3D SIMULACIJSKA IGRA ŽIVOTA U TREĆEM LICU -SKIMS

Topić, Anica

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

ANICA TOPIĆ

Z A V R Š N I R A D

3D SIMULACIJSKA IGRA ŽIVOTA U TREĆEM LICU – SKIMS

Split, rujan 2024.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

Predmet: Objektno programiranje

Z A V R Š N I R A D

Kandidat: Anica Topić

Naslov rada: 3D simulacijska igra života u trećem licu – SKIMS

Mentor: Ljiljana Despalatović, viši pred.

Split, rujan 2024.

Sadržaj

Sažetak

Završni rad opisuje razvoj i implementaciju 3D igre pod nazivom *Skims*, razvijene za igranje u trećem licu. Igra je napravljena korištenjem Unreal Enginea 5 i Visual Studija 2022, a u razvoju su korišteni i vizualno programiranje te programski jezik C++.

Unreal Engine 5 je odabran zbog svojih moćnih funkcionalnosti i širokog spektra opcija unutar uređivača, uključujući revolucionarne tehnologije koje omogućuju stvaranje izuzetno detaljnih i realističnih 3D okruženja.

Razvoj igre se temelji na kombinaciji objektno orijentiranog programiranja u C++ jeziku i vizualnog programiranja putem *blueprint* sustava u Unreal Engineu. Iako je moguće razviti igru koristeći samo *blueprint* sustav, on služi kao dopuna C++ jeziku, a ne kao zamjena. *Blueprint* sustav omogućuje prilagođavanje svojstava u uređivaču, čime se izbjegava tvrdo kodiranje (engl. *hard coding*), koje se smatra lošom praksom u razvoju softvera.

U radu je naglasak stavljen na razvoj i implementaciju lika koji simulira osnovne ljudske funkcije poput hranjenja, higijene i spavanja, pri čemu je posebna pažnja posvećena numeričkim statistikama i funkcionalnostima tog lika.

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

SUMMARY

A third-person 3D life simulation game titled *Skims*

The final paper describes the development and implementation of a 3D game titled *Skims*, designed for third-person play. The game was created using Unreal Engine 5 and Visual Studio 2022, with visual programming and the C++ programming language also employed during development.

Unreal Engine 5 was chosen for its powerful features and extensive range of options within the editor, including groundbreaking technologies, which allow for the creation of highly detailed and realistic 3D environments.

The game development is based on a combination of object-oriented programming in C++ and visual programming through the Blueprint system in Unreal Engine. While it's possible to develop the game using only the Blueprint system, it serves as a supplement to C++ rather than a replacement. The Blueprint system allows for property adjustments within the editor, thus avoiding hard coding, which is considered poor practice in software development.

The paper places special emphasis on the development and implementation of the character, which simulates basic human functions such as eating, hygiene, and sleeping. More specifically, it focuses on the character's numerical statistics and functionalities.

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

1. Uvod

Svijet videoigara skromno je započeo 1950-ih s jednostavnim igrama kao što su *Tennis for Two* i *Spacewar!* na *mainframe* računalima. Već 1979. godine je nastalo prvo takozvano "uskrsno jaje" (engl. *easter egg*) kojeg je ubacio Atari programer, Warren Robinett, u svoje igre zbog manjka akreditacije za svoj rad [1]. Ubrzo nakon toga, 1980-ih, industrija je eksplodirala s hitovima poput *Pac-Man* i sve popularnijim kućnim konzolama. Od tada, svijet videoigara neprestano raste i danas je već ključan dio naše svakodnevice.

Videoigre su značajno napredovale, a ključnu ulogu u tom razvoju odigrao je *Unreal Engine*, kojeg je Epic Games prvi put predstavio 1998. godine. Unreal Engine 3 postao je temelj za popularne naslove poput *Gears of War*, dok su Unreal Engine 4 i Unreal Engine 5 donijeli igre poput *Fortnite*, *Remnant 2* i *Black Myth: Wukong*. Osim u igrama, Unreal Engine se koristi i u filmskoj produkciji i arhitekturi zbog vrhunske vizualne kvalitete. Cilj ovog rada je istražiti mogućnosti razvoja simulacijskih igara kroz izradu 3D simulacijske igre života u trećem licu, koristeći Unreal Engine 5. *Skims* predstavlja jedinstvenu interpretaciju postojećih simulacijskih igara, fokusirajući se na kombinaciju vizualnog skriptiranja i programskog jezika C++, što omogućuje preciznu kontrolu nad mehanikama igre, interakcijom s okolinom i stvaranjem detaljnog 3D okruženja.

U ovom radu detaljno je opisan proces razvoja 3D simulacijske igre *Skims*. Prvo poglavlje opisuje specifikaciju igre i postavlja osnovne mehanike i ciljeve. Drugo poglavlje fokusira se na razvojne alate korištene za izradu igre, uključujući Unreal Engine 5, Visual Studio 2022 i Blender. Treće poglavlje opisuje ključne funkcionalnosti igre, kao što su upravljanje statistikama lika, implementacija interakcije s objektima i dizajn korisničkog sučelja. Na kraju, u zaključku su navedeni najvažniji rezultati i potencijalne mogućnosti za daljnji razvoj igre.

Sva 3D sredstva, izuzev glavnog lika Stiffya, preuzeta su s Unreal Engine Marketplacea. Animacije su preuzete s Mixama i dodatno prerađene za projekt. Izrada projekta ostvarena je uz pomoć službene dokumentacije za Unreal Engine, YouTubea i uz knjigu "Elevating Game Experiences with Unreal Engine 5".

2. Specifikacija igre

Naziv: *Skims*

Platforma: Računalo

Žanr: Simulacija života

Perspektiva: Treće lice

Kratki opis: Igra je inspirirana tematikom Noći vještica. U njoj igrač preuzima ulogu Stiffyja, simpatičnog kostura koji obitava u ukletoj kući unutar sablasnog gradića. Igrač upravlja dnevnim aktivnostima Stiffya, uključujući hranjenje, održavanje higijene i spavanje, dok istražuje svoju okolinu (slika 1). Cjelokupni svijet donosi mračnu atmosferu s mnoštvom vizualnih detalja i dinamičkim osvjetljenjem koje mijenja atmosferu igre u skladu s ciklusom dana i noći.

Slika 1: Prikaz igre i Stiffyja

Cilj igre: Igrač upravlja likom koji mora imati zadovoljene osnovne ljudske potrebe poput hranjenja, higijene i spavanja. Igra započinje u namještenoj kući lika, koji je slobodan istraživati okolinu. Ako igrač ne uspije održati lika na životu, igra se mora ponovno započeti.

Mehanike igre:

- **Kretanje i kontrola nad likom:** Igrač kontrolira kretanje lika kroz okoliš, što uključuje navigaciju kroz različite dijelove okoline.
- **Upravljanje potrebama lika:** Lik ima potrebe poput gladi, higijene i sna koje igrač mora redovito zadovoljavati.
- **Interakcija s okolišem:** Igrač može izvoditi radnje kao što su konzumiranje hrane, korištenje WC-a i spavanje, što je predstavljeno kroz interakciju s objektima u okolišu. Ove radnje su ključne za zadovoljenje potreba lika.
- **Ciklus dana i noći:** Okoliš ima ciklus dana i noći koji utječe na osvjetljenje i atmosferske uvjete.

Okoliš:

- **Modularne građevine i objekti:** Okoliš sadržava modularne strukture i objekte poput ograda, kuća i drugih građevina.
- **Prirodni elementi:** Okoliš sadržava prirodne elemente kao što su drveće, trava, lišće i vodu s pripadajućim efektima i animacijama.
- **Kućni prostor:** Lik živi u kući s određenim rasporedom i dizajnom, uključujući interaktivne objekte poput namještaja i kućanskih aparata. Kuća je ključna za upravljanje potrebama lika i pruža razne interakcije.

Korisničko sučelje:

- **Glavni meni:** Omogućuje pokretanje nove igre i izlaz iz igre.
- **Numeričke statistike:** Na ekranu su prikazane ključne statistike glavnog lika, uključujući informacije o gladi, energiji, higijeni i drugim relevantnim podacima.

- **Sat:** Prikazuje trenutno vrijeme u igri i prati ciklus dana i noći, što utječe na atmosferu igre i njeno osvjetljenje.

Stil: Igra donosi šarmantan ugođaj nalik Noći vještica, spajajući mistične i slatke elemente. Kuće i okoliš su osmišljeni s posebnim teksturama i bojama koje stvaraju tajanstvenu i prijatnu atmosferu. Svaki detalj doprinosi stvaranju ugođaja koji je istovremeno sablastan i šarmantan, a glavni lik, izrađen pomoću Blendera, dodatno obogaćuje ovo iskustvo svojom jedinstvenom i prilagođenom izradom.

Glazba: Atmosferska glazba dodaje dubinu igri i pojačava cjelokupni doživljaj. Melodije su osmišljene kako bi odgovarale mističnom i slatkom tonu igre, stvarajući ugođaj koji je i uzbudljiv i misteriozan.

3. Razvojni alati

3.1. Unreal Engine 5

Unreal Engine 5, službeno predstavljen u travnju 2022. donosi niz revolucionarnih funkcionalnosti koje pružaju realističan doživljaj u igrama. Među najistaknutijim značajkama su Nanite i Lumen. Nanite je virtualizirani sustav geometrije koji omogućuje prikaz izuzetno detaljnih 3D modela s milijunima poligona, dok dinamički prilagođava broj poligona ovisno o udaljenosti objekta od kamere. Bliži objekti zadržavaju visoku razinu detalja, dok se daleki objekti optimiziraju za bolje performanse. Lumen, s druge strane, pruža potpuno dinamičnu globalnu iluminaciju, prilagođavajući se promjenama u scenama i osvjetljenju u stvarnom vremenu, omogućujući tako stvaranje dinamičnih i fleksibilnih okruženja bez potrebe za prethodnim izračunima osvjetljenja.

