Optimizacijske tehnike kod sjenčanja u Unreal Engineu

Doričić, Mia

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetni preddiplomski studij informatike

Mia Doričić

Optimizacijske tehnike kod sjenčanja u Unreal Engine-u

Završni rad

Mentor: v. pred. dr. sc. Vedran Miletić

Rijeka, 23. rujna 2021.

Sažetak

U posljednjih 70 godina, industrija video igara napredovala je u jednu od najprofitabilnijih industrija u svijetu. Od tada se industrija video igara brzo razvijala u svim aspektima. Jedan od najpoznatijih alata za razvijanje video igara je Unreal Engine. U ovom alatu uvelike se koriste programi za sjenčanje. Kvaliteta i optimalnost ovih programa veoma je važna pri izradi video igara. Iz tog razloga ovaj rad obuhvaća glavne metode koje se koriste pri njihovoj optimizaciji. Koristili smo sljedeće tri metode za optimizaciju programa za sjenčanje; rješavanje nepotrebnih karakteristika, prerada matematičkih formula, pakiranje kanala teksture. Uporabom ovih metoda postignuto je prosječno smanjenje broja asemblerskih instrukcija od 7,6% i sveukupna dobit od maksimalno 68% u broju kadrova u sekundi bez promjene u izgledu tekstura obrađene scene. Korištene metode su nužan dio izrade svake video igre kako bi se izbjegla nepotrebna zahtjevnost scene.

Ključne riječi

Unreal Engine, program za sjenčanje, optimizacija, fps, teksture

Sadržaj

1. Uvod

U posljednjih 70 godina, industrija video igara napredovala je u jednu od najprofitabilnijih industrija u svijetu. Prvi primjer sustava kućnih igara svijetu su predstavili Ralph Baer i njegov tim, stvorivši prototip pod nazivom „*Brown Box*“ 1967. Nedugo zatim, Sega, Taito i Atari bile su prve tvrtke koje su predstavile arkadne igre široj publici i izdale elektromehaničke igre „*Periscope*“ i „*Crown Special Soccer*“ 1966. i 1967. godine. Početkom 1970-ih prodaja osobnih računala i igraćih konzola masovne proizvodnje bila je u porastu. Tehničke promjene također nisu zaostajale. Na primjer, Intel je izumio prvi svjetski mikroprocesor, što je dovelo do stvaranja igara kao što je „*Gunfight*“ 1975., prva borbena pucačina za više igrača. [1]

Od tada se industrija video igara brzo razvijala u svim aspektima; umjetnost, programiranje, glazba, dizajn i pisanje. No, još važnije, grafika. Grafika može prikazati video igru na različite načine, a utječe na početni dojam igre. Može dati određenu marku igri i postati nadaleko prepoznatljiva samo po vlastitom stilu. Također, posebno utječe na iskustvo i ambijent igrača. Ako je ambijent odrađen dobro, igrač neće imati problema s povezivanjem uz priču. To se postiže stvaranjem različitih sredstava igre od strane umjetnika u softveru za 3D modeliranje, na primjer Blender i Maya ili u alatima za manipulaciju slikama poput Photoshopa, Gimpa ili nekog drugog. No najviše od svega, grafičkoj komponenti igre pomažu programi za sjenčanje. [2], [3]

Program za sjenčanje se izvršava na grafičkoj procesorskoj jedinici (GPU) i pokreće se kako bi svakom pikselu rekao što bi trebao učiniti i kako bi trebao izgledati u igri. Omogućuje različite vrste efekata iscrtavanja, pa čak i dodavanje stiliziranih crta u finalni prikaz. Razvoj programa za sjenčanje započeo je ranih 1980-ih na LucasFilmu, gdje su angažirali grafičke programere za izradu kompjuteriziranih posebnih efekata. Budući da se to pokazalo uspješnim potezom u filmskoj industriji, mnoge su tvrtke slijedile istu taktiku. [4]

U početku se izraz program za sjenčanje odnosio na program za sjenčanje fragmenata, ali je kasnije bilo mnogo varijanti programa za sjenčanje, poput onih koji utječu na vrhove i geometriju, pa je termin *program za sjenčanje* postao općenitiji za upotrebu. Ogromna upotreba programa za sjenčanje potakla je mnoge važne promjene u grafici koje će biti dodatno objašnjene u nastavku ovog rada. [5]

2. Osnove 3D računalne grafike

2.1. Iscrtavanje

Iscrtavanje (engl. *Rendering*) je važan proces stvaranja slika iz skupova dvodimenzionalnih (2D) i trodimenzionalnih (3D) mreža. Daje informacije o sceni igre, na primjer položaj svjetla ili orijentaciju kamere u igri. Stvorena slika se naziva okvir (engl. *Render*). Okviri i kadrovi su ono što korisnik vidi na ekranu mnogo puta u sekundi. Obično će optimalna igra prikazati novi kadar sa između 30 i 60 slika u sekundi. Za prikaz kadrova, potrebna je kamera. Kamera za igru je struktura podataka koja sadrži informacije o tome odakle treba gledati scenu igre. Ove su informacije važne za utvrđivanje koja su sredstva i objekti trenutno vidljivi na sceni i trebaju se generirati u kadru. [6]

Dvije su kategorije iscrtavanja; iscrtavanje u stvarnom vremenu (engl. *Real-time rendering*) i prethodno iscrtavanje (engl. *Pre-rendering*). Glavna razlika između njih je brzina kojom računaju i finaliziraju slike.

Iscrtavanje u stvarnom vremenu tehnika je koja se koristi u interaktivnim grafikama i igrama koje zahtijevaju brzo stvaranje slike, na primjer igrama u kojima je količina interakcija korisnika velika u visoko interaktivnom okruženju.

Prethodno iscrtavanje je tehnika koja se koristi u okruženjima u kojima se iscrtavanje izvodi na centralnim procesorskim jedinicama (CPU-ima), a ne na GPU-ima. Ova se tehnika koristi kada je potražnja za kvalitetom animacije i vizualnih efekata velika, a foto realizam mora biti najvišeg mogućeg standarda. [7]

Bez obzira na to koja se tehnika za iscrtavanje igre koristi, kôd igre odgovoran za iscrtavanje kadrova naziva se alatom za iscrtavanje (engl. *Renderer*). Ima određeni niz koraka za pretvaranje mrežnih podataka u nove kadrove, koji se naziva tok rada iscrtavanja. [6]

2.2. Tok rada iscrtavanja

Tok rada iscrtavanja (engl. *Rendering pipeline*) slijed je koraka koje OpenGL poduzima pri generiranju objekata. Ovaj se tok rada izvodi na svakoj mreži za koju kažemo da GPU treba uzeti u obzir za sliku koju trenutno iscrtava. Nakon što je svaka od tih mreža poslana kroz tok rada, imamo svoju sliku. Ovi se koraci izvode isključivo na GPU-u, a jedan od razloga zašto su grafičke kartice

toliko važne za igre je taj što su napravljene za što brže izvođenje grafičkog toka rada.

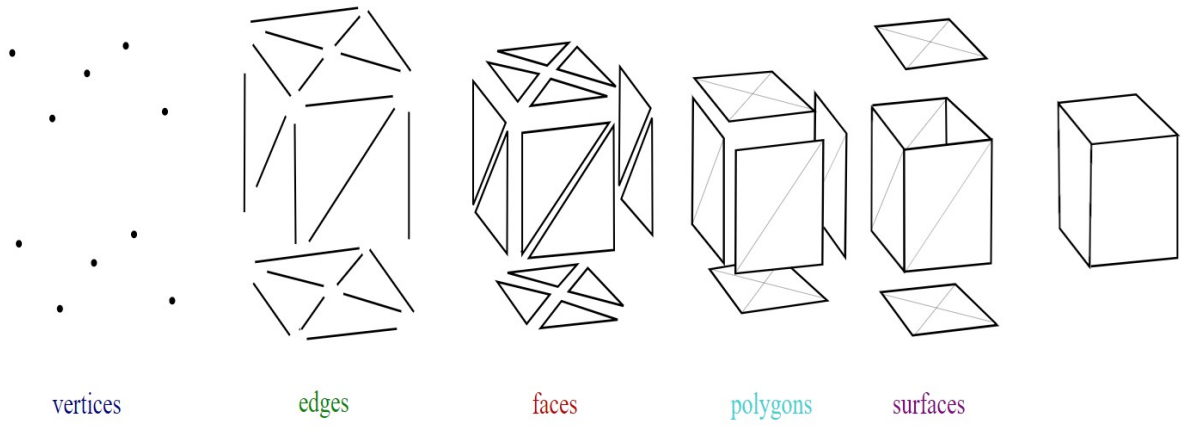
Programi za sjenčanje vrhova, koje detaljnije opisujemo kasnije, prvi su korak u toku rada. Oni se brinu o tome da shvate gdje na ekranu treba nacrtati svaki vrh u mreži koja se trenutno obrađuje. Kad se objekt pomiče u video igri, to je zato što se položaj tog objekta šalje u program za sjenčanje vrhova, koji zatim to kombinira s drugim podacima, poput podataka o položaju i orijentaciji kamere u igri, i koristi sve to da odluči gdje će postaviti vrhove mreže tog objekta na ekranu.

Nakon što program za sjenčanje vrhova završi s obradom svih vrhova u mreži, šalje te podatke u sljedeći korak toka rada, koji se naziva “montaža oblika” (engl. *Shape montage*). Ova faza je odgovorna za povezivanje svih onih vrhova koji su upravo bili obrađeni linijama. Drugim riječima, odgovorna je za stavljanje rubova mreže na zaslon. Sljedeća faza u toku rada je rasterizacija. U ovoj fazi GPU shvaća koje piksele na ekranu bi mreža mogla zauzeti, te za svaki od ovih potencijalnih piksela stvara fragment, koji je struktura podataka koja sadrži sve podatke potrebne za iscrtavanje jednog piksela na zaslonu. Korak rasterizacije ne zna kako izgleda površina mreže; potrebna mu je pomoć od programa za sjenčanje fragmenata kako bi ispunio informacije o tome koje boje svaki od fragmenata mora biti, pa šalje sve fragmente na sljedeći korak u tijeku, gdje programski kod automatski ispunjava te podatke. Naposlijetku, programi za sjenčanje fragmenata su kao boje u linijama koje stvara faza sastavljanja oblika. [6]

2.3. Mreža

Poligonska ili 3D mreža (engl. *Mesh*) je strukturna konstrukcija 3D modela koji se sastoji od poligona. 3D mreže koriste referentne točke na osi X, Y i Z za definiranje oblika s visinom, širinom i dubinom.

Objekti izrađeni poligonskim mrežama moraju pohraniti različite vrste elemenata. To uključuje vrhove, rubove, lica, poligone i površine. U mnogim aplikacijama pohranjuju se samo vrhovi, rubovi i lica. Alat za iscrtavanje može podržavati samo trostrana lica, pa se poligoni moraju izgraditi od mnogo njih.



Slika 1 - elementi modeliranja poligonalne mreže

Vrhovi objekta u 3D prostoru sadrže svoju poziciju na osima X, Y i Z, a rubovi su poveznica između dva vrha. Lica su zatvoreni skup rubova, u kojem trokutasto lice ima tri ruba, a četverostrano lice ima četiri ruba. Površine su češće nazivane grupama za zaglađivanje. To je skupina poligona u poligonskoj mreži koja bi trebala izgledati kao glatka površina. Grupe za zaglađivanje korisne su za opisivanje oblika gdje su neki poligoni glatko povezani sa svojim susjedima, a neki nisu.

Većinu 3D mreža stvaraju umjetnici pomoću programskih paketa; komercijalnih programa poput Maye, 3D Studio Max ili besplatnog otvorenog programa Blender 3D. Kada se modeli stvaraju za animaciju, oni zahtijevaju posebno pažljivu konstrukciju; mogu nastati veoma čudne deformacije ako se poligoni pažljivo ne postave za kontinuirane rubne petlje (sustav koji povezuje sve rubove) oko područja koja će se pomicati. [8]

2.4. Program za sjenčanje

Program za sjenčanje (engl. *Shader*) je dio koda koji se obično izvodi na GPU, za manipulaciju slikom prije nego što se prikaže na zaslon. Programi za sjenčanje dopuštaju različite vrste učinka iscrtavanja, u rasponu od dodavanja rendgenskog prikaza do dodavanja stiliziranih linija u ispis iscrtavanja. Najčešće se koriste za stvaranje osvijetljenih i zasjenjenih područja pri prikazivanju 3D modela.

Razlika od većine programa koje ljudi pišu je upravo to što se programi za sjenčanje obično izvode na CPU-u. To znači da programi za sjenčanje mogu učiniti neke stvari koje regularni programi ne mogu. Najvažnija stvar koju mogu učiniti je kontrolirati dijelove toka rada za iscrtavanje. Većina toka rada ostaje ista bez obzira na vrstu objekta koji se generira. Programi za sjenčanje koji rade na

različitim fazama toka rada dobivaju različita imena, kao na primjer program za sjenčanje vrhova i program za sjenčanje fragmenata/piksela. Ovo su najčešće korištene vrste programa za sjenčanje, budući da su to minimalni zahtjevi za dobivanje nečega kroz tok rada za iscrtavanje i na zaslonu.

Budući da je svaka faza toka rada različita, programi za sjenčanje napisani za svaku fazu također su različiti: program za sjenčanje vrhova se ne može koristiti za obradu fragmenata ili obrnuto. Programi za sjenčanje su napisani u specijaliziranim programskim jezicima koji se nazivaju jezici programa za sjenčanje. Postoji mnogo jezika za sjenčanje, kao na primjer HLSL, GLSL i Cg, ali zapravo su svi vrlo slični. [5], [6]

2.4.1. Jezici programa za sjenčanje

Izvorno su programi za sjenčanje pisani u asemblerskom kodu, ali kao i sa svim drugim programskim poljima postalo je jasno da je potreban jezik više razine. Naime, kada je uveden programibilni tok rada konačno se moglo programirati programe za sjenčanje u asemblerskom kodu. Međutim, programe za sjenčanje je bilo teško pisati i čitati. Tada su se razvili jezici više razine kako bi se lakše moglo razumjeti programe za sjenčanje.

Jezik na kojem su programirani programi za sjenčanje ovisi o cilju koji se želi postići. Službeni jezik sjenčanja za OpenGL i OpenGL ES je OpenGL Shading Language, također znan i kao GLSL, a službeni Direct3D jezik sjenčanja je High Level Shader Language, također poznat kao HLSL. Cvid, jezik sjenčanja treće strane koji emitira i OpenGL i Direct3D programe za sjenčanje razvila je Nvidia; međutim od 2012. je zastario. [9]

2.4.2. Programi za sjenčanje vrhova

Program za sjenčanje vrhova (engl. *Vertex shader*) je funkcija grafičke obrade koja se koristi za dodavanje posebnih efekata objektima u 3D okruženju izvršavanjem matematičkih operacija nad podacima o vrhu objekata.

Svaki vrh može biti definiran s mnogo različitih varijabli, na primjer, vrh je uvijek definiran svojim položajem u 3D okruženju koristeći koordinate X, Y i Z. Vrhovi se također mogu definirati bojama, teksturama i svjetlosnim karakteristikama.

Programi za sjenčanje vrhova zapravo ne mijenjaju vrstu podataka, oni jednostavno mijenjaju vrijednosti podataka, tako da se pojavljuje vrh s drugom bojom, različitim teksturama ili drugačijim položajem u prostoru. [10]

2.4.3. Programi za sjenčanje piksela/fragmenata

Program za sjenčanje piksela (engl. *Pixel shader*) je isto kao i program za sjenčanje fragmenata (engl. *Fragment shader*). Glavna razlika između ovih programa za sjenčanje i programa za sjenčanje vrhova je u tome što program za sjenčanje vrhova može manipulirati atributima vrhova. S druge strane, program za sjenčanje piksela brine o tome kako pikseli između tih vrhova izgledaju. Interpolirani su između definiranih vrhova slijedeći posebna pravila.

Na primjer, ako poligon treba biti potpuno crven, sve vrhove bi trebalo definirati kao crvene. Ako su potrebni određeni efekti poput gradijenta između vrhova, to se čini u programu za sjenčanje fragmenata. Dakle, program za sjenčanje piksela/fragmenata dio je koraka rasterizacije, gdje se slika izračunava i pikseli između vrhova se popunjavaju ili "obojavaju".

