Optimizacija tekstura u 3D videoigrama

Cvitković, Antonio

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Optimizacija tekstura u 3D videoigrama
Texture optimization in 3D video games

Rijeka, rujan 2023.

Antonio Cvitković
0069085517

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski studij računarstva

Završni rad

Optimizacija tekstura u 3D videoigrama
Texture optimization in 3D video games

Mentor: doc.dr.sc. Goran Mauša

Rijeka, rujan 2023.

Antonio Cvitković
0069085517

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA ZAVRŠNE ISPITE

Rijeka, 13. ožujka 2023.

Zavod: **Zavod za računarstvo**
Predmet: **Programsko inženjerstvo**
Grana: **2.09.06 programsko inženjerstvo**

ZADATAK ZA ZAVRŠNI RAD

Pristupnik: **Antonio Cvitković (0069085517)**
Studij: Sveučilišni prijediplomski studij računarstva

Zadatak: **Optimizacija tekstura u 3D videoigrama // Texture optimization for 3D video games**

Opis zadatka:

Proučiti postupke vizualizacije površina u videoigrama, s naglaskom na prikaz površina poligona 3D objekata u stvarnom vremenu. Predložiti rješenje za optimizaciju teksture u ostvarivanju prikaza objekata temeljeno na udaljenosti kamere s ciljem postizanja slike visoke vizualne kvalitete uz minimalno korištenje memorije. Vrednovati predloženo rješenje na skupu standardnih obrazaca u području računalne grafike.

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

Zadatak uručen pristupniku: 20. ožujka 2023.

Mentor:

Doc. Goran Mauša, dipl. ing.

Predsjednik povjerenstva za završni ispit:

Prof. dr. sc. Miroslav Joler

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad.

Rijeka, rujan 2023.

Antonio Cvitković

Zahvala

Zahvaljujem svojem mentoru Goranu Mauši na podršci tijekom pisanja ovoga rada i korisnim raspravama i savjetima. Zahvaljujem svojim roditeljima i bratu na podršci tijekom studiranja.

Sadržaj

Popis slika

Popis tablica

Poglavlje 1

Uvod

U svijetu računalne grafike, teksture su ključne komponente koje daju život digitalnim krajolicima, bićima i predmetima u videoigrama i filmovima. Teksture definiraju njihov izgled, materijalne osobine i razinu detalja, bili detalji realistične prirode ili stilizirani. Uobičajene prakse uključuju korištenje različitih vrsta tekstura, odnosno teksturnih mapa, poput albedo, difuznih, metalnih, spekularnih, mapa hrapavosti, podpovršinskih mapa, mapa ambijentalne okluzije, mapa normala i mapa pomaka, kako bi se simulirale tražene osobine površina. Međutim, visoko-razlučive teksture mogu biti resursno zahtjevne, što dovodi do dužih vremena učitavanja, većeg zauzimanja memorije te smanjenja broja slika u sekundi u videoigrama. Tijekom godina, industrija video igara vidjela je izvanredan napredak u tehnikama optimiziranja tekstura kako bi se održao integritet računalno prikazanog sadržaja uz poboljšano upravljanje resursima. Neke od tih tehnika su atlasiranje tekstura, mipmapiranje te streamanje tekstura. U videoigrama najveći fokus optimizacije je na samim 3D modelima zato što grafička procesorska jedinica (GPU) se najviše koristi za prikaz objekata s obzirom da je to najintenzivniji proces. Optimizacija 3D modela rezultira u veliki porast slika u sekundi pri renderanju u stvarnom vremenu. Optimizaciju 3D modela nazivamo retopologija; proces u kojemu isponova stvaramo manje detaljan 3D model istog oblika, te umjesto prikaza detalja na samom modelu, optimizirani model dobiva informacije o detaljima od tekstura koje su stvorene projekcijom pogleda vektora normala površine neoptimiziranog 3D modela na taj model. [1]. Zanimljivo je da nakon optimizacije 3D modela, same teksture često postaju te koje imaju veću

težinu procesiranja, te mogu zauzimati stotine puta više memorije od 3D modela. Iz tog razloga, bitno je proučiti daljnje ideje optimizacije, gdje je fokus na teksturama, s čime se mogu stvarati moderne igre koje su desetke, pa i stotinu gigabajta manje u memoriji.

U ovom radu istražiti će se metode optimiziranja tekstura, raspraviti određene mane tih metoda i predložiti rješenja kojima se mogu ukloniti te mane. Također, programskim jezikom Java izraditi ćemo programsko rješenje koje će nam služiti za automatizaciju procesa optimizacije tekstura. Optimizaciju će program vršiti sam, vrednovanjem čimbenika poput oplošja 3D modela, umjetničke procjene veličine tekstura i udaljenosti kamere.

1.1 Osnovni pojmovi računalne grafike

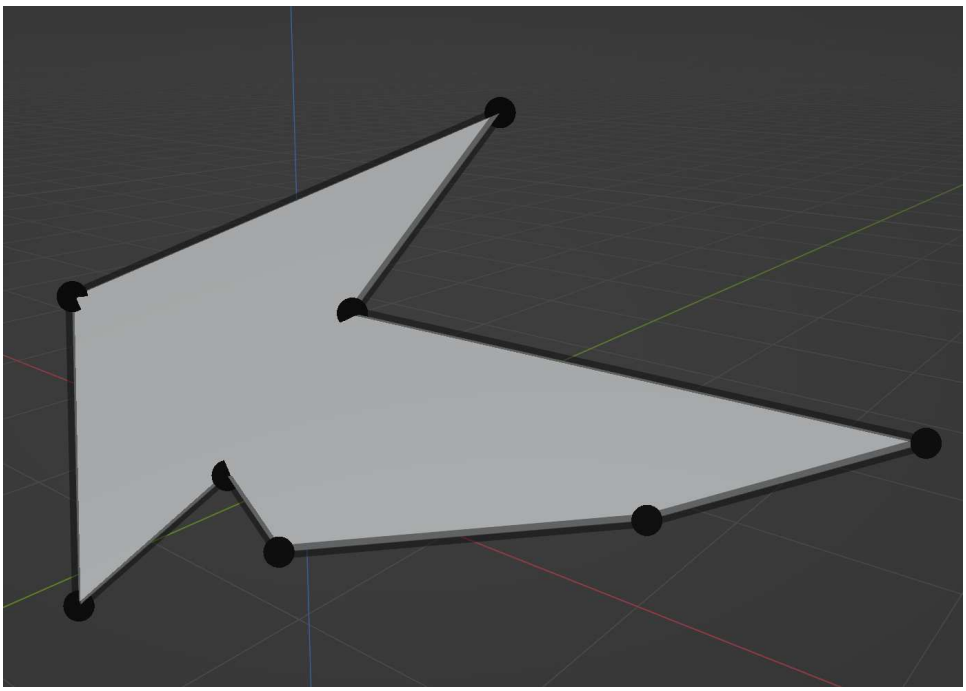
Poligoni, odnosno mnogokuti, su osnovne grafičke komponente u 3D računalnoj grafici koje predstavljaju ravne geometrijske likove s ravnim stranicama koje se koriste za stvaranje trodimenzionalnih objekata. Poligoni se sastoje od vrhova (točaka u prostoru) i bridova (linije koje povezuju te točke). Površinu koju stvaramo između vrhova, nazivamo tijeom poligona. Ovisno o broju bridova, u računalnoj grafici često se spominju tri vrste poligona primjerene za drugačiji sadržaj:

1. Trokut (Tri),
2. Četverokut (Quad),
3. N-terokut.

Trokut je osnovna razina poligona koja služi za prikazivanje objekata u 3D prostoru. Kao najjednostavniji oblik poligona, pogodni su za različite grafičke operacije te su važni za izračunavanje osvjetljenja i sjene u 3D svijetu s obzirom da se osvjetljenje često računa na temelju orijentacije trokuta u odnosu na izvor svijetla. GPU je optimizirana za obradu trokuta, te je njihovo prikazivanje brže i učinkovitije u odnosu na druge geometrijske oblike u računalnoj grafici. Trokuti se koriste za interpolaciju boja, tekstura i drugih svojstava između vrhova što omogućuje lako prikazivanje složenih površina [2].

