RAZVOJ 3D VIDEO IGRE ZASNOVAN NA UNREAL ENGINE 4

Otović, Luka

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij računarstva

Diplomski rad

RAZVOJ 3D VIDEO IGRE ZASNOVAN NA UNREAL ENGINE 4

Rijeka, rujan 2022.

Luka Otović

0069082898

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij računarstva

Diplomski rad

RAZVOJ 3D VIDEO IGRE ZASNOVAN NA UNREAL ENGINE 4

Mentor: Doc. dr. sc. Goran Mauša, dipl. ing

Rijeka, rujan 2022.

Luka Otović

0069082898

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 14. ožujka 2022.

Zavod: **Zavod za računarstvo**
Predmet: **Inženjerstvo kompleksnih programskih sustava**
Grana: **2.09.06 programsko inženjerstvo**

ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Luka Otović (0069082898)**
Studij: Diplomski sveučilišni studij računarstva
Modul: Računalni sustavi

Zadatak: **Razvoj 3D video igre zasnovan na Unreal Engine 4 / 3D video game development based on Unreal Engine 4**

Opis zadatka:

Napraviti povijesni pregled razvoja video igara i upotrebe programsko orijentiranih jezika u njihovoj izradi. Proučiti i opisati programski paket Unreal Engine 4 u svrhu izrade 3D video igara, algoritme za detekciju sudara, graf scene i odgovarajuće algoritme za transformiranje koordinatnih sustava. Predložiti specifikaciju zahtjeva, provesti tehničko oblikovanje i implementirati pokaznu video igru u svrhu demonstracije navedenih pojmova.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 14. ožujka 2022.

Mentor:

Doc. Goran Mauša, dipl. ing.

Predsjednik povjerenstva za diplomski ispit:

Prof. dr. sc. Kristijan Lenac

IZJAVA

Izjavljujem da sam završni rad pod naslovom „Razvoj 3D video igre zasnovan na Unreal Engine 4” izradio samostalno pod nadzorom i uz stručnu pomoć mentora doc. dr. sc. Goran Mauša.

Luka Otović

_____

(potpis studenta)

Zahvaljujem svim prijateljima, kolegama i mentoru na ukazanoj potpori, strpljenju i razumijevanju, a naročito svojoj obitelji koja mi je ovo i omogućila.

SADRŽAJ

1. UVOD....1
2. POVIJEST I RAZVOJ VIDEOIGARA....3
2.1. Usporedba starih videoigara i novih videoigara....6
2.2. Razvoj videoigara....9
2.2.1. Nasljeđivanje....11
2.2.2. Budućnost programiranja videoigara....17
3. UNREAL ENGINE 4....19
3.1. Razvoj videoigara uz UE4....20
3.1.1. Razvoj videoigre kroz pisanje programskog koda....20
3.1.2. Razvoj videoigre kroz stroj UE4....23
3.2. Nacrti (*Blueprints*)....32
3.3. Usporedba klasičnog programiranja i nacrta....36
4. KOLIZIJA....40
4.1. Jednostavna kolizija....40
4.1.1. Linijski segment....40
4.1.2. Ploha....41
4.1.3. Kutija....42
4.1.4. Testovi kolizije....43
4.2. Složena kolizija....48
4.3. Provjera kolizije između dinamičkih objekata....50
4.4. Implementacija kolizije u UE4....52
4.4.1. Dodavanje kolizije kroz programski kod....52
5. GRAF SCENE....55
5.1. Graf scene u videoigrama i 3D aplikacijama....56
5.2. Implementacija....57
5.2.1. Operacije i otprema grafa scene....57
5.2.2. Obilasci....58
5.3. Graf scene i hijerarhija ograničivača volumena....59
5.4. Prikaz grafikona scene unutar UE4....60
5.4.1. Rotacije unutar UE4....63
6. RAZVOJ VIDEO IGRE....66
6.1. Koncept videoigre bez granice....66
6.2. Space Runner....67

6.2.1. Ulazne komande.....67
6.2.2. Dinamičko generiranje razine.....68
6.2.3. Detekcija kolizije sa preprekama.....70
6.2.3. Igračevo naoružanje.....72
7. ZAKLJUČAK.....74
8. LITERATURA.....76
9. POPIS OZNAKA I KRATICA.....78
10. SAŽETAK I KLJUČNE RIJEČI NA HRVATSKOM I ENGLESKOM JEZIKU.....79

1. UVOD

Ovaj se diplomski rad bavi razvojem videoigre za pametne telefone koristeći stroj Unreal Engine 4 (engl. *Unreal Engine 4 engine*; UE4). Cilj ovog rada je razviti videoigru pomoću koje će se demonstrirati osnovni pojmovi i algoritmi koji se koriste u razvoju videoigri. Također će se kroz ovaj rad pokazati najčešći problemi s kojima se programeri suočavaju tijekom razvoja videoigre i na koji ih način otklanjaju. Neki od tih problema su: kolizije, gimbalov problem, prikaz likova na zaslonu.

Na samom početku ovog rada će se prvo prikazati povijest i razvoj videoigara. U tom poglavlju će se najviše govoriti na koji način su videoigre evoluirale od običnih tekstualnih videoigara do masovnih 3D videoigra koje igraju nekoliko tisuća ljudi istovremeno. Na istim videoigrama će se prikazati i razvoj same tehnologije koje se koriste za razvoj videoigara i na koji način bi se u bližnjoj budućnosti moglo razvijati nove videoigre.

Zatim će se u zasebnom poglavlju biti govora o strojevima za razvoj videoigra, a posebna pažnja će se pridati stroju Unreal Engine 4 (UE4). Ovdje će se spomenuti koje su mogućnosti stroja, zašto ih koristimo, koje su njihove prednosti i mane te na koji način ih možemo najbolje iskoristiti za sam razvoj videoigre. U ovom poglavlju će se također spomenuti novi način programiranja kojeg je UE4 uveo, a to je programiranje uz pomoć nacrta (engl. *Blueprints*). Na samom kraju poglavlja će se usporediti klasično programiranje s nacrtima i odgovorit će se na pitanje na koji način bi se trebalo razvijati samu videoigru uz stroj UE4.

Nakon toga će biti govora o jednom od najvećih problema koji se javlja tijekom razvoja videoigre, a to je kolizija. Objasnit će se različiti pristupi problemu i testovi koji se koriste u razvoju videoigre kako bi se otkrilo dodiruju li se dva objekta u virtualnom svijetu. Također će se objasniti razlika između jednostavne i složene kolizije i na koji način ih rješavamo.

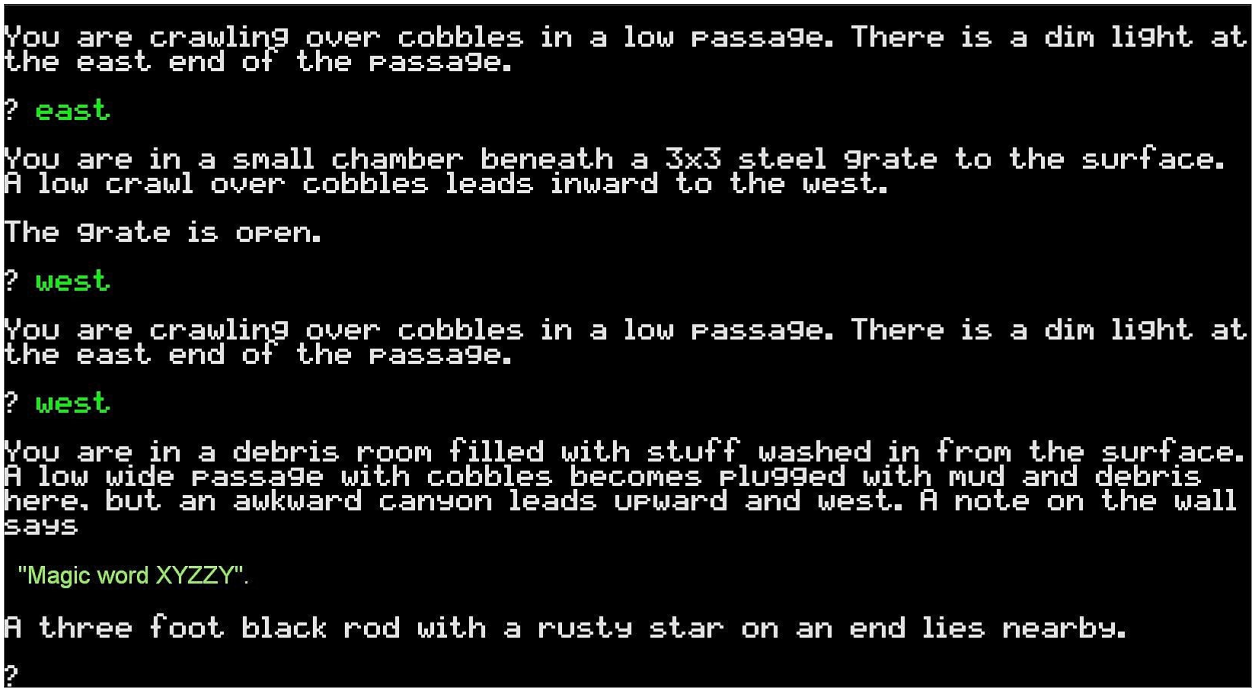
Poslije kolizija će se spomenuti gimbalov problem, koji je čest problem kod objekata koji se mogu rotirati u sve tri osi, kao što su primjerice letjelice . U ovom poglavlju će biti govora o

grafu scene, od čega se sastoji i čemu služi. Zatim će se pokazati na koji način dolazi do gimbalovog problema i na koji način ga rješavamo.

Na samom kraju ovog diplomskog rada će se prikazati cijeli postupak razvoja jedne videoigre što uključuje konceptualizaciju i implementaciju videoigre koristeći metode opisane u ovome radu.

2. POVIJEST I RAZVOJ VIDEOIGARA

Videoigra ili računalna igra je elektronska igra koja uključuje interakciju s korisničkim sučeljem putem kontrolera, tipkovnice, senzora pokreta i sličnih uređaja radi generiranja vizualne povratne informacije. Te povratne informacije najčešće se prikazuju na uređaju za video prikaz kao što su TV prijamnik, monitor, zaslon osjetljiv na dodir ili naočale za virtualnu stvarnost. Također postoje računalne igre koje ne ovise o grafičkom prikazu, na primjer tekstualne avanturističke igre.

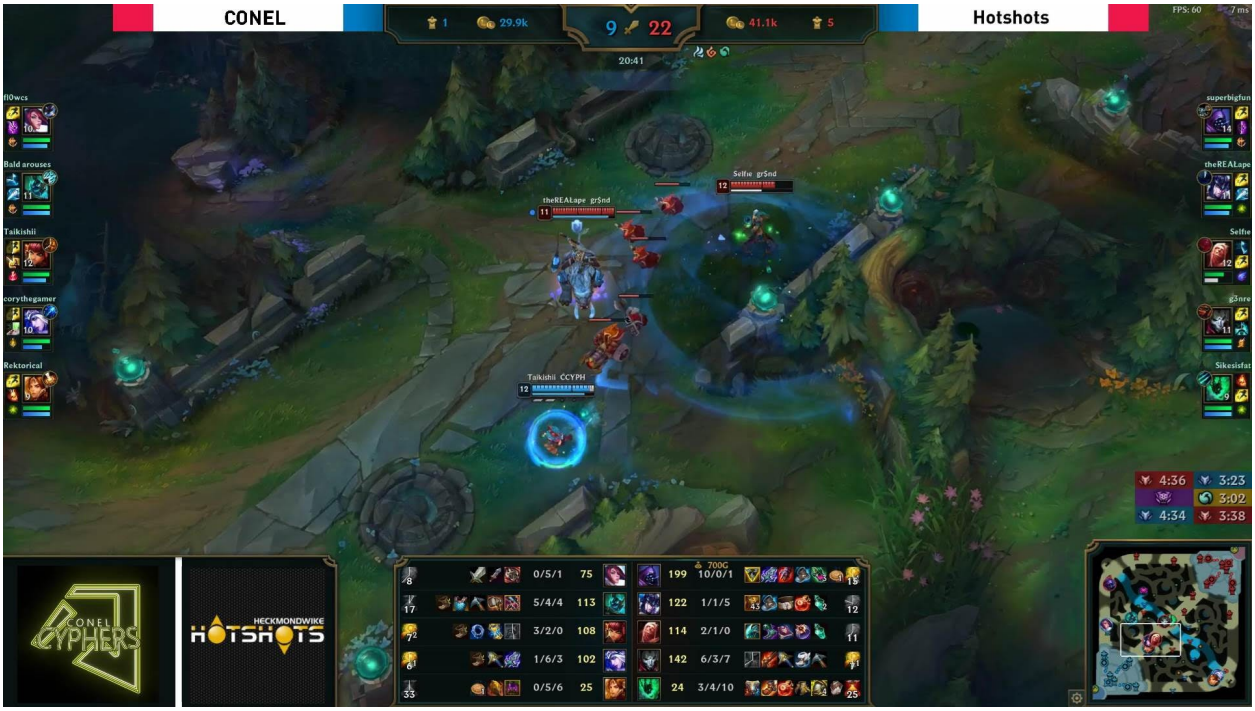


Slika 2.1. Tekstualna avanturistička igra.

Sa Slike 2.1. se može vidjeti primjer jedne od tekstualnih avanturističkih igara. Cijeli svijet je u takvim računalnim igrama opisan tekstualno, a naredbe koje igrač može izdati su veoma jednostavne, primjerice ime prostorije ili smjer u kojem se želi pomaknuti. Igrač izvršava naredbu tako da ju upiše u sučelje naredbenog retka (engl. *command line interface*). Prednosti ovakvih računalnih igrica je što su veoma jednostavne i ovise o igračevoj mašti.

Videoigre su definirane na temelju svoje platforme, što uključuje arkadne videoigre, igre za konzolu, igre za osobno računalo, i u današnje vrijeme igre za pametni telefon (*mobilne igre*). Druga podjela videoigara je na žanrove koji su definirani ovisno o tipu videoigre. Među najpoznatijim žanrovima videoigara danas su: FPS (*first person shooter*), MOBA (*multiplayer online battle arena),* Battle Royal, Horror, Strategije, ...

Slika 2.2. a) FPS videoigra Valorant (2020.).



Slika 2.2. b) MOBA videoigra League of Legends (2009.).

Prva potrošačka video igra je arkadna video igra Computer Space iz 1971. godine, a prva popularna videoigra je izašla 1972. godine pod imenom Pong. U isto vrijeme je izašla prva kućna konzola (Magnavox Odyssey), što je označilo zlatno doba arkadnih videoigara u razdoblju od kasnih 1970-ih do ranih 1980-ih.

1983. godine dolazi kraj zlatnog razdoblja arkadnih videoigara, zbog gubitka kontrole nad izdavaštvom i zasićenosti tržišta. Nakon pada, japanske tvrtke Nintendo, Sony i Sega su uspostavile prakse i metode oko razvoja i distribucije videoigara kako bi se spriječio sličan pad u budućnosti.

Razvoj videoigara u današnje vrijeme zahtijeva brojne vještine kako bi se igra plasirala i zadržala što duže na tržištu. Za razvoj videoigre potrebni su programeri, izdavači, distributeri, trgovci, proizvođači konzola, dizajneri, glazbenici,...

Radi velike konkurencije, na današnjem tržištu ostaju samo najbolje videoigre poput: *Valorant* (Slika 2.2. a)), *League of Legends* (Slika 2.2. b)), *Lost Ark*, *Fortnite*, *Minecraft*,...

U 2000-ima, glavna industrija bila je usredotočena na igrama koje je razvio veliki studio, takozvane “AAA” igre, a to je ostavilo malo prostora za riskantnije i eksperimentalne igre. Zajedno s dostupnošću interneta i digitalne distribucije, to je dalo prostora neovisnom razvoju videoigara (engl. *indie games*) da steknu važnost u 2010-ima [1]. Od tada, komercijalna važnost industrije videoigara raste. Azijska tržišta u nastajanju i mobilne igre na pametnim telefonima posebno mijenjaju demografiju igrača prema ležernom igranju i povećavaju unovčavanje uključivanjem igara kao usluge. Od 2020. godine globalno tržište videoigara ima procijenjene godišnje prihode od 159 milijardi USD na hardveru, programu i uslugama, a to je tri puta više od globalne glazbene industrije 2019. i četiri puta više od filmske industrije 2019. [2].

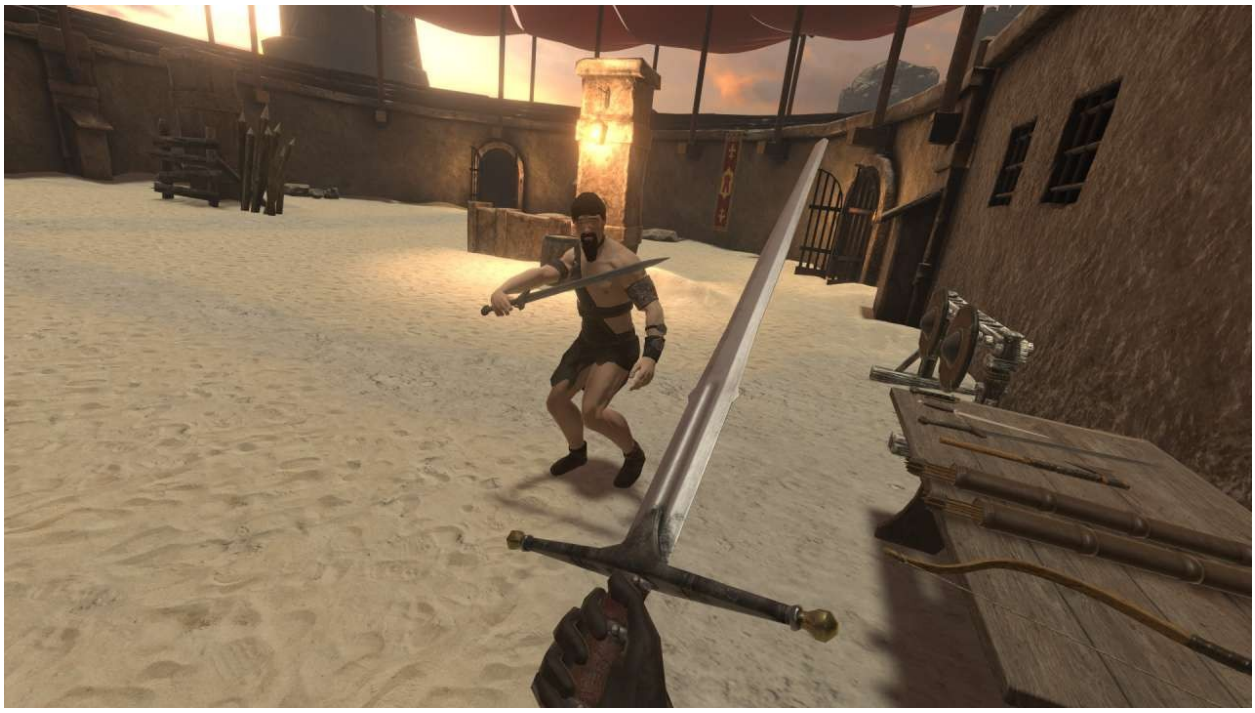
2.1. Usporedba starih videoigara i novih videoigara

U odnosu na svoje početke, videoigre su se do današnjeg dana značajno promijenile, a tehnologija na kojoj su izgrađene i na kojoj se igraju razvila se velikom brzinom. Igre koje poznajemo pretvorile su se u zapanjujuća umjetnička djela. Sa Slike 2.3. se može vidjeti kako igre novijeg datuma izgledaju realističnije u vizualnom smislu u odnosu na starije videoigre. Ali izgled nije jedina stvar koja se promijenila.

Slika 2.3. Usporedba stare inačice videoigre sa novom igrom Tomb raider (2013., 1996.).

Sama igrivost (engl. *gameplay*) se znatno promijenila od videoigara koje su primale jednostavne tekstualne naredbe do videoigra s veoma složenim upravljanjem poput: govora, fizičkog pokreta, kombinacija tipaka na tipkovnici, ...

Takav napredak je jedan od bitnijih faktora koji je doprinio zanimljivosti videoigara, jer s takvim složenim ulazom igrač može kontrolirati lika kao da je to njegovo vlastito tijelo. U tome su najuspješnije virtualna stvarnost (engl. *virtual reality*; VR) videoigre u kojima se lik upravlja pomoću senzora za pokrete što omogućava simuliranje kompletne fizike ljudskog tijela te rezultira preciznim pomicanjem lika u videoigri. Primjerice, igrač može birati proizvoljan kut pod kojim će zamahnuti mačem u srednjovjekovnim videoigrama. Ovakva mogućnost kontrole, i sposobnost igre da adekvatno reagira na takvu kontrolu, je bila nezamisliva prije nekoliko desetaka godina.



Slika 2.4. VR videoigra Blade and Sorcery (2018.).

Druga bitna stvar koja se promijenila u videoigrama jest interakcija igrača s okolinom. U FPS videoigrama starijeg datuma, primjerice simulacijama 2. svjetskog rata, nije bilo moguće uništavati zidove, upravljati vozilima i slično, odnosno akcije koje može poduzeti igrač su bili ograničene na mnoge načine u usporedbi s mogućnostima koje se pružaju u stvarnom životu. U

videoigrama novijeg datuma ne samo da je moguće upravljati razna vozila s veoma realističnom fizikom, već je i moguće totalno devastirati cijelu mapu, odnosno svaki objekt na mapi se može uništiti. Ovakve mogućnosti primarno ovise o količini programerskog posla koji je potreban za njihovu implementaciju, a ograničene su sa hardverskim mogućnostima razdoblja u kojem je videoigra razvijena. Stoga, tematika o uvođenju novih funkcionalnosti u videoigre je tematika na koju se proizvođači videoigara kontinuirano vraćaju kako moćniji hardver postaje pristupačniji ciljanom tržištu.

Primjerice, videoigra *Minecraft* je koncipirana na način da se svaki objekt na mapi može uništiti, sakupiti i iskoristiti u izgradnji nekog novog objekta. Radi takvog koncepta, Minecraft omogućuje igračima veliku slobodu u beskonačnom svijetu, od rađenja oružja do građenja svojih vlastitih kuća, putovanja kroz različite svjetove, borbe i trgovine sa ostalim likovima u igri koji mogu biti kontrolirani od strane drugih igrača ili od strane računala (*non-playable character*; NPC).Primjer ove videoigre se može vidjeti na Slici 2.5.

Slika 2.5. Minecraft (2011.).

S ovakvim ubrzanim razvojem u svijetu videoigara, mnogo ljudi je postalo zainteresirano i rado se upuštaju u taj svijet. Radi popularnosti koja se iz dana u dan povećava, počela su se održavati

natjecanja na globalnoj razini iz različitih videoigara. Jedno od najvećih natjecanja u videoigrama je *League of Legends World Championship* kojeg je 2018. godine pratilo 99.6 milijuna ljudi diljem svijeta, a prikazano je na Slici 2.6. Radi ogromnih uspjeha koja su takva natjecanja osvojila, započelo je osnivanje *Esports-a*.

ESports je elektronski sport koji obuhvaća natjecanje u videoigrama. Svakim danom sve više ljudi smatra ESport pravim sportom i postoji mogućnost da 2026. godine ESport uđe u Olimpijske sportove[3].

Slika 2.6. League of Legends World Championship.

2.2. Razvoj videoigara

Kao što se može zaključiti iz prethodnih potpoglavlja, moderne videoigre se sastoje od velikog broja različitih objekata, poput: igrivih i neigrivih likova, oružja, magije, stabla, trave, životinja, čudovišta i sličnog.