Unreal Engine integrira dva moćna alata za programiranje i skriptiranje, omogućavajući razvoj igara putem C++ jezika i vizualnog skriptiranja (engl. *blueprints*). C++ u Unreal Engineu omogućuje programerima implementaciju složenih mehanika igre i optimizaciju performansi za zahtjevne aplikacije. Vizualno skriptiranje, s druge strane, pruža brz i jednostavan način za kreiranje značajki u igrama bez potrebe za opsežnim znanjem o kodiranju. Korištenje programskog jezika C++ i *blueprintsa* zajedno omogućava timovima da iskoriste prednosti svakog pristupa, omogućujući brzo prototipiranje uz pomoć *blueprintsa* dok se održava fleksibilnost programskog jezika C++ za složenije sustave.

Unreal Editor je razvojno okruženje za izradu igara u Unreal Engineu. Omogućuje kreiranje i uređivanje okruženja, likova i animacija u stvarnom vremenu. Uz editorovu pomoć lako je dizajnirati nivoe i koristiti vizualno skriptiranje. Uz napredne značajke poput dinamičkog osvjetljenja i oblikovanja terena, Unreal Editor pomaže developerima i ljubiteljima igara da izrealiziraju svoje ideje.

Inače, ako se kôd mijenja dok je Unreal Editor pokrenut, znale bi se događati greške koje su nepredvidljive, zato je napravljen *live coding*. *Live coding* je značajka koja omogućuje programerima da mijenjaju C++ kôd dok je Unreal Editor pokrenut, što dopušta korisniku da odmah vidi promjene koje su učinjene. U protivnom bi se trebao pokrenuti ponovno cijeli projekt, što uvelike produžuje proces testiranja i iteracije promjena.

3.1.1 Kreiranje projekta u Unreal Enginu 5

Kreiranje projekta u Unreal Engineu 5 može biti vrlo jednostavno. Instalacija Unreal Enginea 5 se obavlja tako da se skine željena verzija programa sa Unreal Engine stranice.

U pregledniku koji se otvori nakon pokretanja programa može se odabrati prazan projekt ili jedan od ponuđenih predložaka kao što su: pogled iz prvog lica (engl. *first person*), pogled iz trećeg lica (engl. *third person*), pogled iz ptičje perspektive (engl. *top down*).

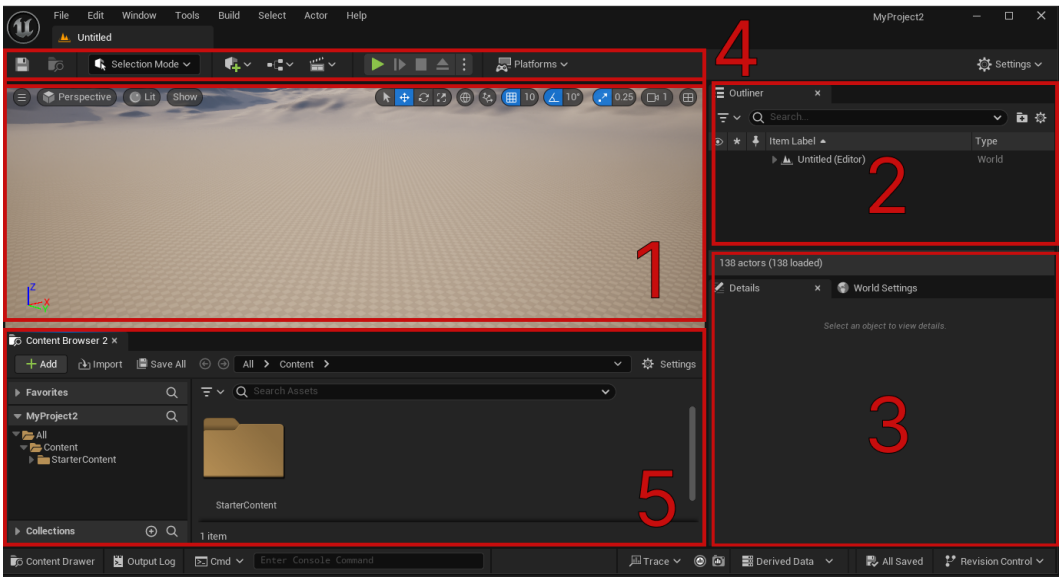
Nakon toga se postavljaju opcije projekta, kao što je odabir platforme. Opcije su: računalo, konzola ili mobilni uređaj. Uz to se može odabrati i kvaliteta i odrediti lokacija i ime projekta. Ako je instalirane više od jedne verzije programa, može se odabrati željena verzija za projekt.

Ključna odluka u razvoju igre je odabir načina razvoja. Može se birati između C++ programiranja i vizualnog skriptiranja (engl. *blueprints*). Ako se odabere jedna od tih opcija, svakako je moguće nadograditi projekt da se može koristiti i druga opcija.

U ovom slučaju korištena je C++ opcija razvoja i predložak pogleda trećeg lica.

3.1.2 Upoznavanje s Unreal Editorom

Upoznavanje sa sučeljem Unreal Enginea ključno je za uspješan rad na razvoju igara. Sučelje (slika 2) sastoji se od nekoliko panela koji omogućuju pristup raznim funkcionalnostima uređivača. U tablici 1 se nalazi detaljna razrada slike 2.



Slika 2: Sučelje Unreal Editora

Tablica 1: Detaljnije objašnjenje slike 2

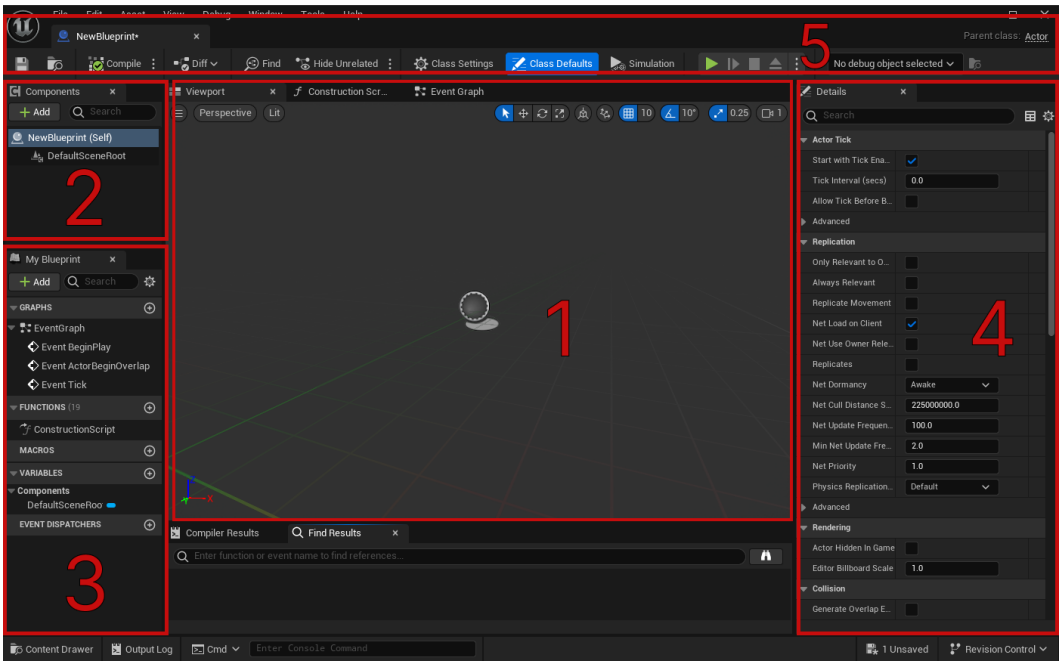
Broj	Naziv	Opis
1	Prikaz scene (engl. *viewport*)	Prikazuje sve objekte dodane na 3D scenu.
2	Prikaz objekata (engl. *outliner*)	Prikazuje hijerarjhiju svih objekata u sceni.
3	Prikaz svojstava objekata (engl. *details*)	Prikazuje i omogućava izmjenu detalja odabranog objekta.
4	Alatna traka (engl. *toolbar*)	Prikazuje alate za izvršavanje radnji unutar uređivača.
5	Sadržajni izbornik (engl. *content browser*)	Prikazuje i omogućava upravljanje svim sadržajem dostupnim unutar projekta.

Prilagodba sučelja u Unreal Editoru omogućava korisnicima da personaliziraju radno okruženje. Kroz promjenu veličine, pozicioniranje i skrivanje pojedinih panela, korisnici mogu stvoriti uvjete koje poboljšavaju produktivnost i olakšavaju pristup često korištenim alatima. Ova fleksibilnost pomaže u ubrzanju razvoja igara i realizacije ideje.

3.1.3 Upoznavanje s *Blueprint* uređivačem

Blueprint uređivač je poseban pod-uređivač unutar Unreal Enginea za *blueprint* klase. Ovdje se mogu urediti svojstva, logika i vizualni izgled za *blueprint* klase, kao i onih nadređenih klasa [2].

Blueprint klase u Unreal Engineu su vizualno skriptirani objekti koji omogućuju korisnicima stvaranje prilagođenih objekata, logike igre i interakcija korištenjem čvorova (engl. *nodes*) unutar *Blueprint* uređivača. Korisnici mogu definirati ponašanje i karakteristike tih objekata na intuitivan način. Vizualno skriptiranje je bitan dio Unreal Enginea, zato je izrazito bitno upoznati se sa sučeljem tog uređivača. Sučelje (slika 3) je sastavljeno tako da korisnici bez programerskog znanja mogu intuitivno stvarati složene interakcije i mehanike pomoću čvorova. U tablici 2 se nalazi detaljna razrada slike 3.



