# Ispitivanje performansi raytracing tehnologije Unreal Enginea

**Lekšan, Dražen**



**SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA**

**Sveučilišni studij**

# ISPITIVANJE PERFORMANSI RAYTRACING TEHNOLOGIJE UNREAL ENGINEA

**Završni rad**

**Dražen Lekšan**

**Osijek, 2021.**

# Sadržaj


1. UVOD ..... 1

1.1. Zadatak završnog rada ..... 1

2. KORIŠTENI ALATI I POSTOJEĆA RJEŠENJA ..... 2

2.1. Računalne igre i razvojna okruženja koja koriste ray tracing ..... 2

2.2. Unreal Engine ..... 2

2.2.1. Razvoj pomoću C++-a ..... 3

2.2.2. Razvoj pomoću Blueprinta ..... 3

2.2.3. Mogućnosti ray tracinga ..... 4

3. RAY TRACING ..... 5

3.1. Način rada ..... 5

3.2. Sjene ..... 6

3.3. Refleksije ..... 8

3.4. Globalno osvjetljenje ..... 11

3.5. Prosvjetljenje ..... 14

3.6. Ambijentalna okluzija ..... 17

3.7. Analiza performansi ..... 21

4. IZRADA RAZINE I IGRE ..... 22

4.1. Korišteni C++ kod ..... 22

4.1.1. Implementacija pomicanja objekata ..... 22

4.1.2. Implementacija mogućnosti nošenja objekata ..... 25

4.2. Korišteni Blueprint ..... 27

4.3. Modeli i materijali ..... 28

5. ZAKLJUČAK ..... 30

LITERATURA ..... 31

SAŽETAK ..... 32

ABSTRACT ..... 32

# 1. UVOD

Završni rad se sastoji od teorijskog i praktičnog djela. U praktičnom dijelu je izrađena jednostavna razina za igru koja dobro prikazuje sve postavke i utjecaj na performanse *ray tracing* tehnologije. Svi modeli i neki materijali potrebni za izradu razine su preuzeti s web trgovine Unreal Enginea. Igrivost je dodana koristeći programski jezik C++ i *Blueprint*. U teorijskom dijelu rada prvo je dan primjer postojećih računalnih igara i razvojnih okruženja koje primjenjuju *ray tracing* tehnologiju te je potom opisan programski alat Unreal Engine, koji je služio za izradu praktičnog dijela rada i njegove mogućnosti. Glavni dio rada je *ray tracing* tehnologija i njen utjecaj na performanse računalnih igara. Opisana je prvo teoretski ne zalazeći preduboko u implementacijske detalje. Svaka postavka je potom pojedinačno opisana te potkrijepljena primjerom iz praktičnog dijela. Na kraju je opisana izrada razine i igre. Opisani su korišteni modeli i materijali od kojih se razina sastoji te C++ kod i *Blueprint* koji omogućuju samu igrivost.

## 1.1. Zadatak završnog rada

Zadatak ovog rada je istražiti *ray tracing* tehnologiju primijenjenu u računalnim igrama i njen utjecaj na performanse. Za to je bilo potrebno izraditi jednostavnu razinu i dodati jednostavne elemente igrivosti koristeći mogućnosti Unreal Enginea 4.26.1.

# 2. KORIŠTENI ALATI I POSTOJEĆA RJEŠENJA

## 2.1. Računalne igre i razvojna okruženja koja koriste ray tracing

U računalnim igrama *NVIDIA* je krenula primjenjivati *RTX* tehnologiju 2018. godine koja omogućuje prikaz *ray tracinga* u stvarnom vremenu. Pojavila se u igrama *Battlefield V* i *Shadow of the Tomb Raider.* Razvojno okruženje koje je korišteno za razvoj *Battlefielda V* je *Frostbite.* Od *ray tracing* postavki u igri su bile podržane samo refleksije koje su imale vrlo velik utjecaj na performanse. *Shadow of the Tomb Raider* je razvijen koristeći *Foundation engine* i od postavki se mogu primijeniti samo sjene.

Napretkom razvojnih okruženja postalo je moguće implementirati sve više postavki. Dobar primjer je igra *Cyberpunk 2077* iz 2020. godine. Od postavki podržava refleksije, ambijentalnu okluziju, sjene i globalno osvjetljenje. Za razvoj je korišteno okruženje *REDengine 4.*

U neke starije igre je kroz ažuriranja dodana podrška za *ray tracing*, a to su *Minecraft* i *Quake II*. *Minecraft* podržava sjene, refleksije i globalno osvjetljenje te se implementacija *ray tracinga* u njegovom razvojnom okruženju smatra jednom od vizualno najupečatljivijih. *Quake II* je izašao 1997. godine i tada je koristio *OpenGL* za iscrtavanje. Ažuriranjem iz 2019. godine je dobio podršku za *ray tracing* refleksije, lom svjetlosti, sjene, ambijentalnu okluziju i globalno osvjetljenje.

## 2.2. Unreal Engine

Unreal Engine je razvojno okruženje za video igre razvijen od strane Epic Games-a. Prva verzija je izašla 1998. godine. Od tada je korišten za razvoj mnoštva 3D igara raznih žanrova. Također se koristi i za neke druge stvari osim razvijanje računalnih igara, npr. u automobilskoj i filmskoj industriji, u arhitekturi i za razne simulacije. Unreal Engine je besplatan za korištenje i otvorenog je koda. Zato ga uglavnom koriste manja poduzeća koja se bave razvojem računalnih igara jer oni nemaju resursa za razvoj vlastitog razvojnog okruženja. Vlasnici uzimaju 5% profita onome tko javno objavi svoju igru za prodaju. Glavni konkurent mu je Unity Engine koji je također besplatan, ali nije otvorenog koda.

Samo razvojno okruženje je napisano u C++ jeziku. Za olakšavanje izrade igara postoji web trgovina s koje se mogu preuzeti gotovi modeli, skripte, zvukovi, materijali i gotovi projekti. Postoje službeno objavljene stvari od strane vlasnike Epic Games-a i stvari objavljene od strane

zajednice. Igre koje se se razvijaju u njemu mogu biti rađene također pomoću C++-a i(li) *Blueprint Visual Scripting* sustava.

### 2.2.1 Razvoj pomoću C++

Unreal Engine 4 nudi dva seta alata razvojnim programerima koji se mogu koristiti zajedno kako bi ubrzali tijekove rada. Novi elemente igrivosti, *Slate* i *Canvas* elementi korisničkog sučelja, te funkcionalnost uređivača mogu se napisati u C++-u i sve promjene će se odraziti Unreal uređivaču poslije kompajliranja u Visual Studiju ili Xcode-u. *Blueprint visual scripting* sustav je robustan alat koji omogućuje klasama da budu napravljene u uređivaču kroz spajanje funkcijskih blokova i referenci svojstava.[1]
Razvoj pomoću C++ je brži za kompajliranje, ali i složeniji. Nudi se veliki broj biblioteka koje sadrže složene tipove podataka i ugrađene funkcije.

### 2.2.2 Razvoj pomoću Blueprinta

*Blueprint Visual Scripting* sustav u Unreal Engineu je cjelovit sustav skriptiranja igranja zasnovan na konceptu korištenja sučelja temeljenog na čvorovima za stvaranje elemenata igranja iz Unreal uređivača. Kao i kod mnogih uobičajenih skriptnih jezika, koristi se za definiranje objektno orijentiranih (OO) klasa ili objekata u stroju. Prilikom korištenja UE4 objekti definirani pomoću *Blueprinta* u kolokvijalnom nazivu nazivaju se samo „nacrtima“. Ovaj je sustav izuzetno fleksibilan i moćan jer pruža mogućnost dizajnerima da koriste gotovo čitav niz koncepata i alata koji su općenito dostupni samo programerima. Pored toga, oznake specifične za *Blueprint* dostupne u implementaciji C++-a Unreal Engine-a omogućavaju programerima stvaranje osnovnih sustava koje dizajneri mogu proširiti.[1]
Blueprinti su sporiji za kompajliranje i jednostavniji za korištenje nego C++.