Četverokut služi kao jedan od osnovnih razina poligona i često služi za lakše opisivanje prostora od trokuta. Koriste se za prijenos podataka iz nekih CAD programa i najčešće služe za modeliranje objekata u videoigrama. Najsavitljiviji su za deformacije u animacijama i deformacije algoritama za skulptiranje. Četverokuti pojednostavljuju proces postavljanja tekstura na 3D modele pri UV odmatanju i mapiranju. Modeliranje koje koristi krivine odnosno Non-Uniform Rational B-Spline (NURBS) modeliranje, krivine prikazuje pomoću četverokuta za stvaranje glatkih i preciznih površina. Dijeljenjem četverokuta na poddivizije u procesu poddivizijskog modeliranja, dobivamo vrlo prirodne rezultate, stoga su najprimjereniji takvom tipu modeliranja [3]. Četverokuti se koriste kao osnovna jedinica retopologije [4]. Iako se pojavljuju kao najčešći element računalne grafike, mnogi pogonski sklopovi videoigara, dijele četverokute na trokute zbog detekcije kolizija [5].

N-terokut, generalno je svaki oblik poligona s n bridova, no u 3D industriji ovaj naziv se primarno koristi za poligone koji imaju 5 bridova ili više. N-terokuti pružaju fleksibinost u 3D modeliranju objekata, pogotovo pri modeliranju organskih površina. Pogodni su za statičke objekte koji nisu animirani te se često dobivaju 3D skaniranjem objekata u stvarnom svijetu. Iako se mogu koristiti i obrađivati, ovaj oblik poligona ima previše nedostataka. Prije renderanja, n-terokuti se moraju teselirati u trokute, što dovodi do nepotrebnog preopterećivanja grafičke procesorske jedinice [6]. Poddivizijsko modeliranje i skulptiranje n-terokuta daje korisniku neočekivane rezultate i neželjene artifakte. Pri animaciji, savijanjem bridova n-terokuta također dobivamo neočekivane rezultate. Neki rendereri, štoviše, odbijaju prihvatiti n-terokute kao prikladan oblik prikaza. Uobičajena praksa u 3D industriji je u cijelosti izbjegavati n-terokute [7].



Slika 1.1 Prikaz n-terokuta u 3D prostoru (n = 8)

1.2 Teksturne mape

Teksture su slike koje mapiramo na površinu oblika ili poligona. Tekstura može biti rasterska ili vektorska, odnosno proceduralna tekstura. Rasterske teksture definirane su poznatim formatima poput Portable Network Graphics (PNG), Joint Photographic Experts Group (JPG/JPEG) i Tag Image File Format (TIFF), dok su proceduralne teksture stvorene matematičkim opisom fraktalnih šumova i funkcija turbulencije kako bi poprimile određene boje i osobine [8]. Iako nam proceduralne teksture pružaju beskonačnu rezoluciju, a zauzimaju vrlo malo memorije, industrijski standard postale su rasterske teksture. Interoperabilnost, izmjenjivost i fleksibilnost samo su od nekih faktora zbog kojeg rasterske teksture dominiraju industrijom, dok u drugu ruku interoperabilnost i uređivanje proceduralnih tekstura kako bi dobili traženi rezultat je često težak zadatak.

U ovom radu, manipulirati ćemo datoteke formata *PNG*, zato što su one najčešće

korištene u industriji video igara. No pokazati ćemo neke rezultate u kojima koristimo format *JPG* gdje je to dopušteno. Format *PNG* je u videoigrama popularan iz 4 razloga [9] :

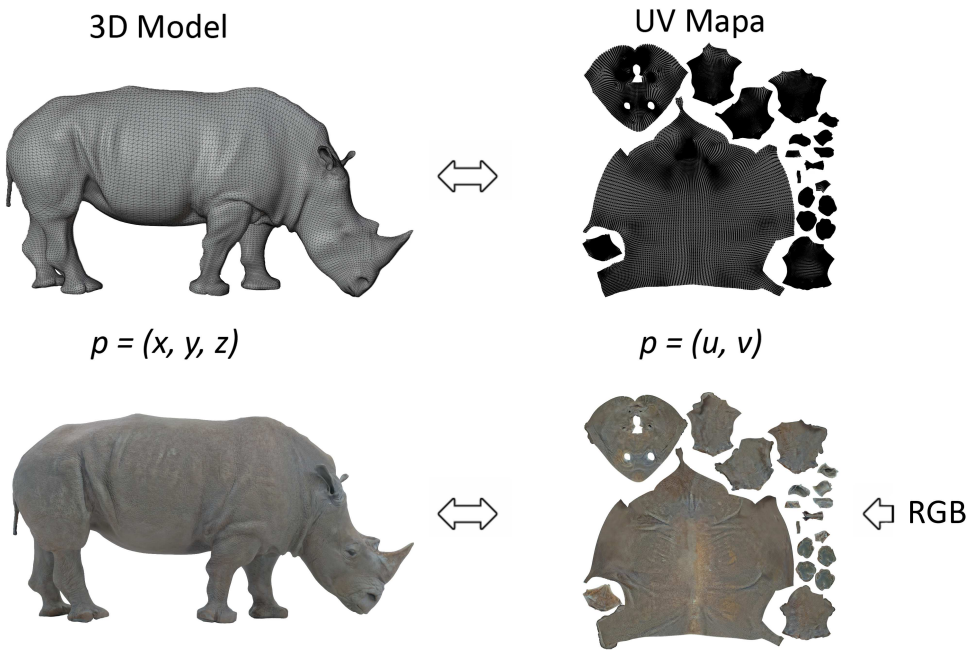
1. Kompresija datoteke bez gubitka kvalitete,
2. Format podržava alfa kanal, odnosno transparentnost,
3. Kompatibilnost sa mnogim programima,
4. 24-bitna RGB i 32-bitna RGBA paleta boja.

U svijetu 3D video igara, osim primjene boje putem albedo mapa sa RGB kanalima, većina mapa koristi se bezbojnim kanalima. Odnosno, vrijednosti sive boje od 0 do 1, gdje 0 predstavlja crnu boju, a 1 predstavlja bijelu boju. Tim numeričkim vrijednostima predstavljamo jačinu svojstva. Bezbojni kanal, iako nazivom nije intuitivno, može značiti da se tekstura koristi samo jednim kanalom RGBa. Iznimka ovom nazivu su mape normale, koje RGB kanala ne koriste za boju nego vrijednosti vektora gdje crvena boja (R) sadrži informacije o X-komponente normale, zelena (G) informacije Y-komponente, te plava (B) informacije Z-komponente normale [10].