Slika 3: Sučelje Blueprint Editora

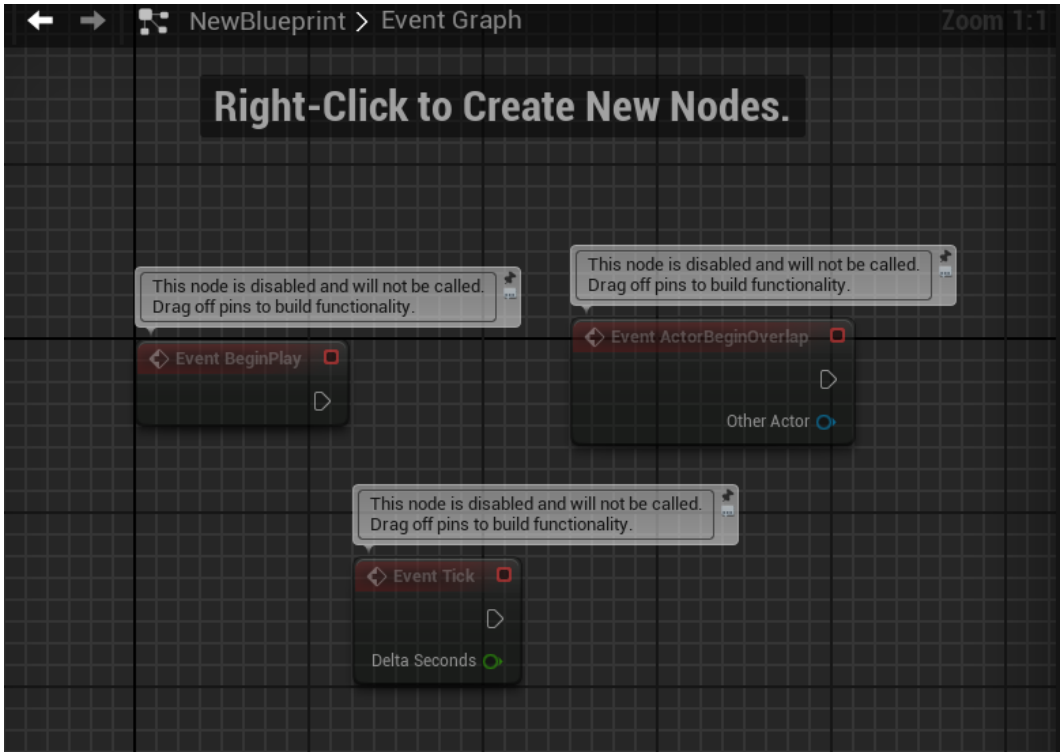
Tablica 2: Detaljnije objašnjenje slike 3

Broj	**Naziv**	**Opis**
1	Prikaz scene	Omogućuje prikaz i uređivanje komponenti u klasi. Sadrži još dvije kartice (engl. *tab*), dijagram događaja (engl. *event graph*) i konstrukcijsku skriptu.
2	Prikaz komponenata (engl. *components*)	Omogućuje prikaz, dodavanje i brisanje svih komponenti u klasi.
3	Prozor "Moj blueprint" (engl. *my blueprint*)	Prikazuje popis svih varijabli, funkcija i događaja unutar klase.
4	Prikaz detalja	Prikazuje i omogućava uređivanje svojstava odabranih čvorova ili komponenti.
5	Alatna traka	Prikazuje alate za izvršavanje radnji unutar uređivača klase.

Nadređena klasa (engl. *parent class*) *blueprint* klase prikazuje se u gornjem desnom kutu

uređivača *blueprinta*. Klikom na naziv nadređene klase otvara se odgovarajuća *blueprint* klasa u uređivaču Unreal Enginea ili C++ klasa u Visual Studiju [2].

Dijagram događaja (slika 4) je mjesto gdje se piše sav kôd za vizualno skriptiranje, stvaraju varijable i funkcije te pristupa drugim varijablama i funkcijama deklariranim u roditeljskoj klasi. Odabirom kartice dijagrama događaja, koja se nalazi desno od kartice prikaza, otvorit će se prozor dijagrama događaja umjesto prozora prikaz.



Slika 4: Sučelje dijagrama događaja

Unutar te kartice, skriptiranje se temelji na spajanju čvorova putem njihovih točaka priključka (engl. *pin*) čime se definiraju akcije unutar igre. Postoje točke izvršenja (engl. *execution*) i varijabilne točke. Točke izvršenja koriste se za povezivanje čvorova u nizu, osiguravajući da se izvršavanje događa u pravom redoslijedu, dok varijabilne točke omogućavaju prijenos podataka između čvorova, omogućujući pristup i manipulaciju varijablama unutar skripte. Ovaj vizualni pristup olakšava razumijevanje logike igre i omogućava brzo stvaranje složenih mehanika bez potrebe za pisanjem kôda.

3.2. C++ u Unreal Engineu

C++ je višenamjenski programski jezik koji se široko koristi u razvoju softvera, igara, sustava i aplikacija. Kao proširenje programskog jezika C, dodaje podršku za objektno-orijentirano programiranje, omogućujući programerima da organiziraju kôd u klase i objekte.

Programiranje u C++ jeziku unutar Unreal Enginea slično je standardnom C++ programiranju, gdje se koriste klase, funkcije i varijable definirane uobičajenom C++ sintaksom. Međutim, postoje značajne razlike koje prilagođavaju C++ za razvoj igara u Unreal Engineu. Jedna od glavnih razlika je integracija posebnih igraćih (engl. *gameplay*) klasa, koje omogućuju da se sve izmjene kôda automatski pokažu u Unreal Editoru nakon što su kompajlirane u Visual Studiju. Druga ključna razlika je implementacija sustava refleksije, koji omogućuje dodatnu enkapsulaciju klasa pomoću posebnih makronaredbi. Ova refleksija omogućuje napredne funkcionalnosti unutar Editora i daje Engineu mogućnost upravljanja memorijom, uključujući dinamičko stvaranje i uništavanje objekata u stvarnom vremenu.

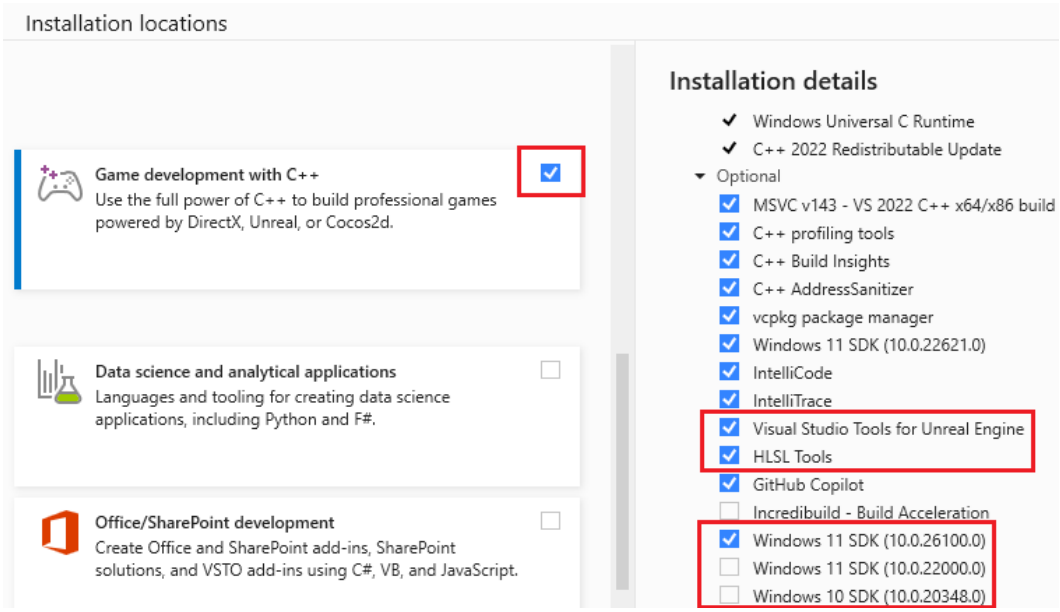
3.3. Visual Studio 2022

Visual Studio je integrirano razvojno okruženje koje nudi raznovrsne alate za razvoj softvera. Podržava širok raspon programskih jezika što ga čini savršenim alatom za različite vrste projekata. Visual Studio pruža funkcionalnosti poput IntelliSense za pametno dovršavanje kôda, integriranih alata za testiranje, upravljanja verzijama i naprednog ispravljanja pogrešaka (engl. *debugging*), čime poboljšava produktivnost korisnika.

Uz komercijalne verzije Visual Studija koje su dostupne već od 1997. godine Microsoft je odlučio objaviti besplatnu verziju 2014. godine. Visual Studio Community je od tada omogućio programerima, studentima i entuzijastima besplatan pristup moćnim alatima koje su dotad bile samo komercijalno dostupne.

Instalacija alata Visual Studio Tools za Unreal Engine unutar Visual Studija omogućuje kreiranje klasa za Unreal Engine, pregledavanje logova te pristup brojnim drugim alatima koji bi inače zahtijevali korištenje samog Unreal Editora.

U Visual Studio instalaciji (slika 5), potrebno je odabrati verziju Visual Studija, zatim kliknuti na "Modify" te pod "Game development with C++" odabrati "Visual Studio Tools for Unreal Engine". Instalacija se dovršava klikom na "Modify".

