidget->SetTimerText` kako bi se postavio prikaz preostalog vremena (ispis 10).

```
void AGameProjectGameModeBase::StopTimer()
{
    TimerManager->StopTimer();
}

void AGameProjectGameModeBase::UpdateTimer()
{
    GameWidget->SetTimerText(TimerManager->GetElapsedTime());
}
```

Ispis 10: Zaustavljanje i ažuriranje tajmera

Implementacija klase `AGameProjectTimer`, koja se koristi za praćenje vremena u igri. Odgovorna je za mjerenje vremena koje igrač ima na raspolaganju za završetak razine.

Konstruktor klase `AGameProjectTimer` postavlja `PrimaryActorTick.bCanEverTick` na `false`, što znači da objekt ove klase ne treba redovito osvježavanje u svakom frameu. Inicijalizira varijable `TimerDuration`, `EndDuration` i `bTimerRunning` na odgovarajuće početne vrijednosti (ispis 11).

```
AGameProjectTimer::AGameProjectTimer()
{
    PrimaryActorTick.bCanEverTick = false;
    EndDuration = 0.0f;
    TimerDuration = 300.0f;
    bTimerRunning = false;
}
```

Ispis 11: Konstruktor tajmera

`EndDuration` predstavlja trajanje vremena koje igrač ima na raspolaganju, postavljeno na 0.0f u ovom slučaju. `TimerDuration` predstavlja proteklo vrijeme, postavljeno na 300.0f (sekunde) u ovom slučaju, što znači da igrač ima 5 minuta za završetak razine. `bTimerRunning` je `bool` varijabla koja označava je li tajmer uključen i u ovom slučaju postavljena na `false`, jer tajmer započinje tek kad se pozove `StartTimer`.

Metoda `StartTimer` se koristi za pokretanje vremenskog mjerenja. Provjerava se je li tajmer već uključen `bTimerRunning`. Ako nije, postavlja se da je tajmer uključen i pokreće se `GetWorldTimerManager().SetTimer` s odgovarajućim vremenom kašnjenja i intervalom. Kada je tajmer postavljen, poziva se metoda `TimerCallback` svake sekunde (1.0f) kako bi se provjerilo jesu li ispunjeni uvjeti za kraj igre. Dok `StopTimer` metoda zaustavlja vremensko mjerenje. Provjerava se je li tajmer aktivan ako je, postavlja se da je zaustavljen i briše se pozivom `GetWorldTimerManager().ClearTimer` (ispis 12).

```
void AGameProjectTimer::StartTimer()
{
    if (!bTimerRunning)
    {
       bTimerRunning = true;
       GetWorldTimerManager().SetTimer(TimerHandle, this,
&AGameProjectTimer::TimerCallback, 1.0f, true);
    }
}

void AGameProjectTimer::StopTimer()
{
    if (bTimerRunning)
    {
       bTimerRunning = false;
       GetWorldTimerManager().ClearTimer(TimerHandle);
    }
}
```

Ispis 12: Start i stop tajmera

A `TimerCallback` metoda (ispis 13) se poziva svake sekunde (1.0f) kad je tajmer aktivan. Smanjuje se proteklo vrijeme `TimerDuration` za 1 sekundu. Ako proteklo vrijeme postane manje ili jednako kraju vremena `EndDuration`, to znači da je igrač premašio dostupno vrijeme i poziva se metoda `Die` na liku igrača. Zatim se zaustavlja tajmer pozivom `StopTimer`, kako se ne bi dalje provjeravali uvjeti za kraj igre.

```
void AGameProjectTimer::TimerCallback()
{
    TimerDuration -= 1.0f;
    if (TimerDuration <= EndDuration)
    {
       AGameProjectCharacter* ProjectCharacter =
Cast<AGameProjectCharacter>(UGameplayStatics::GetPlayerCharacter
(GetWorld(), 0));
       if (ProjectCharacter)
       {
          StopTimer();
          ProjectCharacter->Die();
       }
    }
}
```