1.3 UV mapiranje

UV mapiranje je proces u 3D modeliranju gdje projektiramo odnosno "rasklapamo" površinu 3D modela na 2D površinu kako bi mapirali teksture na poligone. Slova "U" i "V" označuju osi 2D teksture, zato što u alatima za obradu 3D modela, slova "X", "Y" i "Z" već koristimo za osi 3D prostora. Vrijednosti U i V koordinata kreću se od 0 do 1, gdje donji lijevi kut UV prostora je (0, 0), a gornji desni kut (1, 1). U procesu projektiranja, svaki vrh svakog poligona poprima UV koordinate, tako zvane koordinate teksture. UV mapa može biti generirana automatski s alatima aplikacije, može biti postavljena ručno sa strane korisnika ili oboje. Poželjno je zauzeti što veći UV prostor sa projektiranim poligonima, zato što tada optimiziramo koliko piksela teksture se nalazi na poligonima. Kada 3D model poprimi sve UV koordinate, uređivanjem različitim alatima, ovisno koji softver koristimo, možemo primjenjivati boju i svojstva na različite mape. UV mapiranje je fundamentalan dio

procesa izrade 3D modela i stoga se ovom procesu mora vrlo pažljivo pristupiti [11].



Slika 1.2 Povezanost 2D UV prostora sa 3D prostorom

1.4 Format OBJ

Wavefront .obj format (OBJ), najčešće samo zvan **OBJ** ili *obj.* je format datoteke za definiciju geometrije koji je prvi razvila tvrtka Wavefront Technologies oko 1990. godine za svoj paket za animaciju Advanced Visualizer. Format datoteke je otvoren i usvojen je od strane cijele industrije za 3D grafiku te u 2023. godini je još uvijek standard za opisivanje 3D objekata uz popularne formate poput Filmbox (.fbx), COLLADA Digital Asset Exchange(.dae), Stereolithography (.stl) i Polygon File Format (.ply). U svijetu programiranja *OBJ* je popularan izbor za manipuliranje i parsiranje zbog *ASCII* zapisa. Iako format podržava slobodne oblike opisane površinama i krivinama, u videoigrama i filmu koristi se varijanta *OBJ-a* koja opisuje poligonalnu geometriju [12].

Kako bi bolje razjasnili format *OBJ*, ukratko ćemo opisati strukturu jednostavne datoteke koja u sebi ima zapisane podatke o 3D objektu kocke.

Komentari u *OBJ* datotekama započinju znakom "#" i služe za dodatne opise objekata, zapis vlasništva ili kao potpis programa iz kojeg je datoteka izvezena. Tako u kocki koju smo izvezli iz 3D programa Blender, slijedi komentar:

```
# Blender 3.6
# www.blender.org
```

Zatim, u datoteci je zapisano ime materijala koji se veže na 3D model, gdje je mtllib oznaka za knjižnicu materijala, a Cube.mtl naziv datoteke u kojoj su zapisane informacije o materijalima:

```
mtllib Cube.mtl
```

Od opisnih komponenti, slijedi ime 3D objekta, koje će biti prikazano unutar softvera za uređivanje 3D objekata. U našem slučaju znak "o" označava ime, te iza njega slijedi ime objekta:

```
o Cube
```

Oznakom "v" označavamo vrhove i pored toga zapisujemo njihove X, Y i Z koordinate. Pored X, Y i Z koordinate postoji W koordinata, koja ako nije zapisana ima vrijednost 1. W koordinata služi za napredne operacije poput perspektive korekcije, te se vrlo rijetko pojavljuje u 3D modelima.

```
v 1.000000 1.000000 -1.000000
    \#Vrh kocke je definiran koordinatama
    \#X = 1, Y = 1, Z = -1
    \#itd. za svaki vrh koji čini poligone kocke.
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 1.000000
```

Oznakom "vn" definiramo X, Y i Z koordinate normale, odnosno opisujemo u kojem je smjeru okrenuta površina poligona.

```
vn -0.0000 1.0000 -0.0000
    \#Poligon je na X = 0 okrenut prema negativnoj strani
        X osi.
    \#Poligon je na Y = 1 okrenut prema pozitivnoj strani
        Y osi.
    \#Poligon je na Z = 0 okrenut prema negativnoj strani
        Z osi.
    \#itd. za ostale "vn".
vn -0.0000 -0.0000 1.0000
vn -1.0000 -0.0000 -0.0000
vn -0.0000 -1.0000 -0.0000
vn 1.0000 -0.0000 -0.0000
vn -0.0000 -0.0000 -1.0000
```

Oznakom "vt" definiramo koordinate tekstura, odnosno UV mape na u-osi i v-osi. Iako u 3D prostoru imamo samo 8 vrhova, kako bi ih preslikali u 2D prostor, potrebno nam je 14 vrhova u ovako zatvorenom obliku poput kocke.

```
vt 0.625000 0.500000
    \#3D vrh preslikan u 2D prostor, s koordinatama
    \#u = 0.625000, v = 0.500000.
    \#itd. za svaki vrh "vt".
vt 0.375000 0.500000
vt 0.625000 0.750000
vt 0.375000 0.750000
vt 0.875000 0.500000
vt 0.625000 0.250000
vt 0.125000 0.500000
vt 0.375000 0.250000
vt 0.875000 0.750000
vt 0.625000 1.000000
vt 0.625000 0.000000
vt 0.375000 1.000000
vt 0.375000 0.000000
vt 0.125000 0.750000
```

Oznakom "s" označavamo ako je 3D objekt podvrgnut algoritmima za zaglađivanje. U našem slučaju kocke, gdje nema zaglađivanja, zapisano je:

```
s 0
```

Sljedeće oznake "usemtl" i "f" su međusobno povezane. "usemtl" definira koji materijal u knjižnici materijala datoteke formata .mtl se koristi za sve "f" (faces), odnosno površine poligona. Uz oznaku "f" upisujemo vrijednosti "v", "vt" i "vn" za određeni poligon. Uz pretpostavku da na 3 lica kocke postavimo jedan materijal A, a na preostala 3 postavimo materijal B, ostatak *OBJ* datoteke izgleda sljedeće:

```
usemtl A
f 1/1/1 2/2/1 4/3/1 3/4/1
f 3/4/2 4/3/2 8/5/2 7/6/2
f 7/6/3 8/5/3 6/7/3 5/8/3
usemtl B
f 5/8/4 6/7/4 2/9/4 1/10/4
f 3/11/5 7/6/5 5/8/5 1/12/5
f 12/5/6 10/13/6 9/14/6 11/7/6
```

U našem slučaju svako lice je četverokut i stoga je opisano kvartetom vrhova.

Iako smo objasnili sva svojstva ovog vrlo jednostavnog zapisa, unutar programskog rješenja kojeg ćemo kasnije u radu predložiti u svrhu optimizacije tekstura, biti će nam potrebne samo informacije o koordinatama vrhova, te njihov odnos sa poligonima, odnosno licima koje vrhovi stvaraju. Ovo znanje nam je potrebno kako bi lako parsirali *OBJ* datoteke.