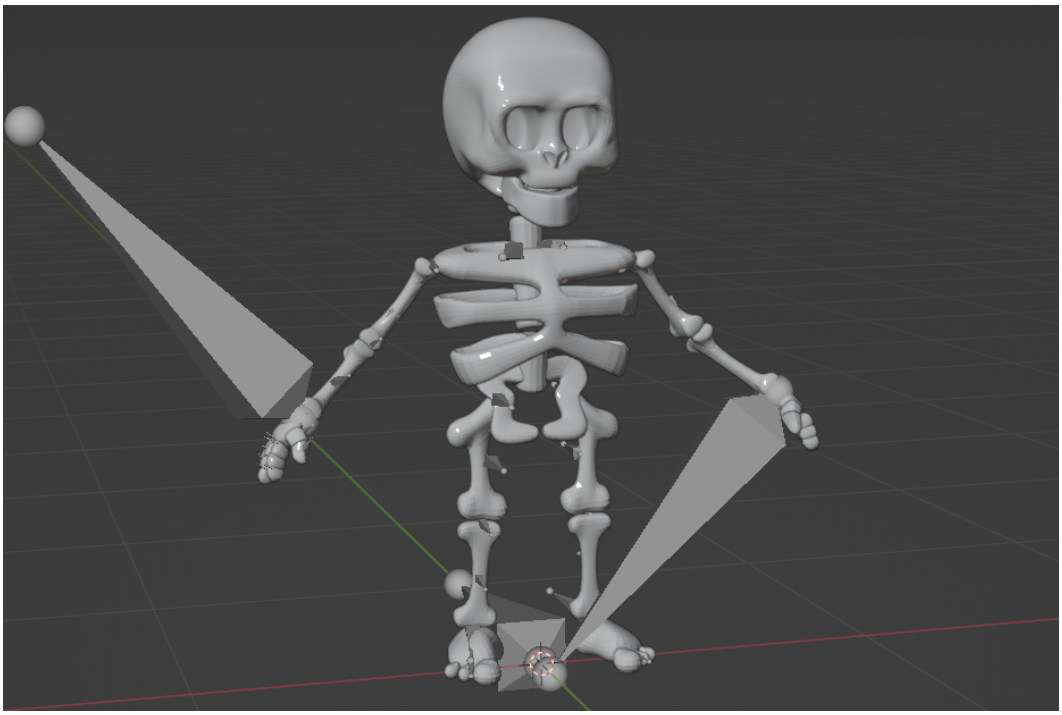


Slika 5: Visual studio installer

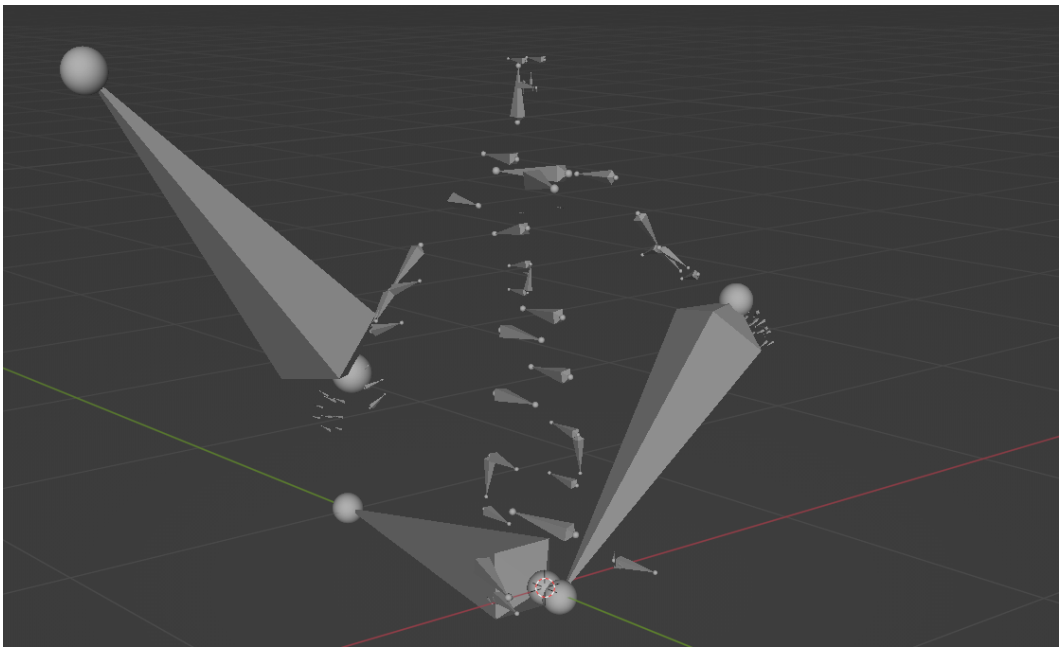
3.4. Blender

Blender je alat koji ima puno različitih upotreba, kao što su 3D modeliranje, animacije i renderiranje. Uz to što je otvorenog kôda, on je ujedno i besplatan, što ga čini dostupnim širokom spektru korisnika, od studenata do profesionalaca u različitim industrijama. Unreal Engine nema ugrađenu mogućnost 3D modeliranja, zato se koriste specijalizirani softveri za 3D modeliranje poput Blendera, čiji se modeli potom mogu uvesti u Unreal Engine za korištenje.

U Blenderu, geometrija (engl. *mesh*) je sastavljena od točaka, rubova i ploha koje definiraju njegov oblik, dok je armatura struktura slična kosturu koja služi za postavljanje kostiju (engl. *rigging*) i animaciju modela. Armatura se sastoji od kostiju koji kontroliraju pokrete geometrije; kada se kost pomakne ili rotira, odgovarajući dio geometrije se također pomiče. Kombinacija geometrije i armature omogućuje stvaranje detaljnih animacija u Blenderu.



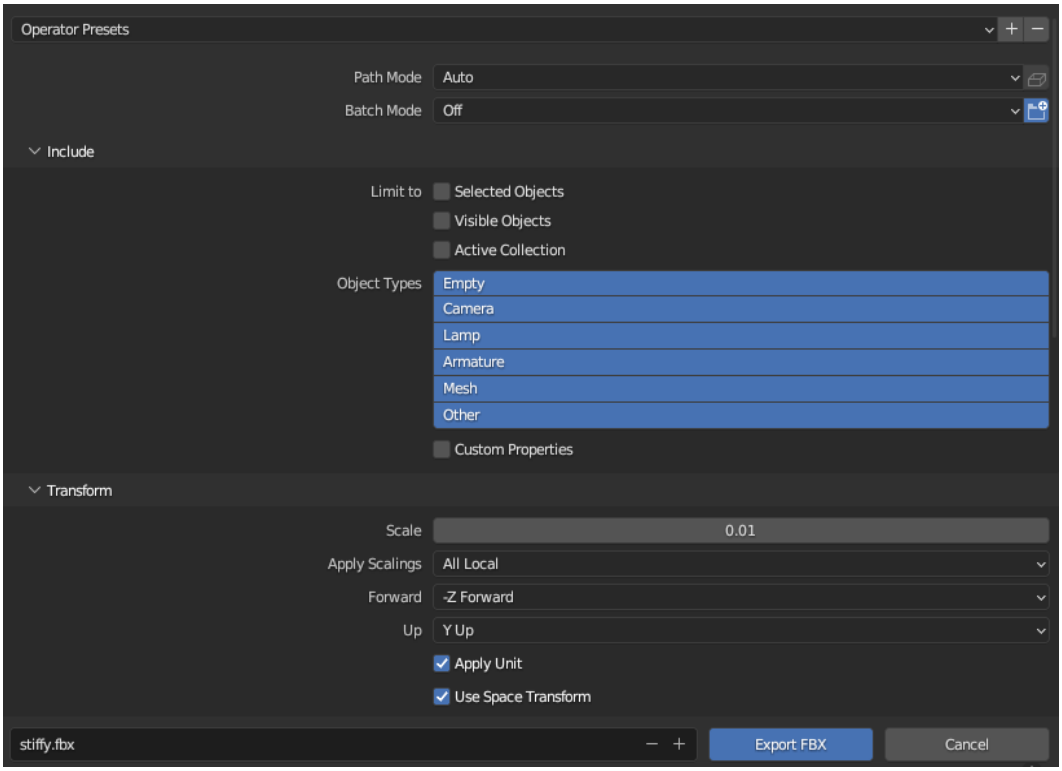
Slika 6: Stiffyjev model i armatura



Slika 7: Stiffyeve kosti

Blender se u ovom slučaju koristio za izradu glavnog lika u igri, Stiffya, zajedno s njegovim kostima koje su esencijalne za animacijski sustav i animacije u Unreal Enginu. Ovaj prilagođeni lik dizajniran je kako bi se uklopio u mističan i simpatičan izgled igre, čime se dodatno naglašava jedinstvena atmosfera i vizualni stil koji igra želi postići.

Prilikom izvoza modela iz Blendera u Unreal Engine, važno je obratiti pažnju na razliku u jedinicama mjere. Blender koristi vlastitu jedinicu mjere, a jedan Blender Unit (BU) često se tretira kao ekvivalent jednog metra. S druge strane, Unreal Engine koristi centimetre kao osnovnu jedinicu mjere. Zbog ove razlike, prilikom izvoza modela iz Blendera u Unreal Engine preporučuje se postavljanje mjere objekta na 0.01 (ili 1/100) u Blenderu, kako bi se osigurala ispravna veličina modela unutar Unreal Enginea (slika 8).



Slika 8: Izvozni prozor Blendera

Izvoz se obavlja u .fbx formatu, jer on pouzdano prenosi geometriju, animacije i kosture, osiguravaju´ci da svi elementi modela budu pravilno uvezeni u Unreal Engine Nakon uvoza modela, preporučuje se provjera mjera kako bi se osiguralo da model ima odgovarajuće dimenzije.

4. Implementacija funkcionalnosti igre

Skims je simulacija stvarnog života u kojoj igrač preuzima kontrolu nad likom po imenu Stiffy. Cilj igre je održavati Stiffyja zdravim i živim, zadovoljavajući njegove osnovne potrebe poput gladi, higijene i energije. Igra započinje u namještenoj kući, ali igrač može istraživati i kretati se izvan kuće, istražujući širu okolinu. Glavni cilj igre je upravljanje svakodnevnim aktivnostima lika kako bi se osiguralo njegovo preživljavanje. Ako igrač ne uspije zadovoljiti potrebe Stiffyja, lik umire i igra počinje ispočetka.

Implementacija *Skimsa* je podijeljena u nekoliko dijelova:

1. Razvoj klase lika i klasa za praćenje njegovih statistika
2. Implementacija interaktivnog objekta
3. Izrada prikaza i ažuriranje statistika lika
4. Razvoj klase svjetskog sata i satno ažuriranje statistika lika
5. Dizajn levela likove kuće i grada
6. Implementacija korisničkog sučelja

Budući da je implementacija igre *Skims* je vrlo opsežna i prelazi okvire ovog završnog rada, nastav rada se fokusira na prve dvije funkcionalnosti.