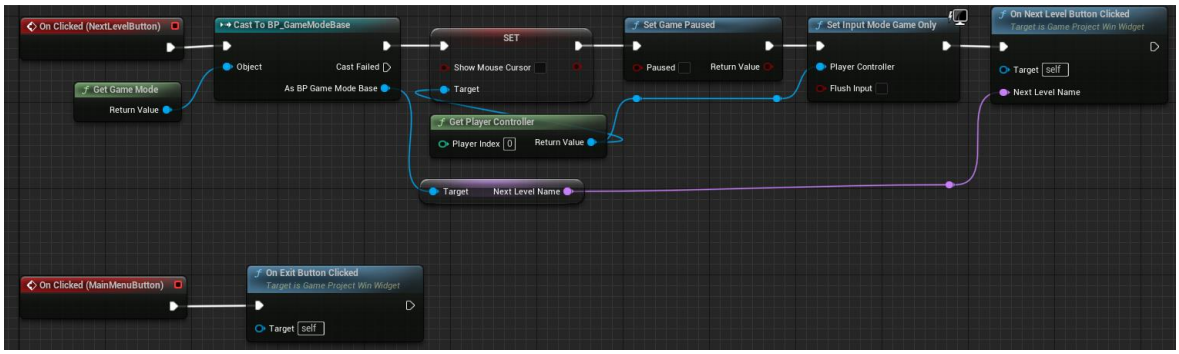
Ispis 13: Metoda `TimerCallback`

Imamo još jedan prikaz widgeta, kada igrač skupi sve zlatne predmete i stane na cilj prikaže se pobjednički widget `ShowWinWidget(FName NextLevel)` sa 2 gumba od koji je jedan za prelazak na sljedeću razinu, jedan za povratak na glavni izbornik. Prvo se dohvaća referenca na `PlayerController`, a zatim se stvara i prikazuje widget `UGameProjectWinWidget`, koji služi za prikazivanje poruke „Congratulation!“. Također, postavlja se pauza na igri i mijenja se način rada unosa kako bi igrač mogao interaktirati samo s prikazanim widgetom (ispis 14).

```
void AGameProjectGameModeBase::ShowWinWidget(FName NextLevel)
{
    NextLevelName = NextLevel;
    APlayerController* PlayerController =
UGameplayStatics::GetPlayerController(this, 0);
    if (PlayerController)
    {
       GameWinWidget =
Cast<UGameProjectWinWidget>(CreateWidget(GetWorld(),GameWinWidge
tClass));

       if (GameWinWidget)
       {
          GameWinWidget->AddToViewport();
          PlayerController->SetPause(true);
          PlayerController->SetInputMode(FInputModeUIOnly());
          PlayerController->bShowMouseCursor = true;
       }
    }
}
```

Ispis 14: Metoda `ShowWinWidget`

U nacrtu se dohvati vrijednost `NextLevelName` varijable i šalje se metodi `OnNextLevelButtonClicked` u `ShowWinWidget` klasi te se ostale funkcije vraćaju na izvorno stanje kao na primjer kursor miša, pauziranje i način unosa (slika 10).



Slika 11: Nacrt Widgeta

Na kraju ove dvije metode pozivaju funkciju `OpenLevel` gdje će otvoriti sljedeću razinu ili se vratiti na glavni izbornik oviseći koji se gumb odabrao (ispis 15).

```
void UGameProjectWinWidget::OnNextLevelButtonClicked(FName
NextLevelName)
{
    UGameplayStatics::OpenLevel(GetWorld(), NextLevelName);
}

void UGameProjectWinWidget::OnExitButtonClicked()
{
    UGameplayStatics::OpenLevel(GetWorld(), FName("MenuLevel"),
true);
}
```

Ispis 15: Metode `UGameProjectWinWidget` klase

4.3. Implementacija različitih objekata i mehanika u igri

Sastoji se od nekoliko različitih klasa koje nasljeđuju `AActor`, što označava da su sve te klase zapravo igrački objekti koji se mogu pojaviti u igri. Evo objašnjenja svake od tih klasa:

4.3.1. AGameProjectItemBase

Klasa koja predstavlja temeljni objekt u igri koji igrač može prikupiti u ovom slučaju zlatni predmet. Ima vizualni mesh kao rotacija oko svoje osi, zvuk koji se reproducira kad igrač prikupi objekt, i također koristi Niagara sustav za prikazivanje posebnih čestica (engl. *particles*) prilikom prikupljanja.