### 2.2.3. Mogućnosti ray tracinga

*Ray tracing* u Unreal Engineu 4 je sastavljen od dvije tehnike:

1. Hibridnog *ray tracer-a* koji spaja mogućnosti *ray tracinga* s postojećim rasterskim efektima.
2. *Path Tracer-a* za generiranje referentnih iscrtavanja. [3]

Prva implementacija je bila u verziji 4.22 koja je izašla 2.4.2019. godine. Značajke koje su trenutno implementirane su sjene, refleksije, prosvjetljenje, ambijentalna okluzija i globalno osvjetljenje. Sjene simuliraju svjetlosne efekte mekog područja za objekte u okolišu. To znači da će na temelju veličine izvora svjetlosti ili kuta izvora sjena objekta imati oštrije sjene u blizini kontaktne površine nego dalje gdje se omekšava i širi. Refleksije simuliraju precizan prikaz okoliša podržavajući višestruke odskoke svjetlosti. *Screen space* refleksije su sposobne samo za jedan skok svjetla i ograničene su onim što je trenutno vidljivo na ekranu. *Ray tracing* refleksije su sposobne za višestruke skokove svjetlosti i nisu ograničene onim što se trenutno nalazi na ekranu, to znači da možemo u refleksiji vidjeti objekte koji nam nisu trenutno u vidnom polju.
Prosvjetljenje točno predstavlja staklo i tekuće materijale s fizički ispravnim refleksijama, apsorpcijom i lomom na prozirnim površinama. Ambijentalna okluzija precizno zasjenjuje područja koja blokiraju ambijentalno osvjetljene i što bolje zasjenjuje uglove gdje se zidovi susreću i mjesta gdje predmeti dodiruju pod. U usporedbi sa *screen space* ambijentalnom okluzijom kutovi i mjesta gdje predmeti dodiruju pod nisu bespotrebno zatamljena. Globalno osvjetljenje daje interaktivno odskočno osvjetljenje u stvarnom vremenu na područja scene koja nisu izravno osvjetljena izvorom svjetlosti.[3]

# 3. RAY TRACING

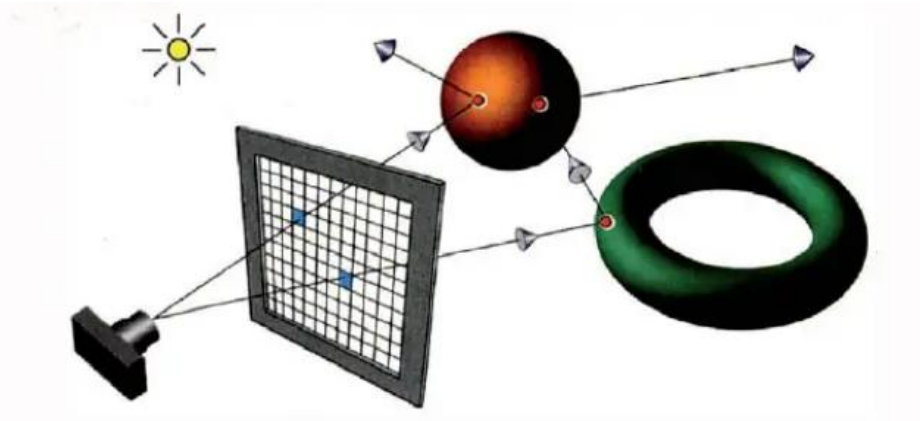
Računalna grafika uključuje simuliranje i distribuciju svjetlosti u 3D prostoru. Postoji samo nekoliko temeljnih algoritama koji su preživjeli test vremena. Mogu se podijeliti u projektivne algoritme i algoritme slikovnog prostora. [2]

Algoritmi slikovnog prostora računaju boju svakog piksela tako što pronađu odakle je došlo svjetlo za taj piksel. Tu je osnovna operacija pronaći najbliži objekt duž vidnog polja. Prateći svjetlo nazad duž linije je dalo ovoj osnovnoj operaciji i pripadajućem algoritmu za sintezu slike ime: *ray tracing*.[2]

U prvom planom znanosti nalazio se 1980-ih godina jer kvaliteta slika koja je mogla biti prikazana sa *ray tracingom* bila zapanjujuća. U području računalnih igara se tek počeo koristiti 2018. godine. Tada je Nvidia predstavila Turing arhitekturu grafičkih kartica koje su bile sposobne pomoću svojih *ray tracing* jezgri prikazati spomenute efekte u računalnim igrama.

## 3.1. Način rada

*Ray tracing* je tehnika u računalnoj grafici koja stvara sliku ispucavanjem zraka što je prikazano na slici 3.1. koja prikazuje kameru, prozor s pikselima, dvije zrake i dva objekta. Zraka ide kroz piksele i testira se presijeca li objekte. Kada zraka pogodi objekt, tragač zraka (engl. *ray tracer*) razrađuje koliko je svjetla reflektirano nazad uz zraku kako bi odredio boju piksela. Koristeći dovoljno piksela, tragač zraka može producirati sliku objekta. Ako su objekti reflektirajući, zrake se mogu odbiti od njih i pogoditi druge objekte.[2]



**Slika 3.1.** *Proces ray tracinga [2]*

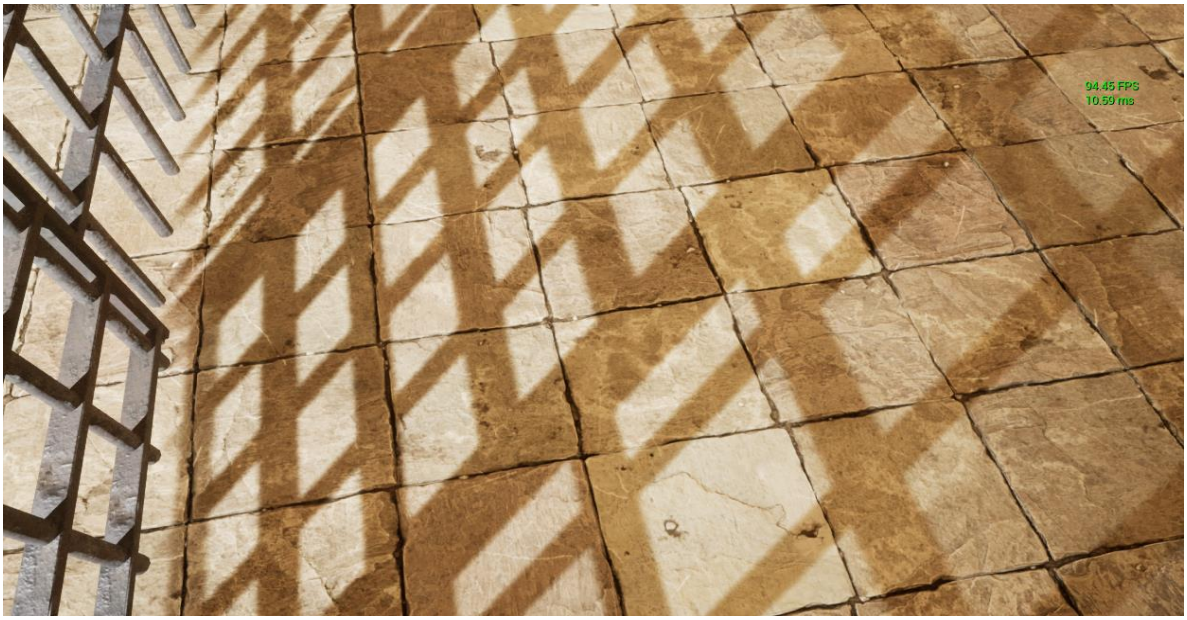
Ovaj proces je konceptualno jednostavan, elegantan i moćan. Omogućuje *ray tracingu* da precizno prikaže refleksije, prozirne objekte, sjene i globalno osvjetljenje (Slika 3.1).
Također ova operacija se može prikazati sljedećim nizom koraka (Slika 3.2):

```
definiraj objekte
odredi materijal za svaki objekt
definiraj izvore svjetla
definiraj prozor čija je površina prekrivena pikselima

Za svaki piksel
ispucaj zraku prema objektu iz centra piksela
izračunaj najbližu točku pogotka zrake se objektima (ako ih
ima)

ako zraka pogodi objekt
koristeći materijal objekta i svjetla izračunaj boju piksela
inače
postavi boju piksela na crnu
```