4.1. Razvoj klase lika i klasa za praćenje njegovih statistika

Funkcionalnosti glavnog lika razvijene su korištenjem C++ jezika u kombinaciji s klasom `ACharacter` i `UActorComponent` iz Unreal Enginea, iz koje je izvedena klasa `ASkimsCharacter`. Ova klasa, `ACharacter`, pruža osnovne funkcionalnosti potrebne za upravljanje likom u igri. Uz klasu lika, koristi se i pripadajuća *blueprint* klasa koja nadograđuje C++ kod, omogućujući dodjeljivanje vrijednosti varijablama i proširenje funkcionalnosti definiranih u klasi. `ASkimsCharacter` je ime klase lika u igri.

Konstruktor `ASkimsCharacter` inicijalizira osnovne postavke lika, uključujući veličinu sudarne kapsule. Konfigurira kretanje, uključujući brzinu hodanja i rotaciju.

Kreira varijablu `CameraBoom` koja je tipa `USpringArmComponent` i postavlja je da prati lika na definiranoj udaljenosti, omogućujući rotaciju armature prema kontroli igrača.

Također se stvara varijabla `FollowCamera` tipa `UCameraComponent`, koja se povezuje s `CameraBoom`, pružajući fiksnu poziciju kamere koja prati lika. Ovaj dio konstruktora naveden je u ispisu 1.

```
CameraBoom = CreateDefaultSubobject<USpringArmComponent>
(TEXT("CameraBoom"));
CameraBoom->SetupAttachment(RootComponent);
CameraBoom->TargetArmLength = 400.0f;
CameraBoom->bUsePawnControlRotation = true;

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

Ispis 1: Dio konstruktora

Važno je istaknuti da, osim prethodno navedenih komponenti, u klasi lika treba definirati i funkcije za akcije lika te funkcije ulazne kontrole (engl. *input controls*). U svakoj klasi lika koja nasljeđuje od `ACharacter` prisutna je funkcija `SetupPlayerInputComponent`, koja služi za uspostavljanje veze između ulaznih kontrola (poput tipkovnice, miša ili gamepada) i funkcija akcije lika. Ova funkcija omogućuje definiciju reakcije lika na različite ulaze od strane igrača, uključujući kretanje, skakanje, napad i druge akcije unutar igre.

4.1.1 Definicija osnovnih statistika lika

`UStatsComponent` je komponenta koja dodaje statistike liku u igri, uključujući energiju, glad i higijenu, nasljeđujući funkcionalnosti od `UActorComponent`. Ova komponenta omogućuje centralizirano upravljanje svim statistikama, što olakšava ažuriranje i praćenje podataka o liku.

Korištenje odvojenog komponenta za statistike lika ima nekoliko prednosti u odnosu na dodavanje statistika direktno u klasu lika. Prvo, centralizacija olakšava organizaciju koda i održavanje, posebno u slučaju promjena. Drugo, omogućava ponovno korištenje istih kom-

ponenti u različitim likovima unutar igre. Treće, enkapsulacija podataka unutar komponenti poboljšava upravljanje statistikama, jer omogućava definiranje funkcija za ažuriranje i manipulaciju varijabli na jednom mjestu. Enkapsulacija također smanjuje složenost klase lika, što je čini lakšom za razumijevanje.

Na kraju, u konstruktoru klase lika, potrebno je inicijalizirati `UStatsComponent`, što značajno proširuje funkcionalnost lika dodavanjem sustava za upravljanje statistikama (ispis 2). Ova komponenta omogućava ažuriranje i modifikaciju statističkih vrijednosti tijekom igre, čime se liku pruža veća interaktivnost.

```
StatsComponent = CreateOptionalDefaultSubobject<UStatsComponent>
(TEXT("Stats Component"));
```

Ispis 2: Inicijalizacija komponenta u konstruktoru

U ovoj slučaju, klasa komponente definira šest različitih statistika: mjehur (engl. *bladder*), glad (engl. *hunger*), energija (engl. *energy*), zabava (engl. *fun*), društvenost (engl. *social*), higijena (engl. *hygene*). Ove statistike predstavljaju ključne aspekte života, samim tim i lika u igri. Uz sve statistike, definiran je maksimalan iznos svake statistike kao što je prikazano na ispisu 3.

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Stats")
float Bladder;
float MaxBladder;
```

Ispis 3: Definicija statistike mjehura

`UProperty` je makronaredba koja označava atribut klase u Unreal Engineu, omogućujući im da imaju različite vrijednosti i karakteristike. Ova makronaredba dolazi s brojnim oznakama koje su važne za definiranje ponašanja i atributa svojstava unutar klasa. Ove oznake omogućuju programerima da precizno kontroliraju način na koji se varijable prikazuju i upravljaju u uređivaču, kao i tijekom razvoja igre.

`EditAnywhere` oznaka omogućuje da se atribut može uređivati s bilo kojeg mjesta

unutar Unreal Editora. Točnije, omogućava korisniku pristup i mijenjanje vrijednosti varijable unutar panela detalja Editora. `BlueprintReadOnly` oznaka označava da je varijabla samo za čitanje unutar *blueprint* sustava. To znači da se vrijednost može koristiti, ali ne i mijenjati iz *blueprinta*. Oznaka `category` definira kategoriju pod kojom će se varijabla prikazati u Editoru. U ovom slučaju, varijabla će biti smještena u kategoriju zvanom "Stats". Kategorizacija pomaže u organizaciji varijabli unutar uređivača, čineći ih lakšima za pronalaženje.

4.1.2 Implementacija metoda za ažuriranje statistika lika

Klasa komponente omogućava dodavanje ili oduzimanje određene vrijednosti svakoj statistici pomoću specifičnih metoda, poput `ModifyBladder` koja je prikazana na ispisu 4. Ove metode ne samo da mijenjaju trenutne vrijednosti statistika, već također osiguravaju da te vrijednosti ostanu unutar postavljenih granica, koristeći `FMath::Clamp` funkciju. To znači da statistike nikada neće pasti ispod nule ili premašiti maksimalnu dopuštenu vrijednost, u ovom slučaju `MaxBladder`. Sve druge statistike, kao što su energija i higijena, također imaju odgovarajuće metode koje reguliraju njihove vrijednosti i osiguravaju da ostanu unutar unaprijed postavljenih granica.

```
void UStatsComponent::ModifyBladder(float Amount)
{
   if (GetOwner()->Implements<UStatsInterface>()){
      if (Bladder == 0.0f){
         IStatsInterface::Execute_OnStatDepleted(GetOwner());
      }
   }
   Bladder = FMath::Clamp(Bladder + Amount, 0.0f, MaxBladder);
}
```

Ispis 4: Metoda modifikacije statistike mjehura

Kako bi se olakšalo upravljanje ponašanjem objekata kada im ponestane bodova statistike, potrebno je implementirati sučelje koje sadrži funkciju koja se poziva u takvim situacijama. `Component` klasa može provjeriti implementira li vlasnik objekta to sučelje i potom

pozvati odgovarajuću funkciju. Time se omogućuje različito ponašanje entiteta pri gubitku bodova statistike: neki entiteti mogu biti uništeni, drugi mogu pokrenuti određeni događaj u igri, dok treći mogu završiti igru, što je slučaj s likom igrača [2].

U ovom slučaju, ako neka statistika dosegne nulu, metode pozivaju implementaciju metode `OnStatDepleted` iz sučelja `IStatsInterface`, koja prima pokazivač na vlasnika ove komponente, a zatim se na tom vlasniku izvršava metoda `OnStatDepleted`.

`IStatsInterface` se koristi za implementaciju događaja vezanih uz statistike u igri. Sučelje je osmišljeno da klasi lika omogući reagiranje na promjene ili specifične uvjete vezane uz njegove statistike, poput trenutka kada neka od njih padne na nulu, kao što je prikazano u prethodnom primjeru.

`BlueprintNativeEvent` predstavlja događaj koji je deklariran u C++ i može imati zadano ponašanje (engl. *default behavior*), koje je također definirano u C++, ali se može nadjačati (engl. *override*) u *blueprintu*. Za deklaraciju izvornog događaja *blueprinta* pod nazivom `MyEvent`, potrebno je definirati funkciju `MyEvent` koristeći `UFUNCTION` makronaredbu s oznakom `BlueprintNativeEvent`, a zatim, kao virtualnu funkciju, potrebno je implementirati funkciju `MyEvent_Implementation` [2]. `MyEvent` u ispisu 4, je nazvan `OnStatDepleted`.

Razlog zašto se moraju deklarirati dvije funkcije je taj što prva služi za *blueprint*, omogućujući nadjačavanje događaja unutar *blueprinta*, dok druga predstavlja C++ potpis, koji omogućuje nadjačavanje događaja u C++. Deklaracija i definicija ovih funkcija prikazane su u ispisu 5.

```
UFUNCTION(BlueprintNativeEvent, Category = "Stats")
void OnStatDepleted();
virtual void OnStatDepleted_Implementation() = 0;
```

Ispis 5: Definicija i deklaracija metode sučelja

Deklaracija funkcije `OnStatDepleted_Implementation` zahtijeva vlastitu implementaciju. Međutim, sučelje ne mora implementirati ovu funkciju, jer bi implementacija biti prazna. Zato je potrebno obavijestiti kompajler da ova funkcija nema implementaciju

unutar ove klase, dodavanjem =0 na kraj njene deklaracije.

Kako bi se uspješno integrirale klasa lika i sučelje IStatsInterface te omogućilo liku da reagira na događaje povezame sa statistikama, klasa lika mora implementirati to sučelje kao što je prikazano na ispisu 6. To bi omogućilo klasi lika da prepozna kada vrijednost neke od statistika, poput gladi, energije ili higijene, dostigne nulu.

Ovakva implementacija omogućava jednostavnu prilagodbu različitim tipovima likova u igri, jer svaki lik može imati svoju jedinstvenu implementaciju funkcije OnStatDepleted, dok i dalje koristi zajedničko sučelje za komunikaciju s komponentama koje upravljaju statistikama.

```
class ASkimsCharacter : public ACharacter, public IStatsInterface
```

Ispis 6: Prikaz zaglavlja klase lika

Implementacija funkcija iz sučelja IStatsInterface u klasi ASkimsCharacter omogućava jednostavno dodavanje novih funkcionalnosti i izmjenu postojećih, bez utjecaja na druge dijelove igre.