U konstruktoru `AGameProjectItemBase` se postavljaju početne vrijednosti i konfiguracija komponenti za ovu klasu (ispis 16).

```
PrimaryActorTick.bCanEverTick = true;

Mesh = CreateDefaultSubobject<UStaticMeshComponent>("Mesh");
RootComponent = Mesh;

CoinSoundComponent =
CreateDefaultSubobject<UAudioComponent>("CoinSoundComponent");
CoinSoundComponent->SetupAttachment(RootComponent);
CoinSoundComponent->SetSound(CoinSound);

NiagaraComponent =
CreateDefaultSubobject<UNiagaraComponent>("NiagaraComponent");
NiagaraComponent->SetupAttachment(RootComponent);
NiagaraComponent->Deactivate();

Mesh->OnComponentBeginOverlap.AddDynamic(this,
&AGameProjectItemBase::OverLapBegin)
```

Ispis 16: Konstruktor

`PrimaryActorTick.bCanEverTick` omogućuje osvježavanje objekta tijekom igre, što znači da će metoda `Tick` biti pozvana svaki frame ako je potrebno. `CreateDefaultSubobject` stvara novu instancu klase `UStaticMeshComponent` koja će se koristiti kao vizualni prikaz objekta. Metoda `CreateDefaultSubobject` se koristi za stvaranje podrazumijevanih pod objekata klase, koji će biti povezani s glavnom objektom (u ovom slučaju klasa `AGameProjectTrapBase`).

`RootComponent` postavlja mesh komponentu kao korijensku komponentu ovog objekta. Korijenska komponenta je glavna komponenta koja drži sve druge komponente koje pripadaju ovom objektu. Postavljanjem mesh komponente kao korijenske komponente, ona postaje centralna komponenta koja će služiti kao osnova za ostale komponente ovog objekta. Sve ostale komponente (npr. kolizijski okviri, zvukovi, itd.) mogu biti pričvršćene na ovu mesh komponentu ili na druge komponente koje su već pričvršćene na nju.

Metoda `BeginPlay` je naslijeđena iz klase `AActor` i poziva se kada se objekt prvi put pojavi u igri. U ovom slučaju, metoda je prazna i nije potrebno izvršiti dodatne

radnje kada objekt počinje igrati. `Tick` metoda također je naslijeđena iz klase `AActor` i poziva se svaki frame kako bi se osvježio objekt tijekom igre.

U ovoj implementaciji, poziva se `AddRotationToActor` (ispis 17), koja dodaje rotaciju objektu na temelju vremenskog koraka `DeltaTime`. Rotacijska brzina je postavljena na 45 stupnjeva po sekundi, a zatim se rotacija primjenjuje na objekt pozivom `AddActorLocalRotation`.

```
void AGameProjectItemBase::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
    AddRotationToActor(DeltaTime);
}


void AGameProjectItemBase::AddRotationToActor(float DeltaTime)
{
    float RotationSpeed = 45.0f;
    FRotator RotationDelta(0.0f, RotationSpeed * DeltaTime,
0.0f);
    AddActorLocalRotation(RotationDelta);
}
```

Ispis 17: Metoda `Tick` i `AddRotationToActor`

Tu je još i `OverLapBegin` metoda koja se poziva kada se ovaj objekt preklapa s drugim objektom koji ima koliziju (ispis 18). Metoda prima informacije o preklapanju i drugom objektu `OtherActor`. U ovoj implementaciji, prvo provjerava je li `OtherActor` tipa `AGameProjectCharacter`. Ako je, reproducira zvuk na poziciji ovog objekta i ako postoji „`NiagaraComponent`“ i nije već aktivan, aktivira se i postavlja se na istu poziciju kao mesh komponenta.