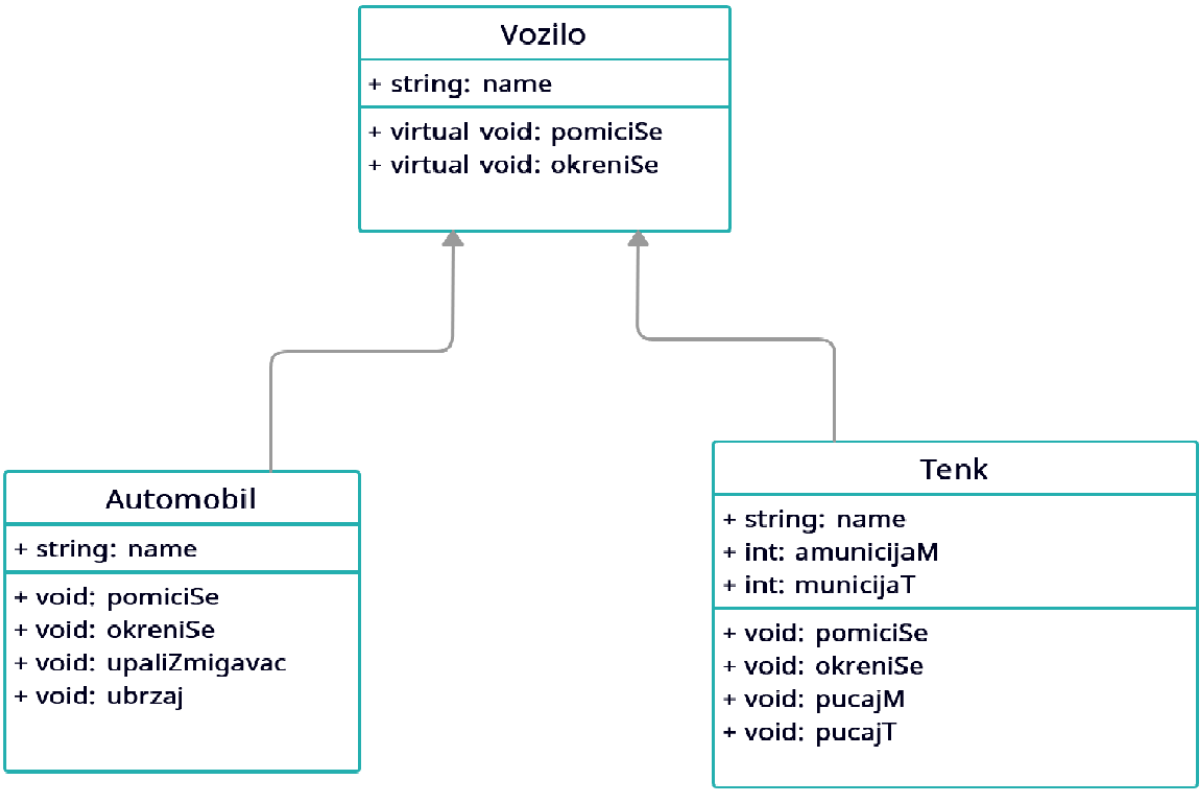
Uglavnom, svi ovi objekti imaju nekakvu hijerarhiju radi koje je najpogodnije koristiti objektno orijentirano programiranje. Takva hijerarhija se implementira na način da svaka klasa predstavlja jedan, uglavnom fizički, predmet unutar videoigre, a zatim se instanciranjem objekata tih klasa popunjava svijet unutar videoigre. Svaka klasa ima vlastite metode i atribute. Metode govore o funkcionalnosti te klase odnosno definira njeno ponašanje, a atributi su varijable te klase u

kojima instance te klase mogu pohranjivati svoje podatke. Uvijek bi trebalo težiti tome da svaka klasa izvodi jednu funkcionalnost, a složene objekte poput glavnog igrača gradimo pomoću više različitih klasa i komponenti koje će se implementirati unutar igračeve klase. Također, unutar klasa se implementiraju dodatne funkcije koje će omogućavati da objekt komunicira sa ostalim objektima koji se nalaze u svijetu.

2.2.1. Nasljeđivanje

Nasljeđivanje je jedna od najvećih prednosti koje objektno orijentirano programiranje pruža. Jedna od prednosti nasljeđivanja su apstraktne klase (engl. *abstract class*) koje se ne mogu instancirati već služe za definiranje svih bitnih varijabli i funkcionalnosti koja će ta klasa imati. Jednom kada se definira takva vrsta klase, nju mogu druge klase naslijediti i na taj način druge klase automatski dobiju sve varijable i funkcionalnosti od apstraktne klase. Druga bitna prednost nasljeđivanja je što potklasa može prilagođavati funkcionalnosti nadklase ili dodati nove funkcionalnosti i na taj način specijalizirati se za svoju namjenu.

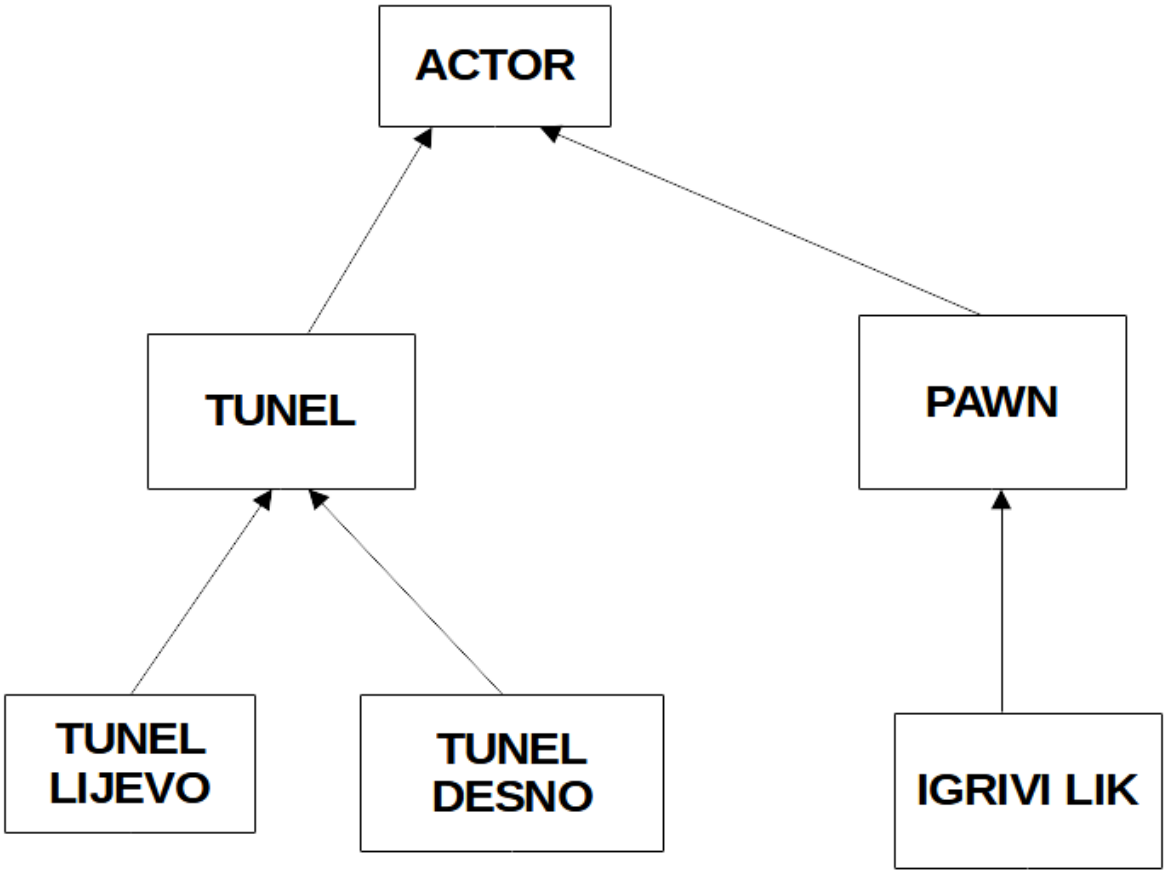
Za primjer sa Slike 2.7., apstraktna klasa *Vozilo* u sebi sadrži osnovne funkcionalnosti vozila. Zatim se može napraviti nova klasa *Automobil* koja bi naslijedila klasu *Vozilo* i proširilo ju na način da promijeni funkcionalnost za pomicanje. Također, može postojati još jedna klasa vozila poput *Tenk* koja bi proširila klasu *Vozilo* na način da bi prvo promijenila funkcionalnost za pomicanje, a zatim bi još dodala funkcionalnost za ciljanje i pucanje.



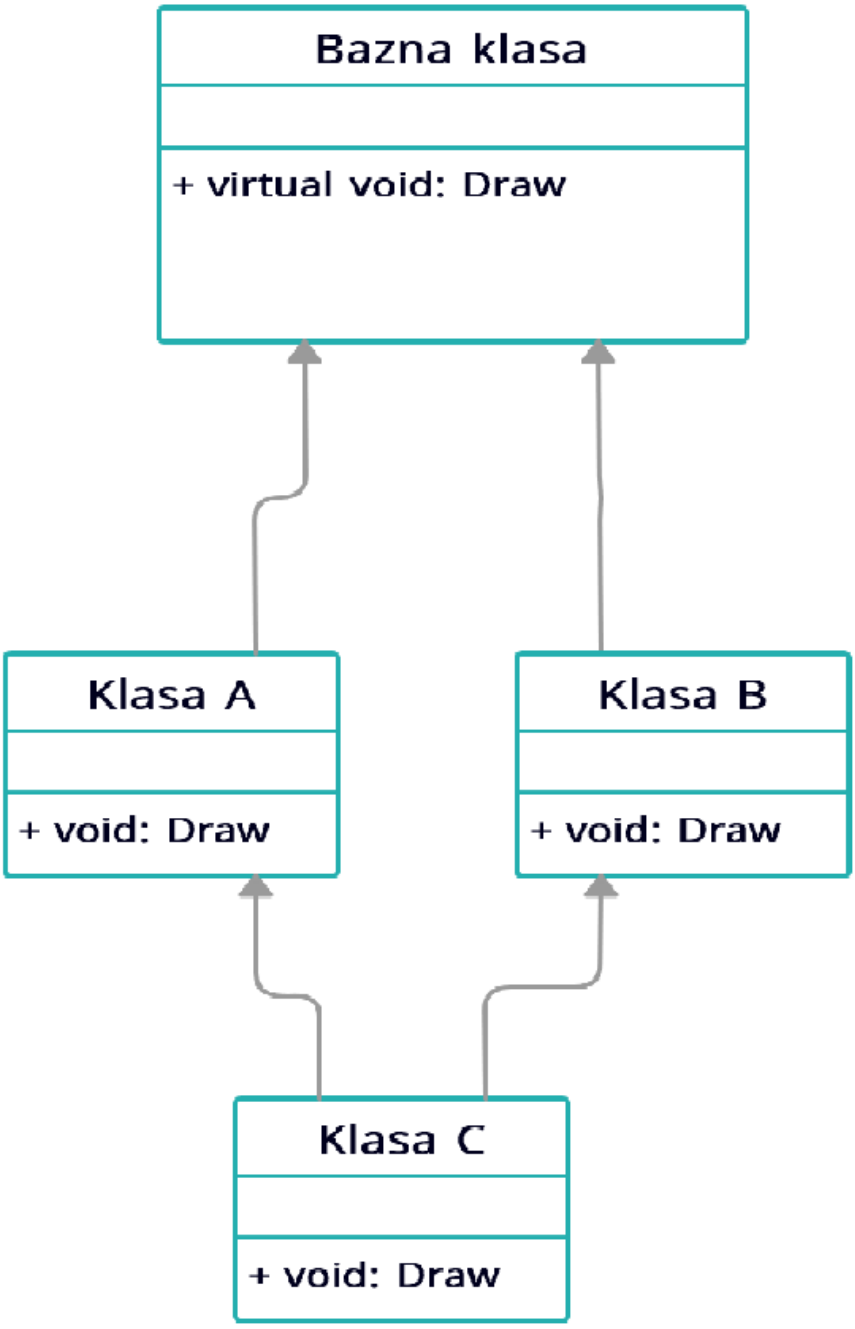
Slika 2.7. Primjer nasljeđivanja. Apstraktna klasa vozilo predstavlja „predložak”, a klase Automobil i Tenk predstavljaju specijalizacije te klase.

Uobičajena je praksa da unutar videoigre sve klase koje predstavljaju objekte u svijetu nasljeđuju od jedne bazne klase, a takav pristup se naziva monolitna klasna hijerarhija (engl. *monolithic class hierarchy*) [4]. Bazna klasa u takvoj hijerarhiji se najčešće naziva *Actor* zbog toga što je nju moguće vizualizirati u svijetu. Stoga, svaka klasa koja se naslijedi od ove klase će se također moći vizualizirati unutar videoigre te će zauzimati pripadajući prostor unutar svijeta. Ono što je bitno naglasiti je to da svaka nova klasa koja nasljeđuje baznu klasu također proširuje baznu klasu i prilagođava ju za svoje koristi.

Najčešće, videoigre imaju klasu *Pawn* koja nasljeđuje klasu *Actor* i implementira sučelje za navigaciju. Zatim svi igrivi i neigrivi likovi koji se mogu pomicati nasljeđuju tu klasu i prerađuju je na način da imaju jedinstveni algoritam za pomicanje, interakciju sa svijetom, itd. Primjer sa Slike 2.8. pokazuje kako klasa municija nasljeđuje izravno klasu *Actor* i implementira sve osnovne stvari koje objekt municija može sadržavati kao što su tip municije i količina. Zatim imamo dvije klase, raketa i metak, koji nasljeđuju klasu *Municija* i proširuju baznu klasu sa materijalima i teksturama. Na istoj slici se može vidjeti kako igrivi lik nasljeđuje klasu *Pawn* kako bi mogao implementirati svoje vlastiti algoritam za pomicanje i interakciju sa ostalim objektima.



Slika 2.8. Monolitna klasna hijerarhija .



Slika 2.9. Dijamantna struktura.

Problem s ovakvom hijerarhijom je što veoma lagano može doći do višestrukog nasljeđivanja (engl. *diamond inheritance*). Taj problem se javlja kada jedna klasa nasljeđuje istoimene metode iz različitih klasa, a program ne može odrediti koju od više istoimenih metoda pozvati. Primjerice, ako svaki objekt ima svoju vlastitu metodu za iscrtavanje na zaslon i ako je definirana klasa koja nasljeđuje tu metodu iz dvije različite klase, tada program neće znati na koji način se taj objekt treba iscrtati na zaslonu. Problem se može vidjeti na Slici 2.9.

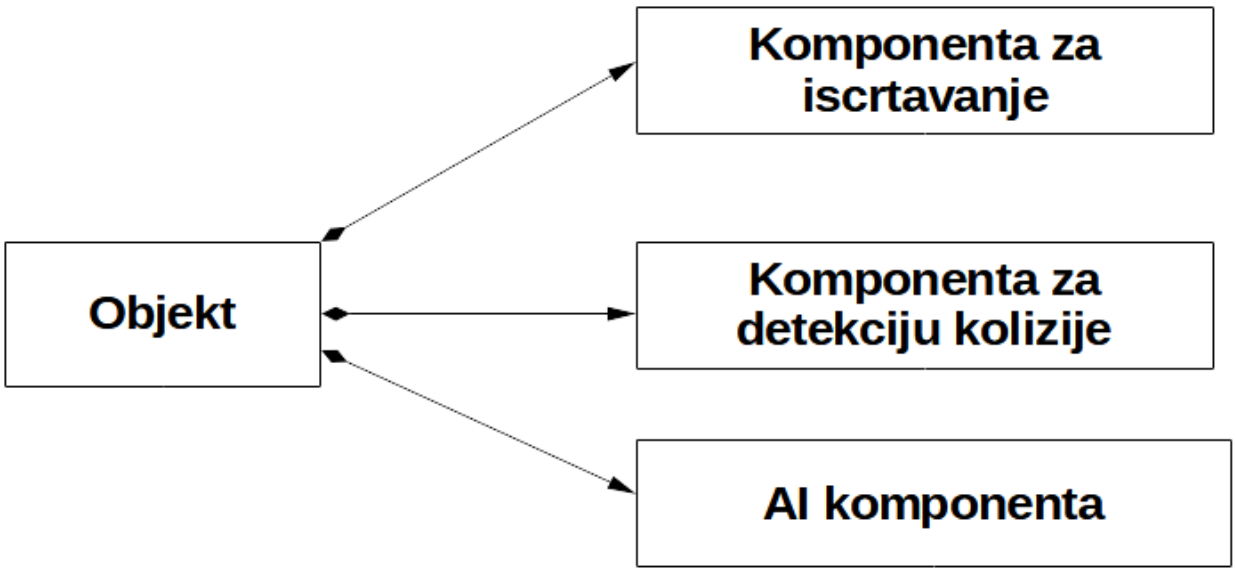
Drugi problem koji se javlja kod čiste monolitne hijerarhije klasa je što kroz nasljeđivanje svaka klasa poprima sve funkcionalnosti natklase, što ponekad može biti problematično. Možemo se zapitati da li svaka klasa treba imate sve funkcionalnosti natklase. U mnogim videoigrama postoje objekti koji su fizički nevidljivi, ali oni u svijetu postoje, a česti primjer toga su nevidljivi zidovi. Može se odmah zaključiti da nevidljivi objekti ne trebaju imati funkcionalnost za iscrtavanje na ekran, a oni će je svejedno naslijediti od svoje natklase. Takav pristup nije najefikasniji jer iako je objekt nevidiljiv, on treba pozvati funkciju za iscrtavanje koja mora učitati *shader*-e i obraditi ih. Na tom postupku se bespotrebno gubi jako puno resursa pa je preporučljivo izbjeći ga.

Kako bi se riješili zadani problemi, koristi se objektno orijentirano programiranje s komponentama koje izvršavaju jednu zadaću. Razlika je što svaki objekt u hijerarhiji može i ne mora implementirati različite komponente. Najčešće korištene komponente su komponenta za iscrtavanje na ekran, komponenta za upravljanje dimenzijama objekta, komponenta za detekciju kolizije, komponenta za umjetnu inteligenciju (engl. *artificial intelligenc*e; AI), itd.

Ako se za vrijeme razvoja videoigre primjeti da više različitih objekata koristi isti kod, tada je najbolje taj dupli kod prebaciti u novu komponentu koju će ti objekti implementirati i koristiti. Na taj način sav kod se nalazi na jednom mjestu i lakše ga je održavati.

Gornji problem se može riješiti definiranjem komponente za iscrtavanje na ekran, a zatim svaki objekt koji se treba prikazati na ekranu koristi tu komponentu i na taj način se rješava problem dijamantne hijerarhije.

Veza između objekta unutar videoigre i komponenata je prikazana na Slici 2.10.



Slika 2.10. Prikaz objekta sa komponentama.

Programski se pridruživanje komponenti objektima implementira na način da svaki objekt posjeduje nesoritrani skup koji sadrži sve komponente vezane za taj objekt, a komponente se mogu dodjeliti ili ukloniti korištenjem metoda za dodavanje ili uklanjanje komponenti. Ovakav način programiranja videoigra se za sada pokazao kao najbolji, jer rješava dva veoma bitna problema te veoma brzo i efikasno se mogu dodavati nove funkcionalnosti klasama [4].

Jedini problem koji se uvodi ovakvim načinom programiranja je problem komunikacije između dvije ili više komponenti. Ovaj problem se javlja jer komponente pojedinog objekta nemaju mogućnost za izravnu međusobnu komunikaciju, već se svaki put ta komunikacija mora odvijati kroz objekt koji prosljeđuje upite i odgovore između komponenti. Primjerice, komponenta za iscrtavanje zahtijeva informacije o lokaciji i orijentaciji objekta u svijetu kako bi ga mogla vizualizirati, a te podatke može saznati upitom prema komponenti za transformacije. Ako se izvodi puno takvih upita, može doći do pada performansi jer se stog veoma brzo puni, a sporo prazni.

2.2.2. Budućnost programiranja videoigara

U današnje vrijeme mnoge kompanije razve vlastiti *Game engine* prije nego što se ubace u razvoj videoigra. *Game Engine* je programski okvir (engl. *software framework*) koji u sebi sadrži različite knjižnice koje olakšavaju i ubrzavaju razvoj videoigra. Stroj se najčešće sastoji od komponenti za iscrtavanje slike na zaslonu (engl. *render*) za 2D i 3D grafiku, komponente sa algoritmima za simulaciju fizike, detekciju kolizije i komponente za reprodukciju zvuka. Razvoj stroja je bio potreban kako bi se mogao uskladiti rad svih osoba koji rade na zajedničkom razvoju videoigre.

Jedna od najvećih prednosti predstavljena u najpopularnijim strojevima je što omogućavaju izravan razvoj videoigra bez potrebe da se prevodi programski kod, što dizajnerima omogućuje mijenjanje teksture objekata bez da se vrše ikakve promjene u programskom kodu ili čekanje programere da naprave potrebne promjene.

U isto vrijeme, moderni strojevi omogućavaju testiranje videoigre unutar samog strojeva, što znači da nije potrebno izraditi izvršnu datoteku svaki put kada je potrebno testirati novu funkcionalnost, već je dovoljno samo pokrenuti određenu razinu (engl. *level*) unutar stroja.

Među najpoznatijim strojevima današnjice su: *Unreal Engine*, *Unity*, GameMaker i Godot[5]. Svi ti strojevi se razlikuju po programskom jeziku u kojem se piše programski kod videoigre. Primjerice, *Unreal Engine* koristi C++ kao programski jezik, dok *Unity* koristi C# kao programski jezik. Iako se razlikuju po programskom jeziku i knjižnicama koje se koriste u samom stroju, većina funkcionalnosti koje nam strojevi pružaju su iste što znači da početnik koji tek započinje sa razvijem svoje prve videoigre može odabrati bilo koji stroj i očekivati da će imat sve glavne funkcionalnosti na raspolaganju, a to uključuje: objektno orijentirano programiranje, komponente, vizualni uređivač za stvaranje razina, testiranje proizvoljne razine u stvarnom vremenu, itd.

Budući da razvoj videoigara postaje sve popularniji iz dana u dan, različite kompanije su počele razvijati svoje vlastite strojeve kako bi olakšale posao svojim zaposlenicima. Iste te kompanije su također odlučile učiniti svoj stroj javno dostupnim svim zainteresiranim ljudima kako bi ih potaknuli da razvijaju svoje vlastite videoigre koristeći njihov stroj. Pošto su se strojevi pokazali veoma efikasni za razvoj manjih videoigara, došlo je do velikog vala razvoja malih videoigra koje su napravljene od strane jedne osobe (engl. *Indie games*).

Pošto je interes za razvoj vlastitih videoigra značajno porastao među pojedincima, kompanije su odlučile nadograditi svoje strojeve dodavanjem takozvanih *nacrta* (engl. *blueprints*). Nacrti omogućavaju osobama bez znanja o programiranju i funkcioniranju programskog koda da razvijaju vlastitu videoigru samo uz poznavanje logike.. Kompanije su takav način razvoja videoigra približile svima, a čak i djeci koja još uvijek ne znaju kako pisati vlastiti programski kod. Ovakav oblik razvoja se više poistovjećuje sa slaganjem puzli nego programiranjem, a slično je programskom jeziku Scratch koji se koristi za podučavanje djece programskim konceptima. Nacrti će u detalje biti objašnjeni u sljedećem poglavlju.

3. UNREAL ENGINE 4

Unreal Engine (*UE*) je stroj za razvoj 3D računalnih videoigra kojeg je razvio *Epic Games*, a prvi put je prikazan u igri pucačine u prvom licu *Unreal* iz 1998. godine. U početku je stroj bio razvijen isključivo za pucačine u prvom licu, a danas se UE može koristiti za razvijanje skoro svih žanrova videoigra te čak za simulaciju fizike kod robota[6,7]. Također je doživio usvajanje u drugim industrijama poput filmske i televizijske industrije zbog mogućnosti izvođenja vrlo realističnih animacija.

UE koristi C++ programski jezik koji mu omogućava visok stupanj prenosivosti i podršku na širokom rasponu platformi za stolna računala, mobilne uređaje, konzole i virtualnu stvarnost.

Izvorni kod UE4 je javno i slobodno dostupan na *GitHub-u* nakon registracije korisničkog računa, a komercijalna upotreba dopuštena je na temelju modela tantijema (engl. *royalty model*) na platformi *EpicGames*. Epic se odriče svoje marže za sve igre koje su objavljene na *Epic Games Store-u* dokle god igra ostvaruje prihode manje od 1 milijuna američkih dolara. Epic je u stroj uključio značajke kupljenih tvrtki poput *Quixela*, čemu je pomogao prihod videoigre *Fortnitea*.

Jedna od glavnih značajki planiranih za UE4 je bilo globalno osvjetljenje u stvarnom vremenu korištenjem *voxel cone tracing-a*, eliminirajući unaprijed izračunato osvjetljenje[8]. Međutim, ova značajka, nazvana *Sparse Voxel Octree Global Illumination (SVOGI),* i prikazana u videoigri Elemental demo-u, je zamijenjena sličnim, ali računalno manje skupim algoritmom zbog problema s performansama[9].

Također, UE4 uključuje novi način programiranja, programiranje korištenjem nacrta. Bolje rečeno vizualni sustav skriptiranja, koji omogućuje brzi razvoj logike igre bez pisanja programskog koda, što rezultira smanjenjem podjela između tehničkih umjetnika, dizajnera i programera, a nacrte je moguće miješati i sa programskim kodom. Stoga, nije potrebno opredijeliti se za jedan od ova dva načina razvoja [10].

Kako se cijeli UE4 pokazao kao veliki hit među programerima i kompanijama, Epic je odlučio izdati UE4 za škole i sveučilišta besplatno, uključujući osobne kopije za studente upisane u akreditirane programe za razvoj videoigre, informatiku, umjetnost, arhitekturu i ostale povezane studijske programe[11].

U ožujku 2015. Epic je izdao UE4, zajedno sa svim budućim ažuriranjima, besplatno za sve korisnike kako bi potaknuli još veći i brži razvoj videoigara [12,13].

Nakon što je Epic odlučio prvo preći na sustav pretplate za UE4, a zatim na besplatnu distribuciju UE4, korištenje stroja je poraslo za 10 puta, te je vjerojatno da će se još više koristiti kako vrijeme bude prolazilo [14].

3.1. Razvoj videoigara uz UE4

Razvoj videoigara u motoru UE4 se odvija kroz dva načina od kojih svaki ima svoje prednosti i mane:

1. Razvoj videoigre kroz pisanje programskog koda,
2. Razvoj videoigre kroz stroj UE4.

3.1.1. Razvoj videoigre kroz pisanje programskog koda

UE4 koristi C++ programski jezik s dodatnim knjižnicama koje omogućavaju još lakši razvoj. Preporuka je da se za pisanje programskog koda koristi *Visual Studio*, ali mogu se koristiti i drugi uređivači.

Čim se stvori novi UE4 projekt, s njime se automatski dobiju sve standardne klase poput *Actor i Pawn*, komponente i funkcije. Stoga, prvi korak u razvoju videoigre je kreirati vlastite klase koje nasljeđuju klasu *Actor* ili neku drugu klasu ovisno o funkcionalnosti koja se želi postići.