**Slika 3.2.** *Pseudokod koji prikazuje način rada [2]*

## 3.2. Sjene

Na svakom pojedinom izvoru svjetlosti može se birati želi li se obična ili *ray traceana* sjena.
Ako se koristi *ray tracing* sjena može se birati koliko se želi uzoraka po pikselu i može se mijenjati volumen izvora svjetlosti. Povećanjem uzoraka po pikselu smanjuje se šum, ali se negativno utječe na performanse. Povećanjem volumena izvora svjetlosti sjene postaju „mekše".




**Slika 3.3.** *Rasterska sjena*

Slika 3.3. predstavlja običnu rastersku sjenu uz volumen izvora svjetlosti od devet $cm^3$.


**Slika 3.4.** *Ray tracing sjena*

Slika 3.4. predstavlja *ray tracing* sjenu s izvorom svjetlosti od devet $cm^3$ i osam uzoraka po pikselu kako bi se eliminirao sav šum. Primjećuje se da se broj okvira po sekundi smanjio za oko petnaest. Sjene su jedna od manje zahtjevnih postavki *ray tracinga*.

**Slika 3.5.** *Šum na sjenama*

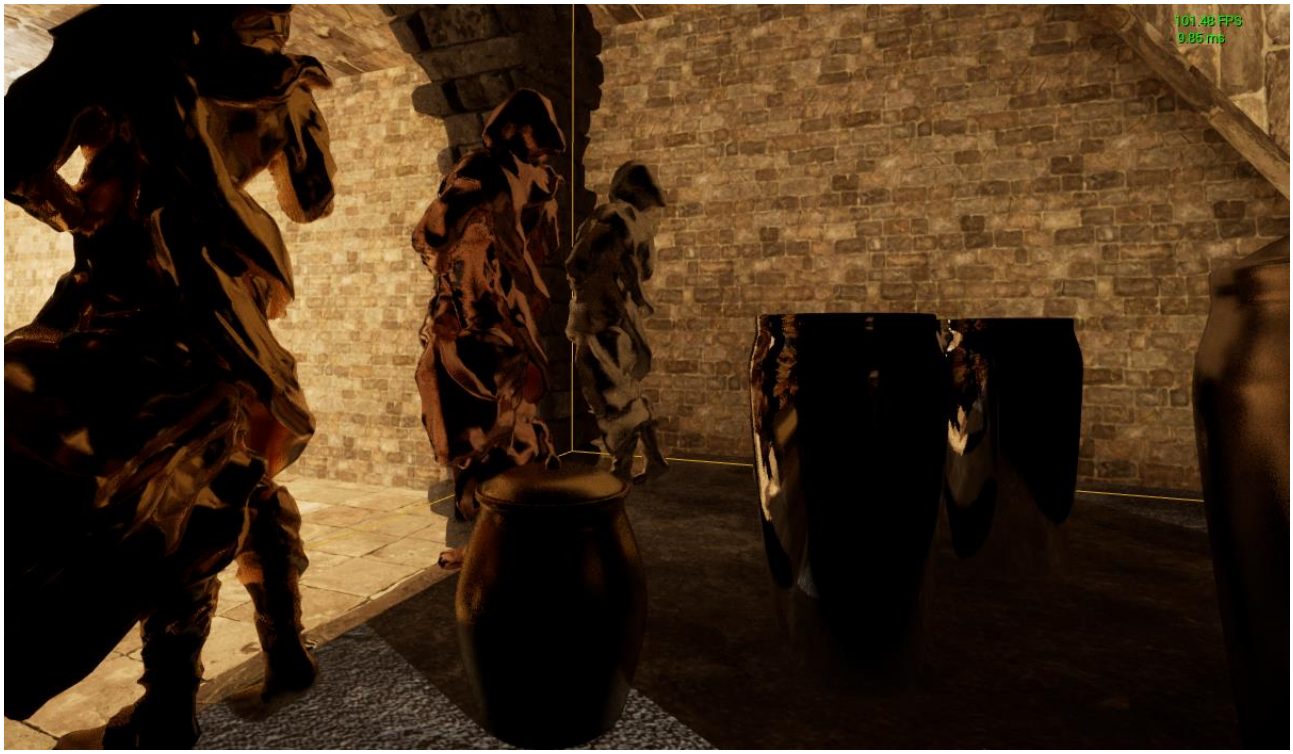
S jednim uzorkom po pikselu se može vidjeti šum po rubovima sjene (Slika 3.5.).

## 3.3. Refleksije

Refleksije se mogu podijeliti u *screen space* refleksije i *ray tracing* refleksije. Za prikaz refleksija se koriste materijali poput plemenitih metala, vode, ogledala. Njihovom kvalitetom se upravlja preko *post process volume-a*.


**Slika 3.6.** *Postavke refleksija*

Slika 3.6 prikazuje sve postavke refleksija, prvo se bira koji tip se želi te za svaki tip postoji ponuđeni niz parametara s kojima se može upravljati.




**Slika 3.7.** *Screen space refleksije*

Mijenjanjem parametara *screen space* refleksija ne utječe se bitno ni na izgled ni performanse (Slika 3.7.).


**Slika 3.8.** *Ray tracing refleksije s jednim skokom svjetlosti i jednim uzorkom po pikselu*

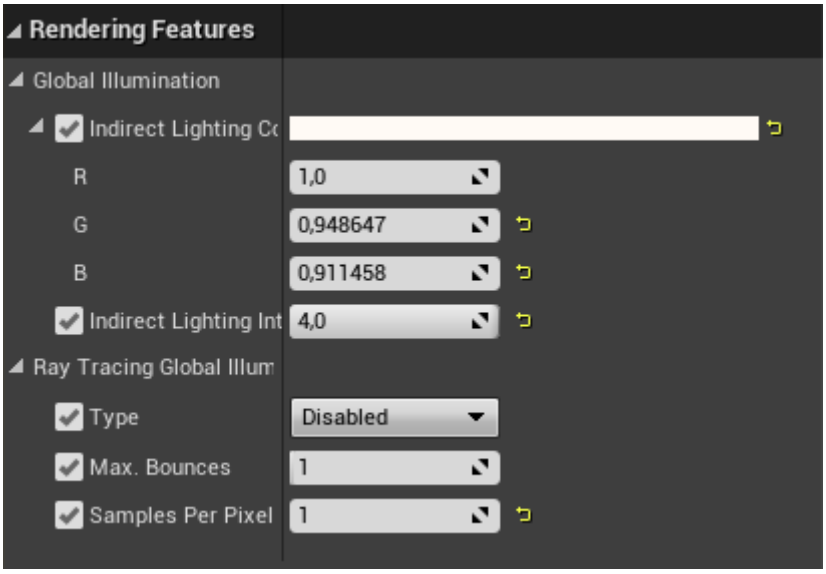

**Slika 3.9.** *Ray tracing refleksije s tri skoka svjetlosti i dva uzorka po pikselu*

Vidi se velika vizualna razlika između screen *space refleksija* i *ray tracing* refleksija(Slika 3.7. i Slika 3.8.). Refleksije su jedna od zahtjevnijih postavki *ray tracinga*. Povećanjem broja skokova svjetlosti i broja uzoraka po pikselu dobije se nešto bolja vizualna kvalitetu, ali izgubi čak

pedeset okvira po sekundi (Slika 3.9.). Smanjivanjem hrapavosti (engl. *roughness*) reflektirajući materijali postaju tamniji na nekim mjestima i povećava se broj okvira po sekundi. Također se može birati hoće li refleksije uključivati sjene.