4.1.3 Implementacija metode akcije pri iscrpljenju statistike

Da bi se napravila posebna akcija nakon što se prepozna pad vrijednosti neke statistike, klasa lika mora implementirati metodu OnStatDepleted, koja je definirana unutar sučelja.

Sadržaj metode prikazan je ispisom 7. Ova metoda sadrži logiku koja se pokreće kada se iscrpi neka statistika, kao što su vizualni efekti, animacije ili obavijesti igraču. U ovom slučaju, to bi bila reprodukcija animacije. Metoda provjerava je li DeathMontage valjan i postoji li geometrija lika i njegov instancirani animacijski objekt. Ako su svi uvjeti zadovoljeni, funkcija poziva Montage_Play koja pokreće animaciju smrti lika. No, ako on nije valjan, funkcija ispisuje upozorenje, ukazujući na problem s učitavanjem animacijske montaže.

UE_LOG je makro koji se koristi u Unreal Engineu za ispisivanje poruke u konzolu, kao i u datoteku poruka igre. Ta funkcionalnost pomaže korisnicima u dijagnosticiranju problema

i praćenju toka izvršavanja programa. `LogTemp` označava kategorija poruke, `warning` označava razinu ozbiljnosti, a `TEXT` omogućava ispis nizova znakova.

```
void ASkimsCharacter::OnStatDepleted_Implementation()
{
   if (DeathMontage && GetMesh() && GetMesh()->GetAnimInstance()){
      GetMesh()->GetAnimInstance()->Montage_Play(DeathMontage);
   }
   else {
       UE_LOG(LogTemp, Warning, TEXT("Failed to
            load animation montage!"));
   }
}
```

Ispis 7: Implementacija metode sučelja

`DeathMontage` je varijabla tipa `UAnimMontage` te predstavlja animacijsku montažu smrti. Deklaracija varijable je prikazana ispisom 8.

```
UPROPERTY(EditANywhere, BlueprintReadOnly, Category = "Animation")
UAnimMontage* DeathMontage;
```

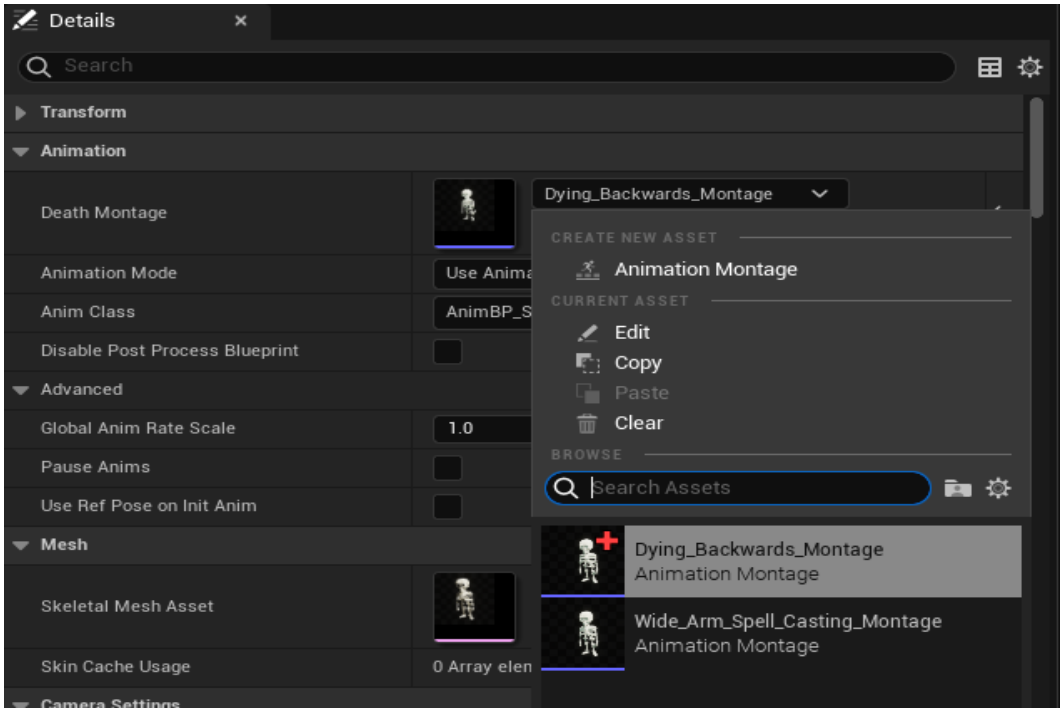
Ispis 8: Deklaracija varijable

Makronaredba `UPROPERTY` omogućava pristup detaljima varijable unutar Unreal Editora, čime se olakšava upravljanje varijablama. Umjesto da se oslanja na tvrdo kodiranje, koje bi zahtijevalo ručni unos putanje do datoteke ili montaže direktno u C++ kôd, korištenje `UPROPERTY` pojednostavljuje proces.

Otvaranjem *blueprinta* nazvanog `BP_TopDownCharacter` unutar Unreal Editora, korisnici mogu jednostavno pristupiti varijabli `DeathMontage` u panelu s detaljima. *Blueprint* naziva `BP_TopDownCharacter` je zapravo vizualna skripta glavnog lika. Ondje se može dodijeliti odgovarajuća montaža animacije umiranja, što omogućuje brzu i jednostavnu prilagodbu ponašanja lika. Na slici 9 prikazana je dodjela montaže animacije, što

dodatno ilustrira kako se varijable mogu lako postaviti u Editoru.

Ako se vrijednost varijable ne dodijeli putem editora, animacija umiranja lika neće se pokrenuti, a u konzoli ispisat će se poruka "Failed to load animation montage!", kako je navedeno u implementaciji funkcije `OnStatDepleted`. Stoga je važno dodijeliti sve potrebne vrijednosti prije pokretanja igre kako bi se izbjegli ovakvi problemi.



Slika 9: Panel detalja glavnog lika

4.2. Implementacija interaktivnog objekta

Baza funkcionalnosti interakcije lika s različitim objektima u igri je ostvarena putem C++ jezika. Klasa `AInteractibleBaseActor`, koja nasljeđuje klasu `AActor` je dizajnirana kao baza za sve aktore (engl. *actors*) u igri koji omogućuju interakciju s likom.

Ova klasa pruža osnovnu funkcionalnost za interakciju, uključujući metode za započinjanje i završetak interakcije i upravljanje prikazom korisničkog sučelja. Nasljeđivanjem ove klase, svaki novi interaktivni aktor u igri automatski dobiva ovu osnovnu funkcionalnost, dok programerima ostavlja mogućnost prilagodbe specifične za svaki pojedinačni slučaj interakcije. Na taj način, `AInteractibleBaseActor` osigurava proširivu osnovu za razvoj raznovrsnih interaktivnih elemenata unutar igre.

Ispis 9 prikazuje dvije bitne funkcionalnosti. Prva jest ta da se za komponentu detekcije sudara (engl. *collision component*) povezuju delegati, koji omogućuju objektu da reagira na događaje sudara. Druga važna funkcionalnost jest da ta ista komponenta za detekciju sudara reagira isključivo na određenom kanalu sudara (engl. *collision channel*), čime se precizno definira s kojim objektima komponenta može stupiti u interakciju.

```
AInteractableBaseActor::AInteractableBaseActor()
{
   CollisionComp->OnComponentBeginOverlap.AddDynamic(this,
         &AInteractableBaseActor::OnOverlapBegin);
   CollisionComp->OnComponentEndOverlap.AddDynamic(this,
         &AInteractableBaseActor::OnOverlapEnd);

   CollisionComp->SetCollisionEnable
         ECollisionEnabled::QueryOnly
   );
   CollisionComp->SetCollisionObjectType(
         ECollisionChannel::ECC_GameTraceChannel1
      );
   CollisionComp->SetCollisionResponseToChannel(
         ECollisionChannel::ECC_GameTraceChannel1,
         ECollisionResponse::ECR_Block
   );
}
```

Ispis 9: Dio konstruktora interaktivnog objekta

4.2.1 Detekcija sudara i interakcije

U Unreal Engineu, delegati omogućuju povezivanje s funkcijama unutar klase koje se pozivaju kada dođe do sudara. U ovom slučaju, delegati `OnComponentBeginOverlap` i `OnComponentEndOverlap` se koriste za upravljanje događajima početka i kraja sudara. Pomoću ključne riječi `AddDynamic`, ovi delegati se povezuju s funkcijama kao što su `OnOverlapBegin` i `OnOverlapEnd`, što omogućava prilagodbu ponašanja objekta tije-

kom sudara. Na ovaj način, može se definirati ponašanje objekta za svaki od tih događaja.

Linijska pretraga (engl. *line trace*) u Unreal Engineu je tehnika kojom se "iscrtava" imaginarna linija u 3D prostoru kako bi se detektirao sudar s objektima duž te linije. Ova metoda se često koristi za provjeru postoji li nešto na putanji te linije, ako da, onda metoda vraća objekt s kojim se linija sudara. Uz to, vraća i informacije o točki sudara, normalnom vektoru površine i referencu na sam objekt.

Linija sudara (engl. *collision channel*) je specifična primjena linijske pretrage. Svaki objekt u Unreal Engineu može biti dodijeljen određenoj liniji sudara, što omogućava filtriranje sudara. Prilikom postavljanja linijske pretrage, može se odrediti koje linije sudara će se uzeti u obzir. Na primjer, ako se želi postići da linijska pretraga detektira samo neprijateljske objekte, može se kreirati posebna linija sudara za njih i konfigurirati linijsku pretragu da reagira samo na taj kanal. Na taj način se filtriraju nepotrebni sudari i samim tim poboljšava performansa i preciznost.