Izraz "fragment" koristi se jer se rasterizacijom svaki geometrijski primitiv, poput trokuta, razbija u fragmente veličine piksela za svaki piksel koji primitiv pokriva. Fragment ima pridruženo mjesto piksela, vrijednost dubine i skup interpoliranih parametara kao što su boja, sekundarna (spekularna) boja i jedan ili više skupova koordinata teksture. [11]

3. Unreal Engine

3.1. Što je Unreal Engine

Unreal Engine (UE) jedan je od najjačih softvera za razvoj igara na svijetu. Razvio ga je Epic Games 1998. godine. Smatra se najjačim zbog širokog raspona alata, uređivača i značajki koje pruža, pa razvoj igara omogućuje svima. U početku je stvoren za pucačine na osobnom računalu, ali se od tada razvio u alat koji se koristi za različite žanrove 3D igara. [12]

S obzirom na to kako je s vremenom napredovao, druge industrije (poput filmske i televizijske industrije) su uključile UE u svoj program rada. Napisan je na C++ programskom jeziku, pa zbog toga podržava mnoge platforme, poput osobnih računala (PC), virtualne stvarnosti (VR), konzola i mobilnih platformi.

3.2. Usporedba s drugim alatima

Kada plan za stvaranje igre nastane, prva od stvari koju programer mora odlučiti je koji alat će koristiti za svoj projekt. Ova odluka može biti problematična s obzirom na količinu alata za razvijanje igara na tržištu. Za usporedbu možemo iskoristiti tri najpopularnija alata za razvijanje igara u 2021. godini; UE, Unity i Godot. [13]

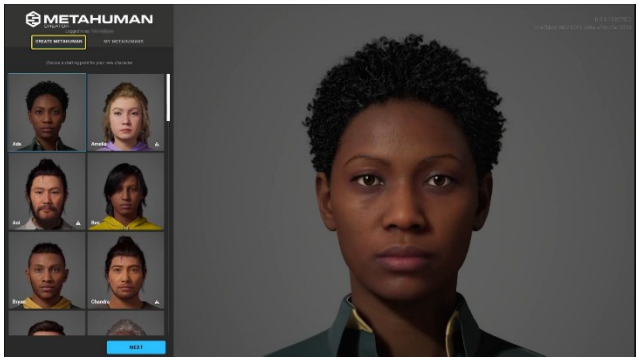
3.2.1. Prednosti i mane Unreal Engine-a

Najpopularniji od trojke je UE. Široko je poznat po svojoj mogućnosti za stvaranje igara visoko kvalitetne grafike. Iako se za programiranje igre koristi jezik C++, nudi alternativnu opciju naziva *Blueprints*. To je vizualni sistem programiranja, omogućavajući programiranje korisnicima koji ne znaju programski jezik. Ovaj način programiranja može biti ponešto stresan za nove korisnike, pogotovo za korisnike koji nisu upoznati sa logikom programiranja. UE najčešće koriste kompanije s velikim budžetima, težeći stvaranju svjetski poznatih igara kao što su npr. *Bioshock, Dishonored* itd.

Slika 2 - prikaz igara napravljenih u UE alatu

Prednost kod ovog alata je još i njihovo vlastito tržište materijalima, mapama i objektima za igru. Često se na njihovoj stranici prodaju objekti za potpuno besplatno, te su ostali popusti sveprisutni. Korištenje tih materijala i objekata garantira njihovu kvalitetu i optimiziranost u UE alatu, s obzirom da su sve te komponente napravljene izričito za UE.

Neke od značajki s kojima se UE ističe su uređivači s dinamičkom fizikom i efektima, foto realističnim iscrtavanjem, realističnim animacijama i još mnogo toga. Ima integriranu medijsku podršku koja donosi veću kontrolu nad video manipulacijama, uređivanjem zvuka i medijskim okvirom. Također, nudi besplatan sadržaj s kojim svatko može stvoriti realne i visokokvalitetne razine i likove sa svojim Megascanima; bibliotekom skeniranja u stvarnom svijetu s tisućama 3D materijala i Metahumanima; aplikacijom za stvaranje foto realističnih digitalnih ljudi s kosom i odjećom. [14], [15]



Slika 3 - prikaz Metahuman kreatora

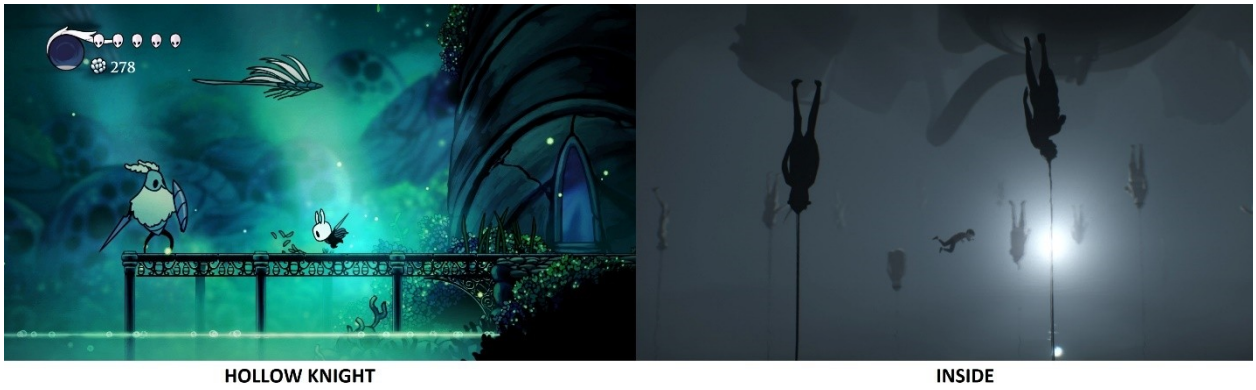
Kao bitna stavka za naglasiti je to da je UE besplatan za koristiti, te nudi punu dokumentaciju za korištenje. Dozvoljeno je objaviti svoju završenu igru na bilo koju platformu, te staviti ju na tržište. Međutim, u pravilima stoji da u slučaju da igra uprihodi preko $1,000,000, potrebno je platiti UE kompaniji 5% prihoda.

3.2.2. Prednosti i mane Unity-a

Unity alat slovi kao najbiraniji alat za samostalne programere koji žele napraviti svoj vlastiti projekt. Razvio ga je Unity Technologies 2005. godine. Radi svoje jednostavnosti, učenje ovog alata ide poprilično brzo. Grafički se poboljšava svakom godinom, međutim i dalje nije na razini grafike koju UE može pružiti. Također, najčešće se koristi u izradi igara za mobitele.

Za programiranje igre koristi jezik C#. Ovaj jezik je mnogo jednostavniji za naučiti, obzirom da su naredbe mnogo jasnije za shvatiti. Unity ima detaljno razrađenu dokumentaciju, kao i vodiče kroz programiranje. Međutim, ako programiranje predstavlja problem, također postoji vizualni sistem programiranja kao i UE.

Radi toga što ovaj alat ima manje mogućnosti i pomoćnih alata za manipulaciju igrom, najčešće se koristi za izradu kraćih igara manje zahtjevnosti. Primjer nekih igara napravljenih u ovom alatu su *Hollow Knight* i *Inside*. Također, Unity ima sam svoje tržište, ali ne nudi visokokvalitetne i realistične objekte besplatno kao što ih nudi UE, upravo zato jer se najčešće koristi za igre manjih razmjera. Postoje besplatni paketi koje korisnik može preuzeti, no to su najčešće početnički paketi i manji alati za započeti igru.



Slika 4 - primjer igara napravljenih u Unity alatu

Jedna od lošijih strana ovog alata je što, za razliku od UE granica zarade od igre je niža. Drugim riječima, ako igra napravljena u besplatnom Unity alatu pod Personal licencom zaradi više od $100,000 u godinu dana, alat više nije moguć za korištenje dok god korisnik ne kupi Professional licencu kojoj cijena varira između $200-$400. Kada se to dogodi, nadogradnje i popravljanje svoje igre nije moguće bez licence.

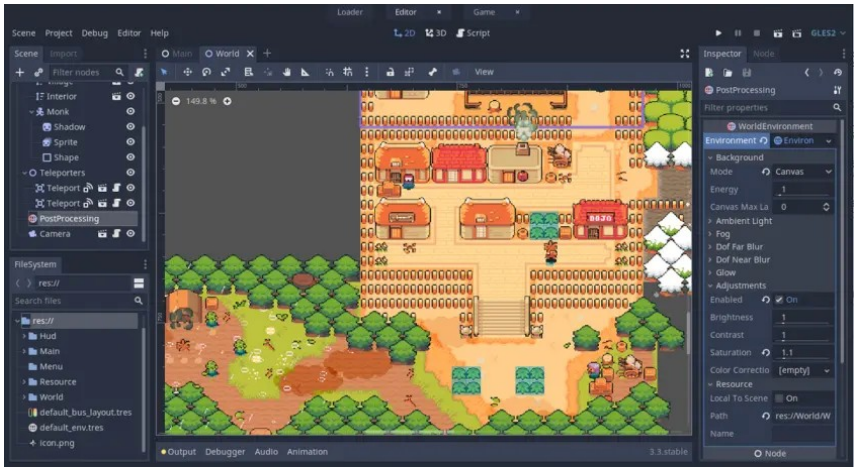
3.2.3. Prednosti i mane Godot-a

Godot je besplatni open-source softver razvijen 2014. godine, čineći ga najmlađim alatom od troje.

Vrlo dobra karakteristika ovog alata je upravo to što je softver otvorenog koda (engl. *open source*), što znači da se korisnik ne mora brinuti oko licence ili plaćanja nakon određene zarade.

Uz to što je besplatan, vrlo je jasan i povoljan za početnike u stvaranju igara. Jezik kojim se programira u ovom alatu zove se GDScript, jezik napravljen konkretno za ovaj alat. Jezik je sličan Pythonu, pa bi nekim korisnicima bilo lakše programirati u tom jeziku više nego u C++ ili C#.

Na drugu stranu, ovaj alat nema mogućnosti za pružiti visoku kvalitetu grafike kao UE ili Unity. Ima svoje tržište, ali u odnosu na ostala dva alata, vrlo je maleno i prazno. Iz tog razloga je najbolji za početnike koji ne žele veliki pothvat iz svoje igre, već žele početi učiti od malih koraka.



Nema konkretnih primjera uspješnih igara za ovaj alat, s obzirom da su to najčešće male igre napravljene s ciljem vježbe, učenja ili kao prezentacija za neku konferenciju.

Slika 5 - primjer izgleda sučelja alata Godot

Kao zaključak usporedbe, sva tri alata su besplatna, imaju svoje tržište, no za stvaranje najkvalitetnijih i grafički najrazvijenijih igara UE se pokazao najpovoljnijim za te svrhe.

3.3. Osnove sučelja

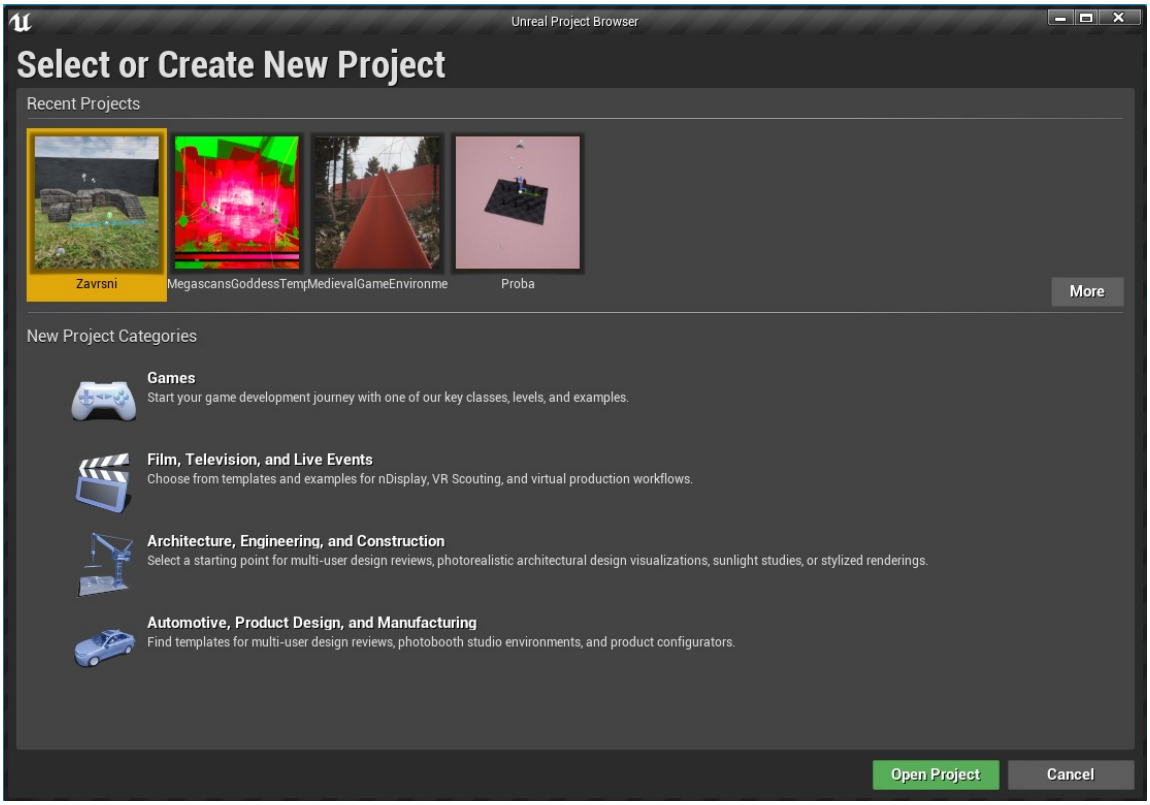
Za bolje shvaćanje toga kakav je alat UE, pogledajmo njegovo sučelje.

U Unreal Editoru scene u kojima se stvara doživljaj igre općenito se nazivaju razinama (engl. *levels*). Razina je 3D okruženje u koju se postavlja niz objekata i geometrija kako bi se definirao svijet koji će igrači doživjeti. Svaki predmet koji se stavi u svijet, bilo da je to svjetlo, mreža ili lik, smatra se Glumcem. Glumac je klasa programiranja koja se koristi u okviru Unreal Engine-a za definiranje objekta koji ima 3D podatke o položaju, rotaciji i mjerilu.

Stvaranje razina počinje postavljanjem objekata na mapu unutar UE-a. Ti predmeti mogu biti geometrija svijeta, svjetla, početne točke igrača, oružja ili vozila.

Budući da je sučelje za Unreal Editor vrlo prilagodljivo, moguće je da se ono može promijeniti od jednog pokretanja programa do drugog. U nastavku možete vidjeti zadani izgled sučelja (slika 6).

Kada korisnik otvori UE, prvi prozor koji se otvori je biranje projekta. Ako korisnik želi napraviti novi projekt, mora izabrati tip projekta.



Slika 6 - biranje projekata

Nakon što korisnik otvori novi ili prijašnji projekt, zadano sučelje koje vidi izgleda ovako (slika 7).

Slika 7 - sučelje UE alata

Vidimo brojem označene prozore:

1 Menu Bar (Izbornička traka)

2 Toolbar (Alatna traka)

3 Place Actor / Modes (Postavite glumca / Načini)

4 Viewports (Okviri za pregled)

5 Content Browser (Preglednik sadržaja)

6 World Outliner (Objekti u svijetu)

7 Details (Detalji)

Izbornička traka u uređivaču trebala bi biti poznata svima koji su već koristili Windows aplikacije. Omogućuje pristup općim alatima i naredbama koje se koriste pri radu s razinama u uređivaču.

Alatna traka prikazuje grupu naredbi koje omogućuju brz pristup uobičajenim alatima i operacijama.

Uređivač razine može se staviti u različite načine uređivanja kako bi se omogućilo specijalizirano sučelje za uređivanje i tijekove rada za uređivanje određenih vrsta glumaca ili geometrije.

Okvir za pregled je prozor u svjetove koje korisnik stvara u Unreal Engine-u. Ova ploča sadrži skup okvira za pregled, od kojih se svaki može povećati tako da ispuni cijeli prozor i nudi mogućnost prikaza svijeta s jednog od tri ortografska prikaza (odozgo, sa strane, sprijeda) ili perspektivnog prikaza koji daje potpunu kontrolu nad onim što se vidi i kako se vidi.

Preglednik sadržaja služi za uvid u učitane objekte u projektu, kako bi korisnik što lakše nalazio potrebne objekte za scenu ili razinu.

Objekti u svijetu se lako mogu vidjeti na ovom popisu, te olakšava posao za korisnika koji koristi veliki broj objekata u sceni. Također, omogućuje korisniku da vidi kako bi scena izgledala bez tog objekta bez da ga mora brisati, i to postiže s opcijom *Hide In World*.