## 3.4. Globalno osvjetljenje

Mogućnosti globalng osvjetljenja (engl.*global illuminationa*) su prikazane na primjeru osvjetljavanja kraja tunela. S jedne strane je izvor svjetlosti koji nije direktno uperen u tunel. Korištenjem običnog globalnog osvjetljenja nije moguće osvijetliti kraj tunela. Postavkama globalnog osvjetljenja se upravlja preko *post process volume-a*.




**Slika 3.10.** *Postavke globalnog osvjetljenja*

U postavkama se kod običnog globalnog osvjetljenja može samo mijenjati intenzitet indirektnog svjetla i njegova boja. Kod *ray tracing* globalnog osvjetljenja može se birati između *final gather* i *brute force* metode. Za svaku se može namjestiti broj skokova svjetlosti i broj uzoraka po pikselu (Slika 3.10).


**Slika 3.11.** *Obično globalno osvjetljenje*

Na slici 3.11. se primjećuje da kraj tunela nije osvijetljen, nema šuma i nikakvog utjecaja na performanse.

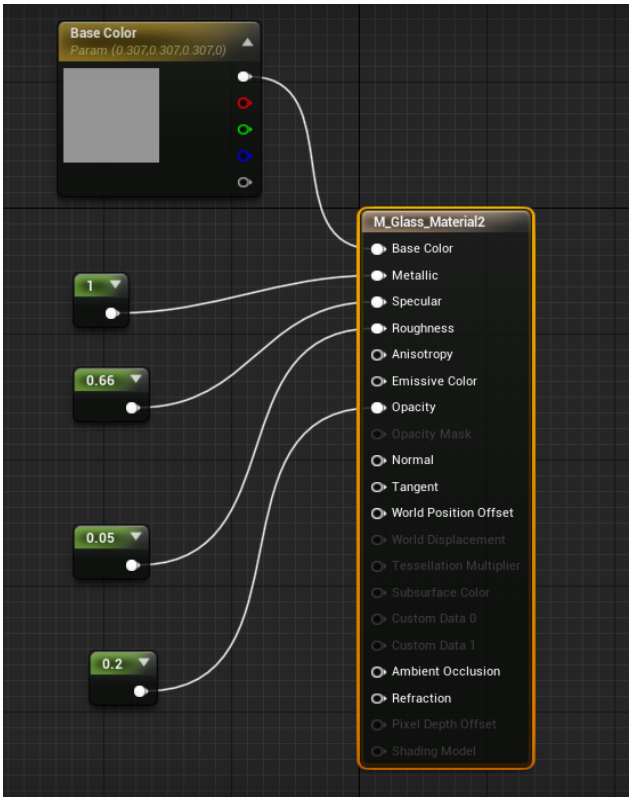

**Slika 3.12.** *Ray tracing globalno osvjetljenje – brute force metoda s tri skoka i tri uzorka po pikselu*


**Slika 3.13.** *Raytracing globalno osvjetljenje – final gather metoda s tri skoka i tri uzorka po pikselu*

Postavljanjem broj skokova svjetlosti na tri uspije se osvjetliti kraj tunela. Ako se koristi *brute force* metoda tada je globalno osvjetljenje najskuplja *ray tracing* postavka (Slika 3.12.), *final gather* drastično manje utječe na performanse, ali se i svjetlost nešto slabije raspršuje (Slika 3.13.). Na slikama se ne može dobro vidjeti šum jer je u obliku promjenjive svjetlosti. Korištenjem *final gather* metode može se povećati broj uzoraka po pikselu do šezdeset četiri i time skoro u potpunosti otkloniti šum ne utječući previše negativno na performanse.

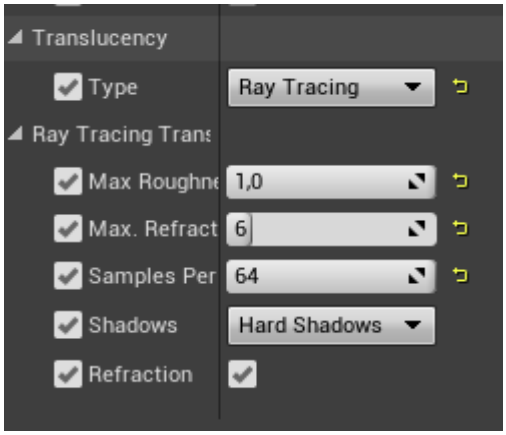
## 3.5. Prosvjetljenje

Prosvjetljenje (engl. *translucency*) je prikazana na primjeru staklene kugle i posude. Na oba modela je postavljen stakleni materijal. Postavkama prosvjetljenja se upravlja preko *post process volume-a*




**Slika 3.14.** *Parametri staklenog materijala*

Materijal je kreiran u Unreal uređivaču. Postavljena mu je osnovna boja u RGB spektru na R: 0.307, G: 0.307, B: 0.307. Pomoću konstanti su mu regulirana svojstva: metalik, spekular, hrapavost i neprosvjetljenje (Slika 3.14.).




**Slika 3.15.** *Postavke prosvjetljenja*

Može se birati između *ray tracing* i rasterske prosvjetljenjosti. Ako se izabere rasterska ne mogu se mijenjati nikakvi parametri (Slika 3.15.).


**Slika 3.16.** *Rastersko prosvjetljenje*

Pri korištenju rasterske prosvjetljenjosti nema loma i refleksije svjetlosti kakve bi inače trebalo biti kod staklenih materijala i zbog toga nema gotovo nikakvog utjecaja na performanse (Slika 3.16.).


**Slika 3.17.** *Ray tracing prosvjetljenje sa jednom zrakom loma*

Ako postoji samo jedna zraka loma objekti su tamni jer svjetlost nije uspjela proći kroz njih (Slika 3.17).

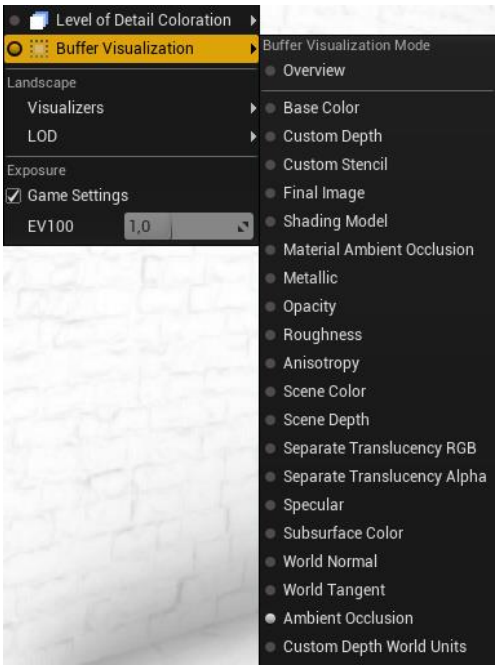

**Slika 3.18.** *Ray tracing prosvjetljenje sa šest zraka loma i maksimalnom hrapavosti*


**Slika 3.19.** *Ray tracing prosvjetljenje sa šest zraka loma i minimalnom hrapavosti*

Postavljanjem broja zraka loma na šest svjetlost bez problema može proći kroz objekt i može se vidjeti lom svjetlosti(Slika 3.18 i Slika 3.19*)*. Broj zraka loma ne utječe na performanse. Performanse gubimo regulirajući hrapavost. Veća hrapavost prosvjetljenosti povećava lom svjetla uz rubove objekta i objekti postaju na nekim dijelovima svjetliji, ali smanjuje performanse (Slika 3.19). Prosvjetljenje nije previše zahtjevna postavka što se tiče performansi