Novokreirana klasa je automatski podijeljena u dvije datoteke: *naziv_klase.cpp* i *naziv_klase.h*. Takva podjela se primarno radi kako bi se programerima olakšalo snalaženje u samom kodu. Datoteka s ekstenzijom .h označava *header* datoteku te u njoj je samo definirana struktura te klase. Bolje rečeno u toj datoteci se nalaze sve varijable, popis metoda koje klasa ima, koju klasu nasljeđuje i koja sučelja ona implementira. Nakon što se definira *header* datatoteka, u datoteci .cpp se implementiraju funkcionalnosti metoda i konstruktor te klase. Primjerice, za implementirati metak, najprije je potrebno definirati klasu *ABullet* i njene varijable i metode kao što je prikazano na Slici 3.1. Sa slike se kod deklaracija varijabli i metoda mogu primijetiti ključne riječi *public* i *protected*. Ključna riječ *public* omogućava izravan pristup unutarnjoj varijabli ili metodi objekta od strane vanjskih objekata. U ovome slučaju je ključna riječ *public* iskorištena za tri člana. Prvi član je konstruktor *ABullet* koji mora imati javni pristup kako bi se objekt mogao instancirati. Drugi član sa javnim pristupom je funkcija *Tick* koju stroj poziva prilikom svakog okvira (engl. *frame*), a u njoj je potrebno implementirati ono što je potrebno izvršiti prilikom svakog iscrtavanja slike na ekran. Posljednja funkcija sa javnim pristupom je *OnBulletHit* koja se pozove kada metak udari u drugi objekt. Ključna riječ *protected* ograničava pristup varijablama i metodama klase samo metodama te klase ili njenih potklasa. Jedina funkcija označena ovom ključnom riječi je BeginPlay koja se izvrši samo jednom i to prilikom stvaranja objekta u svijetu. Ovom riječi je također ograničen pristup za tri komponente: BulletMovement koja upravlja pomicanjem metka, BulletMesh koja definira izgled metka kroz teksture, oblik, kolizije, i slično te MaterialBullet koja omogućava dinamičku promjenu materijala metka. Zaštićena varijabla turretFlag se koristi za označiti je li metak ispaljen od strane igrača ili lika upravljanog od strane računala.

```
public:
    // Sets default values for this actor's properties
    ABullet();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

    UPROPERTY(EditAnywhere, Category = "Components")
    class UProjectileMovementComponent* BulletMovement;

    UPROPERTY(EditAnywhere, Category = "Material")
    class UMaterial* MaterialBullet;

    UPROPERTY(EditAnywhere, Category = "Components")
    UStaticMeshComponent* BulletMesh;


    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Components")
    bool turretFlag;
public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    UFUNCTION()
    void OnBulletHit(AActor* SelfActor, AActor* OtherActor, FVector NormalImpulse, const FHitResult& Hit);

};
```

Slika 3.1. Datoteka (zaglavlja) bullet.h.

Također se sa Slike 3.1. može vidjeti linija koja sadrži „UPROPERTY(EditAnywhere, Category = „Components”)” što stroju UE4 signalizira da se toj varijabli može pristupati direktno iz sučelja stroja UE4 te će se nalaziti unutar kategorije *Components* [15]. Slično vrijedi i za *UFUNCTION()*. Budući da nisu da nisu postavljeni parametri unutar *UFUNCTION()*, stroj UE4 neće moći pronaći odgovarajuću funkciju unutar klase, te toj funkciji se neće moći pristupiti iz grafičkog sučelja stroja UE4.

Sa Slike 3.2.. se može vidjeti implementacija konstruktora i funkcije *BeginPlay*. Kao što je već rečeno, unutar konstruktora se postavljaju varijable na svoje početne vrijednosti. Tu je potrebno prvo instancirati sve komponente koje su navedene u *header* datoteci, zatim reći klasi koja je glavna komponenta te stvoriti graf scene. U ovom slučaju se kao glavna komponenta uzima *BulletMesh* jer je u njoj definiran izgled samog objekta. To znači da se sve druge komponente metka moraju micati u skladu s njegovom fizičkom lokacijom. Primjerice, na metak se može dodati kamera te kako se metak pomiče automatski se i kamera pomiče te ona zadržava svoj relativni odnos u odnosu na komponentu na glavnu kompunentu na koju je vezena.

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

    BulletMesh = CreateDefaultSubobject<UStaticMeshComponent>("BulletMesh");
    SetRootComponent(BulletMesh);

    BulletMovement = CreateDefaultSubobject<UProjectileMovementComponent>("BulletMovement");
    BulletMovement->InitialSpeed = 15000.f;
    BulletMovement->MaxSpeed = 15000.f;
    BulletMovement->ProjectileGravityScale = 0;

    MaterialBullet = CreateDefaultSubobject<UMaterial>("MaterialBullet");

    OnActorHit.AddDynamic(this, &ABullet::OnBulletHit);
    turretFlag = false;
}

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

    AActor* mojLik = GetOwner();
}
```

Slika 3.2. Datoteka bullet.cpp.

U sljedećem potpoglavlju će biti pokazano kako se ovo ovo ponašanje može implementirati koristeći nacrte i stroj UE4.

3.1.2. Razvoj videoigre kroz stroj UE4

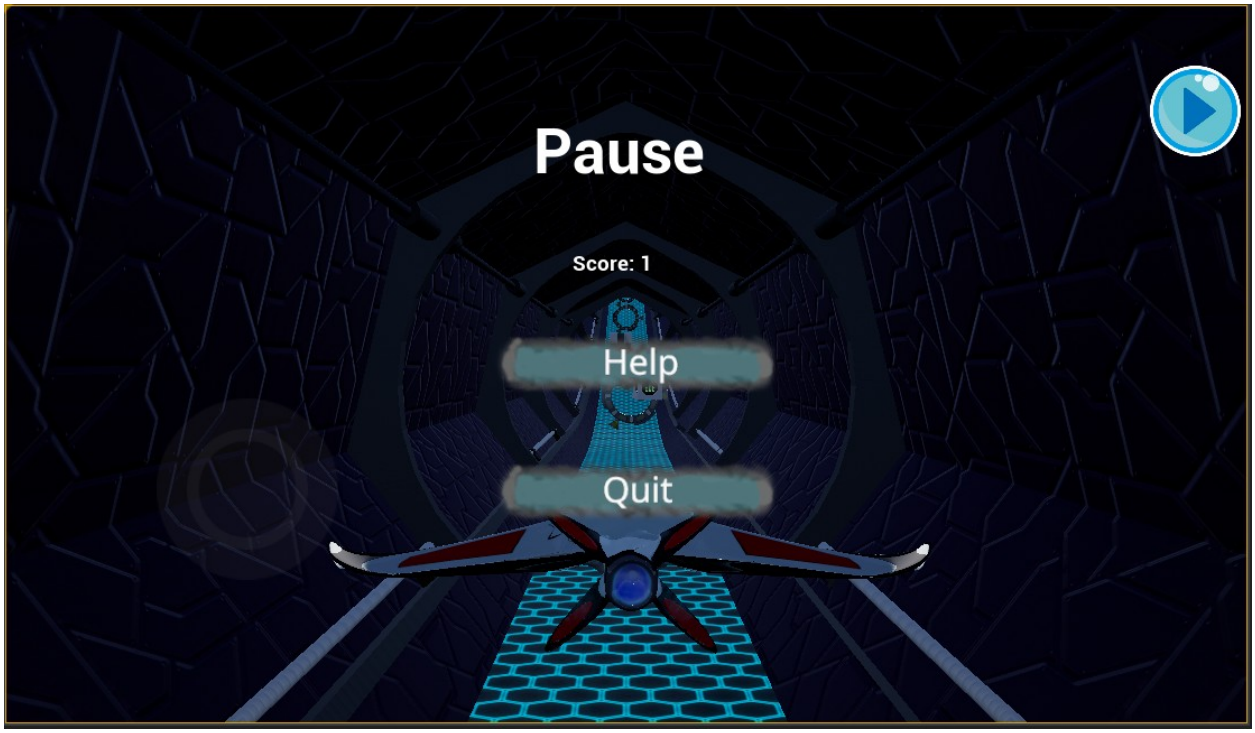
Stroj UE4 omogućava brz i efikasan razvoj videoigara, jer nije potrebno znati programirati. U samom uređivaču može se lagano stvarati i modelirati izgled i ponašanje svake razine. Preporuka je da igra uvijek imam barem tri razine i to su početna razina, igriva razina i krajnja razina.

Početna razina je razina koja se prikaže tek kada igrač pokrene videoigru. Na njoj se najčešće nalazi glavni izbornik po kojem se igrač može navigirati. Početna razina je prikazana na Slici 3.3.

Slika 3.3. Početna razina.

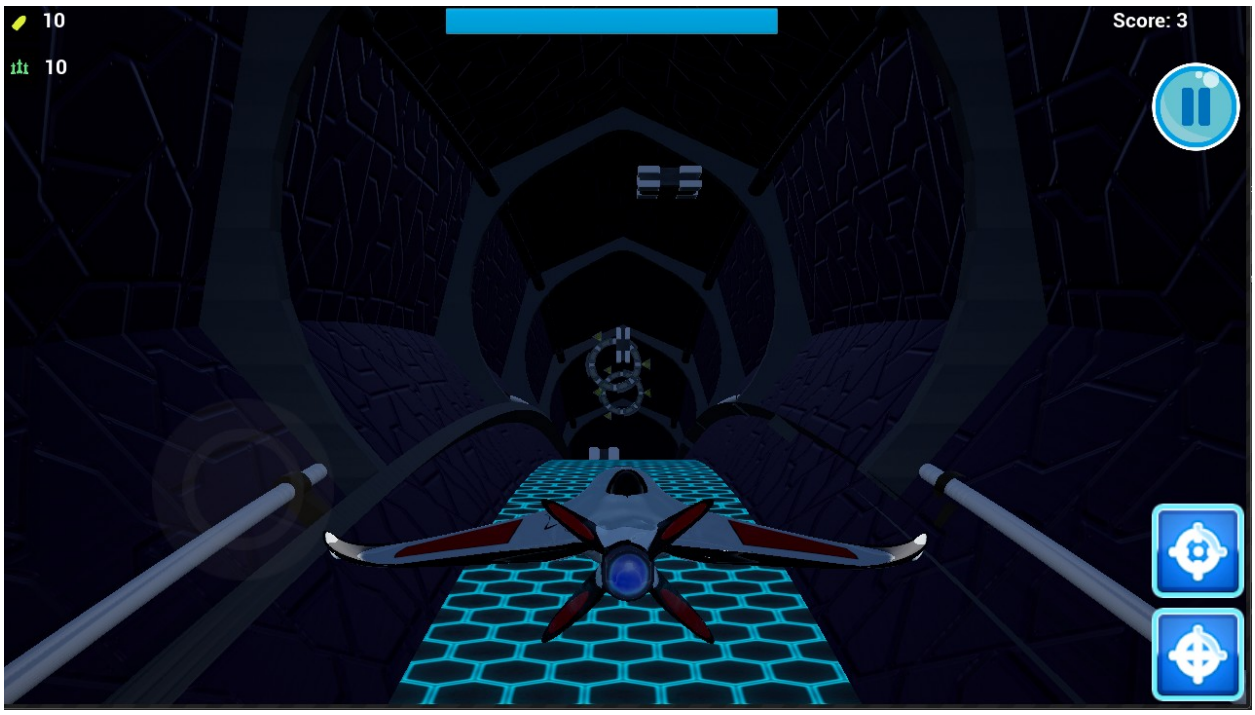
Sa Slike 3.3. se može vidjeti da u izborniku postoji nekoliko gumbova. Prvi gumb (*PLAY NOW*) služi za započet videoigru i prebacuje igrača na igrivu razinu. Drugi gumb (*HELP*) otvara razinu u kojoj se nalaze upute sa objašnjenjima kontrole i na koji način se videoigra igra. Na samom kraju se nalaze gumbovi *SCOREBOARD* i *QUIT* koji služe za prikaz ljestvice sa prvih deset najvećih postignutih uspjeha (engl. *score*) odnosno za ugasiti videoigru.

Sljedeća razina je igriva razina u kojoj igrač treba pomoću zadanih kontrola upravljati svog lika i igrati videoigru. Cilj igrača je preživjeti što dulje tako da izbjegava prepreke ili da ih uništi s mecima i raketama. Igriva razina se sastoji od dva načina rada: *Pause state* i *Unpause state*. Kada je igra u *Pause state-u*, igrač ne može upravljati svojim likom i stanje cijelog svijeta je zamrznuto. U tom trenutku igraču se prikazuje jednostavan izbornik u s napomenom da je videoigra trenutno pauzirana, igračev trenutačni uspjeh, gumb za dobivanje pomoći s uputama za igranje i gumb *QUIT* koji igrača vraća na početnu razinu. Na Slici 3.4. se vidi videoigra u pauziranom stanju.



Slika 3.4. Igriva razina u pauziranom stanju.

Kada se igriva razina nalazi u normalnom stanju, tj. nije pauzirana, igrač koristi virtualnu komandnu palicu za pomicanje broda gore, dolje, lijevo i desno unutar tunela. Na sredini zaslona se nalaze dva gumba jedan za ispucavanje raketa, a drugi za ispucavanje metaka. U gornje desnom kutu se nalazi gumb za pauzirati igrivu razinu, a u gornjem lijevom kutu se nalaze dvije sličice s tekstom. Prva sličica i tekst predstavljaju ikonicu metka te koliko još metaka igrač ima na raspolaganju za ispucavanje, a druga sličica i tekst predstavljaju isto ali za rakete. Na samom vrhu zaslona se vidi plava crta koja predstavlja koliko života igrač još ima. U slučaju da se igrač zabije u zid tunela ili u prepreku, život pada na nulu i igra prelazi u krajnju razinu. U igri također postoje topovske kupole koje gađaju igrača te kod njih količina života dolazi do izražaja budući da igrač može podnijeti više više od jednog pogotka. Na Slici 3.5. je prikazana igriva razina u normalnom stanju.



Slika 3.5. Igriva razina u normalnom stanju.

Na samom kraju je krajnja razina koja se učita nakon što igriva razina završi odnosno igračevi životi padnu na nulu. Na toj razini se prikazuje deset najboljih uspjeha, igračev uspjeh i tri gumba. Prvi gumb (*Play again*) služi za pokretanje igrive razina ispočetka. Sljedeći gumb (*Main Menu*) služi za povratak na početnu razinu te posljednji gumb je za izlazak iz videoigre. Krajnja razina je prikazana na Slici 3.6.

Slika 3.6. Krajnja razina.

Na Slici 3.3. je također prikazan stroj UE4 koji nam omogućava testiranje videoigre unutar samog uređivača. Ova funkcionalnost uvelike ubrzava razvoj videoigre jer nije potrebno prilikom svakog testiranja izgraditi izvršnu datoteku koju je zatim potrebno instalirati na pametni telefon. Osim tri glavne razine koje su dostupne igračima, postoji još razina za testiranje videoigre. Tu razinu programeri sami slažu kako bi mogli brzo i efikasno testirati rubne situacije koje se mogu javiti za vrijeme igranja videoigre. Dobra je navika nove mogućnosti poput funkcionalnosti i klasa najprije implementirati i testirati na testnoj razini prije nego se pusti u igrivu razinu. Kada se nova mogućnost pusti u igrivu razinu, potrebno je dodatno testiranje kako bi se utvrdilo ponašanje u realnoj situaciji, a na samom kraju učinimo novu mogućnost dostupnu svim igračima.

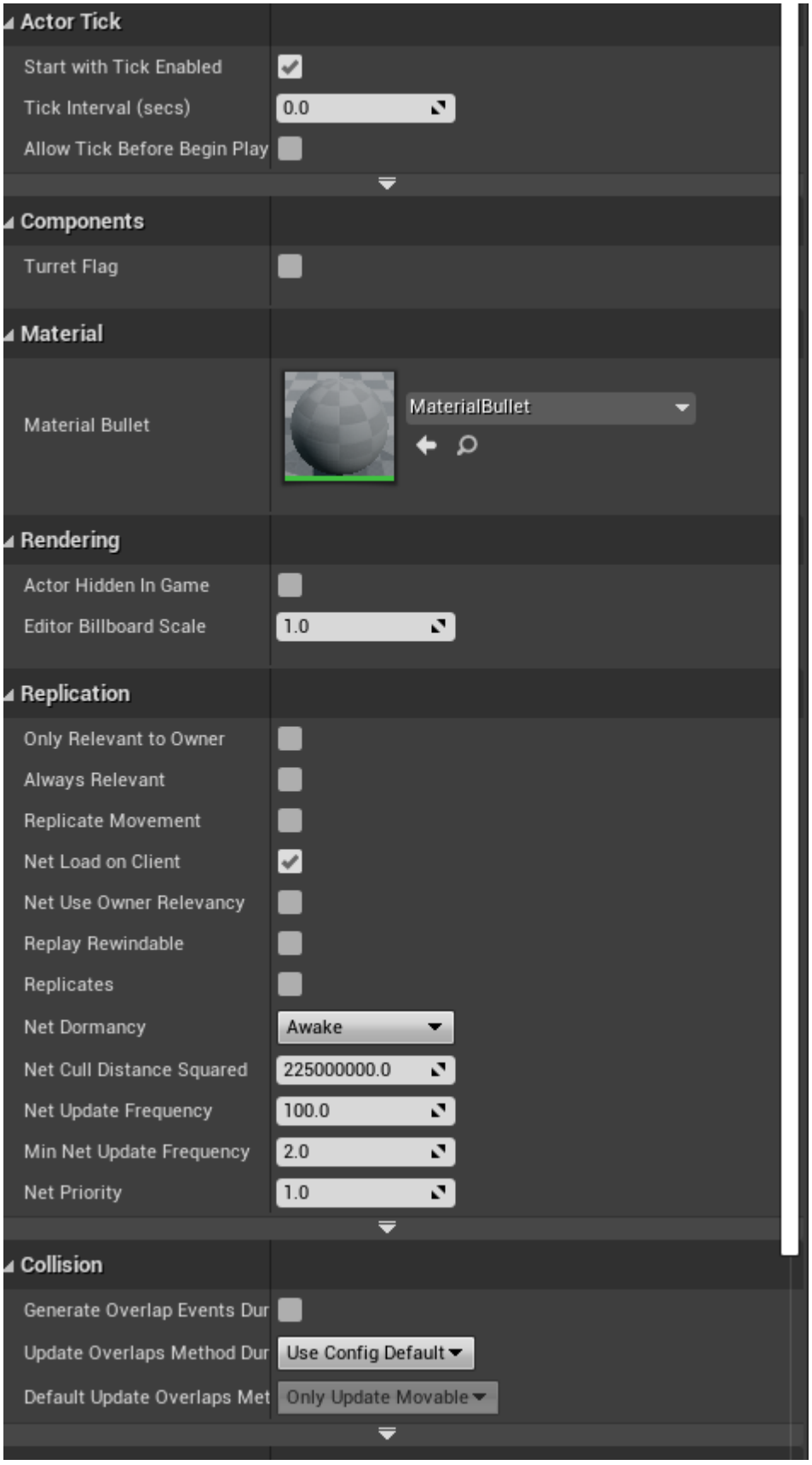
Odmah pored prikazanih razina vidi se struktura cijelog projekta te se tu mogu pronaći C++ klase, razine, teksture, nacrti i ostalo.

Na lijevoj strani zaslona se mogu vidjeti uobičajeni UE4 objekti pomoću kojih se može graditi razine. Također, na tom dijelu uređivača se nalaze kistovi pomoću kojih se može geometrijski „crtati” razinu. Primjerice, ravnu površinu se može s određenim kistovima ispupčiti kako bi se dobio planinski okoliš.

U gornjem dijelu zaslona se nalaze različite opcije koje uređivač nudi, poput: gumba za različite načine testiranja videoigre, prevođenje programskog koda i izgradnju izvršne datoteke,nacrti i ostalo.

S desne strane prikazana je konstrukcija razine, što se sve nalazi na razini. Ispod toga se nalaze različite opcije svijeta ili objekta koji je trenutno označen. U tim opcijama mogu se pronaći sve opcije koje klasa posjeduje uključujući i one opcije koje je sam programer dodao pomoću naredbe *UPROPERTY* [15].

Primjerice, ako otvorimo nacrt metka ili kliknemo na sam metak koji se nalazi na nekoj razini, s desne strane će nam se prikazati prozor sa svim varijablama i komponentama koje smo označili sa UPROPERTY, kao što je prikazano na Slici 3.7.

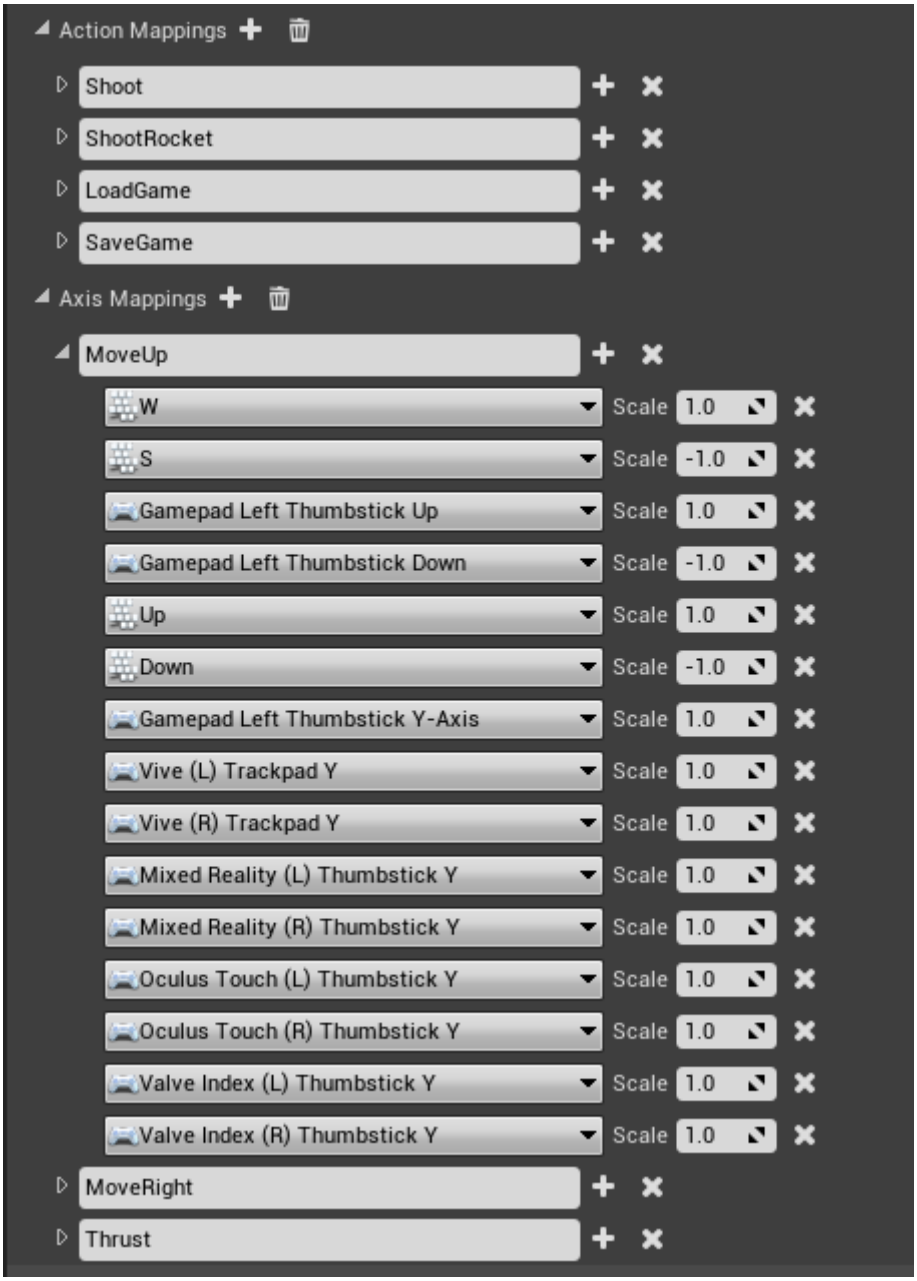


Slika 3.7. Opcije klase Metak unutar nacrta.

Tu se krije najveća prednost uređivača jer opcije objekta definirane u kodu se mogu mijenjati kroz stroj UE4 bez mijenjanja programskog koda. To znatno ubrzava cijeli proces razvoja videoigre, jer programeri mogu definirati varijable o kojima ovisi ponašanje objekta, a tehnički umjetnici zatim prilagođavaju njihove vrijednosti za samu videoigru. Jednom kada se te opcije prilagode, tehnički umjetnik ih odmah može testirati na igrivoj razini i vidjeti je li zadovoljan dobivenim rezultatom. Sve to je moguće bez konstantnog prevođenja i izgradnje izvršne datoteke.

Još jedna prednost stroja UE4 je veoma lagana implementacija ulaznih naredbi u videoigru. Unutar opcija cijelog projekta se nalazi specijalizirano grafičko sučelje za dodavanje ulaznih komandi, a prikazano je na Slici 3.8. Sve ulazne komande se mogu svrstati u jedno od dvije kategorije: *Action Mappings* i *Axis Mappings* [15].

Razlika između te dvije vrste ulaznih komandi je što *Action Mappings* poprimaju vrijednosti 0 ili 1, što označava je li igrač pritisnuo određeni gumb ili nije odnosno informacije o intenzitetu pritiska nisu dostupne. Takve ulazne komande se koriste za akcije koje se trebaju izvršiti nebitno o jačini pritiska, poput: ispucavanje metka ili rakete, skakanje i slično. S druge strane, *Axis Mappings* poprimaju sve vrijednosti u razmaku od 0 do 1 odnosno od -1 do 0. Uglavnom je ta dva slučaju potrebno razlomiti jer najčešće se koriste dva gumba za pomicanje igrača po jednoj osi. Primjerice, pomicanje svemirskog brod gore odnosno dolje su dvije različite komande, jedna za gore (od 0 do 1) i jedna za dolje (od -1 do 0). Međutim, te dvije komande je moguće spojiti u jednu ako je na raspolaganju posebna vrsta kontrolera sa palicom koja se može pomicati gore odnosno dolje. U slučaju kada palica miruje, stanje je 0, a inače se kreće u rasponu od -1 do 1 ovisno o položaju palice. Za videoigru razvijenom kroz ovaj rad, koristi se virtualni kontroler koji omogućava spajanje te dvije komande u jednu. Na Slici 3.8. je prikazano sučelje za jednostavno dodavanje ulaznih komandi.



Slika 3.8. Prikaz ulaznih komandi kroz uređivač.

Kada su sve potrebne ulazne komande mapirane sa imenima funkcija, potrebno je još u programskom kodu odnosno nacrtu implementirati funkcionalnost za ulazne komande. Ukoliko nikakva funkcionalnost nije implementirana, ulazna komanda neće imati nikakav utjecaj. Na Slici 3.9. je prikazan primjer programskog koda za dodavanje funkcionalnosti ulaznim komandama.