Prozor s pojedinostima sadrži informacije, pomoćne programe i funkcije za trenutni odabir u okviru za prikaz. Sadrži okvire za uređivanje transformacija za premještanje, rotiranje i skaliranje glumaca te prikazuje sva svojstva koja se mogu uređivati za odabrane glumce i omogućuje brz pristup dodatnim funkcijama uređivanja ovisno o vrsti glumaca odabranog u prikazu. [16]

3.4. Programi za sjenčanje u Unreal Engine alatu

Kao što smo spomenuli ranije u ovom radu, programi za sjenčanje najčešće se upotrjebljuju za stvaranje zasjenjenih i osvijetljenih područja pri prikazivanju 3D modela. Međutim, model na kojem program za sjenčanje vrši radnju uvijek sadrži neki materijal i teksture koje naknadno korisnik može načiniti svjetlijima, tamnijima, mutnijima itd. uporabom programa za sjenčanje.

Teksture su slike koje se koriste u materijalima, te se spajaju na površine na koje se materijal nanosi. Ovisno o potrebi, teksture se mogu izravno primjenjivati, na primjer, za teksture osnovne boje, ili se vrijednosti piksela teksture koriste unutar materijala kao maske. Uglavnom se teksture stvaraju izvan UE alata u aplikaciji za uređivanje slika, poput Photoshopa, a zatim se uvoze u UE alat putem preglednika sadržaja (engl. *Content Browser*). Jedan materijal može upotrijebiti nekoliko tekstura koje su uzorkovane i primijenjene u različite svrhe. Na primjer, jednostavan materijal može imati teksturu osnovne boje, teksturu refleksije i normalnu mapu. Osim toga, mogu postojati mape emisivnosti i grubоće pohranjene u alfa kanalima jedne ili više ovih tekstura. [17]

3.4.1. PBR teksture

No, uporaba tekstura postala je veoma napredna kroz zadnjih par godina, te cilja što većem realizmu. Jednostavna osnovna boja više nije jedina mapa teksture koja se koristi, već postoji deset različitih mogućih mapa tekstura koje nadodaju efektu PBR materijala. PBR stoji za *Physical Based Rendering*, odnosno renderiranje na fizičkoj osnovi. Koristi se od 1980-ih i razvijeno je za ispisivanje veoma foto realističnih tekstura. PBR koristi točne modele rasvjete za postizanje ovog cilja i polako postaje standard za sve materijale. PBR mape tekstura primjenjuju se na površine 3D modela za stvaranje ponavljajućih uzoraka ili posebnih vizualnih efekata. Oni se mogu koristiti za definiranje specifičnih detalja poput kože, kose, odjeće ili bilo čega drugog. Deset različitih vrsta mapa teksture su:

1 Osnovna boja (engl. *Albedo*)

2 Normalna mapa (engl. *Normal Map*)

3 Grubоća (engl. *Roughness*)

4 Metalnost (engl. *Metalness*)

5 Refleksija (engl. *Specular*)

6 Mapa visine (engl. *Height Map*)

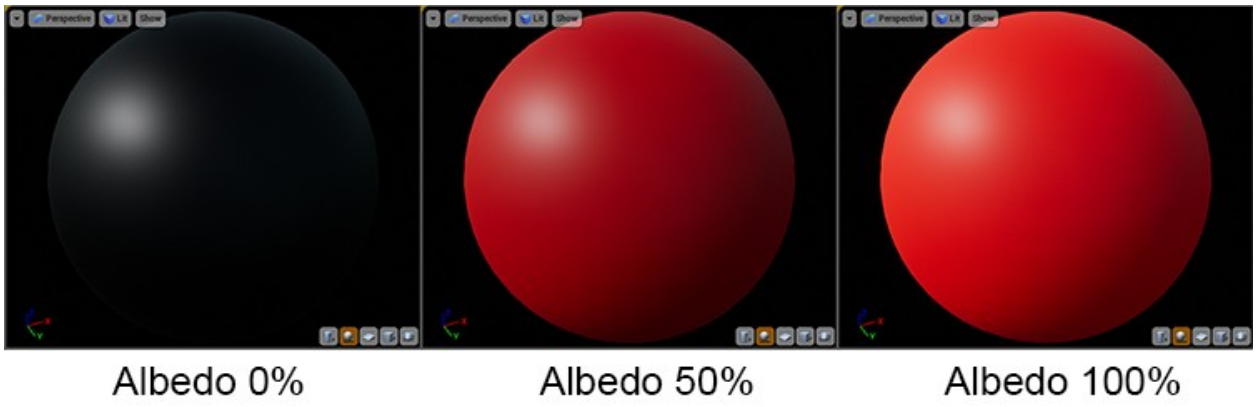
7 Prozirnost (engl. *Opacity*)

8 Ambijentalna okluzija (engl. *Ambient Occlusion*)

9 Prijelom (engl. *Refraction*)

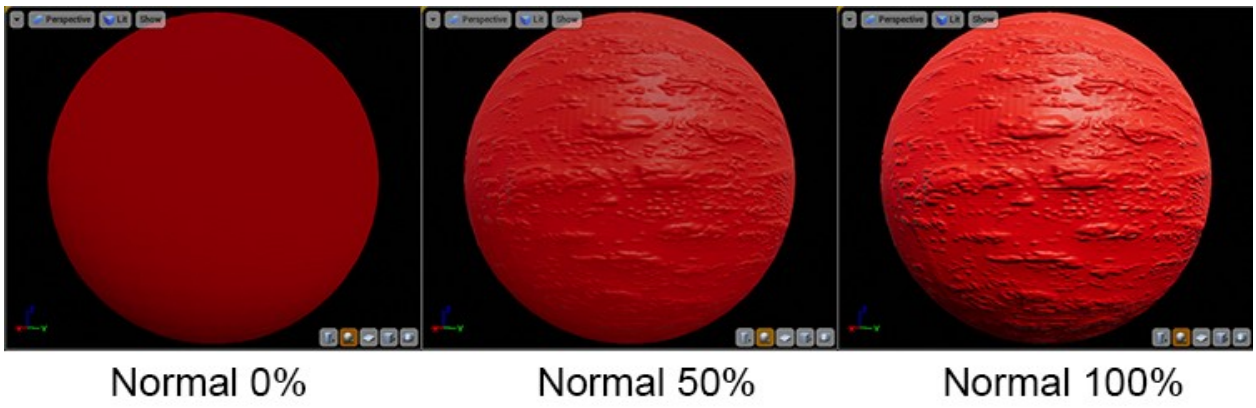
10 Auto-osvjetljenje ili Mapa Emisivnosti (engl. *Self-Illumination* ili *Emissive color*)

Mapa Osnovne boje služi kao baza cijelog materijala. One su ili jednobojne ili ravne svjetlosne slike uzorka s kojim korisnik radi (na primjer cigle). Koriste cijeli RGB (*Red, Green, Blue*) spektar. Pri uporabi ove mape teksture, važno je da je osvjetljenje ravno. Nije poželjno da se prikazuju sjene jer se osvjetljenje u sceni može razlikovati od izvorne fotografije. To će uzrokovati nedosljednost u osvjetljenju teksture i učiniti je nerealnom. Glavna točka ove mape teksture je definiranje boje teksture, no ima i neke sekundarne namjene kao definiranje boje refleksije materijala na metalnim teksturama.



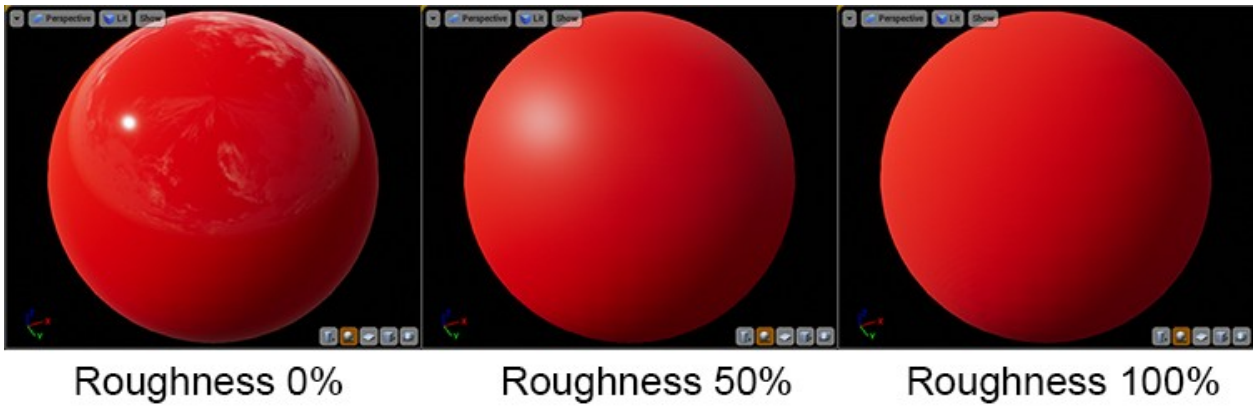
Slika 8 - prikaz utjecaja osnovne boje

Normalne karte važne su za dubinu tekstura. Ova mapa teksture koristi složene izračune za lažiranje načina na koji svjetlo stupa u interakciju s površinom materijala kako bi prividno dale dojam manjih izbočina i udubljenja. Važno je napomenuti da normalna karta neće promijeniti osnovnu geometriju modela na koji se postavlja, već “glumi” da udubljenja postoje. Osnovna boja normalne karte je svijetlo ljubičasta koja predstavlja površinu poligonalne mreže. Odatle se RGB vrijednosti koriste za stvaranje pukotina, izbočina ili pora u modelu. Vrijednosti R, G i B jednake su koordinatama X, Y i Z na osnovnoj mreži.



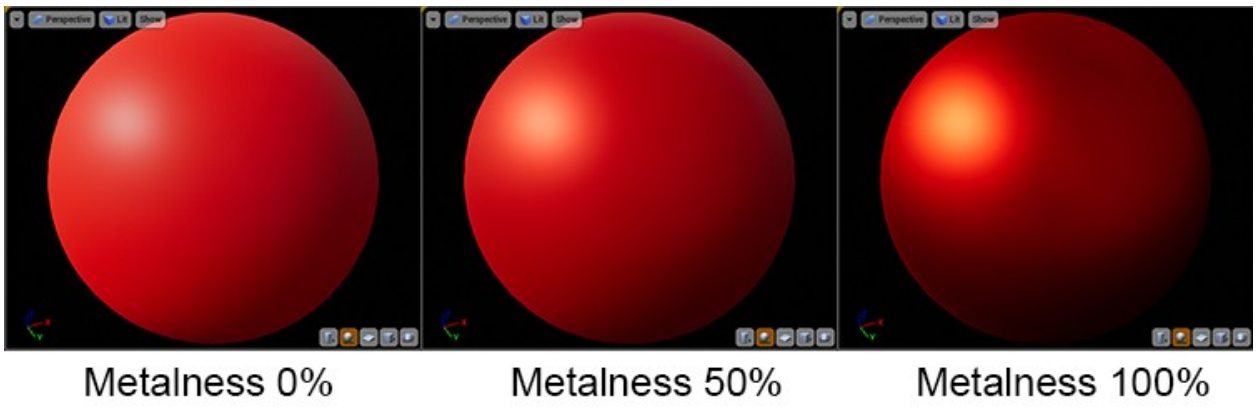
Slika 9 - prikaz utjecaja normalne mape

Grubоća (engl. *Roughness*) definira kako se svjetlost raspršuje po površini modela. Ova mapa teksture počinje s vrijednošću nula pri čemu model uopće neće raspršiti svjetlost, čineći refleksije i osvjetljenje znatno oštrijima i svjetlijima na materijalu. S druge strane, ako se grubоća pojača do kraja, svjetlost će se više raspršiti po materijalu. Zbog toga se osvjetljenje i refleksija šire dalje po modelu, ali izgledaju znatno tamnije. Ove su postavke vrlo važne jer različiti materijali u stvarnom životu imaju vrlo različite hrapavosti. Ove mape su sive boje, s time da je bijela boja maksimalne hrapavosti, a crna glatka i sjajna površina.



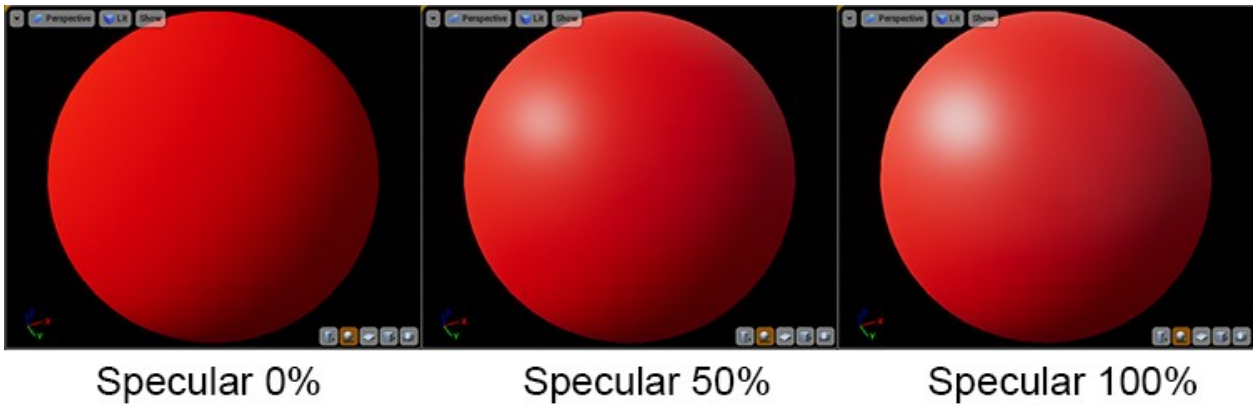
Slika 10 - prikaz utjecaja mape grubоće

Metalnost se koristi za definiranje je li materijal (ili njegov dio) goli metal. Metalne mape su također u sivim tonovima, no crno na mapi metalnosti znači da će dio karte koristiti mapu osnovne boje kao difuznu boju (boja koju tekstura pokazuje kada se pogodi svjetlošću). Bijela će umjesto toga koristiti mapu osnovne boje za definiranje boje i svjetline vaših refleksija i postaviti difuznu boju materijala na crnu. U ovom slučaju difuzna boja više nije potrebna jer će sva boja i detalji tog dijela materijala sada potjecati od refleksija, pa će tako postati crna. Prednost mapa metalnosti je njihova jednostavnost korištenja za simulaciju materijala iz stvarnog svijeta.



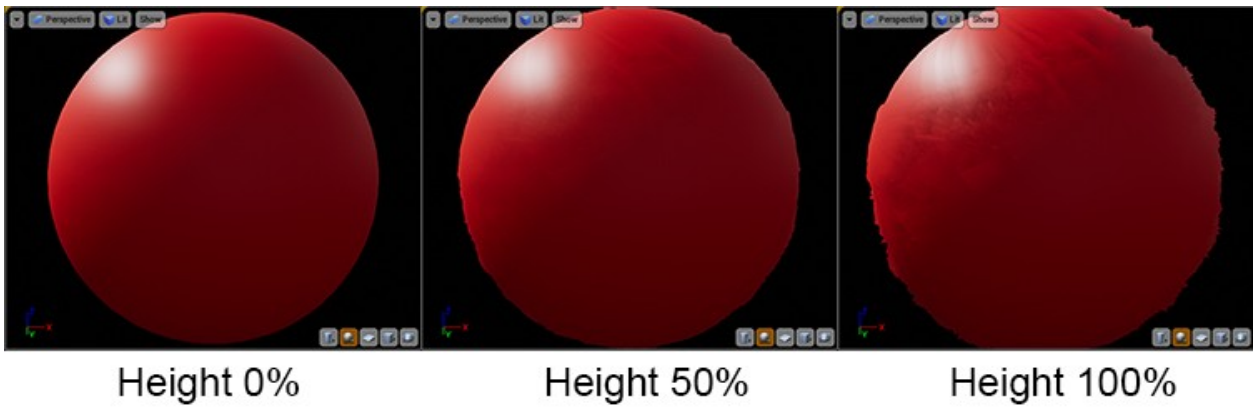
Slika 11 - prikaz utjecaja mape metalnosti

Refleksija se ponekada može koristiti umjesto mape metalnosti. Mape refleksije u PBR-u mogu koristiti punu RGB boju i utjecati na to kako je mapa osnovne boje dizajnirana (ili kako se ona ispisuje iz željenog paketa tekstura). Prednost ove mape je što se ona može koristiti kako bi korisnik utjecao na način na koji se refleksije obrađuju na nemetalnim materijalima, dopuštajući veću fleksibilnost i kontrolu. Nedostatak toga je dodatni sloj složenosti koji ta fleksibilnost dodaje te može biti teže precizno birati željeni rezultat. Mape metalnosti obično izgledaju jednako dobro, ponekad i bolje u određenim uvjetima.