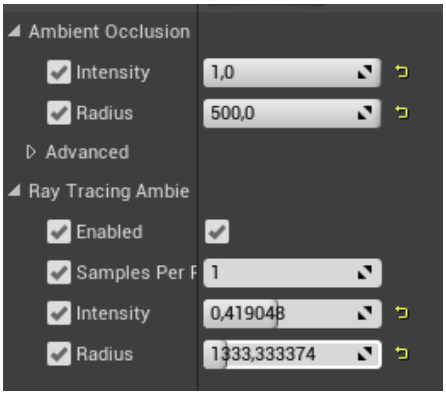
## 3.6. Ambijentalna okluzija

Razlika između *screen space* ambijentalne okluzije (SSAO) i *ray tracing* ambijentalne okluzije (RTAO) je da SSAO bespotrebno zatamnjuje određene dijelove scene. Za najbolji pregled ambijentalne okluzije (AO-a) se koristi *buffer visualization mode* postavljen na AO (Slika 3.20).



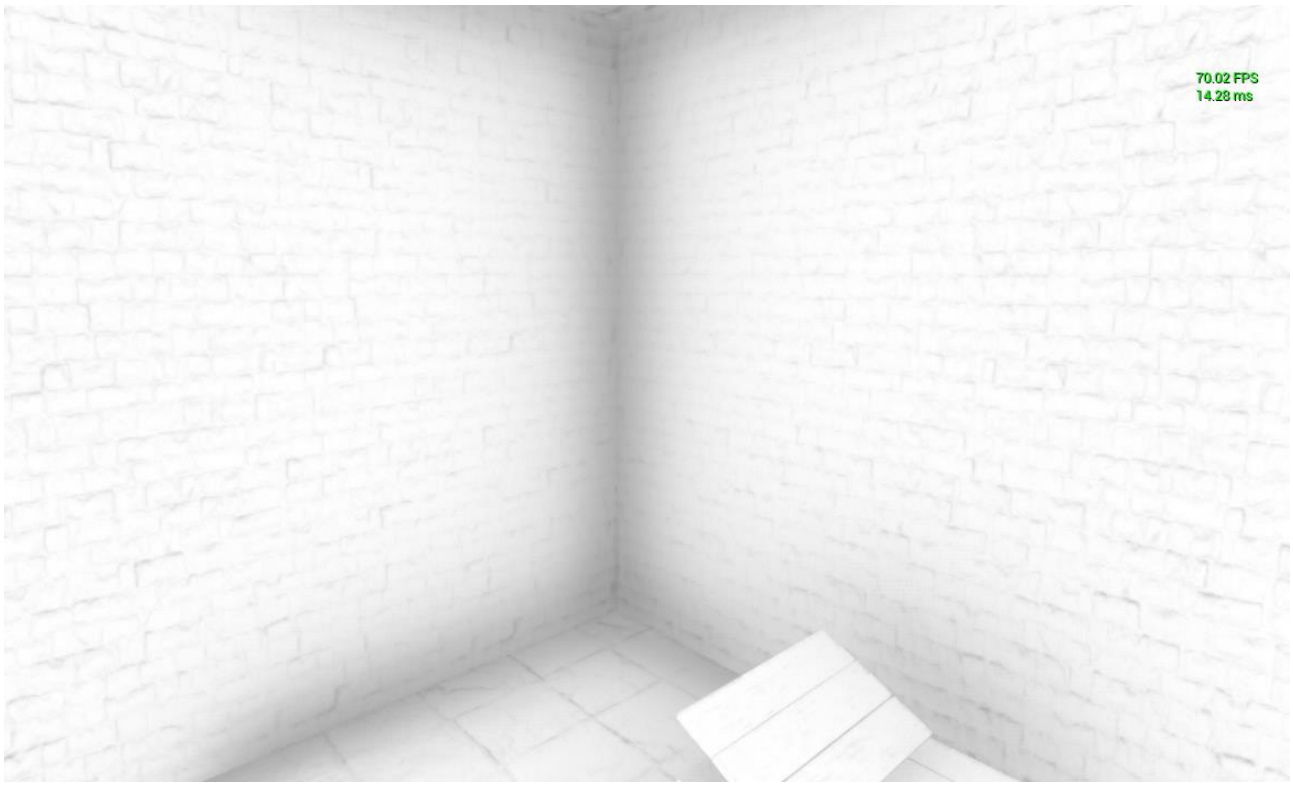

**Slika 3.20.** *buffer visualization mode postavljen na AO*

U ovom načinu slika postaje crno bijela i zasjenjenji dijelovi scene postaju tamniji.

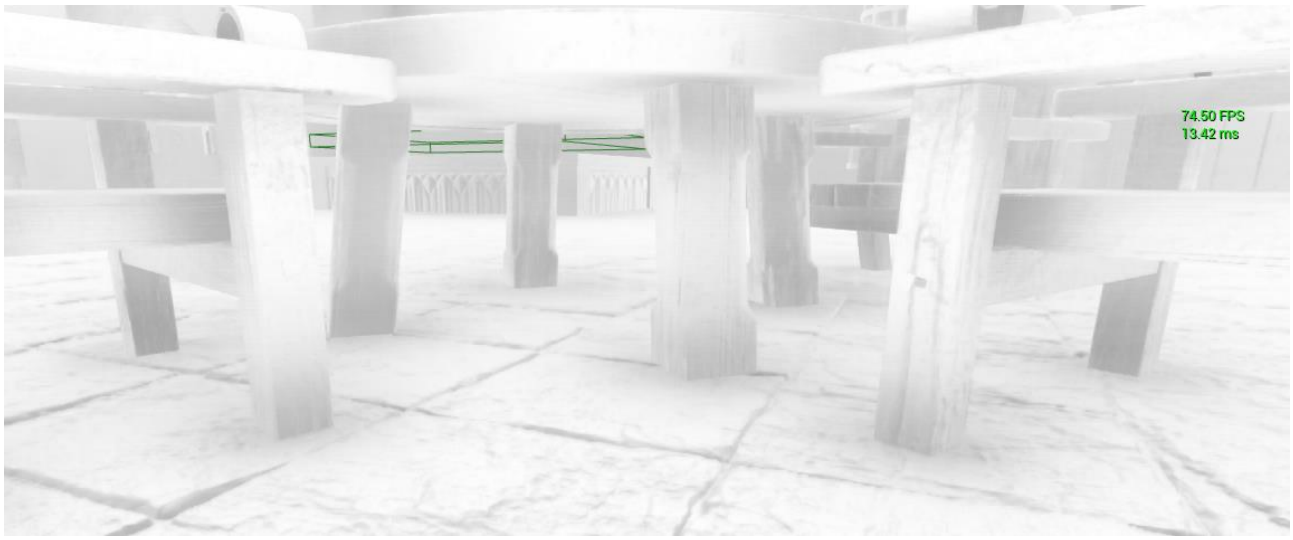



**Slika 3.21.** *Postavke AO*

Intenzitet govori koliko zasjenjeni dijelovi postaju tamniji. Postavke radiusa ovise o veličini scene, ako se koristi manji radius nego veličina scene neće svi dijelovi biti zasjenjeni tj. scena će biti svjetlija. Kada se koristi manji broj uzoraka po pikselu može se javiti šum dok se kreće po sceni (Slika 3.21.).