```
void ARocket_gamePawn::SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent)
{
    // Check if PlayerInputComponent is valid (not NULL)
    check(PlayerInputComponent);

    // Bind our control axis' to callback functions
    PlayerInputComponent->BindAxis("Thrust", this, &ARocket_gamePawn::ThrustInput);
    PlayerInputComponent->BindAxis("MoveUp", this, &ARocket_gamePawn::MoveUpInput);
    PlayerInputComponent->BindAxis("MoveRight", this, &ARocket_gamePawn::MoveRightInput);
    PlayerInputComponent->BindAction("Shoot", IE_Pressed, this, &ARocket_gamePawn::Shoot);
    PlayerInputComponent->BindAction("ShootRocket", IE_Pressed, this, &ARocket_gamePawn::ShootRocket);
    PlayerInputComponent->BindAction("LoadGame", IE_Pressed, this, &ARocket_gamePawn::SetLoadGame);
    PlayerInputComponent->BindAction("SaveGame", IE_Pressed, this, &ARocket_gamePawn::SetSaveGame);
}

void ARocket_gamePawn::ThrustInput(float Val){ ... }

void ARocket_gamePawn::MoveUpInput(float Val)
{
    // Target pitch speed is based in input
    float TargetPitchSpeed = (Val * TurnSpeed * -1.f);

    // When steering, we decrease pitch slightly
    TargetPitchSpeed += (FMath::Abs(CurrentYawSpeed) * -0.2f);

    // Smoothly interpolate to target pitch speed
    CurrentPitchSpeed = FMath::FInterpTo(CurrentPitchSpeed, TargetPitchSpeed, GetWorld()->GetDeltaSeconds(), 2.f);
}
```

Slika 3.9. Dodavanje funkcionalnosti ulaznim komandama.

Sa Slike 3.9. se može vidjeti primjer vezivanja funkcionalnosti za određenu ulaznu komandu, a ovisno o vrsti ulaza se mora koristiti funkcije *BindAxis* i *BindAction*. Zatim se toj funkciji kao parametre šalje ime ulazne komande koje mora odgovarati imenu unutar stroja UE4 kao što je prikazano na Slici 3.8., a s ciljem kako bi bilo jasno koja funkcija i iz koje klase je zadužena za procesiranje ulaznih komandi. Primjerice, ako igrač iskoristi virtualni kontroler za pomicanje svemirskog broda prema gore, pozvat će se funkcija *MoveUpInput* u kojoj se izračunava *CurrentPitchSpeed* koja se dalje koristi za pomicanje i rotiranje svemirskog broda prema gore. Za računanje trenutačne brzine koristi se linearna interpolacija između prošle i nove brzine kako bi se postiglo glatko ubrzanje i rotacija prema gore. Zatim se u *Tick* funkciji izvršavaju rotacije i ubrzavanje svemirskog broda, a za igrivog lika je prikazana na Slici 3.10.

```
void ARocket_gamePawn::Tick(float DeltaSeconds)
{
    const FVector LocalMove = FVector(CurrentForwardSpeed * DeltaSeconds, 0.f, 0.f);

    // Move plan forwards (with sweep so we stop when we collide with things)
    AddActorLocalOffset(LocalMove, true);

    // Calculate change in rotation this frame
    FRotator DeltaRotation(0,0,0);
    DeltaRotation.Pitch = CurrentPitchSpeed * DeltaSeconds;
    DeltaRotation.Yaw = CurrentYawSpeed * DeltaSeconds;
    DeltaRotation.Roll = CurrentRollSpeed * DeltaSeconds;

    // Rotate plane
    AddActorLocalRotation(DeltaRotation);

    float NewForwardSpeed = CurrentForwardSpeed + (DeltaSeconds * Acceleration);
    // Clamp between MinSpeed and MaxSpeed
    CurrentForwardSpeed = FMath::Clamp(NewForwardSpeed, MinSpeed, MaxSpeed);

    Super::Tick(DeltaSeconds);

    SaveGameLogic(DeltaSeconds);
    LoadGameLogic(DeltaSeconds);