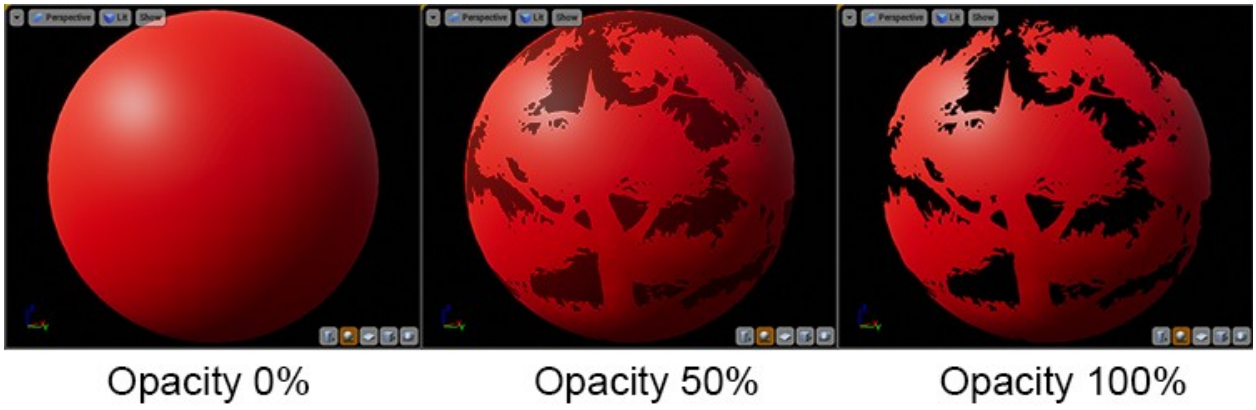
Slika 12 - prikaz utjecaja mape refleksije

Mape visine slične su normalnim mapama jer se koriste za dodavanje manjih detalja u osnovnu mrežu. Velika razlika između ove dvije mape je u tome što će, umjesto da pretvara neravnine i padove poput normalne mape, mapa visine isjeckati mrežu i zapravo dodati podatke u 3D mrežu. Mape visina su još jedne od mapa u sivim tonovima s crnom bojom koja predstavlja dno mreže, a čista bijela najviše vrhove, s nijansama sive koje predstavljaju sve između. Prednost visinskih mapa je velik detalj koji dodaju i koji izgleda ispravno pod svim kutovima i uvjetima osvjetljenja. No, one mogu uzrokovati usporavanje igara ili vrijeme iscrtavanja, te se iz tog razloga preferiraju normalne mape.



Slika 13 - prikaz utjecaja mape visine

Prozirnost je važna vrsta mape jer omogućuje da dijelove materijala učini transparentnim. Ovo je važno ako se izrađuje staklo ili efekt dima. Mape prozirnosti su sive boje. Bijela je potpuno neprozirna, a crna je prozirna. Nijanse sive različite su razine prozirnosti među njima. Ako je materijal samo čisto staklo ili drugi potpuno proziran materijal, tada bi trebalo umjesto ove mape koristiti samo konstantnu vrijednost. 0.0 je transparentna, a 1.0 neprozirna.

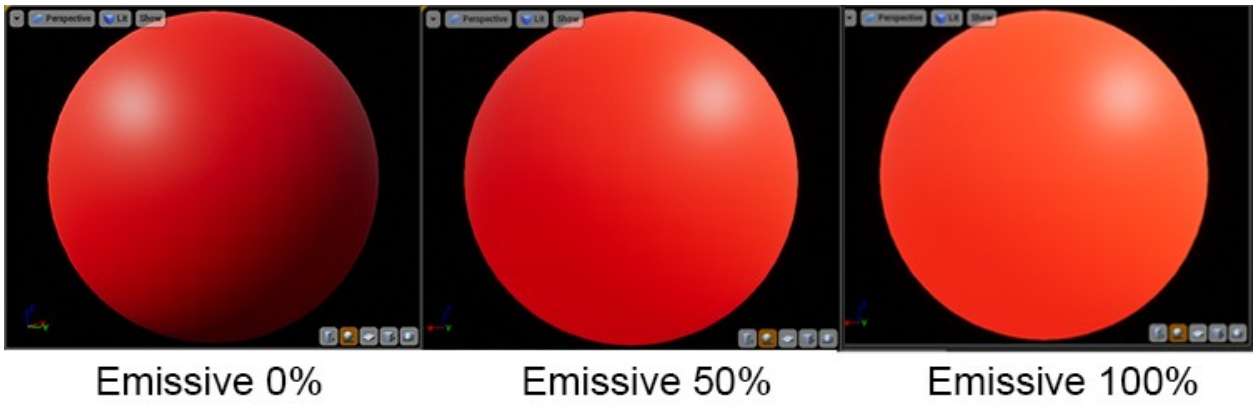


Slika 14 - prikaz utjecaja mape prozirnosti

Ambijentalna okluzija je karta koju PBR alat kombinira s mapom osnovne boje u vrijeme iscrtavanja kako bi definirao kako reagira na svjetlost. To je karta u sivim tonovima s bijelom bojom koja će pokupiti najviše svjetla, a tamnija područja su više u sjeni i manje reagiraju na svjetlo.

Prijelom je proces savijanja svjetlosti pri prolasku kroz krutu tvar, tekućinu ili plin, što narušava način na koji stvari izgledaju kada ih gledamo kroz prozirni objekt. To je efekt koji omogućuje rad povećala i čini da stvari izgledaju drugačije kad se gledaju pod vodom. Ova mapa je važan dio tijeka rada s materijalima jer svi prozirni materijali u stvarnom životu uzrokuju lom svijetla, pa ga je potrebno replicirati kako bi u grafici radio što realnije. Mape prijeloma obično su samo konstantne vrijednosti.

Auto-osvjetljivanje (ponekad se naziva i mapa emisivnosti) se koristi za to da neki dijelovi materijala naizgled emitiraju vlastitu svjetlost pa su i dalje vidljivi u tamnim područjima. Auto-osvjetljivanje je korisno za uključivanje malih LED dioda ili za neke zanimljive svjetlosne efekte, ali ako se koristi previše, dovodi do potpunog ispiranja detalja i uklanja realističnost iz scene. Ove mape koriste RGB spektar. Iako je moguće osvijetliti cijelu scenu pomoću mape za auto-osvjetljivanje, to je loša praksa i puno je teže od dodavanja konvencionalne rasvjete. [17]–[19]



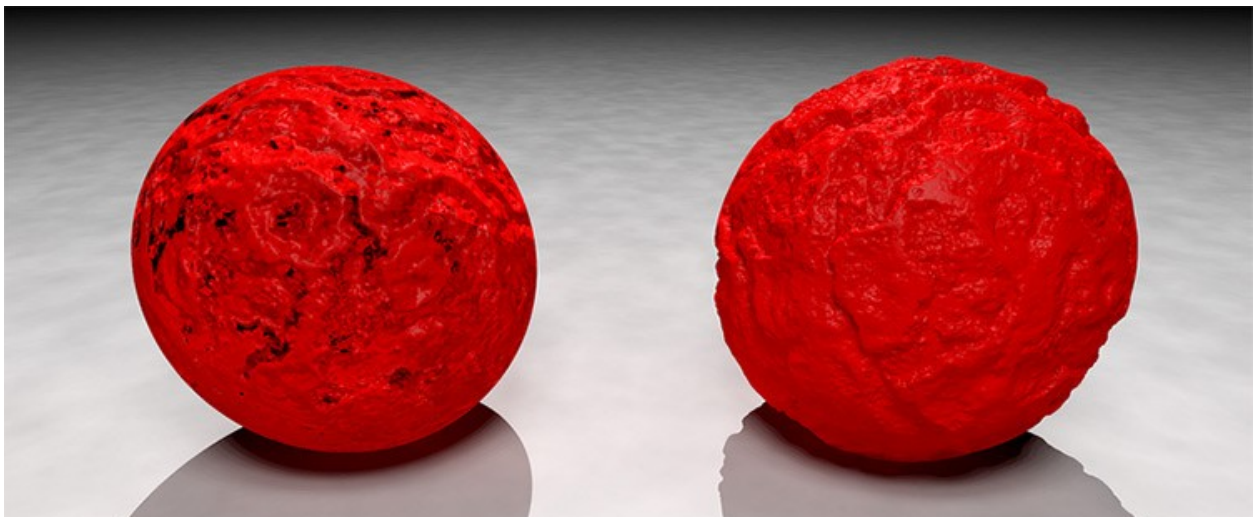
Slika 15 - prikaz utjecaja mape auto-osvjetljivanja

3.4.2. Ne-PBR teksture

Radnje u radnim tijekovima s ne-PBR teksturama nisu standardizirane u 3D programima. Postoji par radnji koje se pojavljuju često, te se ovi primjeri pojavljuju se u Autodesku 3ds Max, ali se odnose i na neke druge 3D softvere.

Difuzne mape (engl. *Diffuse*) su ne-PBR ekvivalent mape osnovnih boja. One definiraju boju materijala kako svjetlost pada na njega. Glavna razlika je u tome što se difuzne mape obično ne izrađuju s ravnim svjetlosnim profilom. Njen rezultat se može objasniti kao mapa osnovne boje i mapa ambijentalne okluzije spojena u jednu sliku, jer su informacije o sjeni dio difuzne karte.

Mape neravnosti (engl. *Bump Map*) su temeljniji oblik normalne mape. Dok normalna mapa koristi puni RGB spektar za približavanje sve 3 dimenzije prostora, mape neravnosti su u sivim tonovima koje rade samo u smjeru gore ili dolje. Kako se karta obavija oko osnovne mreže, može učiniti da se neravnine pojavljuju u svim smjerovima mreže. Rezultati nisu točni kao kod normalne mape, pa se stoga mape neravnosti prestaju koristiti u industriji.



Slika 16 - prikaz efekta mape neravnosti

Zbog nedostatka postavki metalnosti, u ne-PBR radnim tijekovima koriste se mape Odraza. Obično su to konstantne vrijednosti slične mapama prijeloma u PBR-u, a njihovu boju i intenzitet kontrolira mapa refleksije. [17]–[19]

4. Optimizacija programa za sjenčanje

4.1. Zašto provoditi optimizaciju programa za sjenčanje

Razlog za optimizaciju programa za sjenčanje je što je ponekad brzina kadra (engl. *frame rate*) jako niska. Niski broj kadrova u sekundi (engl. *frame-per-second* ili kraće *FPS*) je primjetan ako se igra naizgled sporo ponaša. Ako je broj kadrova dovoljno nizak, igranje igre može se činiti kao da se prikazuje dijaprojekcija jer se samo nekoliko različitih kadrova pojavljuje na ekranu u sekundi.

U manje ekstremnim slučajevima igra će dati osjećaj trzavosti i sporine – upravo suprotno od glatkosti, kako bi i trebalo biti.

Problemi s brojem kadrova u sekundi nisu uzrokovani problemima s mrežom. Ako postoji niska brzina kadrova, to nije igra koja zaostaje - to je računalo koje ne uspijeva pratiti igru. Korisniku će možda trebati brža grafička kartica, više RAM-a (skraćeno za engl. *Random Access Memory*) ili bolji CPU. Također, tvrdi disk može biti prespor, što dovodi do usporavanja igre jer je prisiljen čitati podatke s tvrdog diska, ili možda u pozadini radi previše softvera koji se natječu za resurse.

Drugim riječima, nizak FPS problem je u performansama igre na računalu. A u problemu performiranja igre na računalu veliku ulogu imaju programi za sjenčanje, dajući veliku potrebu za optimizacijskim metodama. [20]

4.2. Metode mjerenja učinkovitosti programa za sjenčanje

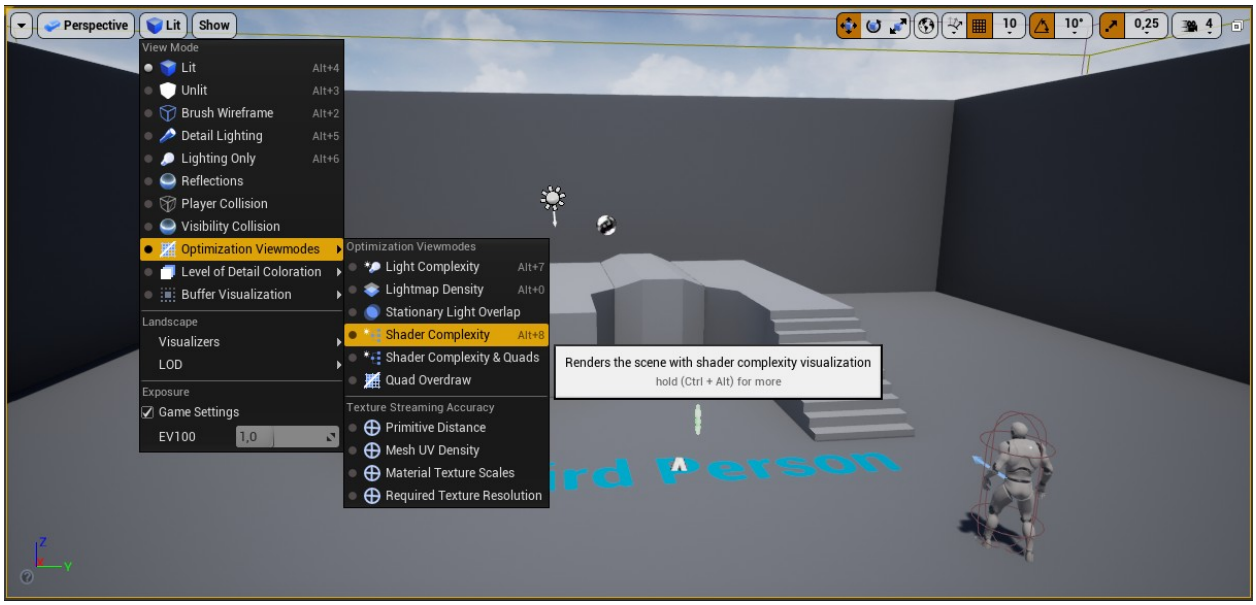
Kako bi smo znali koje tehnike provesti i na koji način optimizirati programe za sjenčanje, prvi korak koji moramo napraviti je izmjeriti koliko su ti programi uopće učinkoviti i koliko se dobro provode.

Postoji više načina kako mjeriti učinkovitost programa za sjenčanje. S obzirom na potrebe korisnika svaka metoda se može koristiti pojedinačno, no u praksi bi najtočnije očitanje bilo provesti sve metode kako bi se korisnik uvjerio da je stanje jednako u svim metodama. [21]

4.2.1. Način prikaza složenosti programa za sjenčanje

Prva metoda je kroz *način prikaza složenosti programa za sjenčanje*. Ovoj opciji korisnik može pristupiti na način da se klikne na gumb *Lit*, zatim *Optimization Viewmodes* te nakon toga *Shader*

Complexity (slika 17).

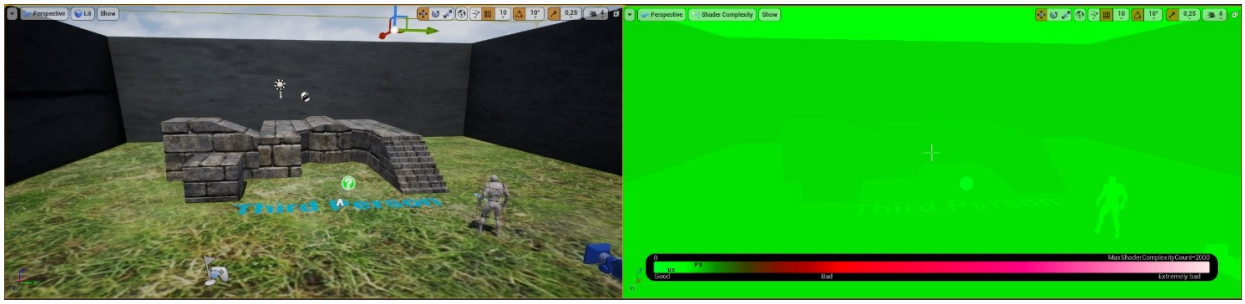


Slika 17 - prikaz opcije u UE alatu

Ova opcija na ekranu pokaže scenu u obojanom stanju. Scena će biti obojana u boju ovisno o programu za sjenčanje; na dnu ekrana može se vidjeti skala od svijetle zelene boje do bijele boje. Boje označavaju koliko je objektov program za sjenčanje kompliciran i težak za provođenje. Što je više objekata na sceni obojeno zeleno, to su programi za sjenčanje manje zahtjevni. Objekti obojeni u kričavo crveno sve do bijele boje, to su programi zahtjevniji i valjalo bi ih optimizirati.