Poglavlje 2

Uobičajene prakse optimizacije tekstura u videoigrama

Optimizacija tekstura u razvoju video igara je krucijalna ako želimo kao developeri video igara ponuditi korisnicima najbolje iskustvo te optimizirati performanse igre za računala sa jako niskom računalnom snagom, s obzirom da velika većina korisnika na svijetu nema jako moćna računala. U 2023. godini, sve češće možemo vidjeti da videoigre postaju sve zahtjevnije za procesirati. Također, memorija koju igre zauzimaju je u proteklih 10 godina jako eskalirala. Iako točan iznos memorije ovisi o tipu igre, sve češće možemo vidjeti popularne AAA igre koje prekoračuju 100, pa čak i 200 GB memorije. Sva ta memorija se naravno ne učitava odmah u jednoj sceni videoigre, no moramo se zapitati koliko su same učitane scene postale složene s obzirom na tu količinu memorije. Dok današni hardver prati složenost igara, developeri se moraju pobrinuti da su igre što manje zahtijevne. Wirthov zakon o performansama računala govori o tome kako softver rapidnije postaje sporiji nego hardver uspijeva postati brži. Dok je možda kontradiktoran Mooreovom zakonu koji govori da broj tranzistora računala se poduplava svake godine, s obzirom na složenost i robustnost današnjih programa, vrijedi primjetiti da se na optimizaciju pazi sve manje [13].

U računalnoj grafici sve češće razvijamo tehnike koje olakšavaju učitavanje i procesiranje 3D svijetova. U procesu prikazivanja tekstura, česte prakse uključuju:

1. Atlase tekstura - Ujedinjavanje nekoliko tekstura u jedan veliki atlas tekstura, tako da pri renderanju različitih modularnih dijelova velikog modela, imamo što manje poziva od strane procesora za izmjenu iz koje se datoteke podaci čitaju [14],
2. Kompresiju tekstura - Korištenje različitih formata kompresije koji optimiziraju teksture unutar kompajliranih igara. Neki od takvih formata su DirectX Texture Compression (DXT), Block Compression (BCn) i Ericsson Texture Compression (ETC) [15],
3. Ponavljanje tekstura - Gdje je moguće, optimalno je na poligonima ponavljati istu teksturu. Time dobivamo visoku razinu detalja s vrlo malom potrošnjom memorije. Bitno je napomenuti da nam ovdje jako pomažu tzv. "teksture bez šavova" (Seamless textures), zato što na rubovima prijelaza kod ponavljanih tekstura vizualno prijelaz nije očit [16],
4. Mipmapiranje - Generiranje unaprijed skaliranih verzija teksture različitih rezolucija unutar iste teksture kako bi pri renderanju objekata smanjivali i povećavali teksture s obzirom na udaljenost od kamere [17],
5. Stvaranje razina detalja teksture - Poput mipmapiranja, unaprijed generiramo skalirane verzije tekstura različitih rezolucija, no spremamo teksture kao zasebne datoteke koje se unutar igara tokom izvođenja isto kao mipmape renderaju s obzirom na udaljenost od kamere. Kreiranje razina detalja je često brži proces nego generiranje mipmapa [18].

Postoji još mnogo tehnika za optimiziranje tekstura prije i poslije kompajliranja video igara. U ovom radu, fokusirati ćemo se na optimiziranje razina tekstura, počinjući od nulte razine detalja teksture, s obzirom da umjetnicima koji rade teksture često nije očito kolika rezolucija je potrebna za koji model i točna vrijednost zahtjeva godine iskustva u polju. Također, dotaknuti ćemo se teme stvaranja slika sa različitim razinama detalja i usporediti performanse i integritet završnog produkta kada pažljivo pristupimo problemu.

2.1 Generiranje razina detalja tekstura

Kako bi se nosili sa izazovom održavanja visoke kvalitete tekstura uz štednju memorije, generiranje Level of Detail (LOD) teksturnih mapa, odnosno slika sa različitim razinama rezolucije pojavljuje se kao najjednostavnije i najrobustnije rješenje. Glavni cilj LODova je balansiranje performansi dok zadržavamo kvalitetu slike kako se udaljavamo od 3D modela. Kako ne bi dolazilo do naglog prijelaza između tekstura, unutar pogonskih sklopova videoigara često se koriste metode poput opadanja LOD-ova i miješanja razina. Bez korištenja ovih metoda, teksture na 3D modelima neprirodno iskaču kada prelazimo određeni prag udaljenosti.

U svijetu video igara postoje 2 različita pristupa LOD-ovima, te se ova tema često mora raspraviti prije početka samo projekta stvaranja određene videoigre. Razlikujemo LOD-ove koji su generirani dinamično, unutar samih pogonskih sklopova za rađenje videoigara, te statičke LOD-ove koji su unaprijed stvoreni od strane umjetnika. Oba tipa LOD-ova imaju svoje prednosti i nedostatke, te su često tema dostupnog vremena za izvedbu projekta. Očito je da dinamični LOD-ovi ne zahtijevaju pretjeranu pozornost od strane programera, no često puta nisu najbolje optimizirani, pogotovo ako se koristimo neispravnom nultom razinom detalja.

2.1.1 Dinamično generiranje razina detalja

Dinamično stvaranje razina detalja uključuje promjene tekstura u stvarnom vremenu putem virtualnih tekstura i streamanja podataka. Ovakve teksture stvaraju se na kriteriju udaljenosti 3D modela od kamere i ovisno o tome koliki dio ekrana korisnika, odnosno igrača, zauzimaju. Ova metoda služi za povećanje performansi renderiranja slika u sekundi u stvarnom vremenu i poboljšava uštedu memorije. Prednosti dinamičnih razina tekstura su:

1. **Prilagodljivost** je najveća vrlina dinamičkih LOD-ova i zbog nje su teksture uvijek prilagođene trenutnoj sceni.
2. **Konzistentna vizualna kvaliteta** je uvijek prisutna oko igrača, i pogonski sklop videoigre se uvijek brine da gdje god igrač pogleda iz blizine, da su površine i 3D objekti detaljni.

3. **Fleksibilna optimizacija performansi** dozvoljava preciznu kontrolu nad optimizacijom performansi, forkusirajući se na 3D objekte koji imaju najveći utjecaj na memoriju, ovisno o gledištu igrača.

Nedostaci dinamičnih razina tekstura su:

1. **Centralna procesorska jedinica (CPU) se više troši** implementiranjem dinamičnog sistema LOD-ova zato što program igre mora konstantno evaluirati situaciju udaljenosti kamere od svih objekata i mijenjati razine virtualne teksture.
2. **Složena implementacija** je težak zadatak. Stvaranje robusnog dinamičnog LOD sustava može biti vrlo tehnički zahtjevno ako se ne koristimo komercijalnim pogonskim sklopovima videoigara koje imaju već isprogramirane sustave za dinamičan LOD sustav.
3. **Potencijalno iskakanje tekstura** se može događati ako pažljivo ne implementiramo LOD sustav. Ako dolazi do iskakanja tekstura, igraču se odmah gubi imerzija, te moramo biti jako pažljivi ako želimo prikazati realističan svijet.

2.1.2 Statično generiranje razina detalja

Statičko ili unaprijed izračunato LOD generiranje uključuje stvaranje više LOD verzija tekstura tijekom faze stvaranja sadržaja igre, koje zatim odabire i koristi pogonski sklop videoigre na temelju unaprijed određenih kriterija. Prednosi dinamičnih razina tekstura su:

1. **Predvidljive performanse**, koje smanjuju potrebu za izračunima u stvarnom vremenu koje inače vrše testeri, umjetnici i programeri, osiguravajući da sadržaj radi kako je zamišljen.
2. **Olakšavanje rada CPU-a** je produkt nestanka izračuna u stvarnom vremenu koji se stalno događa kod dinamičnog generiranja razine detalja.
3. **Konzistentna vizualna kvaliteta** je u rukama umjetnika, za svaku razinu teksture, osiguravajući da tekstura 3D modela izgleda dobro iz različitih da-

ljina.