    URocketGameInstance* GI = Cast<URocketGameInstance>(UGameplayStatics::GetGameInstance(GetWorld()));
    UE_LOG(LogTemp, Warning, TEXT("coins: %d"), GI->coins);
}
```

Slika 3.10. Tick funkcija igrivog lika.

S ovim potpoglavljem smo završili uvod u UE4 i pokazali koje su sve mogućnosti dostupne kroz uređivač i programski kod te kako se njihovim kombiniranjem može iskoristiti punu moć koju nam pruža UE4.

3.2. Nacrti (*Blueprints*)

Nacrt je novi način programiranja unutar stroja UE4, a omogućavaju brzo i efikasno programiranje logike videoigre. Budući da se videoigre sastoje od mnogo različitih komponenti, postoji i više različitih tipova nacrta. Svi tipovi nacrta nasljeđuju standardni nacrt koji daje slobodu programerima i tehničkim umjetnicima da stvaraju nove tipove nacrta za razvoj videoigre.

Tipovi nacrta:

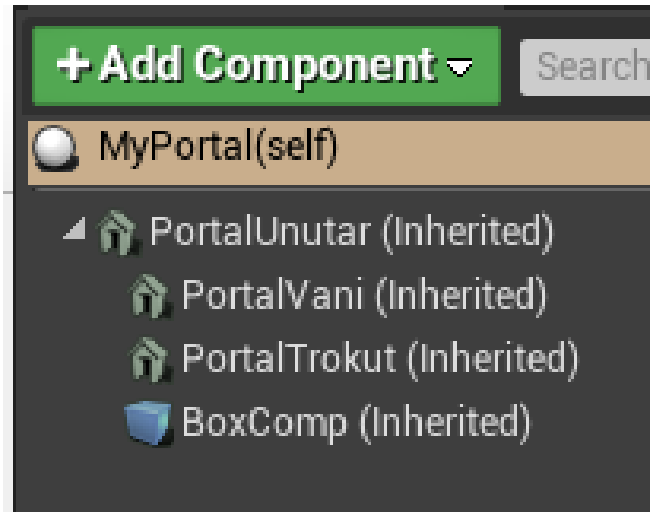
1. standardni nacrt,
2. knjižnica funkcija kao nacrt,

3. sučelje kao nacrt,
4. knjižnica makroa kao nacrt.
5. grafičko sučelje kao nacrt

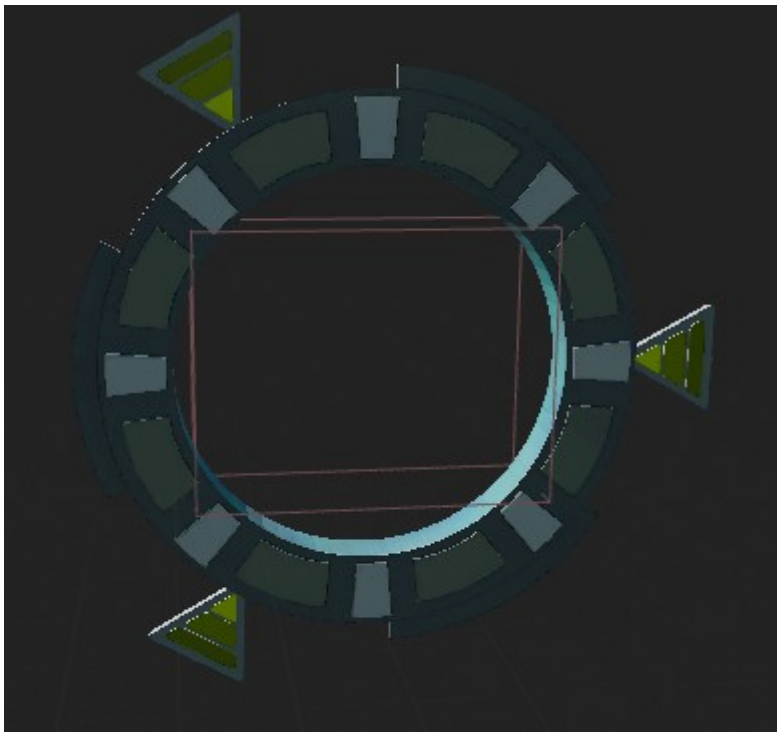
Najčešće se koriste standardni nacrti i nacrti za stvaranje grafičkog sučelja. Nacrti 1. i 5. tipa imaju *Viewport*, dio uređivača u kojem se prikazuje fizički izgled objekta ili sučelja. U tom načinu rada definira se ono što igrač vidi: teksture, veličina i parametre objekta/sučelja i slično. Kroz *Viewport* se također mogu dodavati komponente na objekt i dobiti pregled cijelog stabla komponenti. Klikom na komponentu ili na sam objekt otvaraju nam se njegove opcije u kojima se mogu mijenjati različite parametre objekta/komponente. *Viewport* je prikazan na slici 3.11. a), b) i c).

Nacrti 2. i 4. tipa su nacrti u kojima se čuvaju implementacije funkcionalnosti koje se koriste na više mjesta u projektu, a funkcioniraju poput knjižnica u programskim jezicima.

Razlika između 3. i 4. tipa nacrta je u tome što sučelje kao nacrt je više kao knjižnica u kojoj ćemo čuvati sve funkcionalnosti koje će onda grafičko sučelje koristiti, dok grafičko sučelje kao nacrt nam omogućuje da dizajniramo cijelo grafičko sučelje kako će izgledati i pozivati različite funkcije.



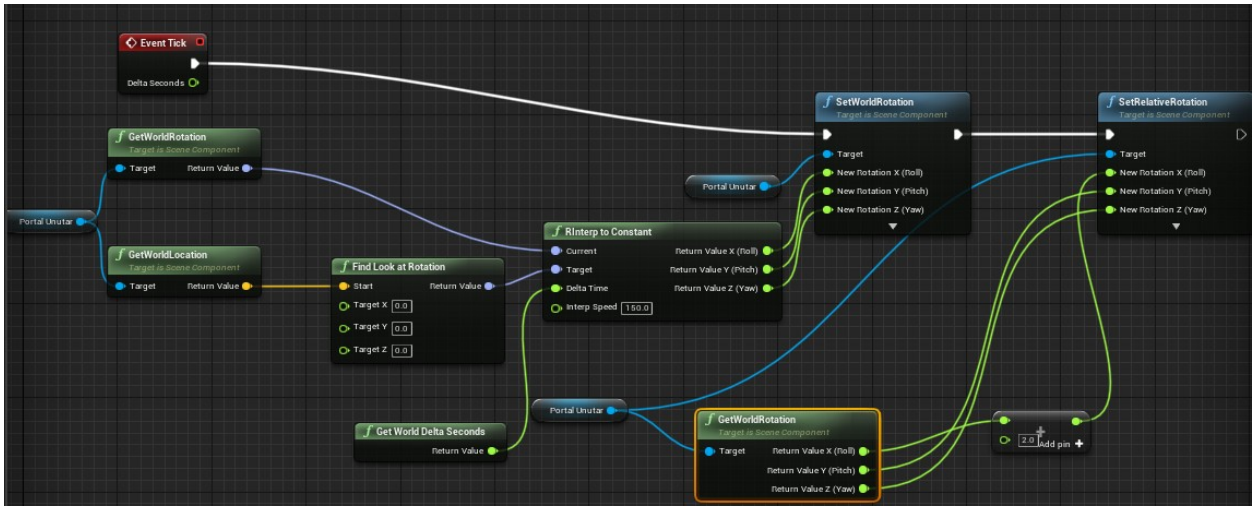
Slika 3.11. a) Viewport - stablo komponenti.



Slika 3.11. b) Viewport - prikaz objekta.

Slika 3.11. c) Viewport - opcije koje sadrži komponenta.

Svi tipovi nacrta imaju graf događaja (engl. *Event Graph*) koji se prikazuje u posebnom prozoru kroz kojeg se dodaju i implementiraju događaji, kao što je prikazano na Slici 3.12.. Događaj je funkcionalnost objekta da reagira na određenu situaciju, a neki od najčešćih događaja su: početak igre, *Tick*, što će se dogoditi ako dođe do kolizije, itd. Bolje rečeno, graf događaja je ono što čini objekt/sučelje interaktivnim.



Slika 3.12: Programiranje korištenjem nacrta.

Kartica skripti koju svi tipovi nacrta imaju koja se koristi implementiranje prilagođene (engl. *custom*) skripte koju će se koristiti kao knjižnicu funkcija ili unutar samog nacrta.

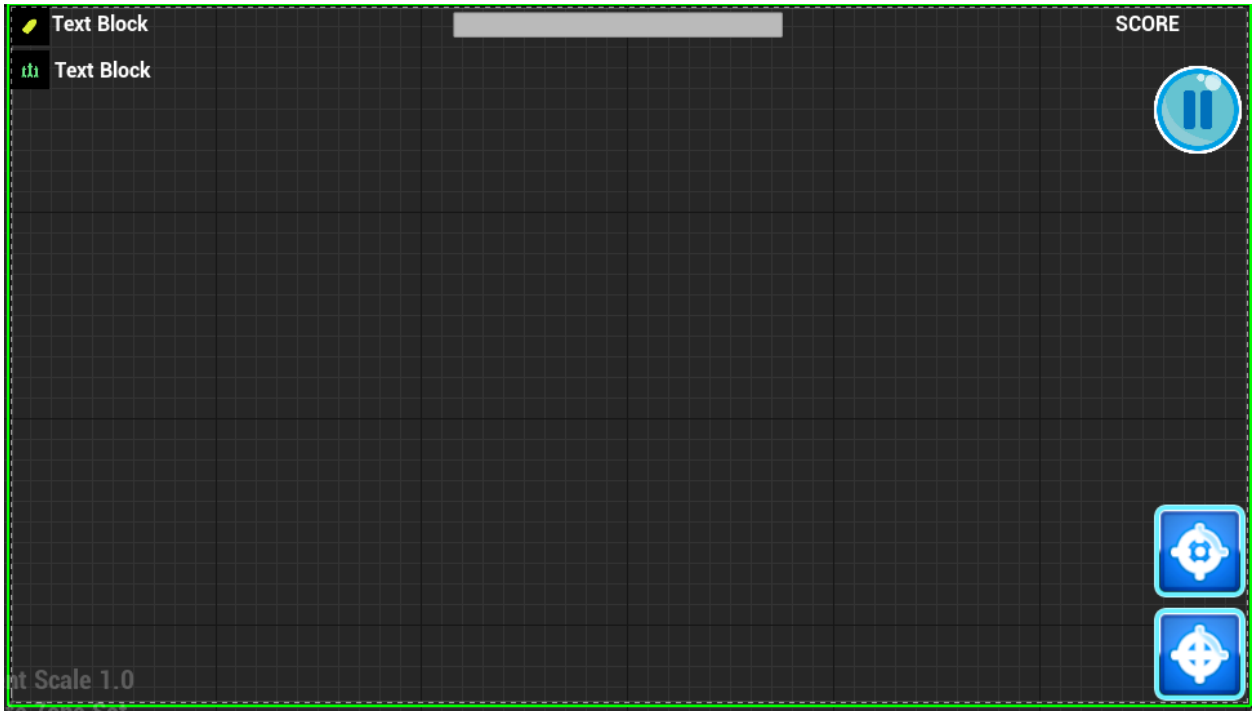
Nacrti su veoma korisni primarno zato što nude puno različitih mogućnosti koje bi bile zahtjevne za implementirati kroz programski kod, ubrzavaju razvoj i smanjuju vjerojatnost pogreške.

3.3. Usporedba klasičnog programiranja i nacrta

Najveća prednost nacrta je što tehnički umjetnik ne mora znati programirati i razumjeti funkcioniranje koda. Iz njegove perspektive, dovoljno je poznavanje logika iza programskog koda odnosno koji je ulaz u algoritam i koji je izlaz iz algoritma. Velika prednost nacrta je što ne zahtijevaju prevođenje koda i stvaranje izvršne datoteke nakon svake promjene, što znači da se može napraviti cijeli objekt unutar nacrta i dodijeliti mu komponente i funkcionalnosti. Objekt iz nacrta je zatim moguće instancirati gdje god je potrebno unutar videoigre. Ovo omogućava da netko tko nema nikakvog iskustava u programiranju i dalje može samostalno razvijati videoigru do određene granice.

Nacrti omogućavaju veoma efikasno građenje interaktivnog grafičkog korisničkog sučelja koje će se koristiti unutar videoigre. Takvo sučelje se također može implementirati kroz sam

programski kod, ali bi iziskivalo viši truda i vremena. Naročito se jednostavna sučelja mogu napraviti značajno brže u usporedbi sa implementacijom kroz programski kod te za to može biti dovoljno ponekad i nekoliko minuta. Na Slici 3.13. je prikazano grafičko sučelje unutar nacrta.



Slika 3.13. Prikaz igrivog sučelja unutar nacrta.

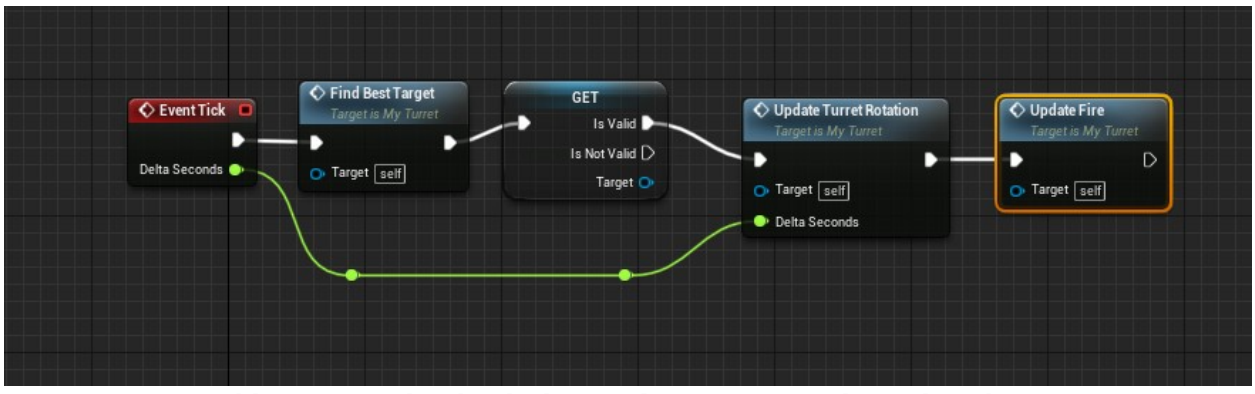
Glavna mana kod korištenja nacrta je što smo limitirani s funkcionalnostima koje nam stroj UE4 pruža. Primjerice, za posložiti polje brojeva po redu , uređivač UE4 pruža svoju funkciju, ali ponašanje te funkcije nije moguće dodatno modificirati u skladu s potrebama projekta budući da nam njen programski kod nije dostupan. Implementacija vlastitog algoritma za slaganje unutar nacrta može biti izazovno jer je ponovno potrebno oslanjati se na druge funkcije kojima nemamo pristup. Može se zaključiti kako razvoj videoigre koristeći isključivo nacrte predstavlja problem ukoliko je potrebno implementirati algoritme specifične za projekt, a koji nisu unaprijed dostupni u uređivaču. Sa druge strane, razvoj videoigre pisanjem programskog koda omogućuje izravan pristup svim funkcijama te se može lagano ponašanje svake funkcije promijeniti u skladu s potrebama projekta. Na Slici 3.12. je prikazano dodavanje nove funkcionalnosti objektu kroz nacrt.

Može se vidjeti kako radi svojih prednosti i mana nijedan pristup nije savršen za samostalno korištenje i nameće se pitanje koji od ta dva pristupa bi se trebao koristiti za razvoj videoigre.

Općenita preporuka je da pojedinci koji se tek upuštaju u razvoj videoigra započnu s nacrtima, a oni koji imaju iskustva u programiranju i razvoju videoigra da kombiniraju nacrte i pisanje programskog koda. Budući da se nacrti i programski kod ponašaju jednako odnosno oba pristupa imaju podršku za nasljeđivanje, objekte, komponente, varijable, funkcije, događaje, itd. Stoga, bilo koja C++ klasa se može pretvoriti u nacrt i obratno.

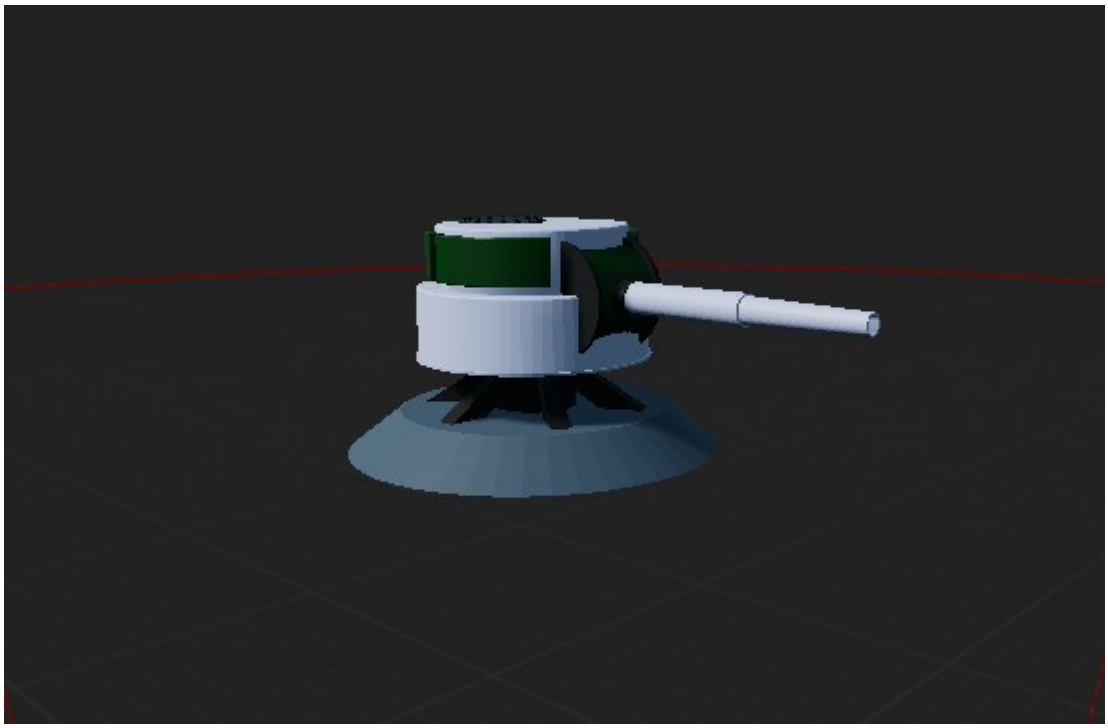
Preporučeni pristup je prvo definirati C++ klasu u kojoj će se definirati sve funkcionalnosti i varijable, a to podrazumijeva pisanje skripti i makroa. Potom se može napravi nacrt te klase koji će imati pristup svim funkcijama i varijablama koje su označene sa *UPROPERTY* i *UFUNCTION* kao što je prikazano na Slici 3.1. Jednom kada je nacrt klase izrađen, može se veoma brzo i efikasno uređivati fizički izgled objekta, implementirati događaje, a sve funkcije napisane u programskom kodu se mogu jednostavno pozivati.

Ovakav hibridni pristup se može pojasniti na sljedećem primjeru. Pretpostavimo da pokušavamo realizirati top koji se rotira prema igraču i pokušava ga pogoditi. Za implementirati ovakvo ponašanje, najprije je potrebno definirati klasu za top i potrebne funkcije poput funkcije za ispaljivanje metka, a potom će tehnički umjetnik pomoću nacrta sagraditi logiku iza topa. Ovo podrazumijeva prosljeđivanje ulaznih podataka skripti koju je programer napisao te dohvaćanje i obradu rezultata skripte u nacrtu. Na samom kraju dizajner dizajnira izgled kupole, metaka i animira ponašanje kupole. Na Slici 3.14. se može vidjeti programiranje uz pomoć nacrta i logiku iza kupole koja je napravljena u zadanoj videoigri.



Slika 3.14: Prikaz logike koja se koristi za upravljanje kupolom.

Sa Slike 3.14. vidi se da se svakog okvira najprije pronalazi najbolja meta, a za to se koristi generički algoritam za pronalazak najbolje mete kako bi se mogao što lakše prilagoditi za gađanje različitih tipova meta. Najbolja meta se bira na način da se prvo pronađu sve mete unutar određenog radijus oko topa te generira polje takvih meta. Zatim algoritam prolazi kroz polje meta i traži najbližu metu koju top fizički vidi, budući da nema smisla gađati metu koju top ne vidi i ne može pogoditi. Nakon što se odabere najbolja meta, top će se početi rotirati u smjeru mete uz prilagodbu visine cijevi kako bi ciljala prema meti. Nakon toga se poziva događaj *Fire* u kojem se hitac ispuca iz cijevi topa i pokreće se animacija u kojoj se cijev topa uvuče u kupolu i zatim vrati na početno mjesto s ciljem simulacije ponašanja stvarnih topova. Nakon što se cijev vrati u početni položaj, ispucava se još jedan hitac po istom principu. Također, meta odnosno igrač može biti zaklonjen iza prepreke te u tome slučaju top nema vizualni kontakt sa metom i neće pucati na nju. Top mora čekati određeno vrijeme nakon svaka dva ispucana hitca prije nego što opet može započeti pucanje. Na Slici 3.15. je prikazan izgled kupole.



Slika 3.15. 3D model topa.

4. KOLIZIJA

Kolizija je događaj kada se dva objekata međusobno dodiruju te predstavlja osnovu za interaktivnost unutar videoigre. Ta interaktivnost se postiže na način da se pokrene posebni događaj kada igrač dođe u koliziju s određenim predmetom unutar igre. U današnjem svijetu je skoro pa nemoguće za napraviti kompletnu videoigru bez kolizije. Kolizija se može podijeliti na jednostavnu i složenu koliziju. U ovom poglavlju će se prvo razmotriti jednostavna kolizija, zatim složena kolizija, a na kraju će se njihova implementacija u UE4.

4.1. Jednostavna kolizija

Jednostavna kolizija u videoigrama koristi različite koncepte geometrije i linearne algebre, a pod njom razmatramo linijske segmente, plohe i kutije (engl. *boxes*). Ove kolizije smatramo jednostavnima jer se za bilo koja dva objekta tih tipova može brzo i efikasno odrediti je li oni presjecaju.

4.1.1. Linijski segment

Linijski segment se definira pomoću dvije točke u prostoru, a između njih je potegnuta crta [16]. Za dobiti bilo koju točku na linijskom segmentu može se koristiti linearnu interpolaciju između početne i krajnje točke prema formuli (4.1.):

$$L(t) = Početak + (Kraj - Početak) * t \quad (4.1.)$$

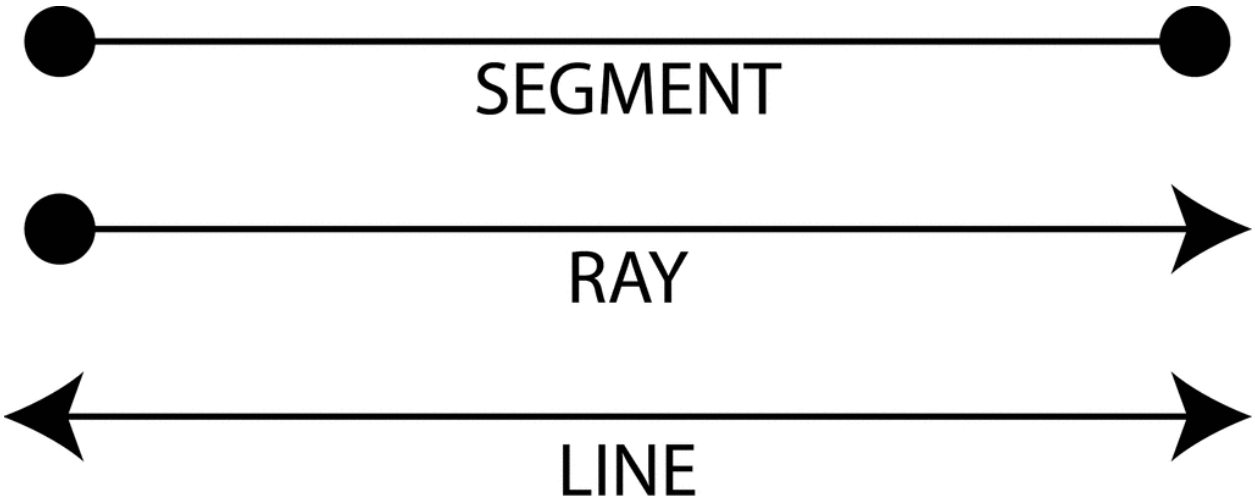
gdje t poprima vrijednost između 0 i 1. Linijski segment se također može poistovjetiti s beskonačnom linijom u pojedinim slučajevima, u takvom slučaju t poprima vrijednost svih realnih brojeva, odnosno: $-\infty \leq t \leq \infty$.

Linijski segment je prikladan odabir za simulaciju ispucanog metka zbog što se metak pomiče toliko brzo da za potrebe razvoja igre je dovoljno znati samo točku u kojoj će pogoditi neki drugi

objekt.. To znači da se jednostavno može konstruirati pravac i provjeriti koji objekt najranije pravac presijeca.
Minimalna udaljenost između linijskog segmenta i određene točke u prostoru dana je izrazom (4.2.) [4].

$$d = \|\vec{AC} - \vec{p}\| \quad (4.2.)$$

gdje je p vektor u smjeru linijskog segmenta, a AC linijski segment između točke A i točke C.



Slika 4.1. Prikaz linijskog segmenta, zrake i linije.

Slika 4.1. prikazuje liniju koja je beskonačna 2D crta, linijski segment koji je dio linije koji ima ograničenu duljinu dok pravac ima početnu točku, a prostire se u beskonačnost.

4.1.2. Ploha

Ploha je ravna 2D površina koja se prostire u beskonačnost. Plohe se najčešće koriste kao objekti koji predstavljaju zidove ili pod. Ploha se matematički može opisati formulom (4.3.)

$$P \cdot \hat{n} + d = 0 \quad (4.3.)$$

gdje je:

P - proizvoljna točka na plohi,

n - normala na plohu,

d - minimalna udaljenost plohe od ishodišta (engl. *origin*).

Proizvoljna točka u prostoru leži na plohi ako zadovoljava formulu koja opisuje plohu.

Za pronaći minimalnu udaljenosti proizvoljne točke i plohe koristi se formula (4.4.)

$$Dist = C * \hat{n} - d \quad (4.4.)$$

gdje je:

C - proizvoljna točka,

n – normala na plohu,

d – minimalna udaljenost plohe od ishodišta.

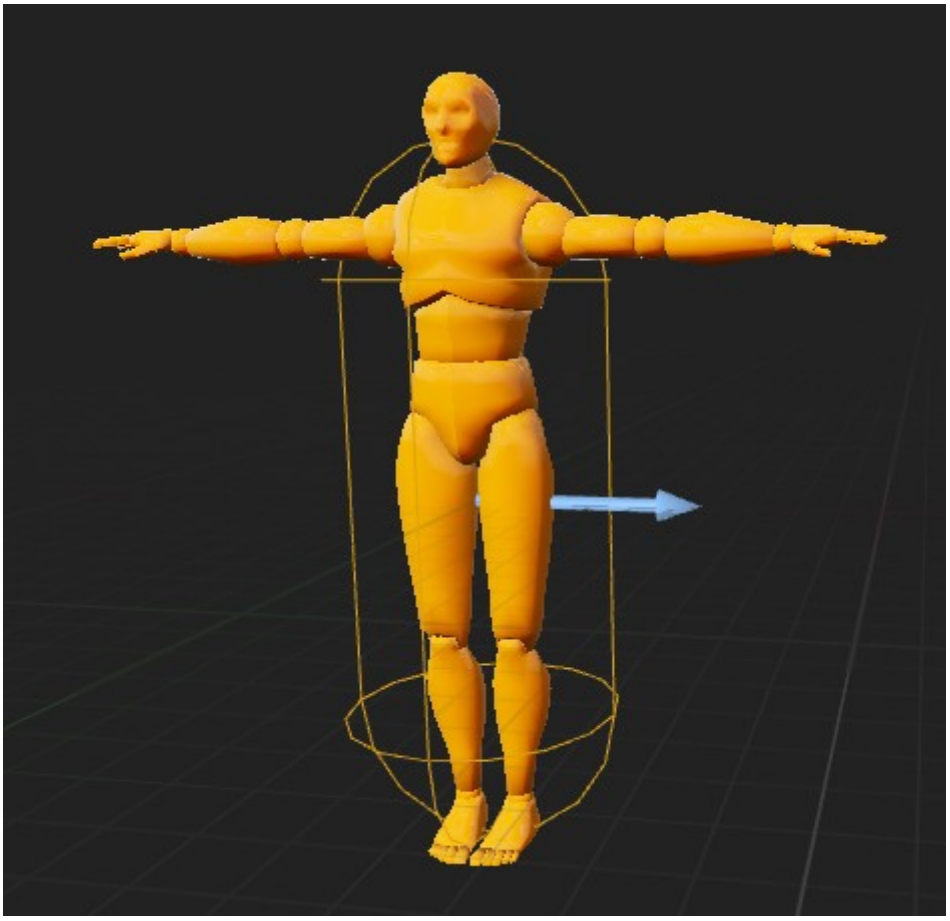
Negativna vrijednost *Dist* znači da se točka nalazi ispod plohe, dok pozitivna vrijednost znači da se nalazi iznad plohe.

4.1.3. Kutija

Pod kutijom smatramo objekt koji se može iskoristiti za predstaviti drugi složeniji objekt. Kutije se još nazivaju ograničivači volumena, a najčešće korišteni ograničivači su: kugla, kvadar i kapsula koju se može tretirati kao linijski segment s polumjerom. S njima se predstavljaju složeniji objekti u 2D ili 3D prostoru.

Prilikom odabira ograničivača polja, preporuka je uvijek odabrati onaj ograničivač polja koji najbolje odgovara objektu. Primjerice, objekte humanoidnog izgleda je najbolje predstaviti kapsulom, a ne kuglom ili kvadrom. Kugla nije dobar odabir jer će se objekt nalaziti u centru kugle te će mu po visini kugla dobro odgovarati, ali po širini i dubini će biti previše praznog prostora. Isti problem se javlja ako se pokuša humanoidne objekte predstaviti kvadrom jer se u tome slučaju gubi jako puno prostora između ruka i tijela jer u tome slučaju je širina kvadra jednaka širini rasponu ruka. Neadekvatni odabir ograničivača volumena rezultira s krivom kolizijom, odnosno dolazi do kolizije kada je u stvari nema.

Na Slici 4.2. se može vidjeti humanoidnog lika unutar kapsule kao ograničivač volumena.



Slika 4.2. Ograničivać volumena – kapsula.

4.1.4. Testovi kolizije

Koristeći prethodno opisanu geometriju i načine predstave različitih objekata u videoigri pomoću geometrije, može se proučiti testove za detekciju kolizije. Provjeravaju se sve moguće kombinacije između svih objekata na trenutnoj razini (svijetu), tako da je bitno uvijek održavati konstantni broj objekata na aktivnoj razini i da ih nema previše kako ne bi došlo do pada

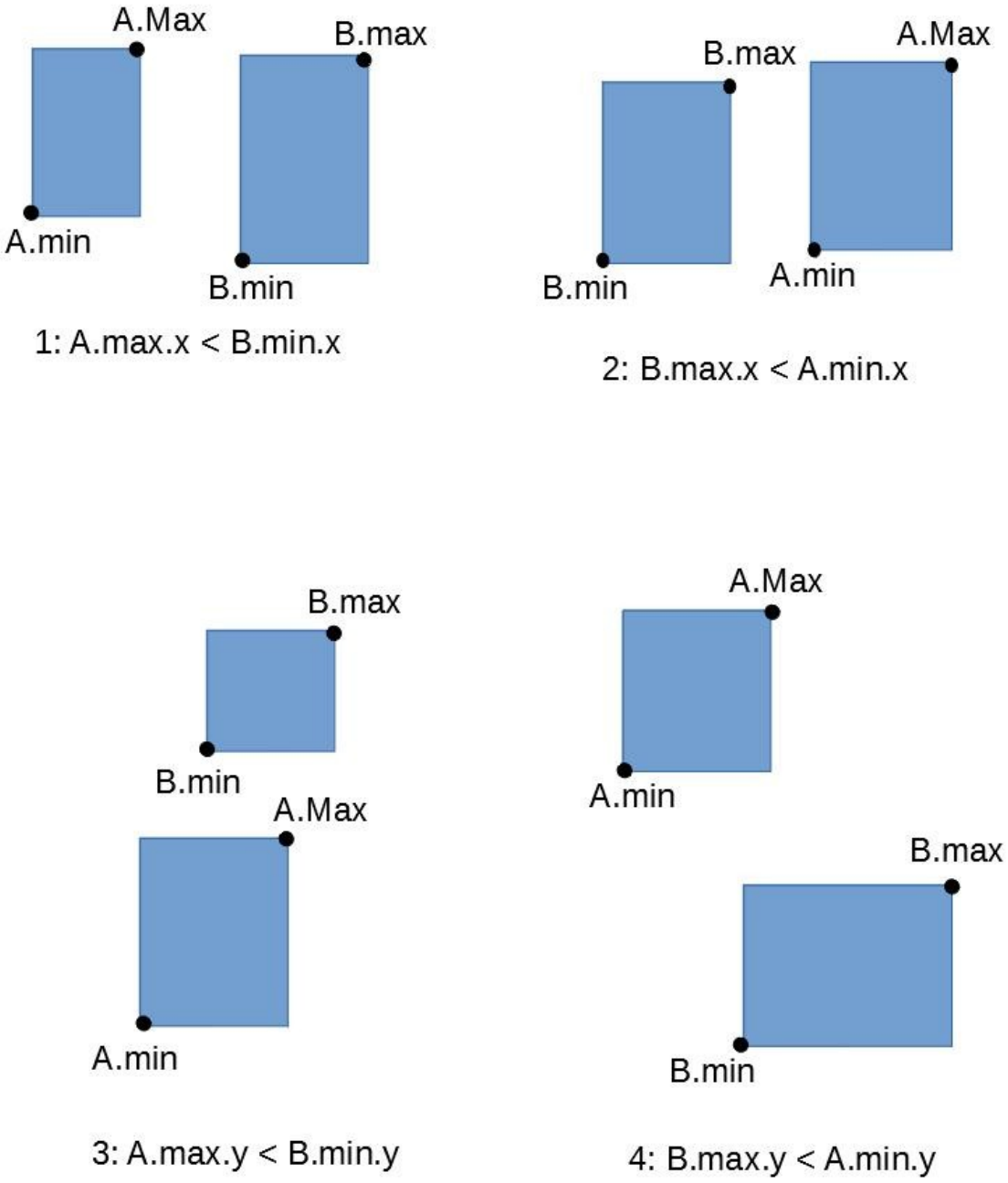
performansi. Sve moderne grafičke kartice danas mogu obrađivati oko 10 milijuna trokuta u realnom vremenu[17].

Kolizija dvaju kvadrova

Test za provjeru kolizije za dva objekta predstavljenih kvadrima je najlakši od svih predstavljenih testova u ovome radu. Za formirati kvadar u 3D svijetu potrebne su samo dvije točke i to donja lijeva točka (X_{min}, Y_{min}, Z_{min}) i dijagonalno nasuprotna gornja desna točka (X_{max}, Y_{max}, Z_{max}). Za provjeriti koliziju između ta dva objekta, potrebno je provjeriti sljedeće uvjete:

1. $A.X_{max} < B.X_{min}$
2. $B.X_{max} > A.X_{max}$
3. $A.Y_{max} < B.Y_{min}$
4. $B.Y_{max} > A.Y_{min}$

gdje oznaka A predstavlja prvi objekt, a oznaka B predstavlja drugi objekt. Kvadri se ne presijecaju ako niti jedan uvjet nije ispunjen [18]. Na Slici 4.3. su grafički prikazani uvjeti kada se dva kvadra ne presijecaju.



Slika 4.3. Uvjeti kada se dva kvadra ne presijecaju.

Kolizija dviju kapsula

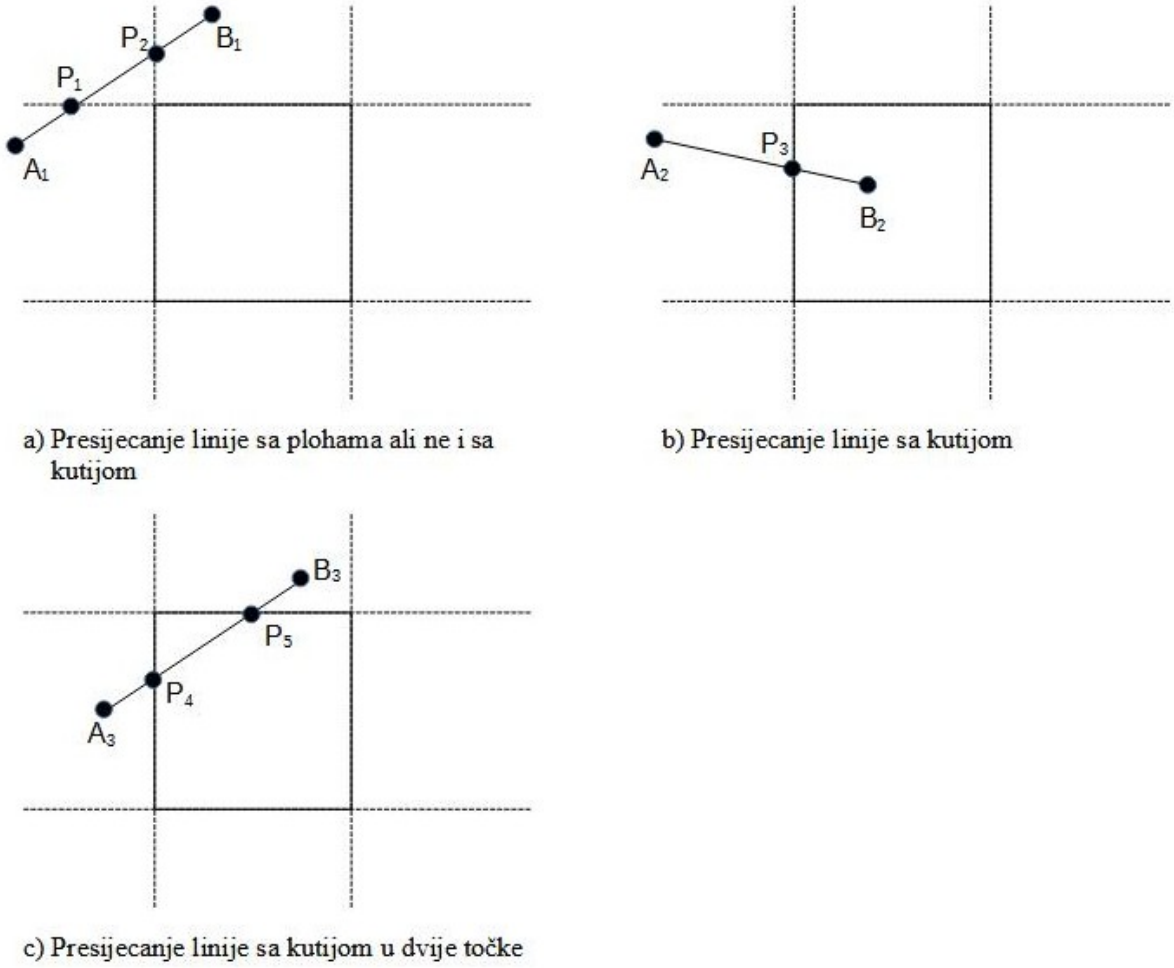
Ova vrsta kolizije je bitna jer većina igara novijeg datuma koristi kapsule kao ograničivače volumena za humanoidne likove. Za detektirati presjecanje između dvije kapsule nije računski zahtjevan proces budući da je u biti kapsula linijski segment s radijusom. Test započinje sa pronalaženjem minimalne kvadratne udaljenosti između dva linijska segmenta. Ako je kvadratna udaljenost manja ili jednaka sumi kvadriranih polumjera, onda se dvije kapsule presijecaju [4].

U nastavku se nalazi implementacija ovog testa u programskom jeziku C++. Funkcija vraća *true* ukoliko se dvije kapsule presijecaju odnosno *false* ako se ne presijecaju.