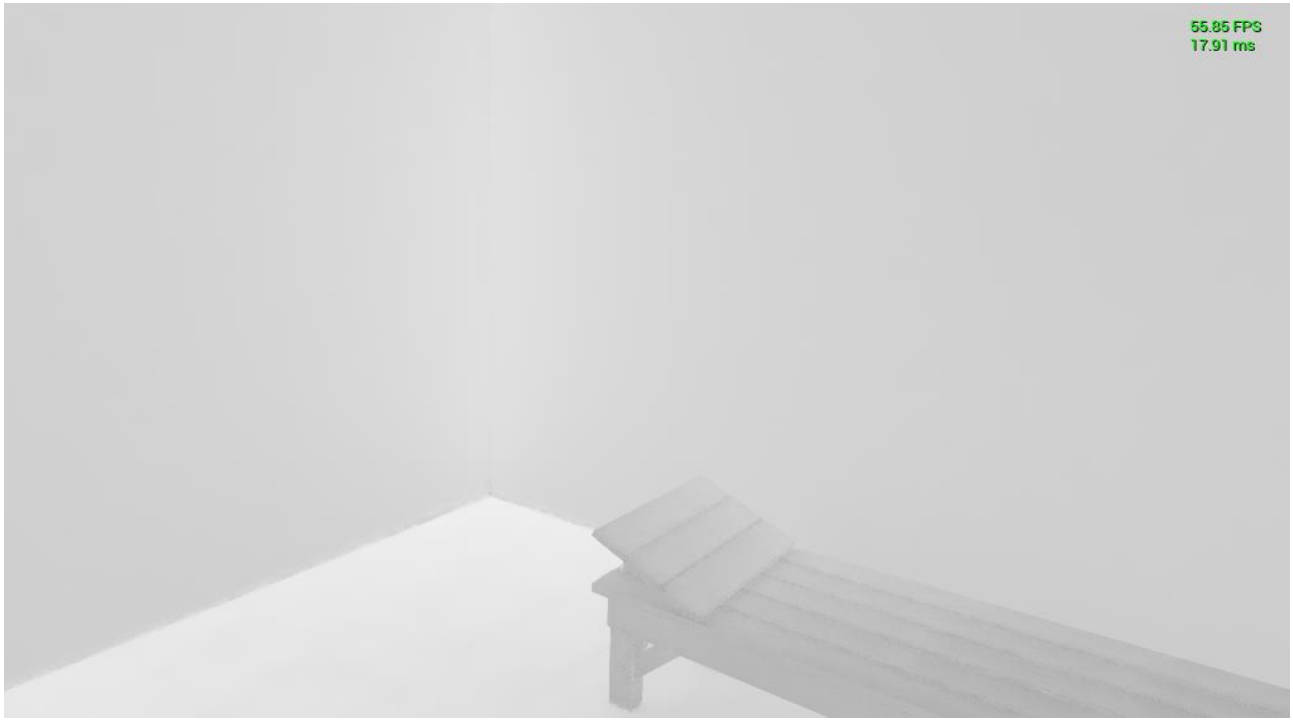



**Slika 3.22.** *SSAO - prikaz kuta*




**Slika 3.23.** *SSAO - prikaz doticanja objekata sa podom*

Na slikama 3.22 i 3.23 se može primijetiti da su kutevi i mjesta dodira sa podom zatamljena kada se koristi SSAO.




**Slika 3.24.** *RTAO - prikaz kuta uz 12 uzoraka po pikselu*


**Slika 3.25.** *RTAO - prikaz doticanja s podom uz 12 uzoraka po pikselu*

Koristeći RTAO se maknulo nepoželjno zatamnjenje s prikazanih mjesta (Slika 3.24 i Slika 3.25). Razlika u performansa između SSAO i RTAO je mala. Kada se koristi RTAO s jednim uzorkom po pikselu dobije se nekoliko okvira po sekundi manje, a povećanjem broja uzoraka po pikselu RTAO postaje sve skuplji što se vidi na slikama 3.24. i 3.25. gdje se izgubi između petnaest i dvadeset okvira po sekundi.

## 3.7. Analiza performansi

Svaka postavka donosi manje performanse u odnosu na rasterske postavke. Kod svih postavki osim prosvjetljenja se pojavljuje šum koji se smanjuje povećanjem broja uzoraka po pikselu, ali to donosi smanjenje performansi. Kod refleksija i globalnog osvjetljenja, koje su ujedno i najzahtjevnije postavke može se regulirati broj skokova svjetlosti. Povećanjem broja skokova svjetlosti samo za jedan se osjeti izuzetan gubitak performansi i praktički nije moguće koristiti više od nekoliko skokova zbog iznimne zahtjevnosti. Efekti globalnog osvjetljenja i ambijentalne okluzije djeluju na cijelu scenu te zbog toga uvijek djeluju i na performanse, dok ostali efekti djeluju na performanse samo kada se nalaze u vidnom polju. Prosvjetljenje i refleksije još imaju mogućnost reguliranja hrapavosti čijim se povećanjem povećava svjetlina reflektirajućih i prozirnih materijala na nekim mjestima te se smanjuju performanse.

# 4. IZRADA RAZINE I IGRE

Za izradu razine su korišteni 3D modeli preuzeti s Epicove trgovine. Neki materijali su samostalno izrađeni, a neki su preuzeti s trgovine. Za dodavanje igrivosti su korištene strukture i klase koje se nalaze u bibliotekama Unreal Engine-a. Za model igrača (engl. *pawn*) je korišten *blueprint*.

## 4.1. C++ kod

Ako želimo dodati pojedinom objektu stanje i ponašanje pridružujemo mu klasu. Svaka klasa se sastoji od .cpp datoteke i datoteke zaglavlja te u .cpp datoteci automatski stvori dvije metode *BeginPlay()* i *TickComponent()*. *BeginPlay()* metoda se pozove jednom kada se igra pokrene, a *TickComponent()* metoda se pozove svaki puta kada se promijeni okvir u igri (npr. ako igra ima šezdeset okvira u sekundi metoda će se pozvati šezdeset puta u sekundi). U programskom kodu su korišteni većinom složeni tipovi podataka koji se nalaze u brojnim bibliotekama Unreal Engine-a.

### 4.1.1. Implementacija pomicanja objekata

Vrata se otvaraju kada se na tlačnoj ploči (engl. *pressure plate*) skupi dovoljna masa. To se postiže zbrajanjem masa svih objekata (engl. *actora*) koji se na njoj nalaze. Masa svakog objekta se može ručno postaviti unutar Unreal uređivača. Tlačna ploča ima ugrađenu metodu za dohvaćanje svih objekata koji se na njoj trenutno nalaze, tj. vraća polje pokazivača tipa *AActor.* S petljom se prolazi kroz to cijelo polje i pomoću ugrađene *GetMass()* metode se dohvati vrijednost atributa mase pojedinog objekta (Isječak koda 4.1.).

```
float UOpenDoor::TotalMassOfActors() const
{
	float TotalMass = 0.f;
	TArray<AActor*> OverlappingActors;
	if (!PressurePlate) { return TotalMass; }
	PressurePlate->GetOverlappingActors(OUT OverlappingActors);

	for (AActor* Actor : OverlappingActors)
	{
	TotalMass += Actor->FindComponentByClass<UprimitiveComponent>()-
	>GetMass();

	}

return TotalMass;
}
```

**Isječak koda 4.1.** *Metoda za računanje ukupne mase svih objekata na tlačnoj ploči[6]*

Vrata se otvaraju mijenjanjem kuta pomoću *Lerp* funkcije koja vrši linearnu interpolaciju između dvije zadane vrijednosti. *OpenDoor()* metoda se poziva unutar *TickComponent()* metode. Tako se promjenom svakog okvira vrata pomaknu za mali kut dok ne dođu do krajnje zadane vrijednosti (Isječak koda 4.2. i Isječak koda 4.3.).

```
void UOpenDoor::OpenDoor(float DeltaTime)
{

	currentYaw = FMath::Lerp(currentYaw, openAngle, DeltaTime*DoorOpenSpeed);
	DoorRotation = GetOwner()->GetActorRotation();
	DoorRotation.Yaw = currentYaw;
	GetOwner()->SetActorRotation(DoorRotation);

}
```