Poziva se metoda `ItemCollected` u „GameMode“ klasi (`AgameProjectGameModeBase`) kako bi se obavijestio `GameMode` da je igrač prikupio ovaj objekt. Zatim se ovaj objekt uništava pozivom `Destroy` jer je prikupljen.

```
AGameProjectGameModeBase* GameMode =
Cast<AGameProjectGameModeBase>(GetWorld()->GetAuthGameMode());
if(Cast<AGameProjectCharacter>(OtherActor))
{
    if (CoinSound)
    {
       UGameplayStatics::PlaySoundAtLocation(this, CoinSound,
GetActorLocation());
    }
    if (NiagaraComponent && NiagaraComponent->IsValidLowLevel())
    {
       NiagaraComponent->SetWorldLocation(Mesh-
>GetRelativeLocation());

       if (!NiagaraComponent->IsActive())
       {
          NiagaraComponent->Activate();
       }
    }
    GameMode->ItemCollected();
    Destroy();
}
```

Ispis 18: Kolizija

4.3.2. AGameProjectTrapBase

Ovo je klasa za zamke u igri. Zamke imaju vizualni mesh i koliziju. Zamke mogu biti aktivne ili neaktivne `bTrapActive`. Kada igrač pređe preko zamke dok je aktivna, zamka će reagirati (npr. podići se). Također ima zvuk koji se reproducira kada se zamka aktivira.

U konstruktoru (ispis 19) se postavljaju početne vrijednosti i konfiguracija komponenti za ovu klasu. `PrimaryActorTick.bCanEverTick = false;` onemogućuje osvježavanje objekta tijekom igre, što znači da metoda `Tick` neće biti pozvana tijekom igre. `TrapCollision` predstavlja kolizijski okvir zamke. Ovaj okvir koristi se za detekciju preklapanja s drugim objektima. `RootComponent` postavlja `TrapCollision` komponentu kao korijensku komponentu ovog objekta.

`TrapMesh` stvara pod objekt klase `UStaticMeshComponent`, koji će prikazivati vizualni prikaz zamke. Postavlja komponentu kao pod komponentu `TrapCollision` komponente. To znači da će `TrapMesh` pratiti `TrapCollision` i kretati i rotirati se zajedno s njom. `TrapSoundComponent` je pod objekt klase `UAudioComponent`, koji će se koristiti za reprodukciju zvuka zamke (ispis 19).

`OriginalMeshLocation` sprema početnu lokaciju `TrapMesh` komponente kako bi se kasnije mogla vratiti na tu poziciju. Varijable `bTrapActive`, `bTrapTriggered`, i `TrapMeshOffsetZ` koriste se za praćenje stanja zamke i pomicanje vizualnog prikaza zamke za određeni iznos `TrapMeshOffsetZ` kada se zamka aktivira (ispis 19).

```
PrimaryActorTick.bCanEverTick = false;

TrapCollision =
CreateDefaultSubobject<UBoxComponent>("TrapCollision");
RootComponent = TrapCollision;

TrapMesh =
CreateDefaultSubobject<UStaticMeshComponent>("TrapMesh");
TrapMesh->SetupAttachment(TrapCollision);

TrapSoundComponent =
CreateDefaultSubobject<UAudioComponent>("ButtonSoundComponent");
TrapSoundComponent->SetupAttachment(RootComponent);
TrapSoundComponent->SetSound(TrapSound);

OriginalMeshLocation = TrapMesh->GetRelativeLocation();

bTrapActive = false;
bTrapTriggered = false;
TrapMeshOffsetZ = 30.0f;
```