```
bool Intersect (const Capsule& a, const Capsule& b)
{
   float distSq = LineSegment::MinDistSq(a.mSegment, b.mSegment);
   float sumRadii = a.mRadius + b.mRadius;
   return distSq <= (sumRadii * sumRadii);
}
```

Kolizija linijskog segmenta i kutije

Najlakši pristup je da kutiju razložimo na više ploha. Primjerice, u 2D svijetu za svaku od četiri stranice kutije se konstruira jedna ploha kao što je pokazano na Slici 4.4. Ako se linijski segment siječe s plohom, ne znači nužno da se sječe i s kutijom zbog toga što su plohe beskonačne. Sa Slike 4.4. b), može se vidjeti da linijski segment siječe kvadar u točki P3 na lijevoj plohi jer kocka sadrži tu točku. Linijski segment ponekada može imati više točaka presjeka kao što je prikazano na Slici 4.4. c). Gdje su obje točke, P4 i P5, točke presjeka. U ovakvom slučaju, funkcija za koliziju bi trebala vratiti onu točku presjeka koja je bliža početnoj točci, odnosno točki iz koje kreće linijski segment.



Slika 4.4. Primjeri detekcije presijecanja linije i kutije u 2D svijetu.

Formula (4.5.) za izračunati presjek linije i plohe je:

$$t = \frac{-početak * n - d}{(kraj - početak) * n} \qquad (4.5.)$$

gdje je:

n – normala na plohu,

d – minimalna udaljenost plohe od postanaka.

Budući da je svaka ploha paralelna s koordinatnim osima u 2D odnosno s koordinatnim ravninama u 3D svijetu, ovu formulu se može optimizirati zbog toga što normala svake ravnine će imati dvije komponente jednake nuli i vrijednost jedan za treću komponentu. Kao posljedica toga, dva od tri člana skalarnog umnoška će uvijek biti nula.

Na primjer, normala lijeve bočne plohe pokazuje u potpunosti lijevo ili desno, a budući da orijentacija nije bitna za ovaj test, *d* se može izračunati na sljedeći način (4.6.):

$$d = -P * \hat{\hat{n}} = -min \cdot \langle 1,0 \rangle = -min_x \quad (4.6.)$$

gdje je:

P – točka presjeka,

n – normala na plohu.

Kombinirajući izraze (4.5.) i (4.6.) dobiva se pojednostavljena formula (4.7.) za provjeru presjeka između linijskog segmenta i plohe [4].

$$t = \frac{-početak \cdot \langle 1,0 \rangle - d}{(kraj - početak) \cdot \langle 1,0 \rangle} = \frac{-početak_x - (-min_x)}{kraj_x - početak_x} = \frac{-početak_x + min_x}{kraj_x - početak_x} \qquad (4.7.)$$

4.2. Složena kolizija

Pod složenu koliziju smatramo situaciju u kojoj se dva složena objekta dodiruju, a to su objekti s velikim brojem stranica i objekti nepravilnog oblika. Sve to čini provjeru kolizije veoma složenom, a postoji nekoliko rješenja za računanje složene kolizije.

Prvo i najjednostavnije rješenje je koristeći ograničivače volumena koji su bili opisani u prošlom potpoglavlju. Predstavljanjem složenog objekta s jednim jednostavnim objektom se uklanja problem složene kolizije. Međutim, programeri moraju biti svjesni da ova metoda uvodi pogrešku u izračunu kolizije zbog praznog prostora između ograničivača volumena i stvarnog objekta u videoigri. Budući da u većini slučajeva takva pogreška ne predstavlja veliki problem, je to najčešća metoda koja se koristi za računanje složene kolizije [19].

Primjerice, prema ovoj metodi metak se može predstaviti linijskim segmentom, a čovjek se može predstaviti kapsulom.

Postoje situacije u kojima se ovaj pristup ne može iskoristiti za izračun kolizije i tada je potrebno problemu pristupiti na drugačiji način. Radi ilustracije problema, pretpostavimo da je zadan Grčki hram prikazan na Slici 4.5. Potrebno je implementirati koliziju za zadani hram i omogućiti kretanje igrača između stupova i po unutrašnjosti hrama.

Slika 4.5. 3D model grčkog hrama.

S prvim rješenjem to nije moguće postići jer to bi značilo da će cijeli hram biti predstavljen jednim ograničivačem volumena koji neće dopuštati igraču kretanje između stupova i kroz unutrašnjost. Rješenje tog problema je da se cijeli objekt hrama predstavi složenim oblikom koji se sačinjava od velikog broja malenih trokuta, a ovisno o preciznosti koju se želi postići za detekciju kolizije, ti trokuti mogu biti veći ili manji. Za postizanje veće preciznosti, trokuti trebaju biti manji, a za manju preciznost trebaju biti veći. S ovakvom metodom se postiže dobro prijanjanje između stvarnog objekta i ograničivača volumena, a prostor prazni prostor između se svodi na malenu mjeru.

Na ovaj način šupljine hrama postaju prohodne, ali pod cijenu smanjenja performansi videoigre jer tijekom svakog okvira videoigre je potrebno provesti test kolizije između svih trokuta i drugih objekata u videoigri. Preporuka je da bi se takvo računanje kolizije trebalo samo

primjenjivati onda kada je neophodno [15]. U današnjim videoigrama se sve češće javlja potreba za ovakvom vrstom detekcije kolizije kako bi se svijet učinio što interaktivnijim.

4.3. Provjera kolizije između dinamičkih objekata

Svi navedeni testovi kolizije se zasnivaju na tome da se tijekom svakog okvira izvršavaju i rade provjere. Osim što ovakav pristup smanjuje performanse, javlja se i problem dinamike za čiju ilustraciju pretpostavimo situaciju u kojoj se igrač kreće prema tankom zidu. Očekivano ponašanje u takvoj situaciji bi bilo da se igrač prisilno zaustavi kada dotakne zid. Međutim, u videoigrama postoji mogućnost da igrač prođe kroz zid budući da se igrač pomiče u diskretnim koracima, a vjerojatnost prolaska kroz zid se povećava sa veličinom koraka. Na taj način, igrač će se tijekom jednog okvira nalaziti sa jedne strane zida, dok će se za vrijeme idućeg okvira nalaziti sa druge strane zida, a testovi kolizije neće detektirati koliziju.

Ovakav tip problema se može riješiti na nekoliko načina, a odabrani pristup ovisi o veličini objekata. Ako su objekti maleni poput metka, dovoljno je samo povući jedan linijski segment i provjeriti je li taj linijski segment presijeca neki drugi objekt. U slučaju velikih objekata, koristi se kontinuirana detekcija kolizije (engl. *CCD – continuous collision detection*)[4] što znači da se uvijek čuva poziciju objekta iz prethodnog okvira i zatim se provodi interpolacija između prethodne i trenutne pozicije. Ovo omogućava provjeru je li objekt došao u koliziju s neki drugim objektom prilikom prelaska iz jednog okvira u drugi.

Razmotrimo situaciju u kojoj se dvije kugle pomiču u proizvoljnim smjerovima te se kontinuirano pamte njihove pozicije iz prethodnog i sadašnjeg okvira. Parovi točaka iz prethodnog i sadašnjeg okvira se mogu predstaviti linijskim segmentom, gdje pozicija iz prethodnog okvira odgovara situaciji kada je t = 0, a sadašnjem okviru kada je t = 1. Time dobivamo jednadžbe (4.8.) i (4.9.):

$$P(t)=P_0+(P_1-P_0)*t \quad (4.8.)$$

$$Q(t)=Q_0+(Q_1-Q_0)*t \quad (4.9.)$$

gdje je:

P_0 – pozicija kugle u prethodnom okviru,

P_1 – pozicija kugle u trenutnom okviru,

t – parametar interpolacije.

U slučaju dodira, udaljenost središta dvije kugle će biti jednak zbroju njihovih polumjera. Stoga, rješavanjem ove dvije jednadžbe tako da je udaljenost između dvije kugle jednaka zbroju njihovih polumjera može se dobiti trenutak u kojem su se one dotaknule, a to je izraženo sljedećom formulom:

$$\|P(t)-Q(t)\|=r_p+r_q \quad (4.10.)$$

gdje je :

r_p – radijus kugle P,

r_q – radijus kugle Q.

Rješavanjem jednadžbe za *t*, dobiva se kvadratna jednadžba te ukoliko je diskriminanta jednaka nuli ili pozitivna, jednadžba ima realna rješenja Ukoliko postoje realna rješenja te se *t* nalazi između nula i jedan, može se zaključiti da se kugle sijeku, a u suprotnom se ne sijeku. Opisani postupak se naziva raskrižje sa swept-sferom (engl. *swept-sphere intersection*)[19].

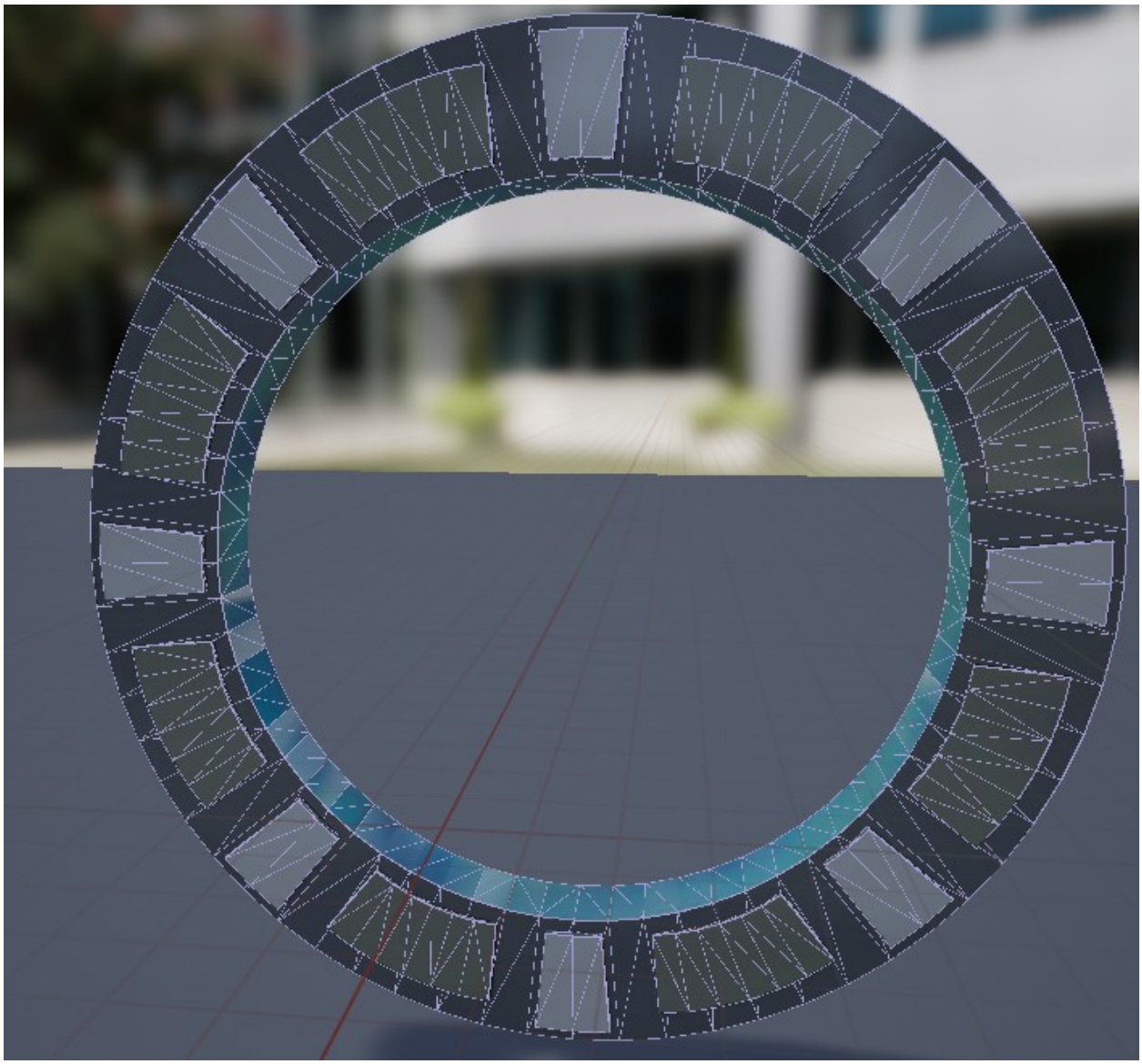
4.4. Implementacija kolizije u UE4

U prethodnim poglavljima se teoretski razmatralo koje vrste kolizija postoje, s kojim problemima se susrećemo te na koje načine se oni rješavaju. U ovome poglavlju će se razmotriti na koje načine se kolizija može implementira kroz UE4, a činjenica da postoji veći broj testova i situacija prilikom provjera kolizija, radi predstavlja jedan od najtežih zadataka u razvoju videoigre. UE4 donekle olakšava taj zadatak budući da sadrži gotovu implementaciju testova za koliziju, a na programeru preostaje briga o preciznosti kolizije i performansima videoigre.

4.4.1. Dodavanje kolizije kroz programski kod

Za dodati koliziju na određeni objekt unutar videoigre, najprije je potrebno dodati ograničivač volumena u datoteku zaglavlja, a potom se dodaje funkcija koja će se pozvati kada dođe do kolizije. Ta funkcija treba sadržavati instrukcije koje definiraju kako videoigra reagira na događaj kolizije. Budući da su ograničivači volumena implementirati kao komponente, potrebno ih je inicijalizirati u konstruktoru klase u kojoj se koriste, a pritom je potrebno definirati i funkciju koja će se pozvati u slučaju kolizije.

Radi primjera, pretpostavimo da je potrebno realizirati portal kao na Slici 4.6 za kojeg je potrebno moći detektirati kada objekt prođe kroz šupljinu u sredini. U ovakvom slučaju je potrebno koristiti jednostavnu i složenu koliziju. Jednostavnu koliziju bi bilo potrebno postaviti na šupljinu u sredini kako bi se detektirao prolazak objekta kroz portal. Složenu koliziju bi bilo potrebno implementirati za rub koji portala kako bi se mogao detektirati susret. U ovome slučaju je bila potrebna složena kolizija u kojoj je objekt aproksimiran trokutima. Na Slici 4.7 je prikazan kod koji se izvrši kada objekt prođe kroz šupljinu u portalu, a na Slici 4.8. je prikazan nacrt koji se izvrši kada objekt dotakne rub portala.



Slika 4.6. Primjer složene kolizije postavljene na rub portala.

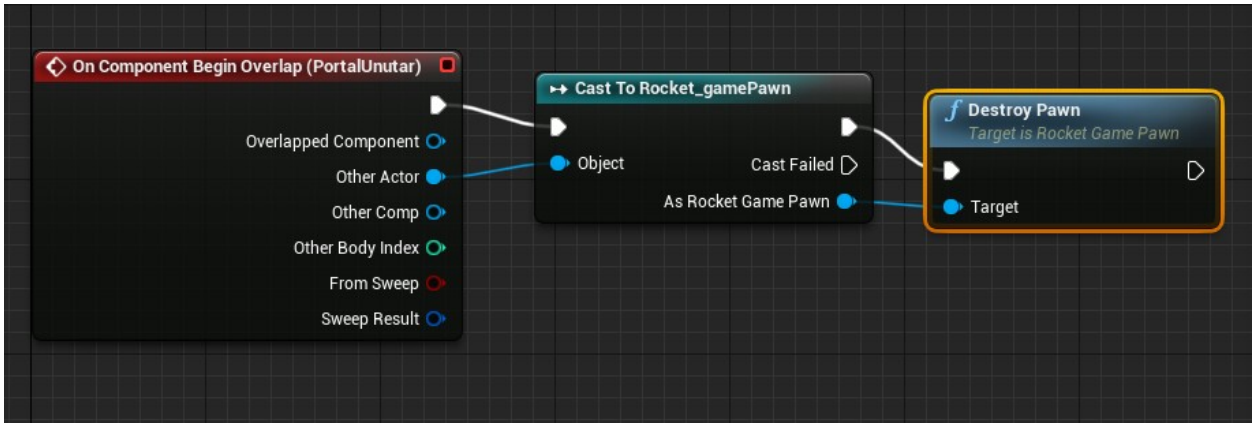
```
if (Cast<ARocket_gamePawn>(OtherActor) != nullptr)
{
    int score = Cast<ARocket_gamePawn>(OtherActor)->GetScoreInt();

    score = score + 5;

    Cast<ARocket_gamePawn>(OtherActor)->SetScore(score);
    GetWorld()->GetTimerManager().SetTimer(DestroyHandle, this, &APortal::DestroyPortal, 2.f, true);
}