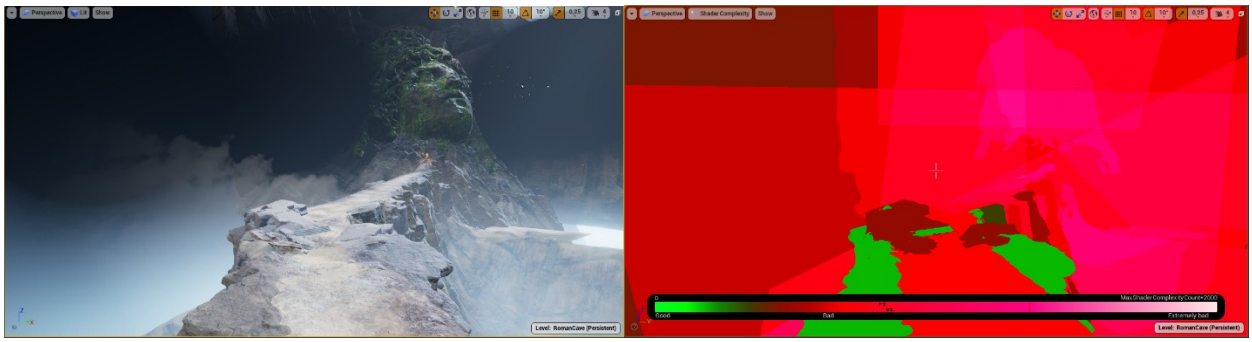
Slika 18 - zahtjevnost programa za sjenčanje u sceni bez tekstura

U našem slučaju, naša scena se sastoji od objekata koji još nemaju teksture, te iz tog razloga je sve zeleno i optimalno. No, kada bismo dodali neke početničke teksture u scenu, izgled i rezultat ovog mjerenja bio bi sitno drugačiji (slika 19).



Slika 19 – Prikaz jednostavne scene sa teksturama. Lijevo: izgled scene, Desno: zahtjevnost programa za sjenčanje.

S obzirom da su dodane teksture iz UE-ovog početničkog paketa objekata, programi za sjenčanje su sitno zahtjevni te razlika sa ili bez tekstura nije velika. Ako bismo učitali već gotovu scenu koja je besplatno dostupna na Epic Games tržištu, vidjeli bismo veliku razliku (slika 20).



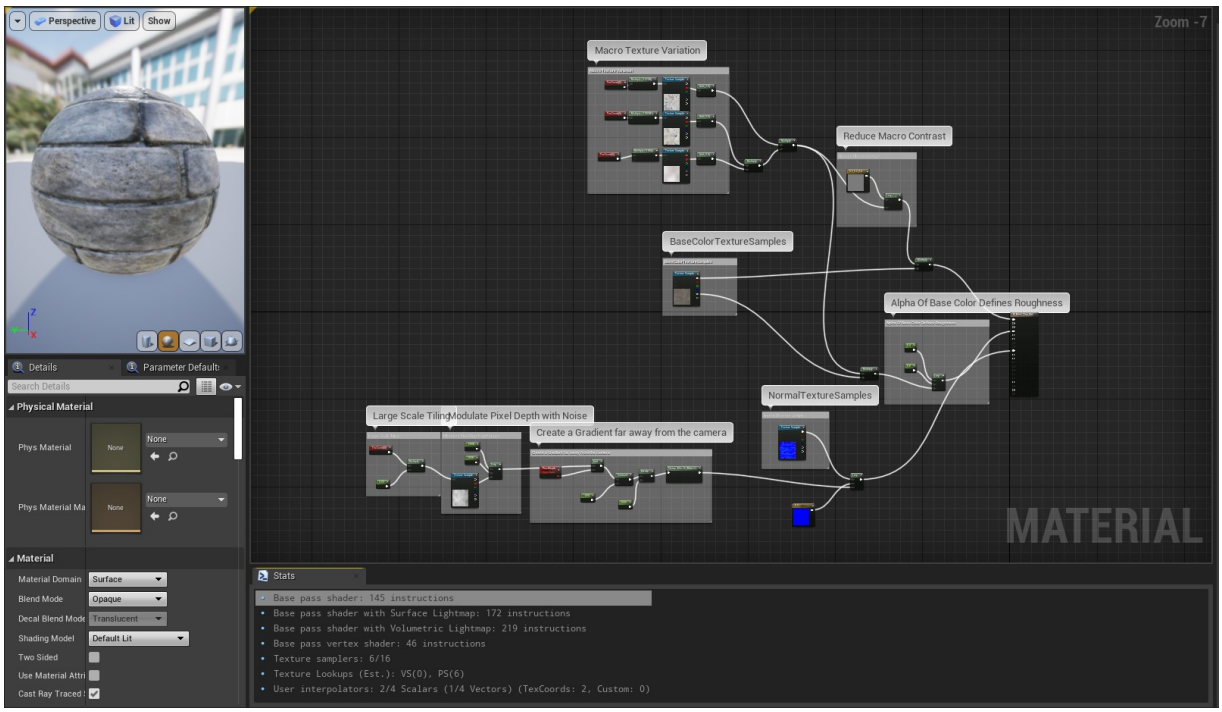
Slika 20 - Prikaz zahtjevne scene. Lijevo: izgled scene, Desno: zahtjevnost programa za sjenčanje.

U ovoj sceni programi za sjenčanje nisu optimizirani te su jako zahtjevni, što se može pročitati iz ružičaste boje koja se nazire na sredini scene, označavajući jako lošu optimizaciju.

Međutim, ova metoda nije savršena iz razloga što daje samo generalnu ideju toga koji program za sjenčanje može biti teži za provođenje te se uvelike oslanja na broj instrukcija koje mora izvršiti svaki program za sjenčanje. Uz to, ne daje detalje o tome što konkretno stvara problem u programu za sjenčanje da bismo ga znali optimizirati. [21]

4.2.2. Broj instrukcija

Sljedeća metoda kojom se može mjeriti koliko je program za sjenčanje zahtjevan je *metoda broja instrukcija*. Sjetimo se velike prednosti kod alata UE, a to su *Blueprints*, vizualni sistem programiranja. Ovaj sistem također funkcionira sa teksturama i programima za sjenčanje. Ako želimo pobliže promatrati neku teksturu, alat UE ju otvori u zasebnom prozoru i pokaže detalje o toj teksturi (slika 21).

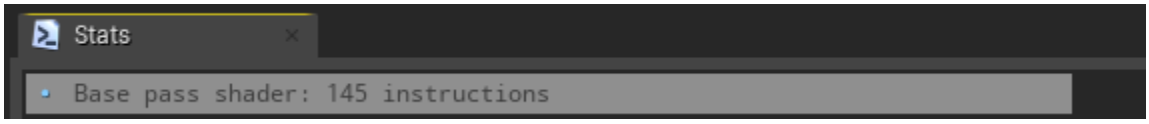


Slika 21 - prikaz grafa programa za sjenčanje

Ovdje vidimo graf sa čvorovima koji sadrže instrukcije rada za programe za sjenčanje. Kada složimo svoj graf sa čvorovima on se kompilira u HLSL kod, nakon čega se taj kod kompilira u asemblerske instrukcije, te se te instrukcije šalju grafičkom upravljačkom programu i grafičkoj kartici.

Dakle, ako znamo koliko asemblerskih instrukcija naš program za sjenčanje sadrži, možemo ocijeniti njegovu zahtjevnost i jednostavnost provođenja. To nas vodi do zaključka da, ako bismo uspjeli smanjiti broj instrukcija u programu za sjenčanje, dobili bismo bolje optimiziran program za sjenčanje.

U našem slučaju, možemo vidjeti da ova konkretna tekstura sadrži 145 instrukcija (slika 22).



Slika 22 - brojač asemblerskih instrukcija programa za sjenčanje

Nažalost, ni ovo nije savršena metoda za ocjenu zahtjevnosti iz razloga što sve instrukcije ne zahtijevaju jednako vremena za izvođenje na GPU-u, te nam to ne garantira brže provođenje programa za sjenčanje. Isto tako, svaka platforma zahtjeva različiti način kompiliranja, što može rezultirati u različitom broju instrukcija na svakoj platformi. [21]

4.2.3. Testiranje projekta na različitim platformama

Najtočnija metoda za mjerenje optimalnosti programa za sjenčanje je testirati projekt na različitim platformama. Iako je najtočnija, ova metoda je i najteža za koristiti jer zahtjeva puno pripreme. Potrebno je postaviti program za sjenčanje na objekt, objekt postaviti u scenu u kojoj se program za sjenčanje koristi, izvesti projekt kao gotovu igru te nakon toga testirati na raznim platformama.

Prednost koju nudi ova metoda je ta da svaka platforma ima drugačije karakteristike za provođenje programa za sjenčanje. Na taj način možemo dobiti precizno očitanje sa svake platforme za isti program za sjenčanje u istoj sceni, sa istim akcijama i postavkama, i vidjeti koliko je program za sjenčanje optimalan.

Mana ove metode uz dugotrajnu pripremu je ta što nema svaki korisnik pristup platformama na kojima želi izvesti svoj projekt. Zato bi uvijek bilo optimalno prvo pripremiti neki način na koji će se projekt testirati na drugim platformama prije nego što se projekt finalizira. [6], [22]

4.3. Metode optimizacije programa za sjenčanje

Nakon što smo utvrdili koji program za sjenčanje stvara poteškoće i koliko, sljedeći korak je optimizirati taj program. Ovisno o problemu koji postoji, birati će se jedna od metoda optimizacije.

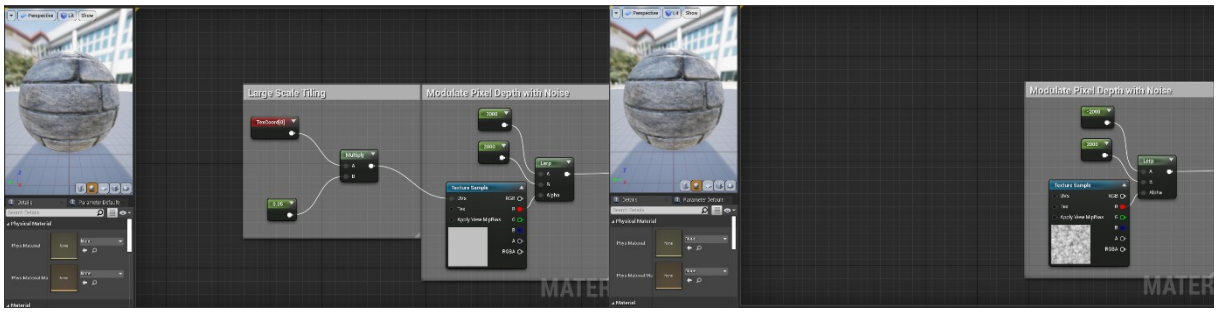
4.3.1. Rješavanje nepotrebnih karakteristika

Kao vrlo jednostavna metoda, program za sjenčanje može postati optimalniji na način da se izbrišu pojedine karakteristike iz njega koje su nepotrebne i zauzimaju određen broj instrukcija koje ne služe ničemu. Na primjer, ako u grafu s čvorovima programa za sjenčanje postoje opcije poput iscrtavanja crnih uzoraka koji bi bez te opcije i dalje bili crni, ta opcija je beskorisna i može se maknuti.

Kao još jedan primjer možemo pogledati jedan od početnih programa za sjenčanje koji dolazi sa UE besplatnim paketom na početku (slika 23).

Na slici možemo vidjeti kako program za sjenčanje ima posebni skup naredbi za *Large scale tiling*. Opcija *TexCoord* stoji skraćeno za *TextureCoordinate*, odnosno koordinate teksture. Ona daje koordinate UV teksture u obliku dvokanalne vektorske vrijednosti, s kojima daje mogućnost mijenjanja skale pločica koje će se pokazivati na sceni te koliko će veliko pločica izgledati na pojedinom objektu. Ovaj skup opcija će jedino biti nužnost ako ovu teksturu stavljamo na objekt velike površine. Broj 0.05 stoji kao konstanta kojom će se množiti vrijednosti koordinata u slučaju

da su velike, kako bi manipulacija položaja teksture bila što lakša.



Slika 23 - prikaz brisanja nepotrebnih karakteristika

No, s obzirom na naš slučaj, ovu teksturu koristimo na malim objektima, te nam je taj skup opcija nepotreban.

Kada bismo izbrisali taj skup naredbi za skaliranje, ne bismo vidjeli nikakvu razliku (slika 24).

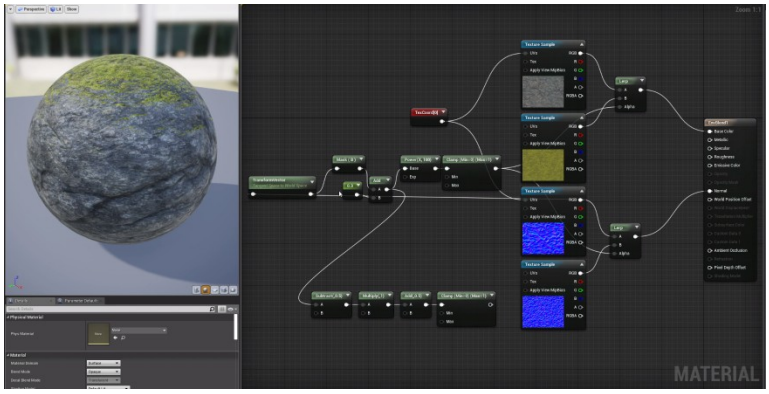
Slika 24 - usporedba teksture prije i poslije brisanja nepotrebne karakteristike

Česti su slučajevi gdje korisnik koristi vrlo napredan program za sjenčanje, no od svih mogućih iskoristivih opcija korisnik ugasi nekolicinu, te mu ostane samo par opcija aktivno. U tom slučaju program za sjenčanje je i dalje opterećen svim opcijama, bez obzira što se u toj sceni ne koriste. Puno optimalnije bi bilo izbrisati neiskorištene opcije kako bi se zahtjevnost programa za sjenčanje smanjila, čak i kada su te opcije ugašene u sceni. [6], [22]

4.3.2. Prerada matematičkih formula

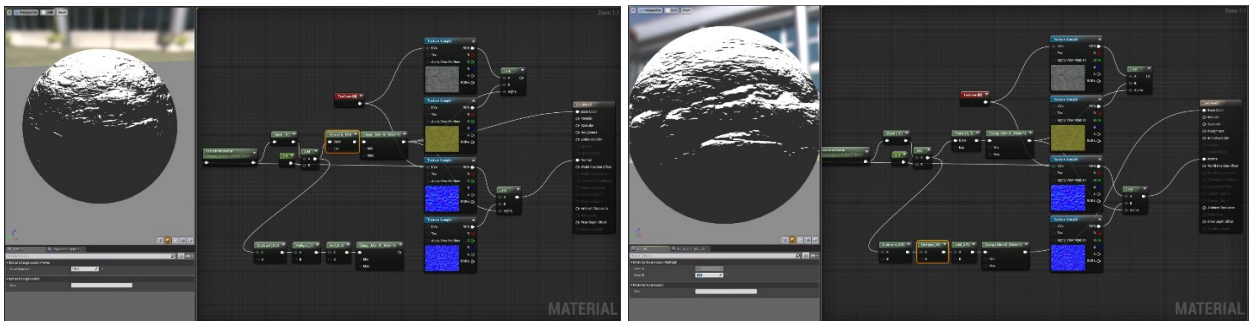
Ponekada se u programima za sjenčanje koriste matematičke operacije za jednostavniju manipulaciju teksturom. Ono što korisnici najčešće ne uzimaju u obzir je da neke matematičke funkcije zahtijevaju više snage nego ostale. Najbolja opcija kod rada sa matematičkim formulama je probati postići svoj cilj sa što manje zahtjevnim operacijama, npr. zbrajanje, oduzimanje, množenje i dijeljenje. Sve operacije iznad toga su zahtjevnije i troše više vremena.

Uzmimo za primjer ovaj program za sjenčanje (slika 25).



Slika 25 - primjer programa za sjenčanje koji koristi kompleksnu matematičku funkciju

Ovdje imamo program za sjenčanje koji imitira realističnu teksturu kamena sa mahovinom u prirodi. Ovaj efekt postignut je na način da se koriste maske. Maske stvaraju gradijent vrijednosti u rangu između 0 i 1, gdje 0 predstavlja kamen (crna boja), a 1 mahovinu (bijela boja).



Slika 26 - prerada matematičke funkcije i njen utjecaj na masku. Lijevo: potenciranje, Desno: prerađena funkcija.