**Isječak koda 4.2.** *Metoda za otvaranje vrata[6]*

Metoda za zatvaranje vrata je implementirana na gotovo identičan način, jedina razlika su parametri unutar *Lerp* funkcije. Također se poziva unutar *TickComponent()* metode uz provjeru određenog uvjeta.

```
void UOpenDoor::TickComponent(float DeltaTime, ELevelTick TickType,
	FActorComponentTickFunction* ThisTickFunction)
{
	Super::TickComponent(DeltaTime, TickType, ThisTickFunction);


	if (TotalMassOfActors()>MassToOpenDoor)
	{
		OpenDoor(DeltaTime);
		DoorLastOpened = GetWorld()->GetTimeSeconds();
	}

	else
	{
		if((GetWorld()->GetTimeSeconds())-DoorLastOpened>=DoorCloseDelay)
		CloseDoor(DeltaTime);
	}

}
```

**Isječak koda 4.3.** *TickComponent() metoda[6]*

Dizalo ne zbraja mase nego kada detektira da igrač stoji na njegovoj tlačnoj ploči krene se pomicati. Također se vrši linearna interpolacija između početne i krajnje vrijednosti. Provjerava se stoji li na tlačnoj ploči igrač i dodatna dva uvjeta potrebna za ispravan rad algoritma. Umjesto rotacije ovdje se linearno interpolira lokacija na Z osi koordinatnog sustava. Spuštanje dizala je implementirano na gotovo identičan način samo su promijenjeni parametri unutar *Lerp* funkcije i parametri unutar uvjeta (Isječak koda 4.4.).

```
void UElevator::LiftElevator(float DeltaTime)
{
	if (PressurePlate->IsOverlappingActor(ActorThatOpens) && elevatorMovementDirection == true
	&& moveElevator == true)
	{
	 if (GetElevatorHeight() == (int)upperElevatorLocation.Z)
	{

		elevatorMovementDirection = !(elevatorMovementDirection);
		moveElevator = false;
	}
	FVector CurrentLocation = GetOwner()->GetActorLocation();
	FVector TargetLocation;
	TargetLocation = FMath::Lerp(CurrentLocation, upperElevatorLocation, 0.01f);
	GetOwner()->SetActorLocation(TargetLocation, false, 0, ETeleportType::None);
	}
}
```

**Isječak koda 4.4.** *LiftElevator() metoda*

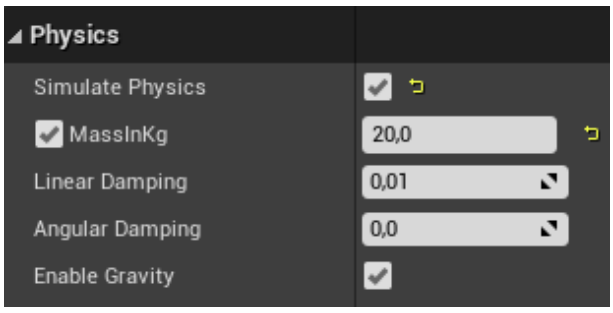
U *BeginPlay()* metodi se dohvaća objekt koji predstavlja igrača, postavlja početna lokacija dizala i inicijaliziraju se atributi. Ta metoda se poziva samo jednom kada se igra pokrene (Isječak koda 4.5.).

```
void UElevator::BeginPlay()
{
	Super::BeginPlay();
	ActorThatOpens = GetWorld()->GetFirstPlayerController()->GetPawn();
	SetElevatorTransform();
	ElevatorMovementDirection = true;
	MoveElevator = true;
}
```

**Isječak koda 4.5.** *BeginPlay() metoda*

### 4.1.2. Implementacija mogućnosti nošenja objekata

Svi objekti koji se mogu pomicati imaju postavljeno simuliranje fizike. Unreal Engine prema zadanim postavkama koristi *PhysX* za vođenje svojih proračuna fizičke simulacije i za izvršavanje svih proračuna sudara. Podsustav *Physics engine* izvodi točno otkrivanje sudara i simulira fizičke interakcije između objekata u svijetu.[4] Moguće je birati masu objekta te linearno i kutno prigušenje. Svojstvo mase se koristi kod otvaranja vrata, linearno prigušenje određuje koliko se objekt opire translacija dok kutno određuje koliko se objekt opire rotaciji (Slika 4.6).




**Slika 4.6.** *Svojstva fizike objekta*

Ova mogućnost je implementirana u klasi *Grabber()* koja je pridružena igraču za razliku od ostalih koje su pridružene objektima na sceni. Prvo je potrebno dohvatiti trenutnu lokaciju igrača

i njegov domet, ti podatci će služiti kao argument metodi koja projicira liniju za detektiranje objekata.

```
FVector UGrabber::GetPlayersReach() const
{
	FVector PlayerViewPointLocation;
	FRotator PlayerViewPointRotation;

	GetWorld()->GetFirstPlayerController()->GetPlayerViewPoint(
	OUT PlayerViewPointLocation,
	OUT PlayerViewPointRotation);

	return PlayerViewPointLocation + PlayerViewPointRotation.Vector() *
	reach;
}
```

**Isječak koda 4.7.** *GetPlayersReach() metoda[6]*

Povratni tip metode je *FVector* što je složeni tip podatka koji se sastoji od X,Y,Z komponente vektora s preciznošću pomičnog zareza. Domet igrača dobije se tako da se njegovoj trenutnoj lokaciji nadoda njegova rotacija pretvorena u jedinični vektor koji ima smjer i orijentaciju u kojem trenutno igrač gleda te se taj jedinični vektor pomnoži sa skalarom koji predstavlja atribut *reach* (Isječak koda 4.7.). Igračeva pozicija se dohvaća na način da se u metodi vrati samo *PlayerViewPointLocation*.

Za dohvaćanje objekata koji imaju postavljeno simuliranje fizike koristi se metoda *GetFirstPhysicsBodyInReach()* koja ima povratni tip *FHitResult*. *FHitResult* je struktura koja sadrži informacije o jednom pogotku linije, poput točke udarca i vektora normale površine u toj točki. Objekti se detektiraju pomoću linije koju baca metoda *LineTraceSingleByObjectType()*. Izlazni parametar joj je objekt hit, još kao argumente prima igračevu poziciju i igračev domet koji služe za određivanje početka i kraja linije, posljednja dva argumenta određuju koje tipove objekata prepoznaje linija i dodatne parametre korištene za liniju (Isječak koda 4.8.).

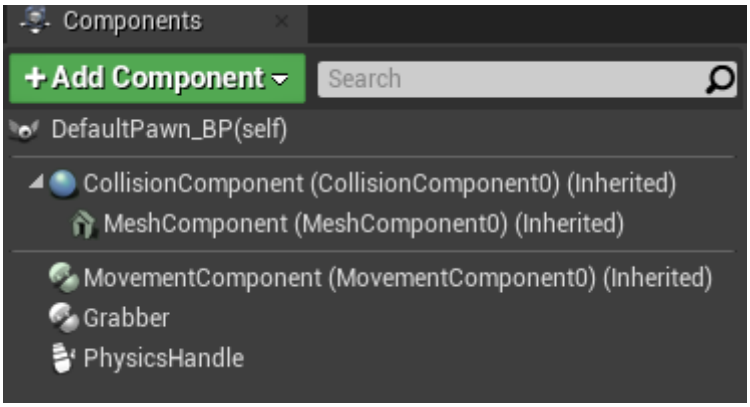
```
FHitResult UGrabber::GetFirstPhysicsBodyInReach() const
{
	FHitResult hit;
	//Ray cast out to a certain distance (Reach)
	FCollisionQueryParams TraceParams(FName(TEXT("")), false, GetOwner());

	GetWorld()->LineTraceSingleByObjectType(
	OUT hit,
	GetPlayersWorldPos(),
	GetPlayersReach(),
	FCollisionObjectQueryParams(ECollisionChannel::ECC_PhysicsBody),
	TraceParams
);
	return hit;
}
```

**Isječak koda 4.8.** *GetFirstPhysicsBodyInReach() metoda[6]*

Iz izlaznog parametra se može dobiti pogođeni objekt koristeći *GetComponent()* ili *GetActor()* metodu. Kada se predmet podigne koristeći desni klik miša njegova se lokacija postavlja na lokaciju igračevog dometa u *TickComponent()* metodi, tj. promjenom svakog okvira se postavlja pozicija objekta na lokaciju igračevog dometa.