U ovom slučaju koristeći `ECollisionEnabled::QueryOnly` označeno je da komponenta sudara interaktivnog objekta samo odgovara na kompleksnije sudare, kao što je linijski sudar, ali neće fizički reagirati na sudar. To omogućava da se komponenta koristi za identifikaciju sudara, bez utjecaja na fiziku.

`ECollisionEnabled::ECC_GameTraceChannel1` je prilagođena verzija linije sudara koja je osmišljena da reagira samo na sudare s komponentama sudara interaktivnih objekata. Kada se koristi `ECR_Block`, to definira da linija sudara uvijek blokira komponentu sudara interaktivnog objekta, sprječavajući daljnju interakciju s objektima na tom kanalu. Druge moguće reakcije uključuju `ECR_Ignore` (zanemaruje sudare s objektima na ovom kanalu) i `ECR_Overlap` (pokreće događaje preklapanja, ali ne blokira interakciju).

```
UPrimitiveComponent* OverlappedComponent,
AActor* OtherActor, UPrimitiveComponent* OtherComp,
int32 OtherBodyIndex,
bool bFromLine,
const FHitResult& LineResult)
```

Ispis 10: Zaglavlje funkcije `OnOverlapBegin`

Pošto funkcija `OnOverlapBegin` slijedi delegat `OnComponentBeginOverlap`, mora imati isti potpis (engl. *signature*) funkcije kao i delegat. Ti parametri (ispis 10) nam daju detaljne informacije o događaju.

1. **Preklopljeni komponent**: Pokazivač na komponent koji inicira preklapanje, bitan za identifikaciju izvora sudara.
2. **Drugi aktor**: Označava aktora koji je preklopljen, važan za određivanje odgovora na sudar.
3. **Druga komponenta**: Preklapajuća komponenta na drugom aktoru, pruža kontekst o sudaru.
4. **Normalni impuls**: Predstavlja primijenjeni impuls tijekom sudara, koristan za realistične fizičke reakcije.
5. **Pogodak**: `FHitResult` struktura koja sadrži detaljnije informacije o preklapanju.

```
if (GetWorld()->LineTraceSingleByChannel(HitResult, StartLocation,
            EndLocation, ECC_GameTraceChannel1, TraceParams))
{
   if (HitResult.GetActor() == InteractingCharacter)
   {
      TogglePlayerInput(true);
      ToggleWidgetVisibility(true);
   }
}
```

Ispis 11: Dio funkcije `OnOverlapBegin`

Ispis 11 prikazuje izvođenje linijske pretrage za provjeru postoji li objekt između početne točke i krajnje točke. `StartLocation` u ovom slučaju predstavlja poziciju s koje će se izvesti linijska pretraga, dok je `EndLocation` zbroj `StartLocation` i smjera koji je pomnožen s duljinom linije pretrage. Ova linijska pretraga omogućava detekciju objekata koji se nalaze na putu, što može biti korisno za različite mehanike igre, kao što su interakcije s okolinom ili provjera sudara. `LineTraceSingleByChannel` se koristi za provjeru

postoji li objekt između `StartLocation` i `EndLocation` koji odgovara specifičnoj liniji kolizije `GameTraceChannel1`. Ako se dogodi sudar, rezultat se sprema u `FHitResult`. U slučaju da je vraćeni aktor iz strukture `FHitResult` jednak liku koji je pokrenuo događaj kolizije, igraču će biti omogućeno da pritisne gumb za interakciju koji se prikaže na sučelju.

4.2.2 Upravljanje interakcijom lika

Funkcija `TogglePlayerInput` igra bitnu ulogu, jer se u njenoj implementaciji (ispis 12) nalazi poziv na funkciju koja sadrži kôd koji definira ponašanje igre tijekom trajanja interakcije lika i objekta.

```
if (bShow){
      PlayerController->EnableInput(PlayerController);
      if (!bIsInteractedWith){
         PlayerController->InputComponent->BindAction(
            "Interact",IE_Pressed, this,
            &AInteractableBaseActor::StartInteraction
         );
      }
   }
   else{
      PlayerController->DisableInput(PlayerController);
      PlayerController->InputComponent->ClearActionBindings();
   }
}
```

Ispis 12: Dio definicije funkcije `TogglePlayerInput`

Kontroler igrača (engl. *player controller*) u Unreal Engineu je klasa koja upravlja interakcijama između igrača i igre. Ponaša se kao most između korisničkog unosa (poput tipkovnice, miša ili kontrolera) i akcija koje se događaju u igri. Kontroler igrača prati i obrađuje sve unose koje igrač šalje i koristi ih za kontroliranje lika, kamere ili drugih aspekata igre. Ovaj parametar određuje hoće li unos putem kontrolera biti omogućen; unos će biti aktivan samo ako je vrijednost `true`. Parametar `bIsInteractedWith` provjerava je li

lik već u interakciji sa objektom i nastavlja dalje samo ako nije.

Nakon toga, koristi se `BindAction` za povezivanje pritiska tipke "Interact" s funkcijom `StartInteraction`. Ova metoda omogućava automatsko pozivanje funkcije kada igrač pritisne tipku "Interact", pokrećući proces interakcije s objektom. Tipka je definirana unutar postavki projekta u Unreal Editoru.

Ispis 13 prikazuje definiciju metode `StartInteraction` unutar klase interaktivnog objekta, koja definira ponašanje objekta i lika koji interaktira s objektom. Nadjačavanje funkcionalnosti u *blueprint* klasama koje su izvedenice ove klase je omogućeno korištenjem oznake `BlueprintNativeEvent`. S obzirom na to da je u ovoj klasi napisana implementacija osnovne logike interaktivnog objekta u `StartInteraction_Implementation` funkciji, ona se koristi kao osnova za druge izvedene klase.

```
UFUNCTION(BlueprintNativeEvent, Category = "Interaction")
void StartInteraction();
virtual void StartInteraction_Implementation();
```

Ispis 13: Definicija funkcije `StartInteraction`

Funkcija `startInteraction_Implementation` (ispis 14) definira ponašanje igre tijekom interakcije između lika i interaktivnih objekata, koji su implementirani unutar funkcije `AInteractibleBaseActor`. Varijabla `bIsInteractedWith` se postavlja na `true`, što označava da je lik započeo interakciju s ovim interaktivnim objektom.

```
bIsInteractedWith = true;
if (bIsInteractedWith)
{
   TogglePlayerInput(false);
   ToggleWidgetVisibility(false);
   InteractingCharacter->EnableCharacterMovement(false);
   StartDelay();
}
```

Ispis 14: Implementacija metode `StartInteraction_Implementation`

Provjera varijable `bIsInteractedWith` služi tome da spriječi lika u ponovnom ulasku u interakciju, prije nego se izvedena implementacije funkcije izvrše. Referenca na lika koji je pokrenuo interakciju je spremljena `InteractingCharacter`. Varijabla je definirana tako se može koristiti u Unreal Editoru (ispis 15). Ova varijabla se koristi i u funkciji `OnOverlapBegin` za rad s likom koji je u interakciji s objektom. Kada se lik približi objektu i dođe do sudara, referenca na taj lik se pohranjuje u `InteractingCharacter`, što omogućuje daljnje manipulacije (poput onemogućavanja kretanja ili prikazivanja sučelja).

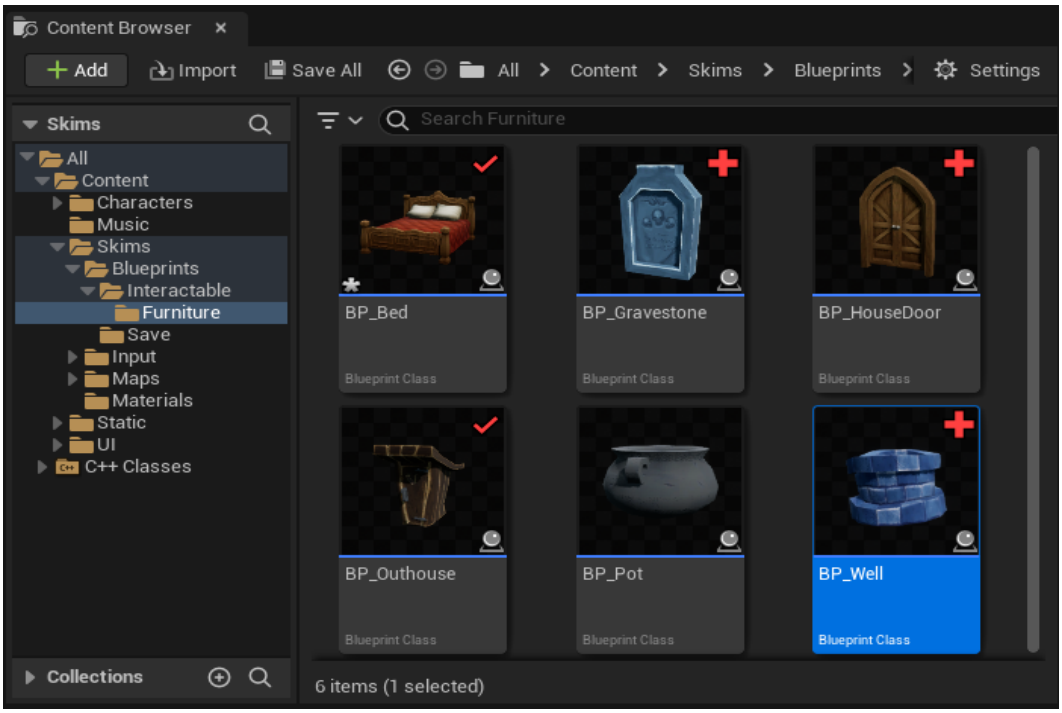
```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly,
                    Category = "Interactable")
class ASkimsCharacter* InteractingCharacter;
```

Ispis 15: Definicija varijable `InteractingCharacter`

Bitno je istaknuti kooperaciju između ove dvije funkcije. Interakcija se omogućava ili onemogućava na temelju stanja igre korištenjem funkcije `TogglePlayerInput`. Kada je unos omogućen, funkcija `BindAction` osigurava da pritisak tipke za interakciju vodi do poziva funkcije `StartInteraction`, koja se zatim brine o onemogućavanju daljnjeg unosa dok se interakcija ne završi, sprječavajući potencijalne konflikte u igri.