Ispis 19: Dio `Trap` konstruktora

Kada dođe do kolizije poziva se `OnTrapOverlapBegin` metoda (ispis 20). Ako je `bTrapActive` postavljen na `false`, tada se provjerava je li preklapanje s igračem `PlayerCharacter`. U tom slučaju, postavlja se `bTrapActive` na `true`, što označava da je zamka aktivirana. Kad je `bTrapActive` već `true`, tada metoda

provjerava ponovno preklapanje s igračem. Ako je `bTrapTriggered` također `true`, to znači da je zamka već aktivirana i igrač je ostao u zamci, pa se poziva metoda `Die` na igraču, što znači da igrač umire. Zatim se `bTrapTriggered` postavlja na `false` kako bi se osiguralo da se metoda `Die` ne poziva više puta dok je igrač u zamci.

```
AGameProjectCharacter* PlayerCharacter =
Cast<AGameProjectCharacter>(OtherActor);
if (!bTrapActive)
{
    if (PlayerCharacter)
    {
        bTrapActive = true;
    }
}
else
{
    if (PlayerCharacter && bTrapTriggered)
    {
        PlayerCharacter->Die();
        bTrapTriggered = false;
    }
}
```

Ispis 20: Metoda `OnTrapOverlapBegain`

Nakon toga kada se prestaje preklapati s ovom zamkom poziva se metoda `OnTrapOverlapEnd` (ispis 21). Ako je `bTrapActive` `true`, to znači da je zamka već aktivirana i provjerava se je li preklapajući objekt igrač. Tada se reproducira zvuk zamke, a `TrapMesh` se postavlja na početnu lokaciju `TrapMeshOffsetZ`, čime se postiže efekt da se zamka podiže nakon što je igrač prošao preko nje. Također, `bTrapTriggered` se postavlja na `true` kako bi se mogla obraditi situacija kada igrač ostane u zamci.

```
if (bTrapActive && OtherActor && OtherActor-
>IsA<AGameProjectCharacter>())
{
    if (TrapSound)
    {
        UGameplayStatics::PlaySoundAtLocation(this, TrapSound,
GetActorLocation());
    }
    TrapMesh->SetRelativeLocation(OriginalMeshLocation +
FVector(0,0,TrapMeshOffsetZ));
    bTrapTriggered = true;
}
```

Ispis 21: Metoda `OnTrapOverlapEnd`

4.3.3. AGameProjectARotatingPlatform

Klasa predstavlja rotirajuću platformu u igri. Platforma ima također vizualni mesh koji se može rotirati oko svoje osi s određenom brzinom i kutom rotacije. Također ima koliziju za detekciju sudara s drugim objektima. Platforma ima zvuk koji se reproducira dok se rotira. Također ima funkcije za obradu preklapanja i rotiranje platforme.

Osim što ima `PlatformCollision`, `PlatformMesh`, `PlatformSoundComponent` definirano u konstruktori tu su još i varijable `OriginalMeshRotation`, `RotationAngle`, `RotationSpeed`, `TargetRotation` i `bIsRotating` koriste se za praćenje stanja rotacije platforme i upravljanje rotacijom (ispis 22).

```
OriginalMeshRotation = PlatformMesh->GetRelativeRotation();
RotationAngle = 90.0f;
RotationSpeed = 1.0f;
TargetRotation = FRotator(0, RotationAngle, 0);
bIsRotating = false;
```

Ispis 22: Dio konstruktora platforme

`bIsRotating` se postavlja na `true` u metodi `OnoverlapBegin` ako se preklapa s igračem, što znači da će se platforma kasnije rotirati kaka prestane preklapanje jer se tada poziva metoda `OnOverlapEnd`. Tada se reproducira zvuk platforme, platforma se rotira na sljedeću stranu, i postavlja `bIsRotating` na `false` kako bi se osiguralo da se rotacija ne odvija neprekidno (ispis 23).

```
AGameProjectCharacter* PlayerCharacter =
Cast<AGameProjectCharacter>(OtherActor);
if (bIsRotating && PlayerCharacter)
{
    if (PlatformSound)
    {
        UGameplayStatics::PlaySoundAtLocation(this,
PlatformSound, GetActorLocation());
    }
    PlatformMesh->SetRelativeRotation(OriginalMeshRotation +
TargetRotation);
    RotationAngle += 90.0f;
    TargetRotation = FRotator(0, RotationAngle, 0);
    bIsRotating = false;
}
```

Ispis 23: Metoda `OnOverlapEnd`

4.3.4. AgameProjectARotatingButton

Predstavlja gumb u igri. Ovaj gumb ima dvije glavne funkcije: aktiviranje rotirajućih platformi kada igrač preklopi s njim, te prilagodbu svog vizualnog stanja kad se pritisne.