## 4.2. Korišteni Blueprint

Za model igrača (engl. *default pawn-a*) je promijenjena C++ klasa koja mu je po zadanom bila pridružena u *blueprint*. *Blueprint* je u ovom slučaju omogućio mijenjanje parametara koji su inače postavljeni u kodu klase direktno u Unreal uređivaču. Neki od podešenih parametara su visina gledišta, simuliranje fizike što uključuje podešavanje mase i prigušenja kod kretanja te 3D model što predstavlja igrača.




**Slika 4.9.** *Komponente dodane blueprintu*

*Blueprintu* je dodana ranije opisana *Grabber* komponenta i *PhysicsHandle* komponenta koja olakšava pomicanje objekata koji imaju uključeno simuliranje fizike. Ostale komponente su automatski dodane (Slika 4.9.).

## 4.3. Modeli i materijali

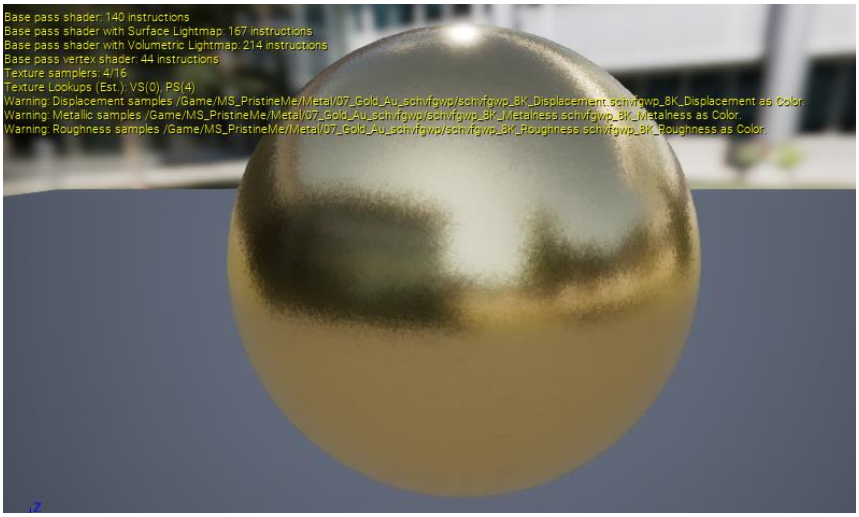
U Unreal Engine-u naziv za 3D modele koji su korišteni za izradu scene je statička mreža (engl. *static mesh*) (Slika 4.10.). To su modeli koji su napravljeni u aplikacijama za 3D modeliranje (kao što su *3dsMax, Maya, Softimage* itd.) i uveženi u Unreal uređivač. Velika većina svake scene se sastoji od njih. Statička mreža je dio geometrije koji se sastoji od skupa poligona koji se mogu predmemorirati u video memoriji i prikazati grafičkom karticom. To im omogućuje učinkovito iscrtavanje, što znači da mogu biti mnogo složenije od drugih vrsta geometrije. Budući da su predmemorirane u video memoriji, statičke mreže se mogu translatirati, rotirati i skalirati.[7]


**Slika 4.10.** *Primjer korištene statične mreže u Unreal uređivaču*

Materijal je sredstvo koje se može primijeniti na statičku mrežu radi kontrole vizualnog izgleda scene (Slika 4.11.). Na visokoj razini vjerojatno je najlakše zamisliti materijal kao "boju" koja se nanosi na objekt. No čak i to može biti pomalo zavaravajuće jer materijal doslovno definira vrstu

površine od koje se čini da je objekt napravljen. Može se definirati njegova boja, koliko je sjajna, može li se vidjeti kroz objekt i još mnogo toga.[8]

Tehnički rečeno, kada svjetlost sa scene pogodi površinu, materijal se koristi za izračunavanje interakcije te svjetlosti s tom površinom. Ovi izračuni se izvode pomoću dolaznih podataka koji se unose u materijal iz različitih slika (tekstura) i matematičkih izraza, kao i iz različitih postavki svojstava svojstvenih samom materijalu.[8]




**Slika 4.11.** *Primjer korištenog materijala u Unreal uređivaču*

# 5. ZAKLJUČAK

*Ray tracing* tehnologija se u nekoliko prošlih desetljeća koristila za prikaz fotorealističnih slika jer hardver nije bio dovoljno napredan da bi se mogla prikazati u stvarnom vremenu. Kada se proizveo hardver sposoban za prikaz *ray tracinga* u stvarnom vremenu razvojni programeri razvojnih okruženja za video igre su krenuli implementirati njegove mogućnosti u razvojna okruženja. Osim Unreal-a implementiran je i u druge razvojna okruženja. Cilj implementacije je da utječe što manje na performanse i da se primijeti jasna vizualna razlika u odnosu na rastersko iscrtavanje. *Ray tracing* se može primijeniti u više grafičkih opcija koje su vezane za osvjetljenje. Svaka opcija drugačije utječe na performanse i donosi drugačiju vizualnu razliku ovisno o tome s kojim parametrima je primijenjena te kakvi su algoritmi korišteni za implementaciju u samom okruženju.

## LITERATURA:


[1] https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ [12.07.2021]

[2] Kevin Suffern, Ray tracing from the ground up, 2007

[3] https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/ [28.08.2021]

[4]https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/Physics/ [26.08.2021]

[5]https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/Engine/ [26.08.2021]

[6]https://www.gamedev.tv/p/learn-unreal-engine-c-developer-4-22-for-video-game-development/ [27.08.2021]

[7]https://docs.unrealengine.com/4.26/en-US/WorkingWithContent/Types/StaticMeshes/ [28.08.2021]

[8]https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/ [28.08.2021]

[9]https://www.nvidia.com/en-eu/geforce/rtx/ [07.09.2021]

## SAŽETAK

Cilj ovog završnog rada je bio izraditi razinu za jednostavnu računalnu igru na kojoj se može prikazati primjena *ray tracing* tehnologije i njen utjecaj na performanse. Rad je izrađen koristeći Unreal Engine. Podijeljen je na tri glavna poglavlja. U prvom poglavlju je opisan korišteni alat za izradu igre i razine te njegove mogućnosti. U drugom poglavlju dan je teorijski opis *ray tracinga* te njegov prikaz na izrađenoj sceni uz prikaz vizualne razlike i utjecaja na performanse. Treće poglavlje prikazuje način na koji je dodana igrivosti kroz opis programskog koda i *blueprinta* te modele i materijale korištene za izradu razine.



Ključne riječi: C++, računalne igre , ray tracing, Unreal Engine



## ABSTRACT

The goal of this final paper was to create a level for a simple video game on which the applied ray tracing technology and its impact on performance can be demonstrated. It is divided into three main chapters. The first chapter describes the game engine tool used to create the game and the level and its capabilities. In the second chapter, a theoretical description of ray tracing is given, as well as its presentation on the created scene, with a presentation of the visual difference and the impact on performance. The third chapter shows how to add gameplay through a description of program code and blueprints and models and materials used to create the level.



Keywords: C++, ray tracing, Unreal Engine, video games