4.2.3 Proširenje funkcionalnosti interakcije korištenjem *blueprint* klase

Nakon što je C++ klasa napisana i kompajlirana, može se koristiti kao roditeljska klasa za novu *blueprint* klasu. U Unreal Engine editoru, prilikom stvaranja nove *blueprint* klase, moguće je odabrati željenu C++ klasu kao roditeljsku. *Blueprint* klasa tada nasljeđuje sve funkcionalnosti definirane u C++ klasi, s mogućnošću dodatnog prilagođavanja ili proširivanja kroz *blueprint* sučelje. Ovaj proces omogućava jednostavno kombiniranje performansi C++ programiranja s pristupačnošću *blueprint* sustava.

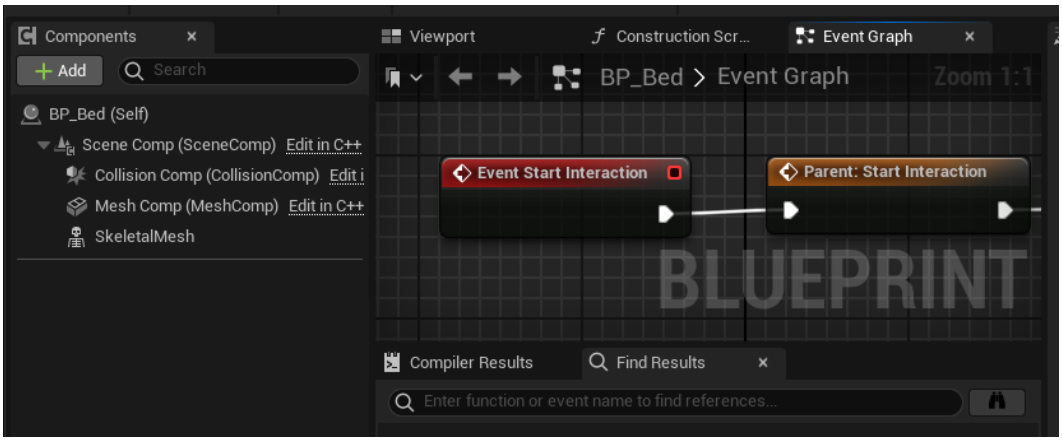


Slika 10: *Blueprint* klase koje nasljeđuju od interaktivnog objekta

Svaka od ovih *blueprint* klasa (slika 10) funkciju `StartInteraction` implementira na svoj način, jer se statistike lika mijenjaju ovisno o objektu s kojim se dolazi u kontakt. U ovom primjeru, imamo objekte koji pripadaju kategoriji namještaja; promjene u statistikama ovise o konkretnom komadu namještaja. Na primjer, kada lik koristi krevet (engl. *bed*), energija će mu se povećati, dok će kontakt s bunarom (engl. *well*) rezultirati porastom higijene.

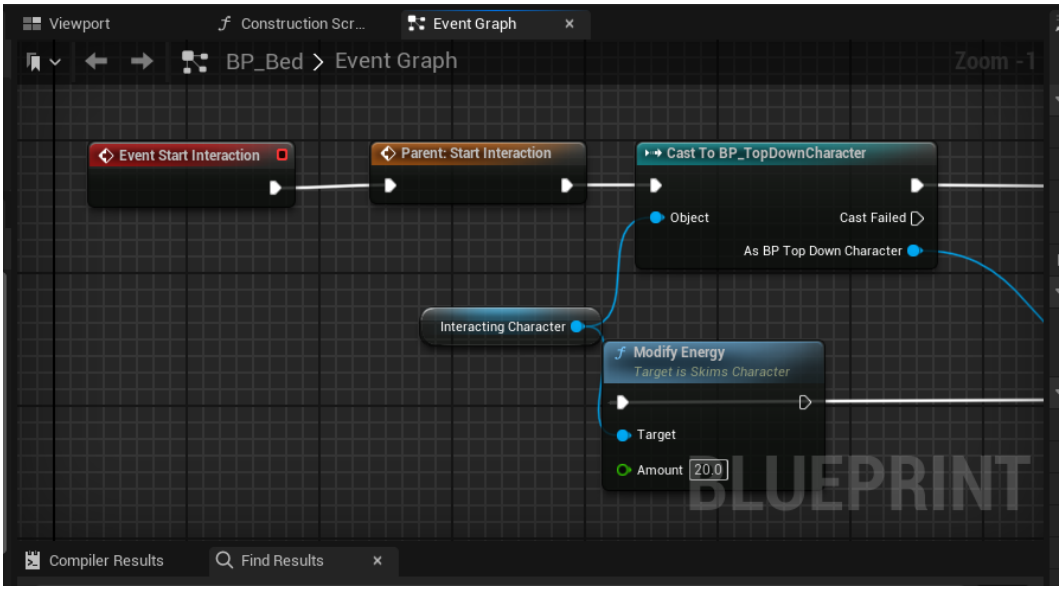
Zahvaljujući ovom sustavu, korisnici mogu lako proširiti funkcionalnost interakcije. Na primjer, mogu stvoriti novu klasu koja nasljeđuje `AInteractableBaseActor` i u *blueprintu* definirati vlastitu verziju `StartInteraction` koja će, osim osnovne funkcionalnosti, dodati posebne efekte, animacije ili druge akcije specifične za taj objekt.

Postupak nadjačavanja funkcije `StartInteraction` unutar blueprinta pod nazivom `BP_Bed` se radi tako da se otvori dijagram događaja unutar *blueprinta*. Zatim desnim klikom na prozor i odabirom opcije `Add Event for StartInteraction` se stvara čvor koji se može koristiti za definiranje logike. Budući da je u ovom slučaju zadržana funkcionalnost funkcije, iz tog čvora se poziva čvor naziva `Parent:StartInteraction` kao što je prikazano na slici 11.



Slika 11: Nadjačavanje `StartInteraction` funkcije

Nakon toga, dodaju se dodatni čvorovi koji predstavljaju novu logiku, kao što je prikazivanje poruka ili aktiviranje animacija. U ovom slučaju potreban je čvor `Cast To BP_TopDownCharacter` jer omogućuje pristup funkcijama za mijenjanje statistika lika. Na taj način, uz referencu `InteractingCharacter`, moguće je ažurirati statistike lika koji je prikupljen u sudaru s interaktivnim objektom. Specifično, koristi se za pozivanje funkcije `ModifyEnergy` u klasi lika. Na kraju sve to izgleda kao na slici 12



Slika 12: Dijagram događaja `BP_Bed` klase

Definicija funkcije `ModifyEnergy` u C++ (ispis 16) s `BlueprintCallable` oznakom omogućava da se ova funkcija može pozvati iz *blueprint* klase. Funkcija kroz parametar `Amount` prima iznos koji se koristi za smanjenje energije ili povećanje energije. U

klasi lika, postoje ovakve funkcije za sve ostale statistike, tako da se u različitim *blueprint* klasama mogu mijenjati različite statistike lika, ovisno o potrebi. Ovo predstavlja jednu od ključnih funkcionalnosti igre koja omogućava igraču da održava život lika kroz interakciju s različitim objektima unutar igre.

```
UFUNCTION(BlueprintCallable, Category = "Stats")
void ModifyEnergy(float Amount);
```

Ispis 16: Definicija funkcije `ModifyEnergy`

Kada se dovrši uređivanje *blueprint* klase, njenim kompajlirajem se potvrđuje da je funkcija uspješno nadjačana, omogućavajući da se prilagodi način na koji `BP_Bed` reagira na interakcije.

5. Zaključak

Projekt *Skims* predstavlja ambicioznu igru koja je razvijena unutar Unreal Enginea 5. Korištenje ovog naprednog alata omogućilo je implementaciju tehnologija i funkcionalnosti koje podržavaju stvaranje složenih 3D okruženja te simulaciju osnovnih ljudskih funkcija. Tijekom rada na projektu istraženi su različiti aspekti dizajna igre, uključujući razvoj 3D okruženja i likova, kao i implementaciju kompleksnih akcija likova i interakciju s objektima, što doprinosi ukupnom doživljaju igre.

Unreal Engine, iako jako izazovan, istovremeno je i iznimno zanimljiv alat. Učenje o naprednim funkcionalnostima, kao što su sustavi animacija i fizike, može djelovati zastrašujuće, no donosi i velike nagrade u obliku mogućnosti za kreativno izražavanje.

Uz malu prilagodbu besplatnih modela i animacija, uspješno je stvorena jedinstvena atmosfera s prepoznatljivim stilom. Gotovi modeli i animacije omogućili su da se više vremena posveti oblikovanju cjelokupne vizije igre, umjesto fokusiranja na pojedine aspekte. Uz prilagođeni model lika koji je razvijen za potrebe ove igre, dostupni resursi omogućili su da se stvori jedinstveni i zabavan ugođaj.

Kombinacija C++ programiranja i vizualnog skriptiranja omogućila je postizanje visoke razine preciznosti. Iako povezivanje vizualnog skriptiranja s C++ može biti zbunjujuće, ovaj spoj omogućava novu razinu prilagodbe koja ne bi bile dostupna isključivo putem C++ programiranja.

Iako je projekt *Skims* postigao ključne ciljeve, postoji značajan prostor za nadogradnju i poboljšanje. Daljnji razvoj igre mogao bi uključivati proširenje svijeta igre, dodavanje složenijih mehanika poput interakcije s drugim likovima, ili razvoj naprednih AI sistema koji bi omogućili interakciju s okolinom na temelju statistika lika. Poboljšanje grafičkih elemenata i optimizacija performansi također bi doprinijeli boljem iskustvu igranja.

Literatura

[1] IGN, “The first easter egg,” https://www.ign.com/wikis/atari-adventure/The_First_Easter_Egg, [Online; posjećeno 20-August-2024].

[2] G. Marques *et al.*, “Elevating game experiences with unreal engine 5: Bring your game ideas to life using the new unreal engine 5 and c++,” 2022.