Kao i do sad u ovom konstruktoru ima `ButtonCollision`, `ButtonMesh`, `ButtonSoundComponent` te varijable `OriginalMeshLocation`, `bIsBtnPressed`, `bIsBtnActive` i `ButtonMeshOffsetZ` slične kao i kod klase `AGameProjectARotatingPlatform` koriste se za praćenje stanja gumba i upravljanje njegovim vizualnim stanjem.

Metoda `OnOverlapBegin` (ispis 24) se poziva kada se objekt preklapa s drugim objektom, u ovom slučaju, s igračem. Ako je gumb aktivan i ako gumb nije pritisnut, tada se gumb označava kao aktivan, a zatim se aktiviraju sve rotirajuće platforme `RotatingPlatform->RotatePlatform` (ispis 25) koje su prethodno registrirane i povezane s ovim gumbom. Također, postavlja se tajmer koji će nakon 0.5 sekundi pozvati metodu `ResetButtonActivity`, koja će ponovno postaviti `bIsBtnActive` na `false` kako bi se spriječilo ponavljanje aktivacije platformi pri ponovnom preklapanju s igračem.

```
AGameProjectCharacter* PlayerCharacter =
Cast<AGameProjectCharacter>(OtherActor);
if (bIsRotating && PlayerCharacter)
{
    if (PlatformSound)
    {
        UGameplayStatics::PlaySoundAtLocation(this,
PlatformSound, GetActorLocation());
    }
    PlatformMesh->SetRelativeRotation(OriginalMeshRotation +
TargetRotation);
    RotationAngle += 90.0f;
    TargetRotation = FRotator(0, RotationAngle, 0);
    bIsRotating = false;
}
```

Ispis 24: Metoda `OnOverlapBegin`

```
void AGameProjectARotatingPlatform::RotatePlatform()
{
    PlatformMesh->SetRelativeRotation(OriginalMeshRotation +
TargetRotation);
    RotationAngle += 90.0f;
    TargetRotation = FRotator(0, RotationAngle, 0);
}
```

Ispis 25: Metoda `RotatePlatform` u `AGameProjectARotatingPlatform` klasi

Za promjenu stanja gumba korist se `UpdateButtonState` metoda koja ažurira stanje gumba, tj. mijenja njegovo vizualno stanje i postavlja zastavicu `bIsBtnPressed` ovisno o tome je li gumb pritisnut ili ne (ispis 26).

```
void AGameProjectARotatingButton::UpdateButtonLocation(bool
bIsPressed)
{
    if (bIsPressed)
    {
        ButtonMesh->SetRelativeLocation(OriginalMeshLocation);
    }
    else
    {
        ButtonMesh->SetRelativeLocation(OriginalMeshLocation +
FVector(0, 0, ButtonMeshOffsetZ));
    }
}

void AGameProjectARotatingButton::UpdateButtonState(bool
bIsPressed)
{
    if (!bIsPressed)
    {
        UpdateButtonLocation(!bIsPressed);
        bIsBtnPressed = true;
    }
    else
    {
        UpdateButtonLocation(!bIsPressed);
        bIsBtnPressed = false;
    }
}
```