if (Cast<APrateci_zid>(OtherActor) != nullptr)
{
    Destroy();
}
```

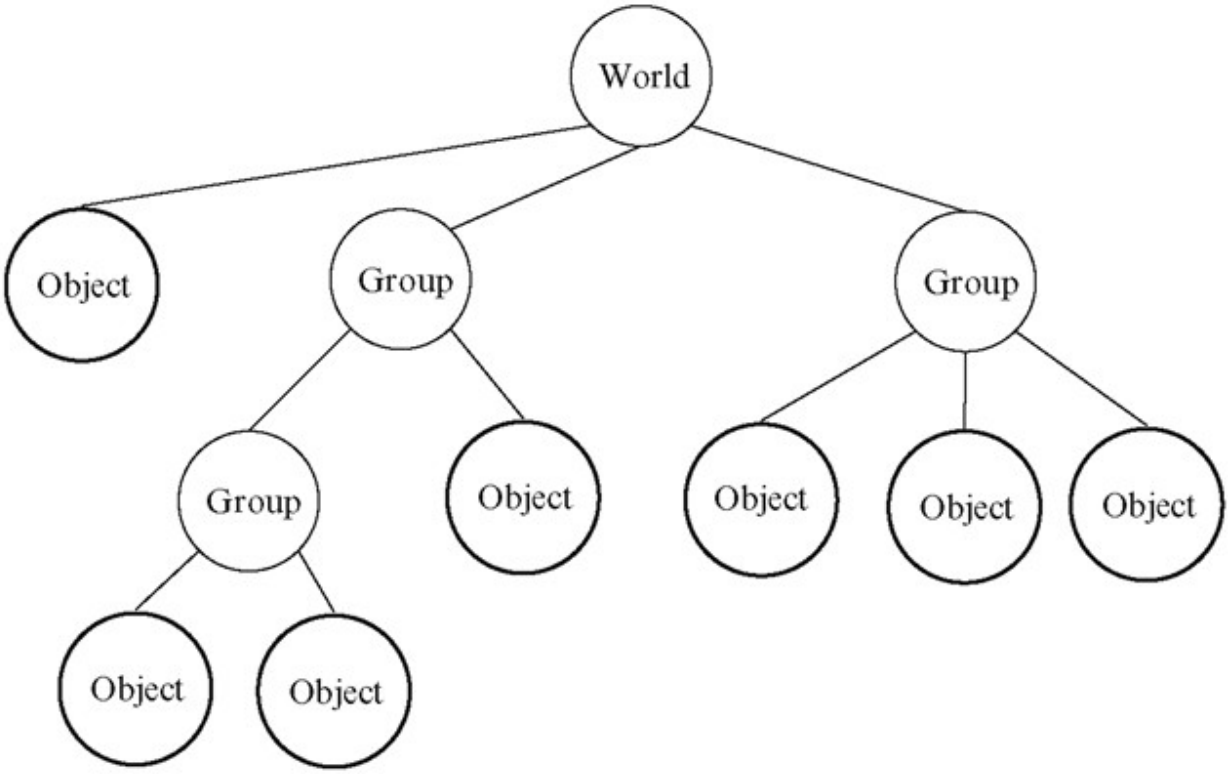
Slika 4.7. Programski kod za rukovanje kolizijom u šupljini portala sa Slike 4.6. Prikazani kod omogućava prolazak za objekte tipa ARocket_gamePawn dok svi ostali objekti uništavaju portal.



Slika 4.8. Rukovanje kolizijom kroz nacrt i UE4 uređivač za detekciju sudara objekta sa rubom portala sa Slike 4.6.

5. GRAF SCENE

Graf scene je opća struktura podataka koju uglavnom koriste aplikacije za uređivanje vektorskih grafika i moderne računalne videoigre, a uređuje logičnu i često prostornu reprezentaciju grafičke scene. Graf scene je u svojoj biti zbirka čvorova u strukturi stabla. Čvor stabla može imati više djece, ali samo jednog roditelja, a učinak roditelja se primjenjuje na sve njegove čvorove djece. Bolje rečeno, operacija/akcija izvedena nad grupom automatski širi svoj učinak na sve njezine članove. Na Slici 5.1. prikazan je grafički prikaz grafa scene.



Slika 5.1. Graf scene.

U mnogim programima, pridruživanje matrice geometrijske transformacije na razini svake grupe i ulančavanje takvih matrica zajedno je učinkovit i prirodan način za obradu takvih operacija. Prednost toga je što se više jednostavnih objekata mogu grupirati u jedan složeni objekt kojim se zatim može manipulirati jednako lako kao sa jednim objektom.

U uređivanju vektorske grafike, svaki lisni čvor u grafu scene predstavlja neku osnovnu jedinicu dokumenata, obično oblik poput elipse ili Bézierove staze. Iako se sami oblici mogu dalje rastaviti na čvorove kao što su čvorovi *spline-a*, praktično je zamisliti graf scene kao sastavljen od oblika, bez razmatranja niže razine reprezentacije.

Jedan od korisnijih koncepta čvora je sloj koji se ponaša kao čvor koji ne unosi vizualne promjene poput praznog lista papira, a na kojeg se može postaviti bilo koji broj oblika i grupe oblika. Dokument tada postaje skup slojeva, od kojih se svaki sloj može jednostavno učiniti nevidljivim, zatamnjenim ili zaključanim, odnosno da nije moguće unositi promjene. Neke aplikacije postavljaju sve slojeve u linearni popis, dok druge podržavaju slojeve unutar slojeva do bilo koje željene dubine.

Interno, možda uopće ne postoji stvarna strukturna razlika između slojeva i grupa, budući da su oboje samo čvorovi graf scene. Ako su razlike potrebne, uobičajena deklaracija tipa u programskom jeziku C++ bila bi napraviti generičku klasu čvorova, a zatim izvesti slojeve i grupe kao potklase. Član vidljivosti bio bi značajka sloja, ali ne nužno i grupe.

5.1. Graf scene u videoigrama i 3D aplikacijama

Grafovi scene korisni su za moderne videoigre koje koriste 3D grafiku i sve veće svjetove ili razine. U takvim aplikacijama čvorovi u grafu scene predstavljaju entitete ili objekte u sceni. Na primjer, igra može definirati logičan odnos između viteza i konja tako da se vitez smatra produžetkom konja. Graf scene imao bi čvor „konj” s čvorom „vitez” koji je na njega priključen. Graf scene također može opisati prostorni, kao i logički odnos različitih entiteta: vitez se kreće kroz 3D prostor kao što se konj kreće.

U ovako velikim aplikacijama, zahtjevi za memoriju glavna su razmatranja pri dizajniranju grafa scene. Kako bi se smanjili zahtjevi za memoriju, odmah na početku se instancira objekt sa svim teksturama, a zatim se njega samo kopira kroz svijet. Taj način rada znatno smanjuje zahtjeve za memoriju jer umjesto da se svaki put na novo učitavaju i spremaju podatci o vitezu poput

tekstura i materijala, koriste se postojeće podatke i sprema ih se samo jednom. U primjeru iznad, svaki je vitez zaseban čvor scene, ali je instanciran grafički prikaz viteza koji je sastavljen od 3D mreže, tekstura, materijala i shadera. To znači da se čuva samo jedna kopija podataka, na koju se zatim pozivaju svi čvorovi „vitez" u grafu scene. Ovo omogućuje smanjen proračun memorije i povećava brzinu, budući da kada se stvori novi viteški čvor, podaci o izgledu ne moraju se duplicirati.

5.2. Implementacija

Najjednostavniji oblik grafa scene koristi polje ili povezanu listu, a povezivanje njegovih oblika jednostavno je stvar linearnog ponavljanja čvorova jedan po jedan. Druge uobičajene operacije, poput provjere koji oblik siječe pokazivač miša, također se izvode putem linearnih pretraga. Ovakav pristup je obično dovoljan za malene grafove scene.

5.2.1. Operacije i otprema grafa scene

Primjena operacije na graf scene zahtijeva neki način slanja operacije na temelju tipa čvora. Na primjer, u operaciji iscrtavanja, čvor transformacijske grupe akumulirao bi svoju transformaciju množenjem matrice, vektorskim pomakom, kvaternionima ili Eulerovim kutovima. Nakon toga, lisni čvor šalje objekt na iscrtavanje. Neke implementacije mogu izravno iscrtati objekt, što poziva temeljni API za iscrtavanje, kao što je *DirectX* ili *OpenGL*. Budući da osnovnoj implementaciji API-ja za iscrtavanje uglavnom nedostaje prenosivost, mogli bi se grafikon scene i sustavi za iscrtavanje odvojiti. Kako bi se postigla ova vrsta otpremanja (engl. *dispatch*), može se primijeniti nekoliko različitih pristupa.

U objektno orijentiranim jezicima poput C++, to se lako može postići virtualnim funkcijama, gdje svaka funkcija predstavlja operaciju koja se može izvesti na čvoru. Virtualne funkcije je jednostavno napisati, ali uglavnom je nemoguće dodati nove operacije čvorovima bez pristupa izvornom kodu. Alternativno se može koristiti obrazac posjetitelja. (engl. *visitor pattern*) kod kojeg je na sličan način teško dodavati nove tipove čvorova.

Druge tehnike se zasnivaju na korištenju RTTI (*Run-Time Type Information*). U ovome pristupu se operacija može realizirati kao klasa koja se prosljeđuje trenutnom čvoru, a zatim postavlja upit o tipu čvora koristeći RTTI i traži ispravnu operaciju u nizu povratnih poziva ili funktora (engl. *functors*). To zahtijeva da se mapa tipova za povratne pozive ili funktore inicijalizira tijekom izvođenja, ali nudi veću fleksibilnost, brzinu i proširivost.

Postoje varijacije ovih tehnika, a nove metode mogu ponuditi dodatne prednosti. Jedna alternativa je ponovna izgradnja grafa scene, gdje se graf scene ponovno izgrađuje za svaku od izvedenih operacija. Međutim, to može biti vrlo sporo, ali rezultira visoko optimiziranim grafikonom scene. Može se vidjeti da dobra implementacija grafikona scene uvelike ovisi o pojedinostima aplikacije u kojoj se koristi.

5.2.2. Obilasci

Obilasci (engl. *traversals*) su metode za obilaženje stabla i zato su one ključ za primjenu transformacijskih operacija nad grafom scene [20]. Obilasci se općenito sastoje od započinjanja u nekom proizvoljnom čvoru koji je često korijen grafa scene, primjene operacija kod kojih se često operacije ažuriranja i iscrtavanja primjenjuju jedna za drugom i rekurzivnog pomicanja niz graf scene do čvorova djeteta, dok se ne dosegne čvor lista. U listu, mnogi strojevi za graf scene zatim se vraćaju stablom, primjenjujući sličnu operaciju. Na primjer, razmotrimo operaciju iscrtavanja koja uzima u obzir transformacije, dok se rekurzivno kreće niz hijerarhiju grafikona scene, poziva se operacija prediscrtavanja. Ako je čvor transformacije, on dodaje vlastitu transformaciju trenutnoj matrici transformacije. Nakon što operacija završi s obilaskom svih potomaka čvora, poziva operaciju čvora nakon iscrtavanja tako da transformacijski čvor može poništiti transformaciju postavljanjem izvorne transformacijske matrice koju je čvor dobio od roditelja. Ovaj pristup drastično smanjuje potrebnu količinu množenja matrica.

Neke operacije grafikona scene zapravo su učinkovitije kada se čvorovi obilaze različitim redoslijedom – to je mjesto gdje neki sustavi implementiraju ponovnu izgradnju grafikona scene kako bi promijenili redoslijed grafikona scene u format ili stablo koje je lakše raščlaniti.

Na primjer, u 2D slučajevima, grafovi scene obično se sami prikazuju počevši od korijenskog čvora stabla, a zatim rekurzivno crtaju podređene čvorove. Listovi stabla predstavljaju objekte koji su u prvom planu. Budući da se crtanje nastavlja odostraga prema naprijed gdje se bliži objekti jednostavno iscrtavaju preko udaljenih, proces je poznat kao slikarski algoritam (engl. *Painter's algorithm*). U 3D sustavima, koji često koriste dubinske međuspremnike, učinkovitije je prvo nacrtati najbliže objekte, budući da udaljenije objekte često je dovoljno samo testirati po dubini umjesto da se iscrtavaju, jer su zaklonjeni bližim objektima.

5.3. Graf scene i hijerarhija ograničivača volumena

Hijerarhije ograničivača volumena (engl. *Bounding Volume Hierarchies; BVHs*) korisne su za brojne zadatke – uključujući učinkovito uklanjanje i ubrzavanje otkrivanja sudara između objekata.

BVH je stablo ograničivača volumena.Veličina volumena na dnu hijerarhije je dovoljno velika da usko obuhvati jedan objekt, ili možda čak i neki manji dio objekta u BVH-ovima visoke rezolucije. Kako se uspinjemo po hijerarhiji, svaki čvor ima svoj volumen koji obuhvaća sve volumene ispod sebe. U korijenu stabla nalazi se volumen koji obuhvaća sve volumene u stablu, odnosno obuhvaća cijelu scenu.

Ako granični volumen objekta ne presijeca volumen viši u stablu, onda niti ne može presijecati niti jedan objekt ispod tog čvora. Na taj način se postiže ubrzanje u otkrivanju sudara između objekta. Na primjer, pretpostavimo *FPS* videoigru u kojoj svaki igrač ima svoj ograničivač volumena koji u grubom predstavlja lika, a zatim svaki dio tijela ima dodatno svoji ograničivač volumena kako bi se moglo točno odrediti gdje je metak pogodio igrača te koliko štete će on primiti. Ako metak ne presijeca najveći ograničivač volumena koji predstavlja kompletnog lika, onda ne presijeca niti jedan drugi ograničivač volumena i zato se može sa sigurnošću odmah nakon toga odbaciti sve ostale testove kolizije za te ograničivače volumena.

Radi sličnosti između BVH i grafa scene, moguće je prilagoditi graf scene da postane BVH ako svaki čvor ima pridruženi volumen ili postoji namjenski izgrađen „vezani čvor” dodan na

pogodnom mjestu u hijerarhiji. Iako ovo nije tipično za graf scene, postoje prednosti uključivanja BVH u njega.

5.4. Prikaz grafikona scene unutar UE4

Unutar stroja UE4 postoje više različitih grafova scene. Najčešće se grafovi scene dijele na globalni i relativni graf scene. Razlika između njih je u tome što globalni graf scene u sebi sadrži cijelu igrivu razinu i sve što se nalazi na toj razini, dok relativni graf scene sadrži jedan složeni objekt koji ima svoj vlastiti graf scene. Bolje rečeno, u igrivoj razini postoji samo jedan globalni graf scene i beskonačno mnogo relativnih grafova scene.

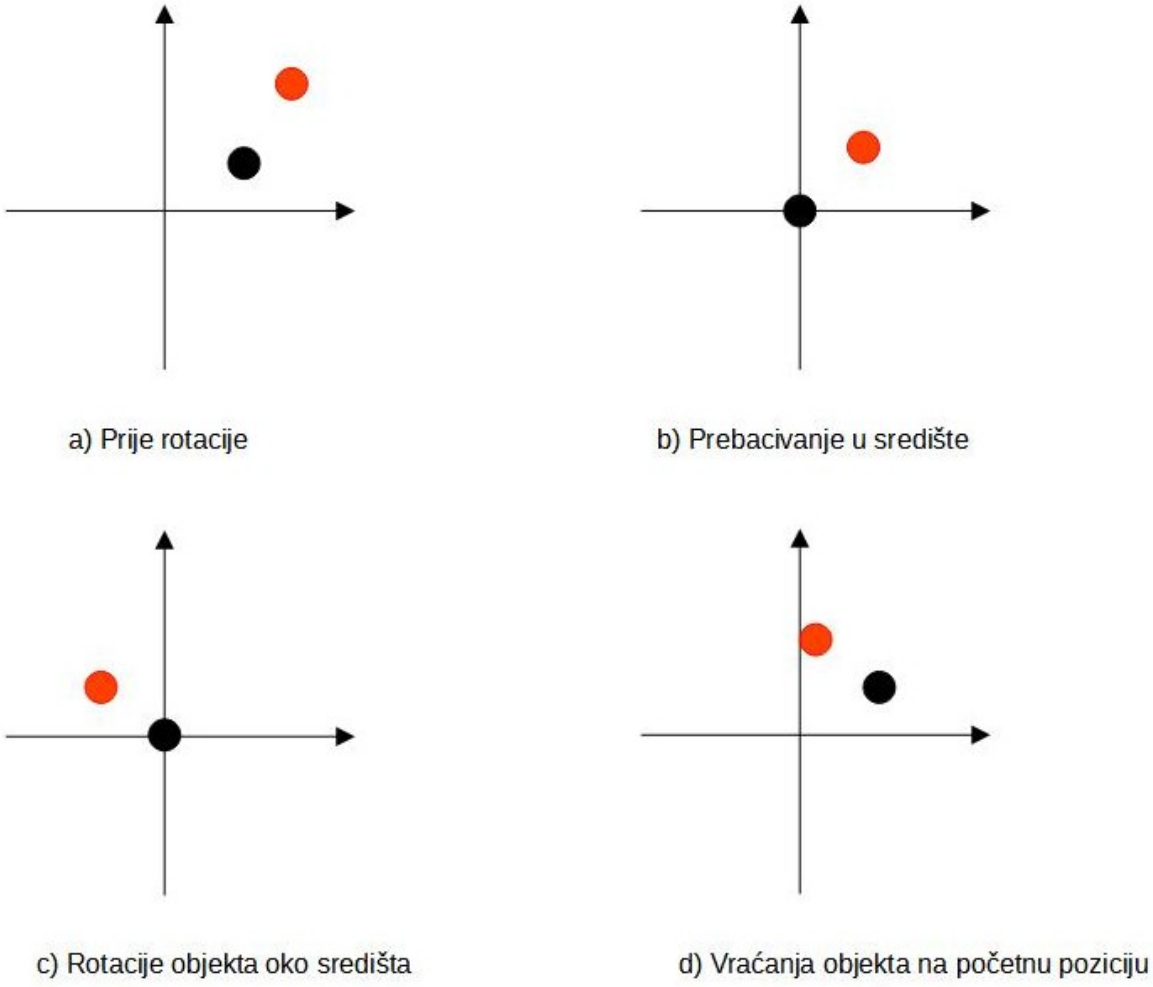
Dodavanjem komponenti na određeni objekt unutar videoigre, kao što je kroz rad bilo opisano, odmah se slaže složeni objekt koji ima svoj vlastiti graf scene, a to je prikazano na Slici 3.11. a). S iste slike se može vidjeti da korijen grafa scene je komponenta „PortalUnutar" koji ima tri djeteta.

Tijekom razvoja videoigre, važno je voditi brigu o tome je li se akcije/operacije izvršavaju unutar relativnog ili globalnog grafa scene. Na primjer, rezultat neće biti isti ako se komponenta „PortalTrokut" rotira unutar globalnog koordinatnog sustava ili se rotira unutar relativnog koordinatnog sustava. Ako se tu komponentu rotira u globalnu koordinatnom sustavu, onda će se ta komponenta rotirati oko središta tog koordinatnog sustava, a u drugom slučaju će se rotirati oko središta relativnog koordinatnog sustava.

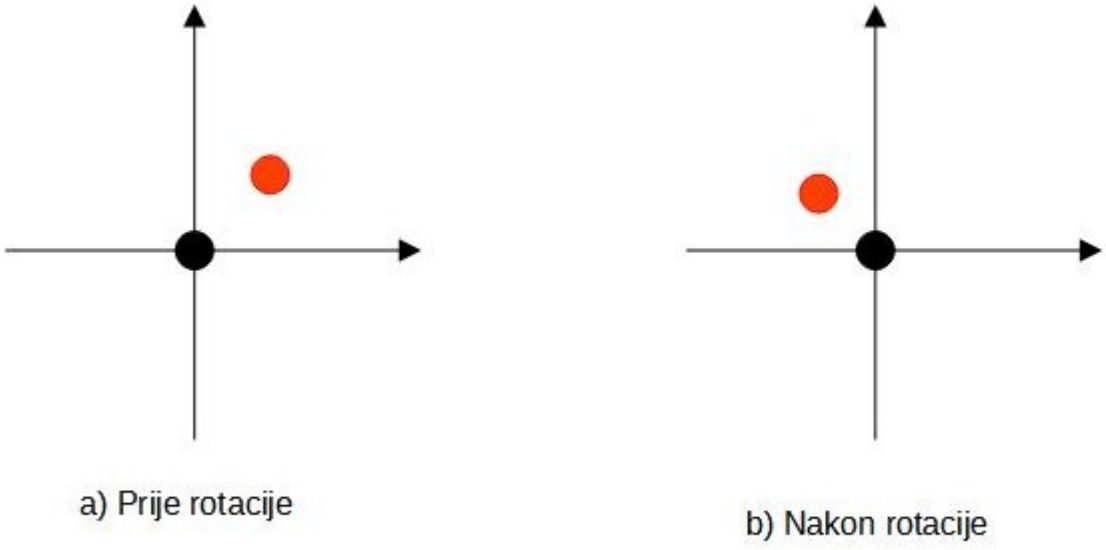
Recimo da želimo rotirati komponentu „PortalTrokut" oko komponente „PortalUnutar" na Slici 3.11 b) kako bi se dobio efekt da se portal rotira oko svoje osi. Ovakvo ponašanje se može postići rotiranjem unutar globalnog ili rotiranjem unutar relativnog koordinatnog sustava.

Za realizaciju te rotacije unutar globalnog sustava, najprije je potrebno translatirati cijeli portal u središte globalnog koordinatnog sustava. Zatim se komponentu „PortalTrokut" rotirati oko središta koordinatnog sustava te na kraju se portal mora translatirati natrag na svoju početnu poziciju. U konačnici je za ovaj pristup bilo potrebno iskoristiti tri akcije: translacija, rotacija i opet translacija, kao što je prikazano na Slici 5.2.

U slučaju da se takvo ponašanje htjelo realizirati u relativnom koordinatnom sustavu, portal ne bi bilo potrebno translatirati jer se već nalazi u središtu koordinatnog sustava. Stoga, dovoljno je samo rotirati komponentu „PortalTrokut" oko središta. U usporedbi sa prethodnim pristupom u kojem je bilo potrebno iskoristiti tri operacije, u ovome postupku je bilo potrebno iskoristiti samo jednu operaciju kao što je prikazano na Slici 5.3..

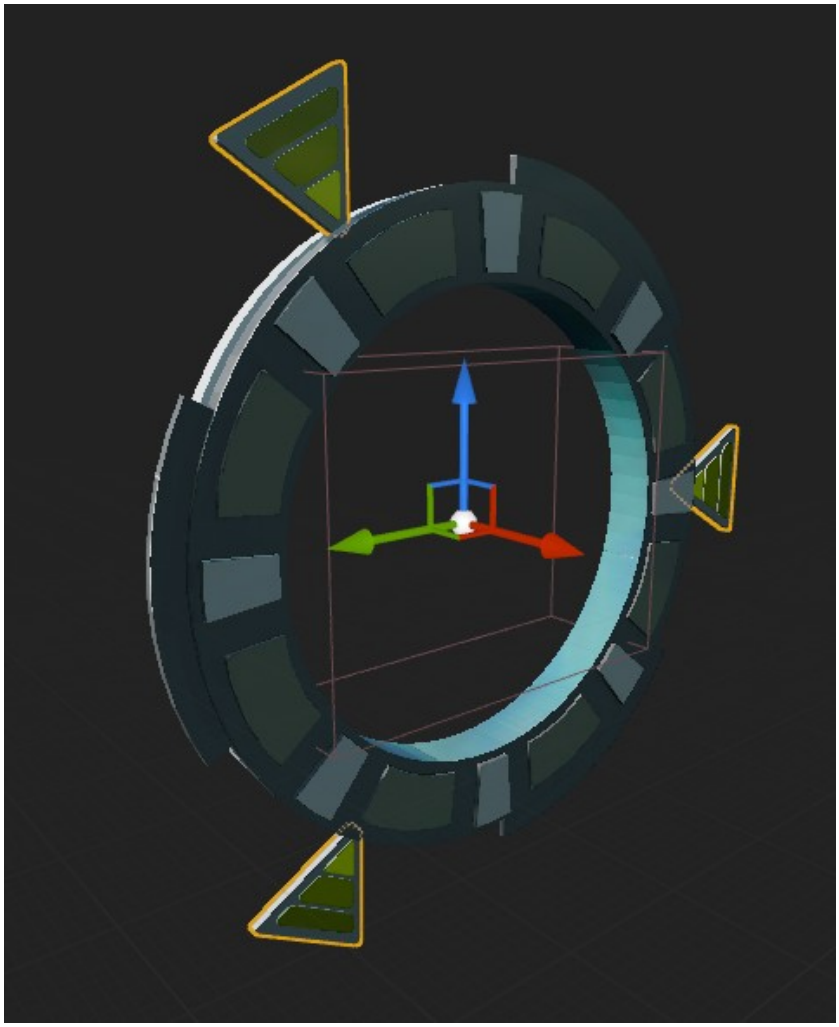


Slika 5.2. Postupak rotiranje crvene oko crne točke u globalno koordinatnom sustavu.



Slika 5.3. Postupak rotiranje crvene oko crne točke u lokalnom koordinatnom sustavu.

Isti efekt se mogao realizirati tako da se obavi rotacija komponente „PortalUnutar” oko središta koordinatnog sustava budući da je to korijen stabla. To znači da sve operacije/akcije koje se izvrše nad njom će se odmah preslikati i na druge komponente unutar tog objekta. Jedini razlog zašto je ovaj pristup nepovoljniji je taj što se rotiraju četiri komponente, dok se u prvobitnom rješenju rotira samo jedna komponenta, a oba pristupa proizvedu isti rezultat. Sa Slike 5.4., se također može vidjeti relativni koordinatni sustav za komponentu „PortalTrokut”.



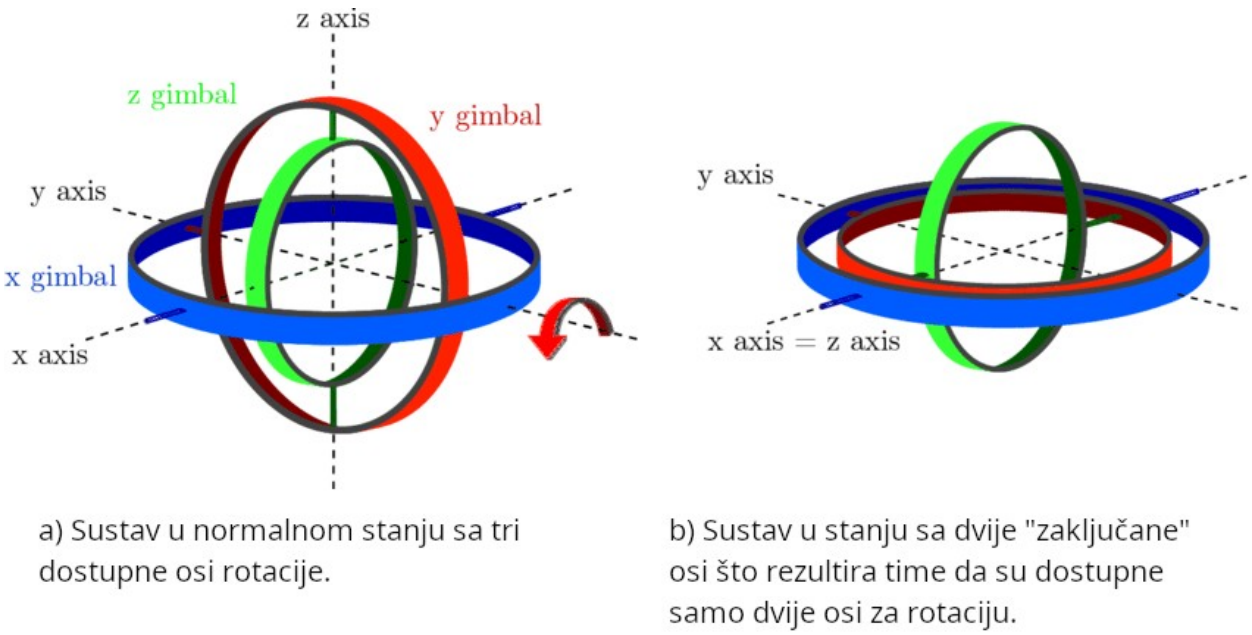
Slika 5.4. Relativni koordinatni sustav.

5.4.1. Rotacije unutar UE4

Postoje mnogi načini za prikazati rotacije, a dva najraširenija su pomoću kvaterniona ili pomoću Eulerovih kuteva. UE4 podržava prikaz rotacija i u jednom i u drugom obliku, ali interno se operacije i čuvanje rotacije uvijek vrši u kvaternionima. Razlog za ovo je taj što Eulerovi kutovi imaju gimbalov problem.

Gimbalov problem je gubitak jednog stupnja slobode u troosovinskom kardanskom sklopu koji se javlja kada se dviju od tri osi ubace u paralelnu konfiguraciju, „zaključavajući" sustav u rotaciju u degeneriranom dvodimenzionalnom prostoru. Ovaj problem se manifestira u videograma koje interno koriste Eulerove kutove kada dvije osi postanu paralelne te se radi toga

kompletno izgubi jednu os rotacije Od kada se dogodi takav slučaj, objekt se može rotirati samo oko dvije osi, a ne oko sve tri osi. Taj problem se najčešće javlja kod objekata koji zahtijevaju rotaciju oko sve tri osi, a taj problem nije vezan samo za domenu videoigra već općenito za sve primjene Eulerovih kuteva kao što je na primjer u zrakoplovima. Na Slici 5.5. a) se može vidjeti tri stupnja rotacije u normalnom slučaju, dok se na Slici 5.5. b) može vidjeti gimbalov problem gdje su dostupna samo dva stupnja rotacije umjesto tri [21].

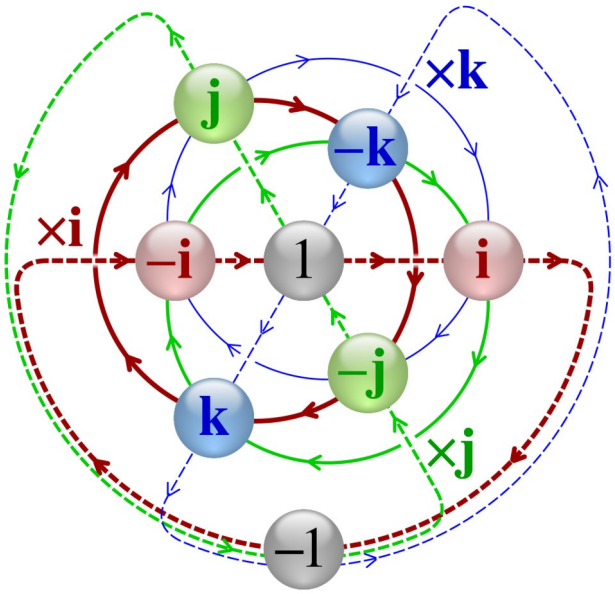


Slika 5.5. Gimbalov problem.

Kvaternioni su algebarsko proširenje kompleksnih brojeva [22], a razlika između kompleksnih brojeva i kvaterniona je što kvaternioni imaju tri imaginarne jedinice, koje se označavaju s i, j i k dok kompleksni brojevi imaju samo jednu imaginarnu jedinicu kao što se može vidjeti sa Slike 5.6. Na ovaj način se orijentacija tretira kao jedna vrijednost, a ne kao tri zasebne vrijednosti. Nakon izvođenja rotacija, kvaternion se mora normalizirati kako bi se spriječilo nakupljanje pogreške zbog pomičnog zareza prilikom uzastopnih transformacija. Budući da se orijentacija kod kvaterniona prikazuje kao vrijednost, ne može doći do gimbalovog problema te je to razlog zašto se kvaternioni danas koriste za zapis orijentacije u mnogim programima.

Iako UE4 interno obrađuje rotacije pomoću kvaterniona, programeru su na raspolaganju dostupna dva skupa funkcija za rad sa rotacijama: one koje rukuju sa Eulerovim kutovima i one koje rukuju sa kvaternionima. Ukoliko programer koristi funkcije koje su namijenjene za

Eulerove kutove, UE4 će najprije Eulerove kutove pretvoriti u kvaternione, a zatim izvršiti traženu operaciju.



Slika 5.6. Grafički prikaz množenja kvaterniona sa imaginarnim jedinicama: i (crvena putanja), j (zelena putanja) i k (plava putanja). Ova slika radi jednostavnosti uzima u obzir samo situacije u kojima su svi članovi kvaterniona osim jednog nula.

6. RAZVOJ VIDEO IGRE

6.1. Koncept videoigre bez granice

Prije početka razvoja videoigre, najprije je potrebno osmisliti ideju (koncept) za videoigru koja treba obuhvaćati jednostavnu i složenu koliziju, rotaciju igrača i ostalih objekata u sve tri osi kako bi se demonstrirao rad sa kvaternionima te vidoeigra mora biti razvijena u stroju UE4 koristeći objektno programiranje u kombinaciji sa nacrtima.

Videoigra koja će se razviti pripada vrsti videoigra bez granica (engl. *endless running game*) koje se zasnivaju na konceptu da videoigra nema kraja, a igraču je cilj sakupiti što više bodova. Budući da je razina neograničena, ona se mora dinamički generirati na način da s vremenom postaje sve izazovnije za igrača, a to ujedno čini videoigru zanimljivom. Odlučeno je razvijati videoigru za pametne telefone kako bi se pojednostavio broj ulaznih komandi, jer će se koristiti samo virtualna komandna palica prikazan na zaslonu uređaja (engl. *joystick*) za upravljanjem glavnim likom.

Videoigra pod nazivom *Space Runner,* koju sam razvio u sklopu diplomskog rada, se sastoji od beskonačnog dinamički generiranog tunela i svemirskog broda kojeg igrač mora navigirati kroz tunel te je cilj stići što dalje. Jedan od najbitnijih zahtjeva videoigara koje su klasificirane kao videoigre bez kraja je da sa vremenom postaju sve izazovnije. Stoga je odlučeno u videoigru uvesti nekoliko faktora koji je čine izazovnijom kroz vrijeme: ubrzavanje glavnog lika sa vremenom, pojavljivanje prepreka u tunelu, pojavljivanje kupola koje napadaju glavnog lika te generiranje različitih oblika tunela koji su zahtjevniji za navigaciju. Sa strane implementacije, ovo je značilo imati povezanu objektnu strukturu jer svaka vrsta tunela nasljeđuje od apstraktne klase. Također je bilo potrebno razviti algoritam po kojem će kupole pronaći metu i ponašat se u njezinoj okolini. Sve te funkcionalnosti u svojoj osnovi uključuju rad sa kolizijama i međusobnu komunikaciju između objekata različitih vrsta.. Igrač na raspolaganju ima dvije vrste naoružanja kojima može uništiti prepreke u tunelu te u igru su uvedeni sporedni ciljevi koje igrač može izvršiti kako bi bio nagrađen dodatnim bodovima.

Kako bi se demonstrirala detekcija različitih vrsta događaja u videoigri, uvedeni su bodovi koje igrač može sakupiti na tri različite načina:

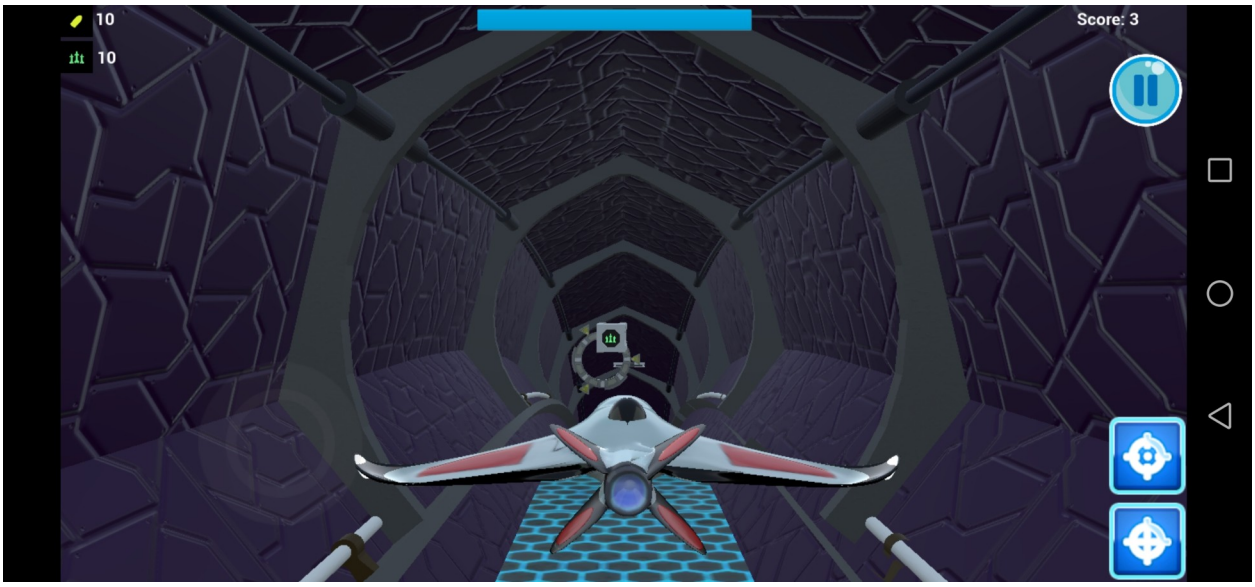
1. Igrač se nagrađuje bodovima proporcionalno vremenskom trajanju igre.
2. Igrač dobiva dodatne bodove za svaku uništetu prepreku ili kupolu.
3. Igrač može dobiti dodatne bodove prolaženjem kroz kružne portale koji se pojavljuju u tunelu.

Iz perspektive igrača, bodovi uvode dodatnu razinu igrivosti i motiviraju ga da osim samog preživljavanja u tunelu, nastoji uništiti što više prepreka i topovskih kupola te tako maksimizirati količinu bodova u što kraćem vremenu. Kako bi igrač mogao pratiti svoj uspjeh, odlučeno je napraviti ljestvicu deset najboljih ostvarenih rezultata koja će se držati u lokalnoj memoriji.

6.2. Space Runner

6.2.1. Ulazne komande

Budući da je ova igra bila zamišljena za pametne telefone, sve kontrole će se odvijati preko zaslona na dodir. Kada igrač dodirne zaslon, videoigra će prepoznati ulaz i izvršiti akcija ovisno o kontroli koju je igrač dodirnuo. Na samom zaslonu pametnog mobitela je prikazana virtualna komandna palica koja omogućava igraču da upravlja smjerom kretanja svemirskog broda te dva gumba koji služe za ispaljivanje običnih metaka odnosno raketa. Na Slici 6.1., je prikazan zaslon mobitela na kojem se mogu opisane kontrole. Ovakve ulazne kontrole se smatraju intuitivnima i prikladne su za sve uzraste.



Slika 6.1. Prikaz videoigre na pametnom telefonu.

6.2.2. Dinamičko generiranje razine

Dinamički generirani tunel se dobiva nasumičnim kombiniranjem pet različitih statičkih segmenata: ravni segment i segmenata koji skreću lijevo, gore, desno i dolje. Podešavanjem vjerojatnosti za pojavljivanje svakog segmenta i ovisno o kombinaciji različitih vrsta segmenata, može se generirati lakše odnosno zahtjevnije oblike tunela za igrača. Ovo se temelji na iduće dvije pretpostavke:

1. Za igrača najlakša situacija je kada se tunel sastoji od ravnih segmenata jer tada igrač ne treba pratiti krivinu tunela, već samo zaobilaziti prepreke.
2. Za igrača je manje zahtjevna situacija kada su dva segmenta iste vrste jedan za drugim kao što su dva skretanje lijevo, nego kada su različitih vrsta kao kada skretanje lijevo prethodi skretanju gore. U slučaju dva jednaka segmenta, igrač može relativno jednostavno nastavit pratiti krivinu tunela zbog toga što su oba segmenta jednako zakrivljeni te nije potrebno prilagoditi kretanje broda prilikom prelaska iz jednog segmenta u drugi. U slučaju kada su segmenti različite vrste, izazovnost proizlazi iz činjenice da igrač mora prilagoditi kretanje broda prilikom prelaska iz jednog u drugi segment, odnosno zbog promjene smjera tunela.

Kako bi generirani tunel postajao sve teži s vremenom, algoritam za generiranje tunela treba u početku preferirati kombiniranje ravnih segmenata. S protokom vremena treba počet preferirati kombiniranje zakrivljenih segmenata iste vrste te u konačnici segmente različitih vrsta kako bi se postigla maksimalna zahtjevnost. Može se primijetiti kako prilikom generiranja generiranja tunela, vjerojatnost za dodavanje idućeg segmenta ovisi isključivo o posljednjem postavljenom segmentu. Primjerice, na samom početku igre uvjetna vjerojatnost da će se pojaviti ravni segment nakon ravnog segmenta je veća nego za pojavljivanje ostalih tipova segmenata. Ovakva formulacija upućuje na to da se algoritam za generiranje tunela može formulirati u obliku Markovljevog lanca sa promjenjivom matricom prijelaza. Markovljev lanac je pogodan za ovaj slučaj jer se temelji na ideji da vjerojatnost prelaska u stanje x ovisi isključivo o prethodnom stanju. Ako je sustav redom prošao kroz stanja x_1, x_2, .., x_n, vjerojatnost da prijeđe u stanje x ovisi isključivo o trenutnom stanju x_n, dok prethodna stanja x_1, x_2, …, x_{n-1} nemaju utjecaj. Matematički se to može zapisati na idući način:

$$P(X_{n+1}=x \mid X_n=x_n, ..., X_1=x_1) = P(X_{n+1}=x \mid X_n = x_n) \quad (6.1.)$$

Stanja Markovljevog lanca u ovome slučaju su segmenti tunela koji se mogu pojaviti. Matrica koja opisuje uvjetne vjerojatnosti se naziva matrica prijelaza, a uobičajeno sadrži konstante vrijednosti. Promjenjiva matrica prijelaza je potrebna iz tog razloga kako bi se postigao željeni efekt generiranja lakših parova segmenata pri početku igre te generiranje zahtjevnijih parova kako vrijeme protječe. Takvo ponašanje je ostvareno na način da su definirane dvije matrice prijelaza: jedna koja se koristi pri samom početku igre te generira za igrača najlakše oblike tunela i druga matrica prijelaza koja generira za igrača najzahtjevnije oblike. Kako vrijeme protječe, provodi se linearna interpolacija iz prve u drugu matricu te na taj način se mijenjaju uvjetne vjerojatnosti koje se koriste za generiranje tunela. Interpolacija ovisi o parametru *t* koji na početku igre iznosi 0, a dostiže maksimalnu vrijednost 1 nakon što prođe 100 segmenata. Stoga, nakon 100 segmenata tunel postaje maksimalno zahtjevan za igrača. Matrice prijelaza korištene u interpolaciji su prikazane na Slici 6.2, a postupak interpolacije je prikazan na Slici 6.3.