Nedostaci statičnih razina tekstura:

1. **Nedostatak prilagodljivosti** pri statičnim LOD-ovima znači da se ne mogu prilagođavati dinamičnim promjenama u 3D svijetu. Iz tog razloga nisu uvijek prikladni za igre sa potpunom kontrolom kamere i za igre gdje je okruženje proceduralno generirano.
2. **Više datoteka i više zauzete memorije** ponekad se može pokazati kao problem pri izgradnji igre, jer sam proces postavljanja manualnog prijelaza između LOD-ova traje i može biti zbunjujuć s nekoliko datoteka. Zauzimanje više memorije, dok nije kritično, svejedno je nedostatak statičnih LOD-ova.
3. **Dugotrajna izrada LOD-ova** se događa zato što za stotine, ponekad i tisuće tekstura još moramo stvarati nekoliko manje rezolutnih instanci.

2.2 Zanemareni aspekti generiranja razina detalja

Sada kada smo naveli sve definicije, formate i prakse koje će biti bitne u ovom radu, možemo se fokusirati na miskoncepcije čestih opisa i pronaći riješenje za iste. Do sada smo detaljno opisali proces generiranja različite razine tekstura, no srž problema optimizacije pojavljuje se prije - u samoj izradi nulte razine detalja, odnosno početne teksture. Umjetnici često nasumično s obzirom na pretpostavljenu veličinu 3D modela postavljaju određenu rezoluciju tekstura, no kako često rade iz jako bliske udaljenosti od modela, znaju zanemariti koliko zapravo male rezolucije teksture mogu biti iz pogleda kamere igre. Naravno, u igrama iz prvog lica i trećeg lica, koje imaju slobodan pokret kamere, visoka kvaliteta je vrlo bitna, no čak i tu postoje slučajevi gdje postavljamo 3D modele do kojih se igrač nikada neće približiti.

Kako je praksa 3D industrije pretpostaviti veličinu modela i time stvoriti početnu rezoluciju, odnosno što je model veći, tekstura je veće rezolucije kako bi se prikazalo više detalja na tako velikom oplošju, te što je model manji, koristimo manju rezoluciju. Koristeći programsko riješenje, u programskom jeziku Java, koje ćemo opisati u poglavlju 3., više nećemo morati pretpostaviti veličinu modela, nego s obzirom

na bilo koju duljinu objekta iz stvarnog svijeta, možemo izračunati površinu svih poligona koji opisuju 3D model. Programsko riješenje će također pokrivati slučaje udaljenosti kamere, koje vrlo lako možemo namjestiti.

Bitna priprema za korištenje 3D modela je skaliranje objekta na točnu veličinu kakav se on pojavljuje u stvarnom svijetu, ili po procjeni umjetnika u slučaju stiliziranih i fantastičnih svijetova. Točna veličina objekta u pogonskim sklopovima videoigara i ostalim programima za renderanje, stvara realističnu svijetlost i sijenu, te je jedan od bitnih faktora fotorealizma, koji se rijetko spominje pri učenju 3D-a. Kada imamo preciznu veličinu, algoritmi za odbijanje zraka svjetla poput "ray tracinga" i radiometrije daju realističnije rezultate.

Poglavlje 3

Predloženo rješenje

U nastavku ćemo objasniti kod za računanje 3D oplošja uz dodatne rubne slučajeve koji se baziraju na pojmovima koje smo do sada opisali, poput n-terokuta, trokuta, vrhova, te ćemo se osvrnuti na neke nove pojmove poput degeneriranih poligona, triangulacije i diskretnog skaliranja tekstura. Također, objasniti ćemo parsiranje 3D modela putem *OBJ* formata koji je opisan *ASCII* zapisom.

Kako ovaj rad ne bi zagušili sa nepotrebnim informacijama, kod za korisničko sučelje, uvoz i izvoz slika i za vizualne povratne informacije nećemo opisivati.

3.1 Računanje oplošja 3D objekata

3.1.1 Parsiranje OBJ datoteka

Za parsiranje OBJ datoteka, metodi *calculateSurfaceArea()* unutar klase *ObjSurfaceAreaCalculator* ćemo poslati OBJ datoteku *objFile* kao običan tekst. Zatim, dokada još teksta postoji, čitamo svaku liniju koja započinje sa slovom "v" što označava vrhove poligona, te u polje vrhova zapisujemo X, Y i Z koordinate, dijeleći ih u odnosu na razmake i tabulatore, što god stoji između vrijednosti. Također, svaki poligon odnosno lice označeno sa slovom "f" zapisujemo u polje lica, rastavljajući zapis na isti način kao i kod vrhova, zanemarujući slovo "f" i znak "/" te namještamo da svaki indeks polja lica korespondira indeksu polja vrhova gdje su X, Y i Z koordinate

zapisane.

```
public static double calculateSurfaceArea(File objFile)
   throws FileNotFoundException, IOException {
    List<double[]> vertices = new ArrayList<>();
    List<int[]> faces = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new
       FileReader(objFile))) {
        String line;
        while ((line = reader.readLine()) != null) {
                if (line.startsWith("v ")) {
                    String[] parts = line.split("\\s+");
                    double x = Double.parseDouble(parts[1]);
                    double y = Double.parseDouble(parts[2]);
                    double z = Double.parseDouble(parts[3]);
                    vertices.add(new double[]{x, y, z});
                } else if (line.startsWith("f ")) {
                    String[] parts = line.split("\\s+");
                    int[] faceVertices = new int[parts.length
                        - 1];
                    for (int i = 0; i < faceVertices.length;
                       i++) {
                        String vertexData = parts[i + 1].
                           split("/")[0];
                        faceVertices[i] = Integer.parseInt(
                           vertexData) - 1;
                    }
                    faces.add(faceVertices);}}}
```

3.1.2 Triangulacija N-terokuta

Nakon parsiranja informacija datoteke, moramo znati da ćemo kasnije računati oplošje modela kao sumu površina svakog triangla. Triangulacijom eliminiramo svaki

potencijalni četverokut i n-terokut. Tako, unutar iste metode *calculateSurfaceArea()* zapisujemo:

```
double totalSurfaceArea = 0.0;
for (int[] face : faces) {
    if (face.length >= 3) {
    List<int[]> triangles = triangulatePolygon(vertices, face
        );
    for (int[] triangle : triangles) {
        double[] v1 = vertices.get(triangle[0]);
        double[] v2 = vertices.get(triangle[1]);
        double[] v3 = vertices.get(triangle[2]);
        totalSurfaceArea += calculateTriangleArea(v1, v2, v3)
            ;
        }
    }
}
```

Nakon triangulacije ćemo poslati vrijednosti svih trokuta u metodu *calculateTriangleArea*, klase *ObjSurfaceAreaCalculator*, kako bi izračunali oplošje cijelog modela. To ćemo spomenuti kasnije u podpoglavlju 3.1.3.