S obzirom na to koju vrijednost maske proizvedu, dalje se koristi matematička funkcija potenciranja kako bi se odredilo gdje će se prikazivati kamen, a gdje mahovina (slika 26). Što je vrijednost bliža nuli, to će biti više kamena, te isto vrijedi za jedinicu i mahovinu. Ikakva vrijednost između nule i jedinice proizvodi gradijentni prijelaz. U prirodi se neće dogoditi lagani prijelaz iz kamena u mahovinu, nego oštar prijelaz bez gradijenata. Kako bismo to osigurali, moramo imati što više vrijednosti blizu nule i blizu jedinice, a što manje vrijednosti koje teže sredini.

Međutim, sama operacija potenciranja je poprilično zahtjevna, te je potrebno više vremena da se izvede. Uz to, ne pruža dobar način za manipulaciju omjera koliko se mahovine nalazi na vrhu kamena. Ako bismo smanjili vrijednost potencije sa 100 na 1, dobili bismo ovakav slučaj (slika 27).

Slika 27 - prikaz neželjenog utjecaja uporabe funkcije potenciranja

Ne samo da postoji veliki gradijentni prijelaz između mahovine i kamena, već nam se pozicija mahovine proširila po cijelom objektu, što nama nije poželjno. Ovo se može izbjeći na način da se iskoristi novi, manje zahtjevan set matematičkih funkcija. S obzirom da maske proizvode svoje vrijednosti u rasponu od 0 do 1, možemo staviti operaciju oduzimanja 0.5 sa te početne vrijednosti, pomnožiti ju s nekim cijelim brojem, te vratiti tih oduzetih 0.5. Ovime je osigurano da, koji god broj dobili natrag, biti će ili ispod nule ili iznad jedinice. Maske samo gledaju kojem broju je vrijednost blizu, što znači da se svi minusi računaju kao nula, i svi brojevi iznad jedinice kao jedinica. Također, dodatna prednost ovakvog pristupa je u tome što ne postoji pomak maske, dok veće potencije lokaliziraju bijeli dio maske na vrhu teksture.

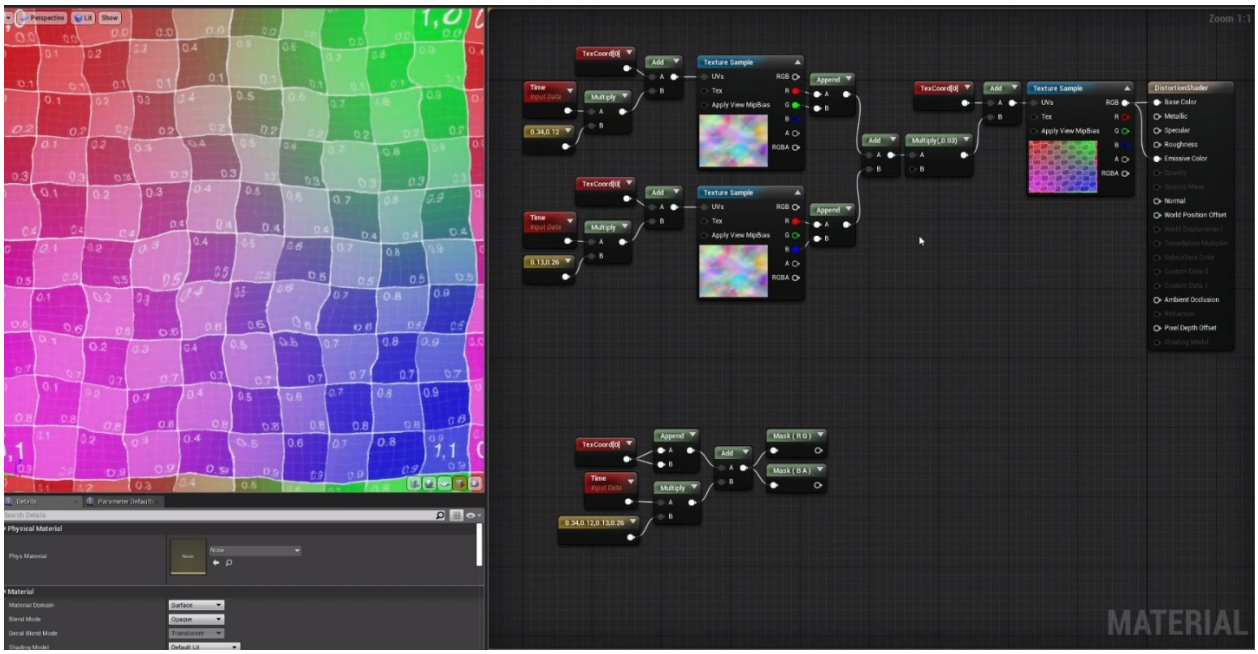
Kao što je vidljivo na gornjoj slici (slika 22), rezultat je dobar i precizan. Ako pogledamo pobliže broj instrukcija, nakon ovih operacija imamo manji broj instrukcija nego prije, bez obzira što su tri odvojene matematičke funkcije. Ovime možemo zaključiti da smo zaista optimizirali program za sjenčanje preradom matematičkih funkcija iz više zahtjevnih u jednostavne. [6], [22]

4.3.3. Kombiniranje komponenata, manje matematike

Metoda kombiniranja komponenata služi za postići rezultat sa manje matematike i varijabli, a s tim i manje zahtjevnosti. Ako postoji način da identičan segment naredbi pojednostavimo mijenjanjem tipa varijabli, program za sjenčanje je uredniji i jednostavniji.

Za ovu metodu optimizacije programa za sjenčanje možemo koristiti primjer na sljedećoj slici (slika

28). Ovdje možemo vidjeti program za sjenčanje koji predstavlja distorziju. Sadrži se od dvije teksture koje koriste komponentu vizualni šum, odnosno *noise*. Taj šum postoji kako bi distorzirao UV koordinate programa za sjenčanje, te se sve na kraju spaja u treću teksturu.



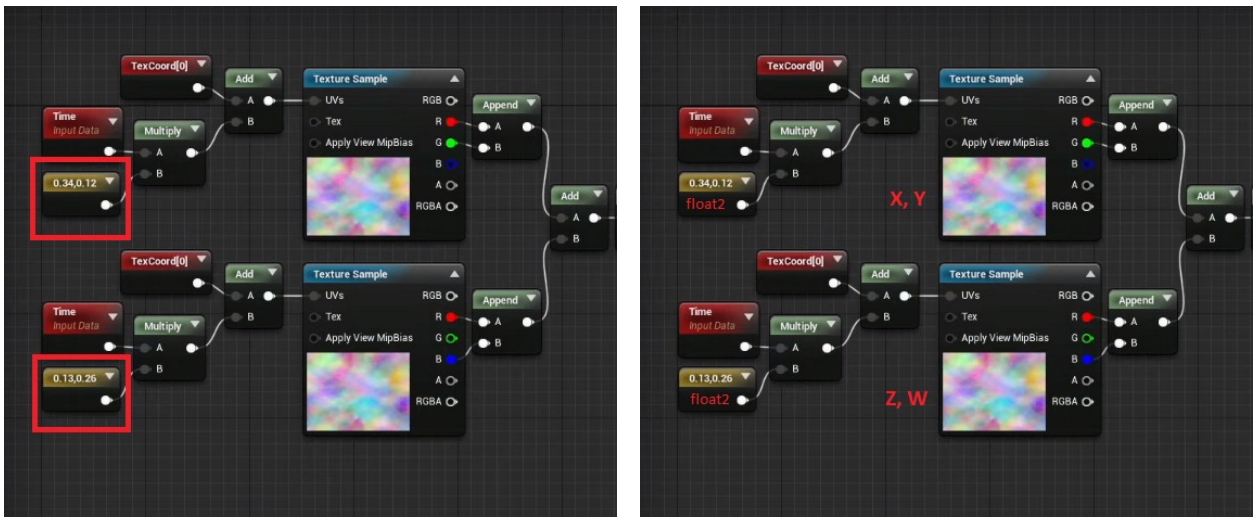
Slika 28 - primjer programa za sjenčanje sa ponavljajućim matematičkim izrazima

Bitno je za primijetiti da je početni matematički izraz dva dijela ovog programa za sjenčanje duplikat. Jedina razlika su konstante koje ulaze u teksture. Dakle, unosimo različite vrijednosti, no sve ostale karakteristike su iste.

Korisniku koji cilja na optimizaciju svog programa za sjenčanje ovakav slučaj je znak za to da postoji način za ovo pojednostaviti kako bi smanjio zahtjevnost programa za sjenčanje.

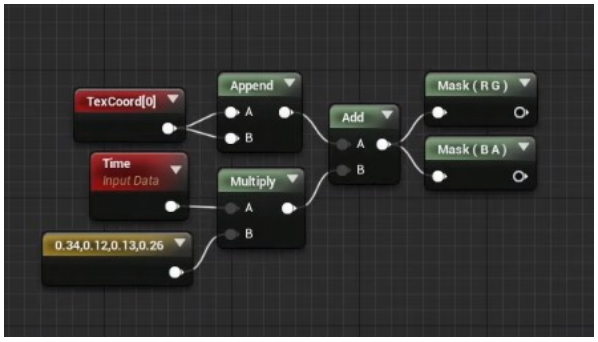
Najviše računski zahtjevan dio u ovom primjeru su matematičke funkcije množenja i zbrajanja, odnosno *multiply* i *add*. Neovisno o tome što su te funkcije malo zahtjevne, s obzirom na cijeli program za sjenčanje troše najviše snage. Cilj optimizacije je što više smanjiti potrošnju memorije. U ovom slučaju to možemo napraviti na način da kombiniramo te dvije teksture u jednu te promijenimo tip varijable kojim unosimo svoje vrijednosti konstanti.

Trenutno se konstante pohranjuju u tipu float2. Na početku programa za sjenčanje vrijednosti gornje linije naredbi pohranjuju se u X i Y kanal UV koordinata, a donje u Z i W (slika 29).



Slika 29 - prikaz korištenih float2 varijabli

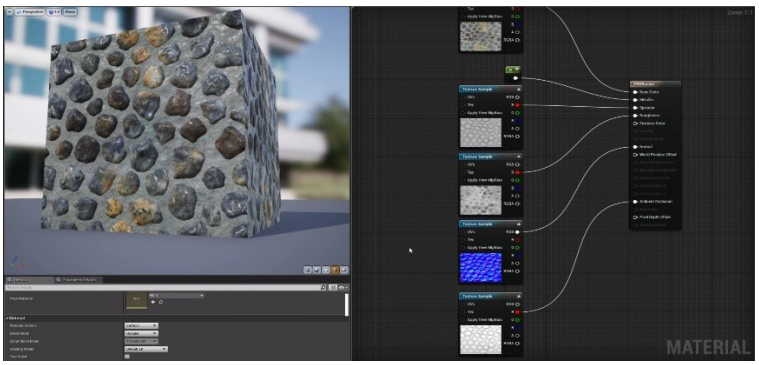
Način na koji ovo možemo promijeniti je stvoriti novu varijablu tipa float4 gdje ćemo pohraniti sve vrijednosti koje su originalno spremljene u dvije različite float2 varijable. Kako je potrebno imitirati prijašnju dodjelu varijabli određenim koordinatama, postoji funkcija *AppendVector*, koja omogućuje kombiniranje kanala za stvaranje vektora s više kanala od originalnog. Dakle, korištenjem te funkcije možemo iste vrijednosti dodijeliti istim koordinatama kao prije na puno uredniji i manje zahtjevan način. Na kraju, možemo razdvojiti te kanale u dvije maske koje nam predstavljaju teksture s kojima smo počeli na početku, no na puno kompaktniji način (slika 30). [6], [22]



Slika 30 - prikaz rješenja problema ponavljajućih izraza

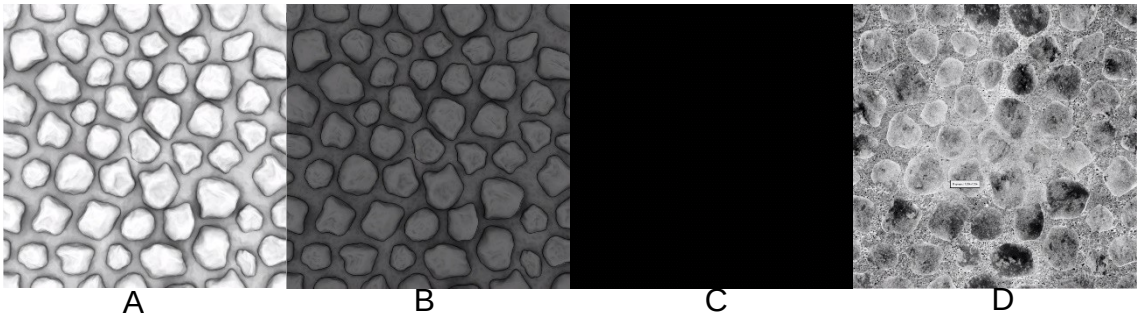
4.3.4. Pakiranje kanala teksture

Primjerak teksture je jedna od najzahtjevnijih karakteristika programa za sjenčanje. Kada god postoji više primjeraka teksture koji koriste isti UV kanal, najisplativije je komprimirati ih sve zajedno u jednu teksturu da postoji što manje instrukcija u asemblerskom kodu. Na sljedećoj slici možemo vidjeti primjer jednog takvog programa za sjenčanje koji koristi više primjeraka teksture (slika 31). Redom; boja, metalnost, refleksija, grubоća, mapiranje teksture i ambijentalna okluzija.



Slika 31 - prikaz programa za sjenčanje s više ulaznih tekstura

Koristimo pet različitih primjeraka teksture, te time radimo 102 instrukcije asemblerskom kodu. To je poprilično malo instrukcija, ali može biti i manje. Ono što možemo napraviti je koristiti neki vanjski alat za manipulaciju slikama, npr. Photoshop, kako bismo spojili sve primjerke teksture zajedno u jednu teksturu. Naime, *Physically Based Rendering* teksture (kraće PBR teksture) koriste mnogo crno-bijelih slika, što možemo vidjeti i u našem slučaju (slika 32). Unutar alata poput Photoshopa vrlo je jednostavno umetnuti određene primjerke tekstura u R, G, B kanale (Red, Green, Blue). S obzirom na poziciju primjeraka u kanalu, alat za stvaranje igara može koristiti određeni kanal kao primjerak teksture.

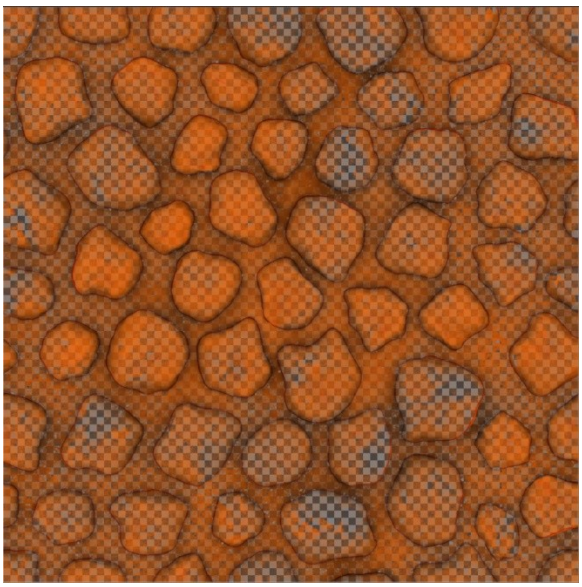


Slika 32 - teksture u sivom spektru. A: okluzija, B: refleksija, C: metalnost, D: grubоća

Pogledajmo kako to izgleda na ovom primjeru. Umjesto četiri različitih primjeraka teksture, napravili smo komprimiranje u jednu teksturu sa R, G, B kanalima. Ako pogledamo samo crveni kanal, vidjeti ćemo primjerak ambijentalne okluzije.

Ako pogledamo samo zeleni kanal, vidjeti ćemo primjerak refleksije. Ako pogledamo samo plavi kanal, vidjeti ćemo metalnost. Naime, na kamenju nema metala, dakle metalnost je postavljena na vrijednost 0, te izlazi kao potpuno crna slika. Ako pogledamo samo Alpha kanal (predstavlja informacije o transparentnosti po pikselu), vidjeti ćemo primjerak grubоće.

Ovime smo postigli spajanje četiri primjerka teksture u jednu. Ovisno o kanalu gdje je koji primjerak teksture postavljen, spajamo odgovarajuće primjerke na odgovarajuće pozicije UV kanala (slika 33).