```
float start_mat[5][5] = {
    {0.8, 0.05, 0.05, 0.05, 0.05}, // Straight
    {0.25, 0.5, 0.15, 0.05, 0.05}, // Up
    {0.25, 0.15, 0.5, 0.05, 0.05}, // Down
    {0.25, 0.05, 0.05, 0.5, 0.15}, // Left
    {0.25, 0.05, 0.05, 0.15, 0.5}, // Right
};

float interpolated_mat[5][5] = {
    {0.8, 0.05, 0.05, 0.05, 0.05}, // Straight
    {0.25, 0.5, 0.15, 0.05, 0.05}, // Up
    {0.25, 0.15, 0.5, 0.05, 0.05}, // Down
    {0.25, 0.05, 0.05, 0.5, 0.15}, // Left
    {0.25, 0.05, 0.05, 0.15, 0.5}, // Right
};

float end_mat[5][5] = {
    {0.05, 0.2375, 0.2375, 0.2375, 0.2375}, // Straight
    {0.05, 0.2375, 0.2375, 0.2375, 0.2375}, // Up
    {0.05, 0.2375, 0.2375, 0.2375, 0.2375}, // Down
    {0.05, 0.2375, 0.2375, 0.2375, 0.2375}, // Left
    {0.05, 0.2375, 0.2375, 0.2375, 0.2375}, // Right
};

FString labels[5] = { "straight", "up", "down", "left", "right" };
```

Slika 6.2. Prikaz matrica prijelaza za generiranje beskonačnog tunela.

Sa Slike 6.2. se možemo vidjeti da sve matrice imaju pet stupaca i pet redaka, a svaki stupac odnosno redak odgovara jednoj vrsti segmenta: ravno, gore, dolje, lijevo i desno. Primjerice, s prve matrice se može očitati da ako je zadnji postavljeni segment bio ravan, da postoji 80% vjerojatnosti da će idući segment također biti ravan dok postoji samo 5% vjerojatnosti za segmente ostalih tipova.

```
void ARocket_gameGameMode::interpolate()
{
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            float t = tunelBR / 100.f;
            FGenericPlatformMath::Min(t, 1.f);
            FGenericPlatformMath::Max(0.f, t);
            interpolated_mat[i][j] = start_mat[i][j] * (1 - t) + end_mat[i][j] * t;
        }
    }

}
```

Slika 6.3. Interpolacija između početne i krajnje matrice prijelaza.

Primjeri generiranih segmenata za tunel se nalaze u nastavku:

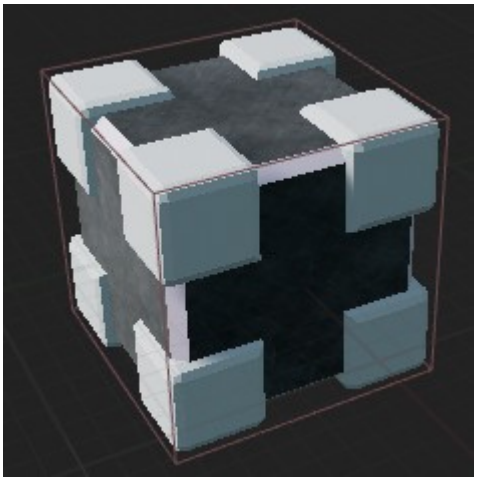
- Pri početku igre: ravno; ravno; lijevo; ravno; ravno; lijevo; lijevo,
- Nakon jedne minute igranja: ravno; lijevo; lijevo; gore; dolje; desno; desno,
- Nakon dvije minute igranja: lijevo; dolje; lijevo; desno; gore; gore; dolje.

6.2.3. Detekcija kolizije sa preprekama

Iako oblik tunela sa vremenom postaje sve teži za navigirati, nakon 100 segmenata on doseže svoju maksimalnu zahtjevnost te od tog trenutka postaje “jednoličan” za igrača. Zbog toga su u tunel uvedene dvije vrste prepreka sa svrhom da nastave otežavati kretanje igrača.

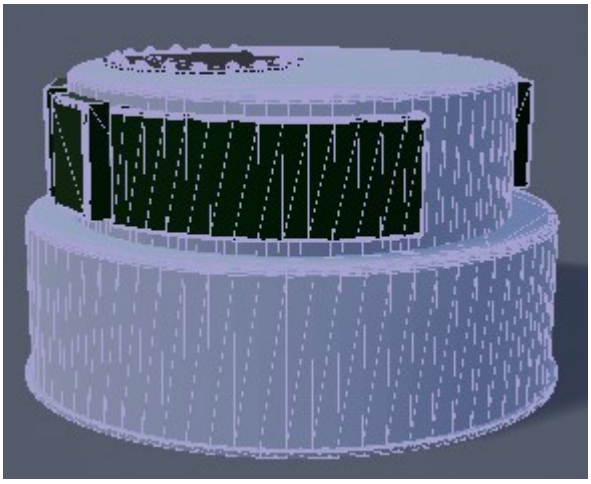
Prva vrsta prepreka su blokovi koji se nasumično pojavljuju u tunelu te na taj način pokušavaju otežati prolazak igraču. Ova prepreka je pasivna u svome ponašanju jer ne nastoji aktivno sabotirati igrača već se računa na to da igrač napravi pogrešku i ne uspije zaobići prepreku. graču je dano naoružanje ograničene količine kojim može uništiti prepreku kako ju ne bi trebao

zaobilaziti. Dodatne količine naoružanja se nasumično pojavljuju u tunelu te ih igrač može sakupiti. Za detekciju sudara sa ovom vrstom prepreke je korištena jednostavna kolizija. Za predstaviti prepreku korišten je kvadrat kao ograničivač volumena budući da u najbolje prijanja obliku prepreke kao što se može vidjeti na Slici 6.4..



Slika 6.4. Prikaz jednostavne kolizije za prepreku.

Druga vrsta prepreke je realizirana u obliku topova koji ispaljuju projektile prema igraču te ga na taj način pokušavaju uništiti. Ponašanje topova je definirano kroz dvije faze. U prvoj topovi čekaju da se igrač približi dovoljno blizu te tada započinje ciljanje. Kada je faza ciljanja gotova, ukoliko se između topa i igrača ne nalazi nikakva zapreka, ispalit će se projektil. Top uvijek ispaljuje dva projektila jednog za drugim te zatim ima pauzu za pripremanje idućeg projektila. Igrač na raspolaganju ima dvije moguće obrane: naoružanjem uništiti topove prije nego ispale projektil ili izbjeći ispaljeni projektil. Topovi se po prvi put počnu pojavljivati nakon što igrač prođe barem 100 segmenata u tunelu, te se pojavljuju sve učestalije što vrijeme više prolazi. Za realizaciju detekcije kolizije za top, bilo je potrebno iskoristiti složenu i jednostavnu koliziju. Zadaće složene kolizije u ovome slučaju je detektirati sudar drugog objekta sa topom. Ukoliko bi se top aproksimirao ograničivačima volumena uz korištenje jednostavne kolizije, uvele bi se nepreciznosti u detekciji kolizije budući da svi ponuđeni ograničivači volumena loše aproksimiraju oblik topa. Ovakve nepreciznosti u detekciji kolizije bi igraču omogućile lakše pogađanje i uništavanje topa. Zbog toga je za detekciju kolizije sa topom bilo potrebno iskoristiti složenu koliziju kao što je prikazano na Slici 6.5. Za detekciju meta koje su u dometu topa, koristi se jednostavna kolizija uz ograničivač volumena u obliku kugle. Radijus kugle predstavlja domet topa, a svi objekti koji se nalazi unutar kugle su potencijalne mete topa.



Slika 6.5. Prikaz složene kolizije postavljene na kupolu topa.

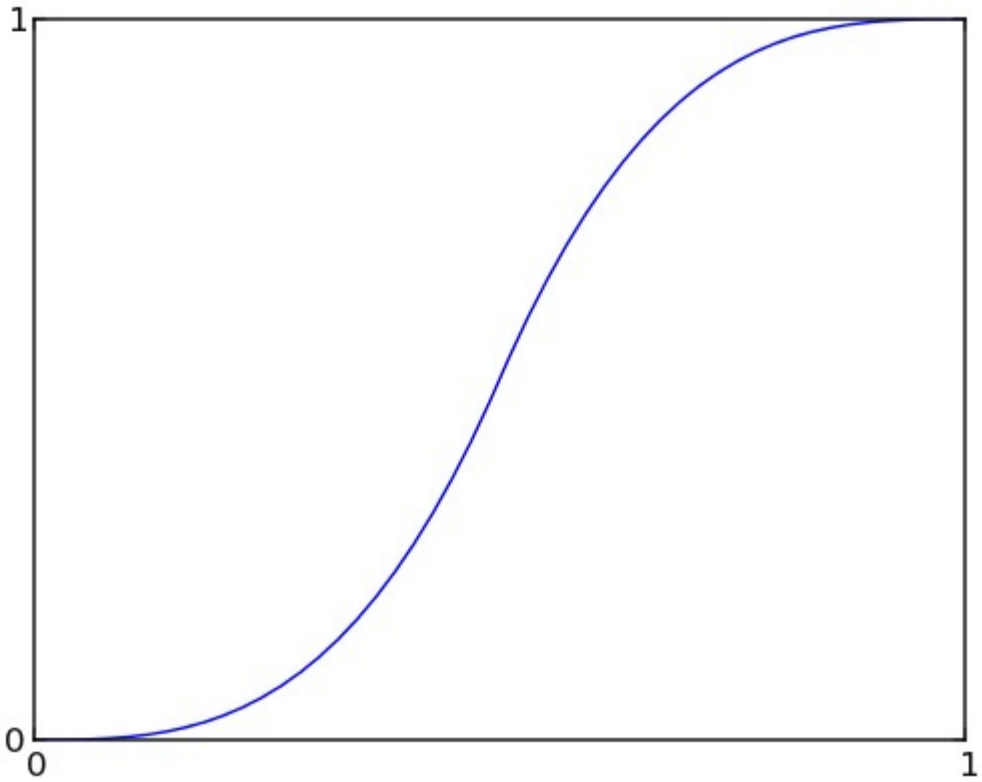
6.2.3. Igračevo naoružanje

Igrač na raspolaganju ima dvije vrste naoružanja za uništavanje prepreka i topova: metke i rakete. Razlika između ove dvije vrste naoružanja je u njihovom ponašanju.

Metak je realiziran na način da se uvijek ispaljuje u smjeru kretanja svemirskog broda te giba se pravocrtno. Kako bi pogodio metu, igrač najprije mora usmjeriti svemirski brod prema meti te zatim ispaliti metak.

Za razliku od metaka, rakete imaju mogućnost automatskog ciljana, odnosno uvijek pogađaju metu te to ih čini više vrijednima igraču. Od trenutka ispaljivanja do trenutka pogotka u metu, raketa prolazi kroz 5 faza. Kada igrač ispali raketu, ona najprije izađe sa donje strane svemirskog broda te tada prelazi u svoju prvu fazu. Tijekom prve faze, raketa se vertikalno udaljava od svemirskog broda te kada postigne zadanu udaljenost, prelazi u drugu fazu. Raketa vozi pravocrtno istom brzinom kao i brod tijekom drugog stanja te prelazi u treću fazu nakon određenog vremena. Raketa na kratko vrijeme uspori u trećoj fazi te zatim započne ubrzanje tijekom četvrte faze. Raketa se nalazi u svojoj četvrtoj fazi sve dok se ne približi na određenu udaljenost do svoje najbliže mete. U tome trenutku, raketa je odabrala svoju metu te prelazi u petu fazu tijekom koje raketa prilagođava svoj kretanje i kut pod kojim će pogoditi metu.

Za izvođenje animacija koje se izvode tijekom opisanih pet faza, koristi se linearna interpolacija s EaseInOutQuad funkcijom prikazanom na Slici 6.6. kako bi se postiglo vizualno zanimljiv efekt ubrzavanja i usporavanja. Trajanje animacije za prve tri faze je predodređeno te one uvijek traju jednako. Četvrta faza u kojoj se traži meta te peta faza u kojoj se pogađa meta same po sebi nisu vremenski ograničene, ali ako raketa ne pogodi metu u određenom vremenu, raketa će se samouništiti. Ovakav mehanizam igra ulogu kada igrač ispali istovremeno ispali dvije ili više raketa jer tada će samo prva raketa pogoditi i uništiti metu, a ostale rakete će izgubiti svoju metu te ih je potrebno samouništiti.



Slika 6.6. Krivulja EaseInOut.

7. ZAKLJUČAK

Ovaj diplomski rad se bavi problemima kolizije, gimbalovim problemom i prikazom likova na zaslonu. Zbog toga je odlučeno pokazati sve probleme na stvarnom primjeru, to jest razvoju videoigre za pametne telefone koristeći stroj Unreal Engine 4. U ovom radu dan je pregled najosnovnijih pojmova i algoritama koji se koriste za razvoj videoigara te sam postupak razvoja jedne videoigre. Sam razvoj videoigre započinje s idejom koja se dalje realizira kroz modeliranje svih objekata i dodavanje funkcionalnosti tim objektima. Postupak razvoja videoigre je gotov kada se kompletna ideja realizira, implementacija testira, a videoigra je spremna za prodaju.

Kroz izradu videoigre, ovaj rad je pokazao koliko sam razvoj može biti težak i izazovan čak i za jednostavnu videoigru prikladnu za osobe svih uzrasta. Tijekom razvoja videoigre sam naišao na mnoge probleme, a jedan od zahtjevnijih je bio implementirati otežavanje igre s vremenom. Ovaj problem sam riješio sustavno postavljanjem pretpostavki što igru čini teškom i definiranje problema matematički pomoću Markovljevih lanaca i linearne interpolacije kako bi se zahtjevnost igre postepeno mijenjala kroz vrijeme.

Kroz ovaj rad sam također predstavio mogućnosti i demonstrirao prednosti korištenja stroja za razvoj videoigre, koliko oni mogu ubrzati razvoj i kako je potrebno čim više iskoristiti napredne postavke koje pružaju. Primjerice, za razvoj grafičkog korisničkog sučelja ili izgleda pojedinih objekta bilo je dovoljno iskoristiti nacrte. Također, kombiniranjem nacrta sa pisanjem programskog koda se olakšava suradnja između programera i tehničkih dizajnera. U ovom hibridnom načinu razvoja, programeri se mogu fokusirati na razvoj željenog ponašanja u programskom kodu, dok dizajnerski tehničari mogu jednostavno modificirati i koristiti ta ponašanja kroz nacrte. Osim toga, nacrti sakrivaju nepotrebne implementacijske detalje te omogućavaju da se dizajnerski tehničari mogu fokusirati na ostvarivanje svojih zadataka.

Razvoj videoigra je izazovan programerski zadatak kroz koji se može vidjeti na koji način osoba pristupa problemu i sposobnost rješavanja problema, a pritom pazeći da performanse konačnog proizvoda ostanu u prihvatljivim granicama. Iako je ideja videoigre izrađene kroz ovaj diplomski

rad bila relativno jednostavna, programski kod i odnosi između objekata su postojali sve kompleksniji dodavanjem svake iduće funkcionalnosti. Kroz izradu ovog rada, uočio sam koliko je bitno programski kod održavati urednim i modularnim s ciljem što lakše implementacije novih i održavanja postojećih funkcionalnosti. Smatram da se konceptualizacijom projekta i sustavnim planiranjem programske strukture može značajno smanjiti spomenute probleme te optimizirati performanse videoigre.

8. LITERATURA

[1] Nadav Lipkin: „Examining Indie's Independence: The Meaning of "Indie" Games, the Politics of Production, and Mainstream Co-optation", 2012.

[2] Stefan Hall, „*How COVID-19 is taking gaming and esports to the next level",* s Interneta, https://www.weforum.org/agenda/2020/05/covid-19-taking-gaming-and-esports-next-level/, 20.08.2022.

[3] Victoria Kennedy, „*Esports coming to 2022 Commonwealth Games*", s Interneta, https://www.eurogamer.net/esports-coming-to-the-2022-commonwealth-games, 20.08.2022.

[4] Sanjay Madhav: "Game Programming in C++: Creating 3D Games", 2018.

[5] P. Mishra, U. Shrawanka: „Comparison between Famous Game Engines and Eminent Games"

[6] M. Herrlich, R. Meyer, R. Malaka, H. Heck, „Development of a virtual electric wheelchair–simulation and assessment of physical fidelity using the unreal engine 3", In *International Conference on Entertainment Computing* (pp. 286-293), 2010.

[7] E. Pukki, „Unreal Engine 4 for Automation", 2021.

[8] Andrew Burnes: "Epic Reveals Stunning Elemental Demo, & Tim Sweeney on Unreal Engine 4", s Interneta, https://web.archive.org/web/20120610175656/http://www.geforce.com/whats-new/articles/stunning-videos-show-unreal-engine-4s-next-gen-gtx-680-powered-real-time-graphics/, 08.05.2022.

[9] John Papadopoulos: "Epic Games' Tim Sweeney Explains Lack Of 'SVOGI' In Unreal Engine 4", s Interneta, https://www.dsogaming.com/news/epic-games-tim-sweeney-explains-lack-of-svogi-in-unreal-engine-4/, 08.05.2022.

[10] Reece A. Boyd: „Implementing Reinforcement Learning in Unreal Engine 4 with Blueprint", 2017.

[11] Tom Sykes: "Unreal Engine 4 now free for academic use", s Interneta, https://www.pcgamer.com/unreal-engine-4-now-free-for-academic-use/, 08.05.2022.

[12] Jordan Sirani: "Unreal Engine 4 is Free for Everyone", s Interneta, https://www.ign.com/articles/2015/03/02/unreal-engine-4-is-free-for-everyone, 09.05.2022.

[13] Christian Nutt: "Unreal Engine 4 is now free-to-download for everyone", s Iterneta, https://www.gamedeveloper.com/business/unreal-engine-4-is-now-free-to-download-for-everyone, 09.05.2022.

[14] John Gaudiosi: "Why Epic Games is giving away its game technology", s Interneta, https://fortune.com/2015/03/03/epic-games-unreal-tech-free/, 10.08.2022

[15] Rachel Cordone: „Unreal Engine 4 Game Development Quick Start Guide", Packt, 2019.

[16] B. Chazelle, H. Edelsbrunner: „An optimal algorithm for intersecting line segments in the plane", 1992.

[17] A. Polak (2016, May): „Counting triangles in large graphs on GPU", 2016.

[18] S. Kockara, T. Halic, K. Iqbal, C. Bayrak and R. Rowe, "Collision detection: A survey," 2007 IEEE International Conference on Systems, Man and Cybernetics, 2007, pp. 4046-4051, doi: 10.1109/ICSMC.2007.4414258.Fgodot

[19] Robert Nystrom: „Game Programming Patterns", 2011.

[20] B. Pfaff: „An introduction to binary search trees and balanced trees", 1998.

[21] David Hoiser: „Avoiding Gimbal Lock in a Trajectory Simulation", 2016.

[22] W. R. Hamilton: „Xi. on quaternions; or on a new system of imaginaries in algebra", 1848.

9. POPIS OZNAKA I KRATICA

- UE4 – Unreal Engine 4
- VR – virtualna stvarnost (engl. virtual reality)
- AI – umjetna inteligencija (engl. Artificial intelligence)
- CCD – kontinuirana detekcija kolizije (engl. Continuous collision detection)

10. SAŽETAK I KLJUČNE RIJEČI NA HRVATSKOM I ENGLESKOM JEZIKU

Sažetak na hrvatskom jeziku

Kroz izradu ovog diplomskog rada, pokazalo se kako je izrada videoigre veoma složen proces koji obuhvaća mnogo različitih znanja, iskustva i osoba različitih profesija, nebitno je li videoigra jednostavna ili složena. Ovaj rad je najprije dao povijesni pregled razvoja videoigra te zatim teoretski razmotrio teme od najvećeg značaja za razvoj videoigre, a to su različiti načini razvoja videoigre, osnove za korištenje Unreal Engine 4, vrste i testovi kolizija, ograničivači volumena, graf scene i njegove implementacije, rukovanje rotacijama i gimbalov problem.

Nakon teoretskog pregleda razvijena je jednostavna videoigra bez granice smještena u svemiru za pametne telefone koristeći Unreal Engine 4. Ova videoigra je iskorištena za demonstraciju koncepata opisanih u teoretskim poglavljima, ali i u cjelini za demonstraciju cijelog procesa razvoja jedne videoigre, od razmatranja koncepta do implementacije algoritama potrebnih za njegovu realizaciju.

Ključne riječi: videoigre, Unreal Engine 4, algoritmi, strukture podataka, vektori, gimbalov problem

Abstract

Through the creation of this thesis, it has been shown that the creation of a video game is a very complex process that includes many different skills, experiences and people from different professions, regardless of whether the video game is simple or complex. This paper first gave a historical overview of the development of video games and then theoretically considered the topics of greatest importance for video game development, which are different ways for a video game development, basics for using Unreal Engine 4, types and tests of collisions, volume limiters, scene graph and its implementation, handling rotations and the gimbal problem.

After a theoretical review, a simple video game without an end set in space for smartphones was developed using Unreal Engine 4. This video game was used to demonstrate the concepts described in the theoretical chapters, but also as a whole to demonstrate the entire process of developing a video game, from the consideration of the concept to the implementation of algorithms necessary for its realization.

Keywords: video game, Unreal Engine 4, algorithms, data structures, vectors, gimbal problem