Triangulaciju vršimo unutar metode *triangulatePolygon()* koja kao argumente prima liste vrhova i poligona koje smo zapisali pri parsiranju OBJ datoteke. Triangulacija je jednostavna provjera gdje poligone koji su već triangli dodajemo u listu triangla, dok za sve poligone koji imaju više od 3 vrha iteriramo kroz vrhove, odnosno indekse poligona, spajajući vertekse na pozicijama "i", te "i+1" sa nultim verteksom poligona.

```
public static List<int[]> triangulatePolygon(List<double[]>
   vertices, int[] polygon) {
    List<int[]> triangles = new ArrayList<>();
    int n = polygon.length;
    if (n == 3) {
        triangles.add(polygon);
        return triangles;
        }
    if (n > 3) {
    for (int i = 1; i < n - 1; i++) {
        triangles.add(new int[]{polygon[0], polygon[i],
           polygon[i + 1]});}
        }
    return triangles;
    }
}
```

Nakon što se metoda izvrši, spremni smo za računanje oplošja 3D objekta.

3.1.3 Računanje površine trokuta

Oplošje 3D modela računamo metodom *calculateTriangleArea()*, klase *ObjSurfaceAreaCalculator*, koja kao argument prima tri vrha trokuta. Podsjetimo se, unutar polja poligona nalaze se X, Y i Z koordinate trokuta. S obzirom da radimo s koordinatama u 3D prostoru, kako bi izračunali površinu trokuta potrebno je koristiti se operacijom vektorskog umnoška. U matematici, sljedećim izrazom definiramo računanje površine trokuta, gdje su "u" i "v" dvije stranice trokuta.

$$P = \frac{1}{2}|\mathbf{u} \times \mathbf{v}| \tag{3.1}$$

Najjednostavniji pristup u programiranju je prvo izračunati stranice trokuta *side1* i *side2*. Prva stranica se izračunava kao vektor prvog i drugog vrha, a druga stra-

nica kao vektor prvog i trećeg trokuta. Zatim se računa vektorski produkt te dvije stranice. U konačnici pratimo matematičku formulu i množimo vektorski produkt sa 0.5.

```
public static double calculateTriangleArea(double[] v1,
   double[] v2, double[] v3) {
    double[] side1 = new double[]{v2[0] - v1[0], v2[1] - v1
       [1], v2[2] - v1[2]};
    double[] side2 = new double[]{v3[0] - v1[0], v3[1] - v1
       [1], v3[2] - v1[2]};

    double[] crossProduct = new double[]{
            side1[1] * side2[2] - side1[2] * side2[1],
            side1[2] * side2[0] - side1[0] * side2[2],
            side1[0] * side2[1] - side1[1] * side2[0]
        };
    double area = 0.5 * Math.sqrt(crossProduct[0] *
       crossProduct[0]
            + crossProduct[1] * crossProduct[1]
            + crossProduct[2] * crossProduct[2]);
    return area;
}
```

Naposljetku u metodu *calculateSurfaceArea()* vrati se vrijednost površine svakog triangla. No vrijednost koja se na kraju zbroji u program za 3D objekte poznatog oplošja nije točna i puno veća nego očekivana. Naime, format *OBJ* prema zadanim postavkama svoje vrhove pohranjuje naopako [19], kako ne bi morao još eksplicitno navoditi smjer pogleda lica. U takvom slučaju, najlakše je riješenje svaki rezultat podijeliti s 4 jer nam je vektorski produkt 4 puta veći nego što bi trebao biti. Stoga u metodi *calculateSurfaceArea()* na kraju pišemo:

```
return totalSurfaceArea / 4;
```

Zanemarivanje degeneriranih poligona

U matematici degenerirani poligon je onaj koji ne čini "valjani" poligon zbog problema kao što su kolinearni vrhovi, nepostojana površina i samopresijecanje. Naime, u prirodi stvarnog svijeta i matematike, objekt u prostoru nikada nije savršeno ravan, no u svijetu računalne grafike, to je jako česta pojava. Pri modeliranju 3D objekata, umjetnici stalno izvlače nekoliko vrhova poligona prema isto smjeru, jer to dopušta lakši proces modeliranja velikih površina. Ti poligoni su često vrlo oštre prirode, te se zaglađivanja kojima poligoni prestaju biti degenerirane prirode (prema matematici) događaju kasnije u procesu, ili ponekad u potpunosti ostanu takvi pri izradi igara gdje je potrebno imati što manje poligona. Česta preporuka pri računanju površine 3D objekata, prema matematici je izbaciti sve degenerirane poligone iz računice, no u svijetu video igara to nije praktično i stoga za takve poligone nismo radili nikakve provjere.

3.2 Algoritam za skaliranje tekstura

U programskom riješenju, nakon što smo učitali teksturu i učitali površinu dodali smo tipku koja daje korisniku mogućnost unosa udaljenosti kamere od modela (više o tome na linku na kraju rada). Sada trebamo odlučiti se za način skaliranja. Ovome problemu moramo pristupiti empirijski, pošto ovdje nema apsolutno točnog riješenja, nego mu se možemo samo potencijalno približiti različitim tipovima diskretnog skaliranja i testiranja. Također, postavljanje određenih eksponencijalnih funkcija opadanja nije se pokazalo u inicijalnom testiranju kao pravi put za skaliranje tekstura. Teksture koje tako diskretno skaliramo, u programskom riješenju prošle su kroz stotine testova do pisanja ovoga rada.

Baza skaliranja je takva da je program postavljen da kao najveću moguću rezoluciju uzima 4096×4096, zato što je to u 2023. godini trenutno maksimum kod velikih igara. Različito modificiranje igara može doprinjeti daljnjim detaljim, no to je često vrlo nepotrebno i jako narušava performanse igre, iako su često takve modifikacije rađene na velikim objektima.

Diskretno skaliranje započinjemo tako, gdje zamišljamo da će završna tekstura

biti visoke rezolucije. Stoga u metodi skaliranja, globalnim varijablama koje smo deklarirali na početku programa zadajemo vrijednosti:

```
newImageResolutionX = 4096;
newImageResolutionY = 4096;
```

Nakon stotine testova granica prijelaza kamere i različitih faktora skaliranja. U srži, kada je 3D objekt udaljen dalje od kamere, iteriramo po novostvorenoj vrijednosti rezolucije sa različitim faktorima. Kod udaljenosti kamere većih od 5 metara udaljenosti, ponvaljamo iteracije više puta nego što je korisnik unio kako bi točno skalirali na granicama prijelaza različitih opadanja rezolucije. Također, na velikim udaljenostima, kako se vrijednost ne bi urušila u nulu, pri svakoj iteraciji dodali smo proizvoljnu vrijednost koja se pokazala najbolja kroz desetke modela različitih veličina i udaljenosti.

```
if (cameraDistance <= 5) {
    // Linearno skaliranje, brzo
    for (int i = 0; i <= cameraDistance; i++) {
        newImageResolutionX = (int) (newImageResolutionX /
            1.26);
        newImageResolutionY = (int) (newImageResolutionY /
            1.26);}}
if (cameraDistance > 5 && cameraDistance <= 20) {
    // Linearno skaliranje, sporije
    for (int i = 0; i <= cameraDistance + 4; i++) {
        newImageResolutionX = (int) (newImageResolutionX /
            1.15);
        newImageResolutionY = (int) (newImageResolutionY /
            1.15);}}




if (cameraDistance > 20) {
```

```
    // Linearno skaliranje, još sporije
    for (int i = 0; i <= cameraDistance + 25; i++) {
        newImageResolutionX = (int) (newImageResolutionX /
            1.1);
        newImageResolutionX += 7;
        newImageResolutionY = (int) (newImageResolutionY /
            1.1);
        newImageResolutionY += 7;}}
if (cameraDistance > 35) {
    // Linearno skaliranje, najsporije
    for (int i = 0; i < cameraDistance + 26; i++) {
        newImageResolutionX = (int) (newImageResolutionX /
            1.05);
        newImageResolutionX += 5;
        newImageResolutionY = (int) (newImageResolutionY /
            1.05);
        newImageResolutionY += 5;}}
```