Slika 33 - prikaz rezultata pakiranja tekstura u RGB kanale

Tekstura izgleda jednako kao što je izgledala i prije ove akcije, a naš cilj je postignut; program za sjenčanje sada javlja da imamo 100 instrukcija u asemblerskom kodu, za razliku od prije gdje smo imali 102 instrukcije.

Iako je mali napredak, pri stvaranju igre koristi se mnogo različitih programa za sjenčanje i tekstura. Kad bismo dozvolili da svaki od tih programa ima dvije instrukcije previše zbog manjka optimizacije, zahtjevnost igre bi eksponencijalno narasla. Taj efekt bi dodatno pojačala činjenica da je učitavanje tekstura vremenski najzahtjevniji zadatak računalu. Kako bismo to izbjegli, ovo je jednostavan način za osigurati se da su programi za sjenčanje što manje zahtjevni. [6], [22]

5. Rezultati

Cilj ovog rada je prikazati optimizacijske metode na neoptimiziranoj sceni te optimizirati programe za sjenčanje kako bi ta scena postala manje zahtjevna. Pomoću 3 optimizacijske metode spomenute ranije u radu optimiziramo sljedeću scenu. Radi načina na koji su programi za sjenčanje napravljeni u ovoj sceni nije bilo potrebe za korištenjem metode kombiniranja komponenata.

Uzeta scena, iako je napravljena za video igru, nije optimizirana. Naime, ova scena koristi tehnike koje se inače koriste za Computer Generated Imagery (kraće CGI). Tolika razina optimizacije nije potrebna jer sami modeli i teksture koje video igra koristi nisu dovoljno detaljni da bi ta razina optimizacije bila uočljiva. Takav način optimizacije može samo preopteretiti scenu i stvara prezahtjevne programe za sjenčanje.

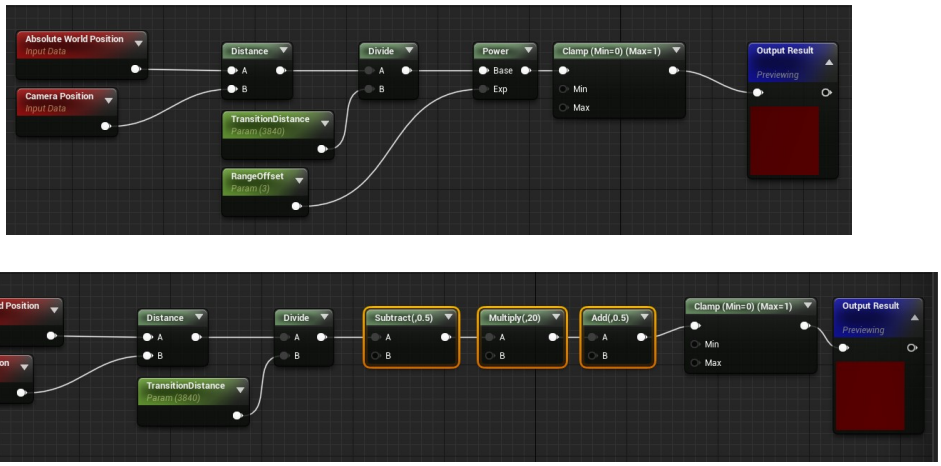
Slika 34 - prikaz zahtjevne scene koja će biti korištena za optimizaciju, Lijevo: izgled scene, Desno: zahtjevnost programa za sjenčanje

Na ovoj razini primjenjujemo sve prije spomenute metode i tehnike optimizacije kako bi prikazali njihovu učinkovitost. Počinjemo od prve metode – rješavanje bespotrebnih karakteristika. Sve teksture koje se koriste za ovu razinu kombinacija su jednostavnijih tekstura koje su zatim kombinirane zajedno. Primjer možemo vidjeti kod teksture *Landscape* koja koristi 5 različitih tekstura: Sand, Gravel, Stones, Rock, RockSide.

Kao primjer, možemo vidjeti dio programa za sjenčanje (tekstura Gravel) koji regulira grubоću teksture (slika 35). Trenutačna grubоća optimizirana je na foto realističan način, čiji rezultat nije direktno uočljiv u razini. Takva optimizacija je pogodnija za CGI u filmovima, a ne za igre. Iz tog razloga možemo ovaj dio ukloniti.

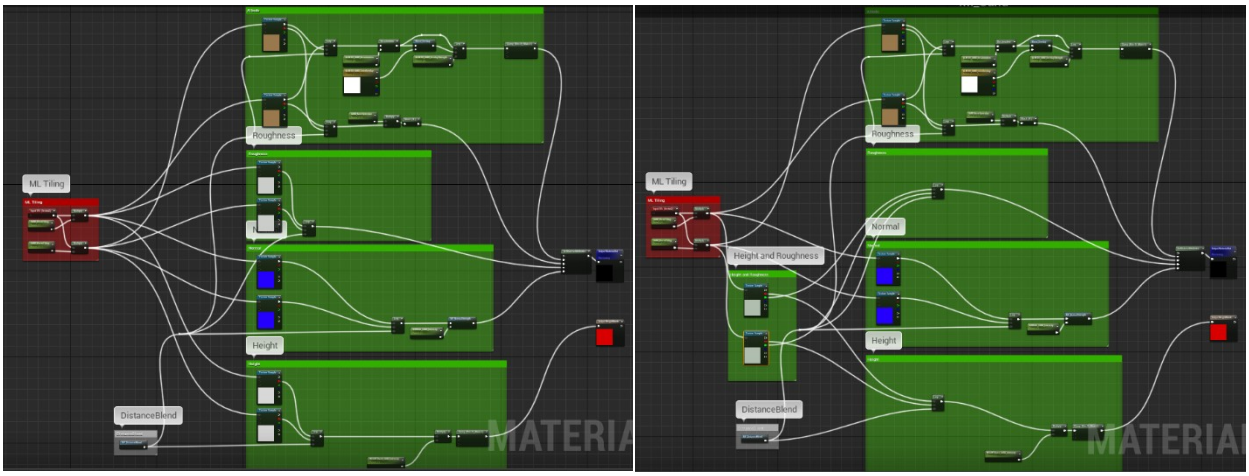
Slika 35 - prikaz upotrijebljene metode rješavanja nepotrebnih karakteristika

Zatim primjenjujemo drugu metodu – prerada matematičkih formula. Potenciranje je računalno veoma zahtjevan proces. Radi toga zamijeniti ćemo ga sa jednostavnijim matematičkim funkcijama (slika 36). Kroz usporedbu dobivenih maski zaključujemo da dobivamo jednaki rezultat.



Slika 36 - prikaz upotrijebljene metode prerade matematičkih formula

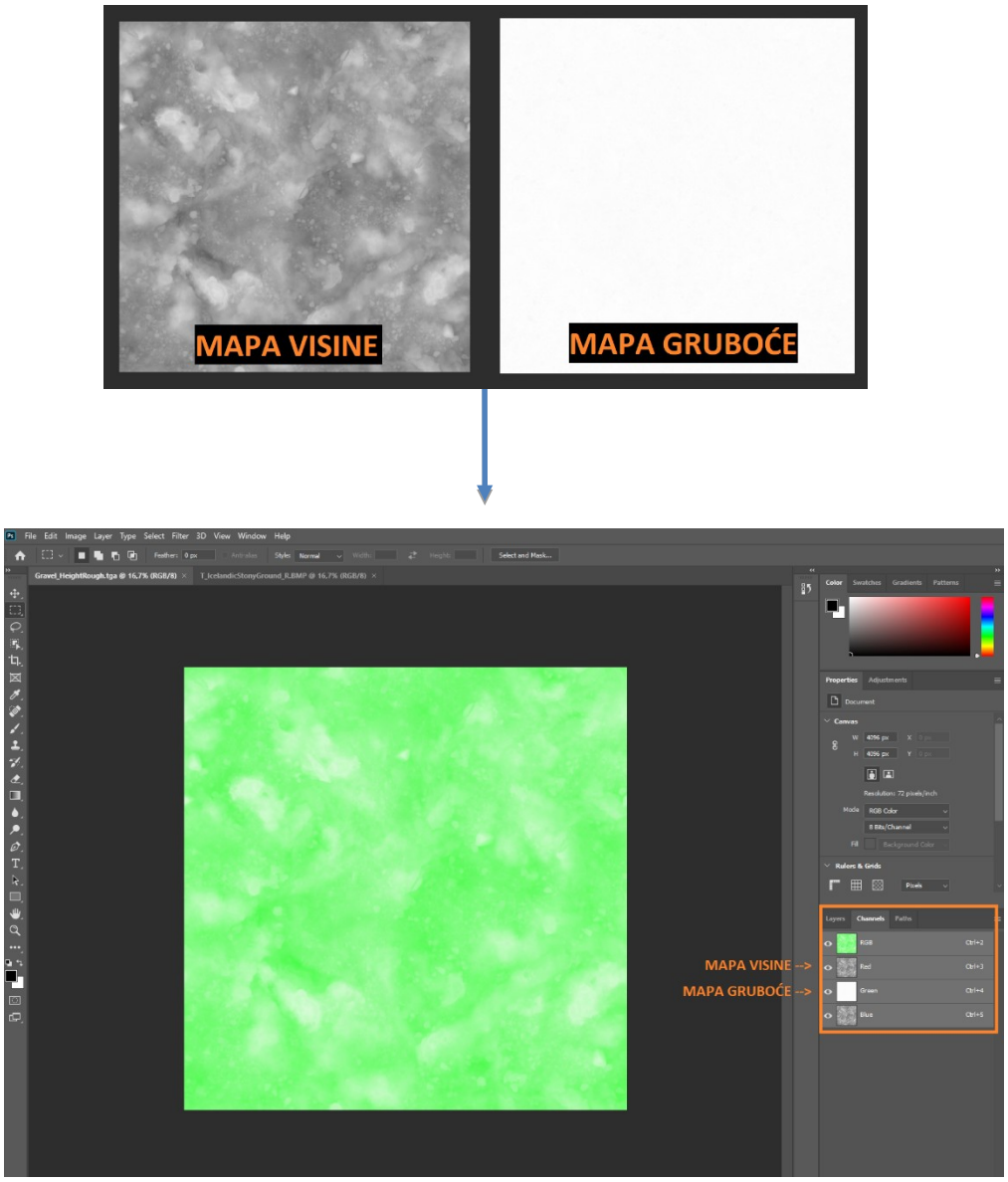
Naposlijetku, krećemo na treću metodu – pakiranje kanala teksture. Najteža instrukcija računalu je pozivanje tekstura. Svaki foto realistični materijal sadrži minimalno 3 teksture koje često imaju veličinu preko 20 megabajta (za dimenziju 2048 x 2048, odnosno 4K). Kao što je napomenuto



Slika 37 - prikaz upotrijebljene metode pakiranja kanala teksture

ranije u radu, *Albedo* i *Normal* mape koriste cijeli RGB spektar. S druge strane, sve ostale (*Roughness*, *Metallic* itd.) su na sivom spektru, odnosno *Grayscale*. Radi toga *Grayscale* teksture možemo spajati u RGB slike s više kanala kako bi u jednoj teksturi spremili dvije ili tri, te time smanjili zahtjevnost programa za sjenčanje (slika 37).

Blue kanal ne moramo mijenjati jer ga u programu za sjenčanje ne spajamo na nikoji čvor (slika 38).



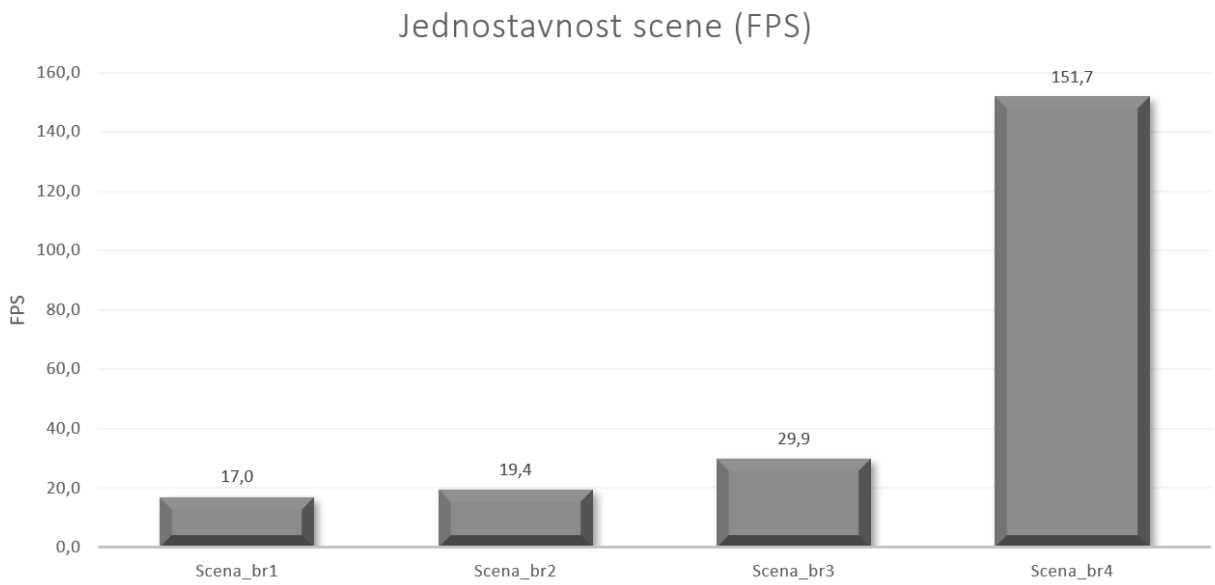
Slika 38 - prikaz pakiranja kanala teksture

Ovom metodom ne smanjujemo broj instrukcija, no uvelike utječemo na brzinu iscrtavanja i time poboljšavamo broj kadrova u sekundi (FPS).

Metode koje smo upravo prikazali naizmjenično primjenjujemo na sve ostale programe za sjenčanje te sve pripadne teksture i njihove pod segmente za najveću moguću optimizaciju.

Nakon primjene metoda provodimo usporedbu razine prije i poslije optimizacije. Uspoređivati ćemo broj kadrova u sekundi i brzinu iscrtavanja pod uvjetom da optimizacija koju smo proveli nije utjecala na izgled razine. U svrhu testiranja odabrala sam četiri kadra scene kojima se redom smanjuje brzina iscrtavanja (zahtjevnost scene se smanjuje). Također, uspoređujemo postoji li razlika u izgledu scene i njoj pripadnih tekstura (slika 39, graf 1).

Slika 39 - usporedba neoptimizirane i optimizirane scene



Graf 1 - prikaz jednostavnosti neoptimizirane scene po broju kadrova u sekundi

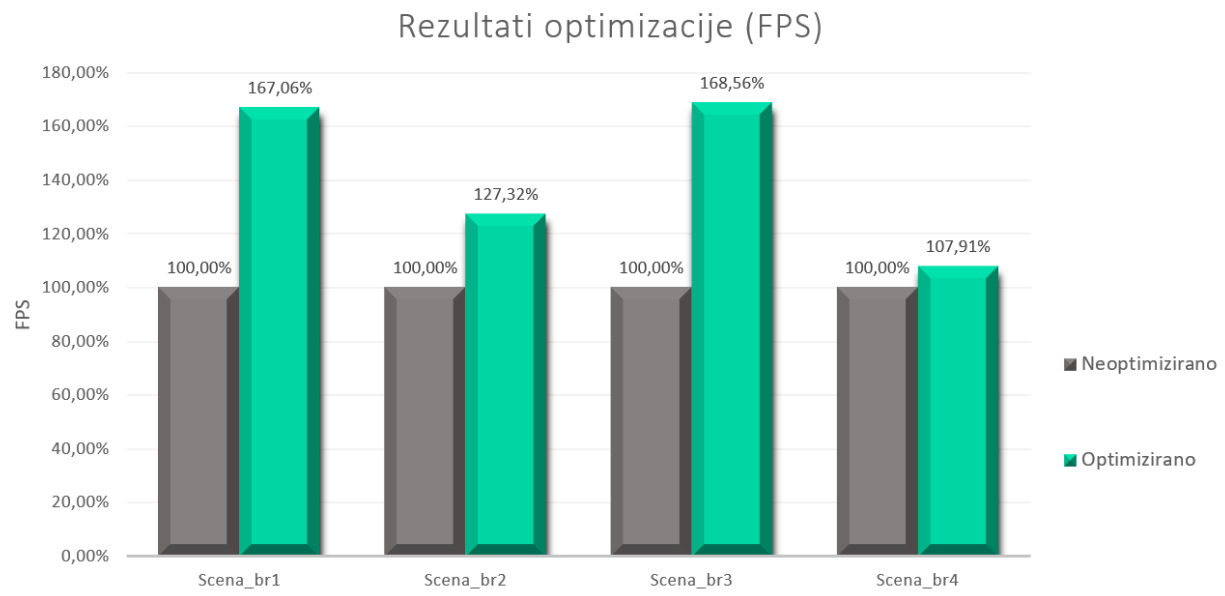
Sljedeće iščitavamo promjenu u količini asemblerskih instrukcija za pojedine programe za sjenčanje na kojima smo provodili optimizacijske metode (tablica 1).