Ispis 26: Metoda `UpdateButtonState`

Dok `CheckButtonState` metoda (ispis 27) provjerava stanje svih gumba u igri. Ako pronađe neki drugi gumb, poziva metodu `UpdateButtonState` na tom gumbu kako bi osigurala da svi gumbi imaju ispravno ažurirano stanje.

```
void AGameProjectARotatingButton::CheckButtonState()
{
    TArray<AActor*> Buttons;
    UGameplayStatics::GetAllActorsOfClass(GetWorld(),
StaticClass(), Buttons);

    for (AActor* button : Buttons)
    {
        AGameProjectARotatingButton* RotatingButton =
Cast<AGameProjectARotatingButton>(button);
        if (RotatingButton)
        {
            RotatingButton->UpdateButtonState(RotatingButton-
>bIsBtnPressed);
        }
    }
}
```

Ispis 27: Metoda `CheckButtonState`

4.3.5. AGameProjectKeyPickup i AGameProjectLockInteraction

Ovo je klasa za prikupljanje ključeva i klasa za interakciju s bravom u igri. Ključevi imaju različite tipove (crveni, plavi, sivi) koji se određuju enumeracijom `EKeyType`. Ključevi također imaju vizualni mesh i koliziju. Kad igrač prikupi ključ, reproducira se zvuk. Također i brave imaju različite tipove (crvena, plava, siva) koji se određuju enumeracijom `ELockType`. Imaju i vizualni mesh i koliziju. Kada igrač ima odgovarajući ključ, može interaktirati s bravom, što će reproducirati zvuk interakcije.

`enum class EKeyType` definira enumeraciju `EKeyType` (ispis 28) s tri moguća tipa ključa „RedKey“, „BlueKey“ i „GreyKey“. Ova enumeracija će se koristiti za označavanje tipa ključa koji ovaj objekt predstavlja.

```
UENUM(BlueprintType)
enum class EKeyType : uint8
{
    KeyType1 UMETA(DisplayName = "RedKey"),
    KeyType2 UMETA(DisplayName = "BlueKey"),
    KeyType3 UMETA(DisplayName = "GreyKey"),
};
```

Ispis 28: `EKeyType`

Glavna metoda OnKeyOverlap se poziva kada se objekt preklapa s drugim objektom, u ovom slučaju, s igračem. Ako je igrač preklopio s ključem, metoda reproducira zvuk ključa, postavlja odgovarajuću zastavicu za tip ključa (prema enumeraciji `EKeyType`) za igrača i uništava sam objekt ključa (ispis 29).

```
AGameProjectCharacter* PlayerCharacter =
Cast<AGameProjectCharacter>(OtherActor);
if (PlayerCharacter)
{
    if (KeyPickupSound)
    {
       UGameplayStatics::PlaySoundAtLocation(this,
KeyPickupSound, GetActorLocation());
    }
    switch (KeyType)
    {
    case EKeyType::KeyType1:
       PlayerCharacter->bHasKeyRed = true;
       break;
    case EKeyType::KeyType2:
       PlayerCharacter->bHasKeyBlue = true;
       break;
    case EKeyType::KeyType3:
       PlayerCharacter->bHasKeyGrey = true;
       break;
    }
    Destroy();
}
```

Ispis 29: Metoda `OnKeyOverlap`

Implementaciju metode `OnDoorOverlap` u klasi `AGameProjectLockInteraction` koristi naredbu `switch` kako bi se ovisno o tipu

`LockType` izvršila odgovarajuća provjera. Na primjer ako igrač ima ključ tipa `RedKey` `PlayerCharacter->bHasKeyRed` je `true`, tada će ključ biti potrošen (postavlja se na `false`), reproducirati će se zvuk odključavanja i objekt će biti uništen `Destroy`. Ako igrač nema odgovarajući ključ, ništa se ne događa (ispis 30).

```
AGameProjectCharacter* PlayerCharacter =
Cast<AGameProjectCharacter>(OtherActor);
if (PlayerCharacter)
{
    switch (LockType)
    {
    case ELockType::LockType1:
       if (PlayerCharacter->bHasKeyRed)
       {
          PlayerCharacter->bHasKeyRed = false;
          if (LockInteractionSound)
          {
             UGameplayStatics::PlaySoundAtLocation(this,
LockInteractionSound, GetActorLocation());
          }
          Destroy();
       }
       break;
    case ELockType::LockType2:
       if (PlayerCharacter->bHasKeyBlue)
       {
          PlayerCharacter->bHasKeyBlue = false;
          if (LockInteractionSound)
          {
             UGameplayStatics::PlaySoundAtLocation(this,
LockInteractionSound, GetActorLocation());
          }
          Destroy();
       }
       break;
    case ELockType::LockType3:
       if (PlayerCharacter->bHasKeyGrey)
       {
          PlayerCharacter->bHasKeyGrey = false;
          if (LockInteractionSound)
          {
             UGameplayStatics::PlaySoundAtLocation(this,
LockInteractionSound, GetActorLocation());
          }
          Destroy();
```

```
        }
        break;
    }
}
```