Na vrijednostima puno većim od 35 metara udaljenosti, veličina teksture se nikada ne ruši u nulu, te njena najmanja vrijednost korespondira specifično veličini 3D modela. Nakon virtualnog urušavanja vrijednosti rezolucije, skaliramo ju s obzirom na oplošje 3D modela koje smo opisali u poglavlju 3.1.

```
// +1 vrijednost na varijabli oplošja spriječava negativno
// skaliranje u slučaju da radimo sa modelima oplošja
// manjim od 1 kvadratnog metra
newImageResolutionX = (int) (newImageResolutionX * (
   surfaceArea + 1));
newImageResolutionY = (int) (newImageResolutionY * (
   surfaceArea + 1));
```

Rubni slučajevi

Postoje dva rubna slučaja o kojima se moramo brinuti. Jedan je slučaj da skaliranjem pređemo rezoluciju dimenzija od 4096. Ako je takav slučaj, to znači da je najveća rezolucija slike od 4096 u redu za dani model.

```
// Radimo provjeru jedne dimenzije, jer su teksture 3D modela
   ,
// prema standardu 3D industrije, uvijek četverokutne
if (newImageResolutionX > 4096) {
    newImageResolutionX = 4096;
    newImageResolutionY = 4096;
}
```

Drugi slučaj je ako po procjeni umjetnika ili određenih limitacija programa, ne želimo imati teksturu veću od originalne nakon skaliranja, u slučaju da je veća rezolucija po standardu 4096 rezolucije, tada skaliramo teksturu na toj udaljenosti natrag na rezoluciju koju je umjetnik postavio.

```
// Ponovo radimo samo provjeru jedne dimenzije
if (newImageResolutionX > imageResolutionX) {
    newImageResolutionX = imageResolutionX;
    newImageResolutionY = imageResolutionY;
}
```

Nakon virtualnog skaliranja rezolucije, primjenjujemo dobivenu rezoluciju na originalnu teksturu koju smo unijeli u program putem *scaleImage()* metode koja kao argument uzima originalnu sliku koju smo unjeli, te skalirane dimenzije. Zatim stvaramo grafiku od slike unutar koje možemo uređivati sliku različitim metodama iz Javinih knjižnica. Metoda *setRenderingHint()* određuje kvalitetu slike pri skaliranju, te kao argumente prima "KEY_INTERPOLATION" i "VALUE_INTERPOLATION_BILINEAR" koja specificira da je ključ interpolacije bilinearna interpolacija koja nam daje glatke prijelaze između piksela pri skaliranju slike. Zatim metodom *graphics2D.drawImage()* napokon skaliramo teksturu i zatim uništavamo prostor za grafiku metodom *graphics2D.dispose()* i vraćamo skaliranu sliku.

```
public BufferedImage scaleImage(BufferedImage originalImage,
    int targetWidth, int targetHeight) {
     BufferedImage resizedImage = new BufferedImage(
         targetWidth, targetHeight, BufferedImage.TYPE_INT_RGB)
         ;
     Graphics2D graphics2D = resizedImage.createGraphics();
     graphics2D.setRenderingHint(RenderingHints.
         KEY_INTERPOLATION, RenderingHints.
         VALUE_INTERPOLATION_BILINEAR);
     graphics2D.drawImage(originalImage, 0, 0, targetWidth,
         targetHeight, null);
     graphics2D.dispose();
     return resizedImage;
}
```

Metodu *scaleImage()* pozivamo pri klikom na gumb izvoz iz programa, kako bi spremili skaliranu teksturu. Korisniku također omogućujemo hoće li datoteku spremiti kao format *PNG* ili kompresirani format *JPG*. U nastavku ćemo pogledati rezultate koristeći oba formata.

Poglavlje 4

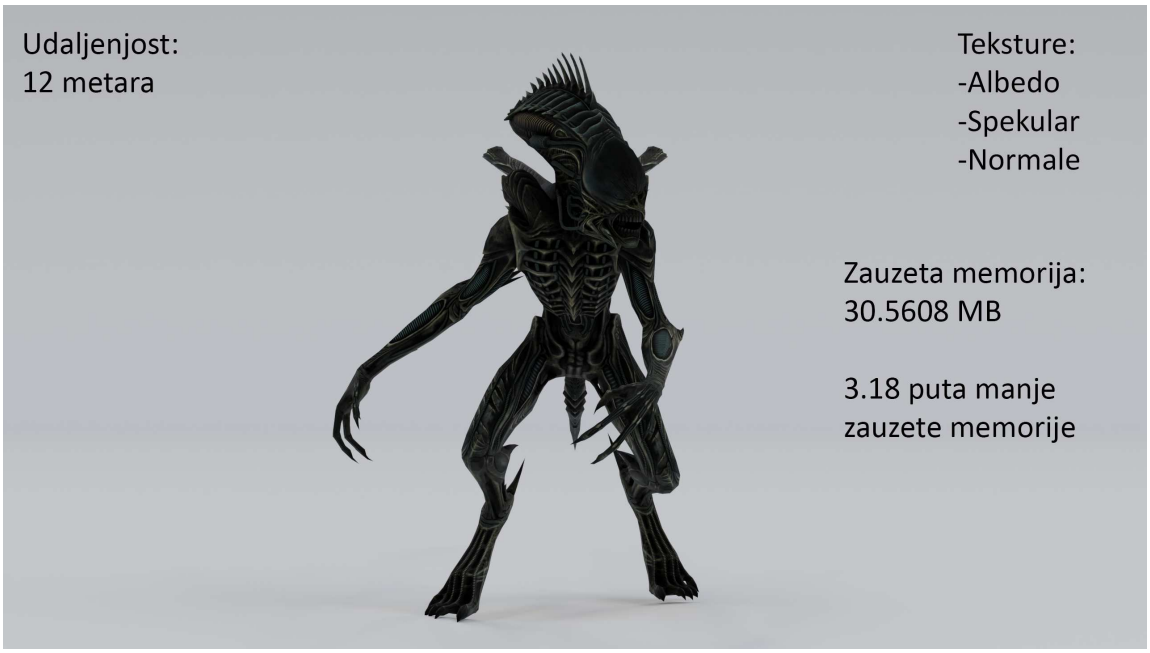
Rezultati

Za prikaz rezultata programskog rješenja, koristiti ćemo Blender Cycles renderer (https://www.blender.org/) za statičke rezultate, te Unity renderer (https://unity.com/) za rezultate u stvarnom vremenu. U programu Blender, rezolucija kojom prikazujemo slike je 2560×1440, kako je to neka gornja granica rezolucije većinu monitora za igranje videoigara.