Tablica 1 – promjena broja asemblerskih instrukcija nakon optimizacije

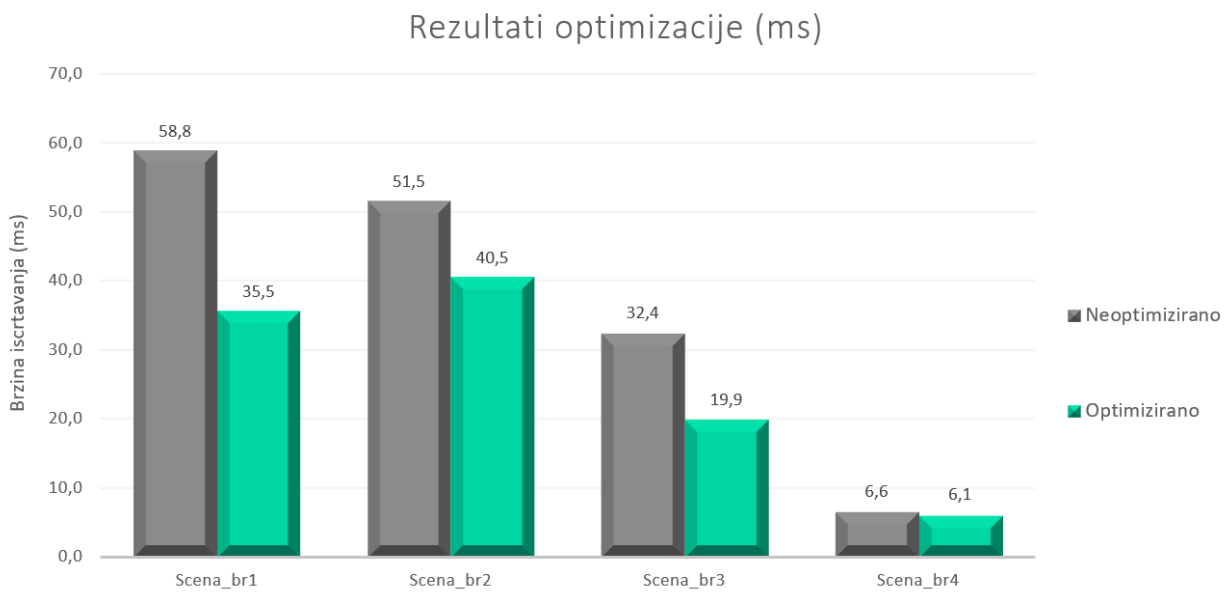
	Broj asemblerskih instrukcija		
Program za sjenčanje	Neoptimizirano	Optimizirano	Poboljšanje (%)
LSMaster	439	373	15,0%
FogCard	157	155	1,3%
CompositeMaster	442	388	12,2%
BlendMaster	452	438	3,1%
2DRaymarched_Loop	95	91	4,2%
StandardMaster	164	154	6,1%
LightShaft	101	90	10,9%

Optimizacija je postigla sveukupno prosječno smanjenje asemblerskih instrukcija od 7,6%. Najveće poboljšanje možemo vidjeti kod programa za sjenčanje naziva *LSMaster*, gdje vidimo smanjenje broja instrukcija od 15%. U isto vrijeme programi za sjenčanje poput *FogCard* nisu omogućavali pretjerane promjene radi načina na koji su složeni.

Svaka asemblerska instrukcija ima svoje vrijeme izvršavanja, te se razlika u vremenu može značajno razlikovati od instrukcije do instrukcije. Radi toga smanjenje broja instrukcija nije potpuni pokazatelj izvršene optimizacije. Kako bi prikazali cijeli utjecaj optimizacije, uz to smo mjerili razliku u broju kadrova u sekundi (FPS) i brzinu iscrtavanja u milisekundama (graf 2, 3).



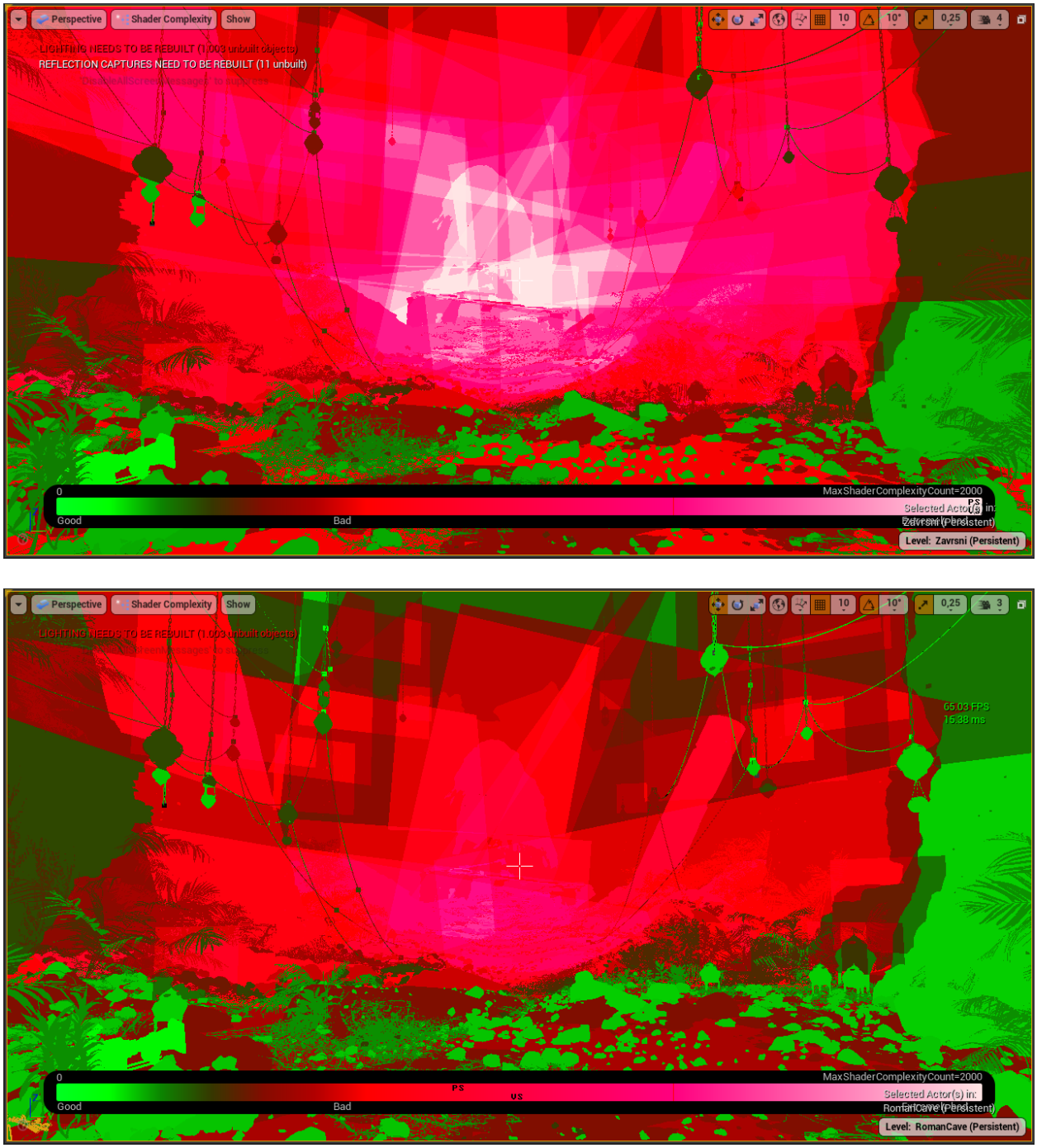
Graf 2 - usporedba broja kadrova u sekundi između neoptimizirane i optimizirane scene



Graf 3 - usporedba brzine iscrtavanja u milisekundama između neoptimizirane i optimizirane scene

Kao što možemo vidjeti, sveukupni utjecaj optimizacije na scenu je značajno veći nego što smo vidjeli kod pojedinačnih programa za sjenčanje i njihovog broja instrukcija. Postigli smo prosječno poboljšanje od 42,7% (po FPS-u), gdje na određenim scenama to poboljšanje iznosi čak skoro 70%. Bitno je za uočiti da ako se kompleksnost scene značajno smanji, razlika u rezultatu optimizacije je manja. Uzrok tome je što pri povećanim udaljenostima UE alat pojednostavljuje izračune svjetlosti, detalja tekstura itd. kako bi uštedio na računalnoj zahtjevnosti. Radi takvih postavki, programi za sjenčanje ne moraju izvršavati većinu koda. Time je utjecaj optimizacije manje vidljiv. [23]

Kada ponovno usporedimo zahtjevnost programa za sjenčanje kroz postavku alata UE, dobivena je vidljiva promjena (slika 40).



Slika 40 - finalna razlika u zahtjevnosti programa za sjenčanje. Gore: prije optimizacije, Dolje: poslije optimizacije

6. Zaključak

U ovom radu upoznali smo se sa osnovama 3D računalne grafike, UE alata i programa za sjenčanje. Programi za sjenčanje često pokazuju nepotrebnu kompleksnost koja značajno povećava zahtjevnost scena. Iz tog razloga koristimo metode i tehnike optimizacije programa za sjenčanje. Kao primjer utjecaja metoda, pri obradi zahtjevne scene primijetili smo smanjenje broja asemblerskih instrukcija od 7,6% i posljedično povećanje u broju kadrova u sekundi (FPS) od 42,7% bez promjene u izgledu tekstura obrađene scene.

Iz toga zaključujemo da su korištene metode nužne za izradu kvalitetnih video igara.

Literatura

[1] "The History Of Gaming: An Evolving Community," *TechCrunch*. https://social.techcrunch.com/2015/10/31/the-history-of-gaming-an-evolving-community/ (accessed Sep. 22, 2021).

[2] B. Dahal, "Meet the Shaders : Vertices, Polygons and Meshes," *Medium*, Mar. 27, 2018. https://medium.com/@darkdreamday/meet-the-shaders-vertices-polygons-and-meshes-1dde115c4bc6 (accessed Sep. 22, 2021).

[3] B. Dahal, "Inside out : A Shader's Anatomy," *Medium*, Mar. 29, 2018. https://medium.com/@darkdreamday/inside-out-a-shaders-anatomy-51d006291a45 (accessed Sep. 22, 2021).

[4] W. E. Carlson, "11.2 Industrial Light and Magic (ILM)," Jun. 2017, Accessed: Sep. 22, 2021. [Online]. Available: https://ohiostate.pressbooks.pub/graphicshistory/chapter/11-2-industrial-light-and-magic-ilm/

[5] "literatuur-shaders.pdf." Accessed: Sep. 22, 2021. [Online]. Available: https://www.cs.vu.nl/~eliens/download/literatuur-shaders.pdf

[6] K. Halladay, *Practical Shader Development: Vertex and Fragment Shaders for Game Developers*. Berkeley, CA: Apress, 2019. doi: 10.1007/978-1-4842-4457-9.

[7] "What is Rendering? - Definition from Techopedia," *Techopedia.com*. http://www.techopedia.com/definition/9163/rendering (accessed Sep. 22, 2021).

[8] M. Botsch *et al.*, "Geometric modeling based on polygonal meshes," in *ACM SIGGRAPH 2007 courses on - SIGGRAPH '07*, San Diego, California, 2007, p. 1. doi: 10.1145/1281500.1281640.

[9] stevewhims, "Asm Shader Reference - Win32 apps." https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx9-graphics-reference-asm (accessed Sep. 22, 2021).

[10] "Vertex Shaders|NVIDIA." https://www.nvidia.com/en-us/drivers/feature-vertexshader/ (accessed Sep. 22, 2021).

[11] "Chapter_1.pdf." Accessed: Sep. 22, 2021. [Online]. Available: http://download.nvidia.com/developer/cg/Cg_Tutorial/Chapter_1.pdf

[12] "What is Unreal Engine? | How It Works | Scope & Career | Advantages," *EDUCBA*, Jun. 01, 2019. https://www.educba.com/what-is-unreal-engine/ (accessed Sep. 22, 2021).

[13] L. I. Wilson, "Godot, Unity, Unreal Engine, CryEngine? Which Game Engine Should I Choose?," *Medium*, Dec. 16, 2019. https://medium.com/@thelukaswils/godot-unity-unreal-engine-cryengine-which-game-engine-should-i-choose-553f8ff7999f (accessed Sep. 22, 2021).

[14] "Unreal Engine | MetaHuman Creator," *Unreal Engine*. https://www.unrealengine.com/en-US/metahuman-creator (accessed Sep. 22, 2021).

[15] "Unreal Engine | Features," *Unreal Engine*. https://www.unrealengine.com/en-US/features (accessed Sep. 22, 2021).

[16] "Level Editor." https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LevelEditor/ (accessed Sep. 22, 2021).

[17] "Textures." https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Textures/ (accessed Sep. 22, 2021).

[18] "What are the different texture maps for?" https://help.poliigon.com/en/articles/1712652-what-are-the-different-texture-maps-for (accessed Sep. 22, 2021).

[19] "Shader Development." https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Rendering/ShaderDevelopment/ (accessed Sep. 22, 2021).

[20] C. Hoffman, "The Difference Between Gaming 'Lag' and Low FPS (and How to Fix Them)," *How-To Geek*. https://www.howtogeek.com/142193/htg-explains-why-lag-and-low-fps-arent-the-same-thing/ (accessed Sep. 22, 2021).

[21] "Performance and Profiling." https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/PerformanceAndProfiling/ (accessed Sep. 22, 2021).

[22] Ben Cloward, *Shader Performance Optimization - UE4 Materials 101 - Episode 7*, (2020). Accessed: Sep. 22, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=D8E47BJOE6E

[23] "Designing Visuals, Rendering, and Graphics." https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/ (accessed Sep. 22, 2021).

Prilozi

Slika 1	http://cse.csusb.edu/tongyu/courses/cs520/notes/mesh.php (Accessed Sep. 22, 2021).
Slika 2	A O'Connor, "A new BioShock is being made at 2K's new studio," Rock, Paper, Shotgun, Dec. 09, 2019. Accessed: Sep. 22, 2021. [Online]. Available: https://www.rockpapershotgun.com/new-bioshock-announced B "Dishonored (Game of the Year) – gamerpick.com." https://gamerpick.com/product/dishonored-game-of-the-year/ (accessed Sep. 22, 2021)
Slika 3	"Unreal Engine | MetaHuman Creator," Unreal Engine. https://www.unrealengine.com/en-US/metahuman-creator (accessed Sep. 22, 2021).
Slika 4	A "Hollow Knight on Steam." https://store.steampowered.com/app/367520/Hollow_Knight/ (accessed Sep. 22, 2021). B "Interaktivna umjetnost #6: 'Inside,'" Inverzija. https://inverzija.net/interaktivna-umjetnost-6-inside/ (accessed Sep. 22, 2021).
Slika 5	Darkcrizt, "Godot, iyo yakavhurwa sosi yemutambo injini yakagadziridzwa kuita vhezheni 3.3," Linux Adictos, Apr. 27, 2021. https://www.linuxadictos.com/sn/godot-yakavhurwa-sosi-mutambo-injini-inovandudzwa-kuchikamu-3-3.html (accessed Sep. 22, 2021).
Slika 8 - 16	"Texture Maps: The Ultimate Guide For 3D Artists," Concept Art Empire, Feb. 25, 2019. https://conceptartempire.com/texture-maps/ (accessed Sep. 22, 2021).
Korištena razina	"Megascans Goddess Temple in Environments - UE Marketplace," Unreal Engine. https://www.unrealengine.com/marketplace/en-US/product/megascans-goddess-temple (accessed Sep. 22, 2021).

Korišteni materijali (slika 21 – 33)	Ben Cloward, Shader Performance Optimization - UE4 Materials 101 - Episode 7, (2020). Accessed: Sep. 22, 2021. [Online Video]. Available: https://www.youtube.com/watch?v=D8E47BJOE6E
Korišteni alati	A “Unreal Engine | The most powerful real-time 3D creation platform,” Unreal Engine. https://www.unrealengine.com/en-US/ (accessed Sep. 22, 2021). B “Official Adobe Photoshop | Photo & Design Software.” https://www.adobe.com/products/photoshop.html (accessed Sep. 22, 2021).