Ispis 30: Metoda `OnDoorOverlap`

Tu je i `AddMovementsToActor` metoda koja dodaje animaciju ključa tijekom igre, rotirajući ključ oko Y osi i oscilirajući po Z osi kako bi stvorio vizualni efekt tijekom igre (ispis 31).

```
void AGameProjectKeyPickup::AddMovementsToActor(float DeltaTime)
{
    float RotationSpeedYaw = -45.0f;
    float OscillationSpeed = 2.0f;
    float OscillationAmplitude = 15.0f;

    FRotator RotationDeltaYaw(0.0f, RotationSpeedYaw *
DeltaTime, 0.0f);
    AddActorLocalRotation(RotationDeltaYaw);

    FVector NewLocation = GetActorLocation();
    float PitchOffset = FMath::Sin(GetGameTimeSinceCreation() *
OscillationSpeed) * OscillationAmplitude;
    NewLocation.Z = 60.0f + PitchOffset;
    SetActorLocation(NewLocation);
}
```

Ispis 31: Metoda `AddMovementsToActor`

5. ZAKLJUČAK

U ovom završnom radu predstavljeno je istraživanje i razvoj igre koristeći mogućnosti Unreal Engine razvojnog okruženja. Rad je pružio uvid u proces izrade igre kroz upotrebu C++ programskog jezika i vizualnog skriptiranja uz korištenje gotovih biblioteka iz pogonskog alata. Cilj je bio pokazati različite načine implementacije funkcionalnosti i prikazati prednosti i svrhu svakog pristupa.

Unutar ovog rada naglasak je bio na programiranju, dok je izgled likova, okoline i zvuka riješen korištenjem gotovih dodataka. Iako to može utjecati na grafičku optimizaciju igre, omogućilo je brži razvoj i testiranje bez potrebe za naprednim znanjem programiranja.

Izrada igre predstavlja zahtjevan zadatak, koji zahtijeva kreativnost u stvaranju priče, ugođaja i zanimljivih igračkih mehanika. Unreal Engine je pružio snažan alat za olakšavanje razvoja igre, omogućivši fokusiranje na implementaciju originalnih ideja bez gubljenja previše vremena na tehničke aspekte.

Kombinirajući znanje programskog jezika C++ i vizualnog skriptiranja, moguće je prilagoditi igru prema potrebama projekta, pazeći na optimizaciju i resurse. Istovremeno, vizualni skriptni jezik nudi jednostavniji pristup za one koji nemaju programersko iskustvo, što može biti korisno prilikom testiranja ili za brzi prototipiranje.

U konačnici, Unreal Engine se pokazao kao snažno razvojno okruženje koje može značajno olakšati proces izrade igara, bez obzira na iskustvo programiranja, nudeći širok spektar mogućnosti za implementaciju raznih ideja.

Dodaci, animacije i zvukovi su besplatno preuzete sa stranice `https://kenney.nl/`, koji su dodatno obrađeni radi potrebe projekta. Izrada projekta provedena je uz pomoć vodiča s `https://www.youtube.com` i službene dokumentacije Unreal Enginea, što je služilo kao relevantan izvor informacija i pomoći prilikom razvoja igre.

LITERATURA

1. Unreal Engine 5.2 Documentation
   https://docs.unrealengine.com/5.2/en-US/