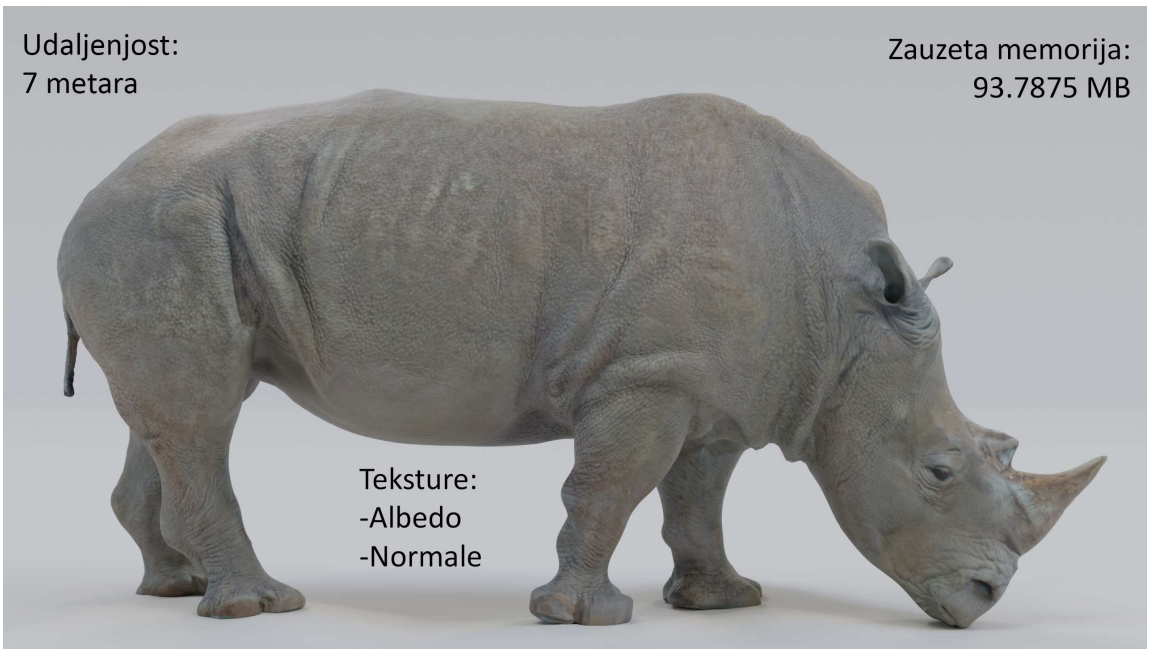
4.1 Cycles renderer

Cycles je popularan renderer otvorenog koda korišten u programu za 3D, Blender. Temeljen je na fizici, čineći ga moćnim alatom za stvaranje realističnih i visokokvalitetnih rendera. Pruža napredne funkcije osvjetljenja putem ray-tracinga, odnosno praćenja i odbijanja zraka svjetlosti. Također podržava visokodinamično snimanje slike (HDRI) za realistično osvjetljenje okoline i globalnu iluminaciju koja doprinosi točnim i prirodnim renderima. Cycles može koristit CPU i GPU za renderiranje, ovisno o korisničkom odabiru. Kod malog uzoraka zraka svjetlosti, koristi se algoritmima za smanjenje šuma kako bi održao i prikazao kvalitetnu sliku [20].

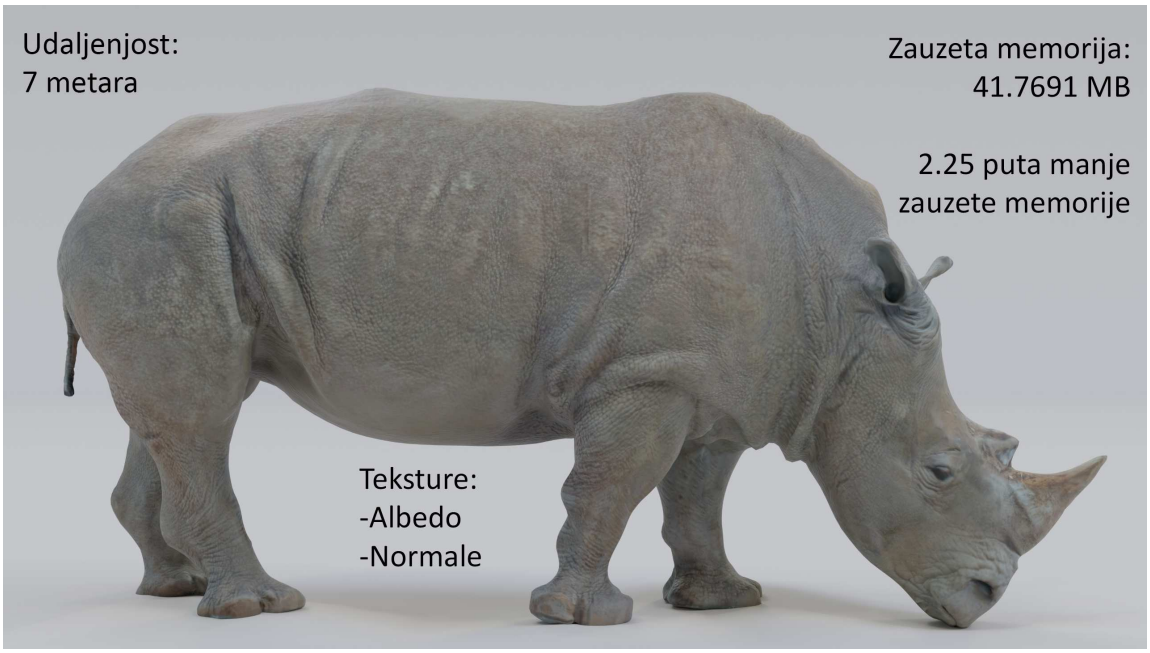
Slika 4.1 3D model Ksenomorfa prema odabiru veličina tekstura umjetnika



Slika 4.2 3D model Ksenomorfa nakon skaliranja tekstura programskim rješenjem



Slika 4.3 3D model nosoroga prema odabiru veličina tekstura umjetnika



Slika 4.4 3D model nosoroga nakon skaliranja tekstura programskim rješenjem

Slika 4.5 3D model kruha prema odabiru veličina tekstura umjetnika

Slika 4.6 3D model kruha nakon skaliranja tekstura programskim rješenjem

Komentar rezultata

Iz dosadašnjih primjera vidimo da su rezultati vrlo uspješni, no bitno je zamijetiti da pri skaliranju tekstura na površinama gdje su vrlo tanke linije piksela tamno sive i crne nijanse, te nijanse postaju dominante pri mješanju s okolnim pikselima, zamračivajući površinu. To najbolje možemo primjetiti ako iz dosadašnjih primjera usporedimo tubu na strani Ksenomorfove glave i prostor ispod naboravanja kože nosoroga na prednjoj desnoj nozi. Razlika je dovoljno mala, da promjenu ne moramo smatrati artefaktom. Primjer 3D modela kruha je ostao identičan i jako memorijski impresivan, te postavlja pitanje češćeg korištenja JPG formata u videoigrama. Bez *JPG* kompresije, teksture svejedno postaju oko 4 puta manje u njihovoj veličini na primjeru kruha. Naravno, čak i u igrama s malo detalja, format *JPG* ne može prevladati u potpunosti, zbog alfa kanala formata *PNG* koji dopušta transparentnost i često se koristi za prikaz listova biljki, i jednostavnih modela kojima putem transparentosti možemo prevariti igrača videoigre da misli da je površina detaljnija nego što zapravo jest. Pri izradi tekstura, vrijedi isprobati JPG format kako bi smo dobili memorijski još puno bolji rezultat, no umjetnik mora biti oprezan u svojoj procjeni formata.

4.2 Unity renderer

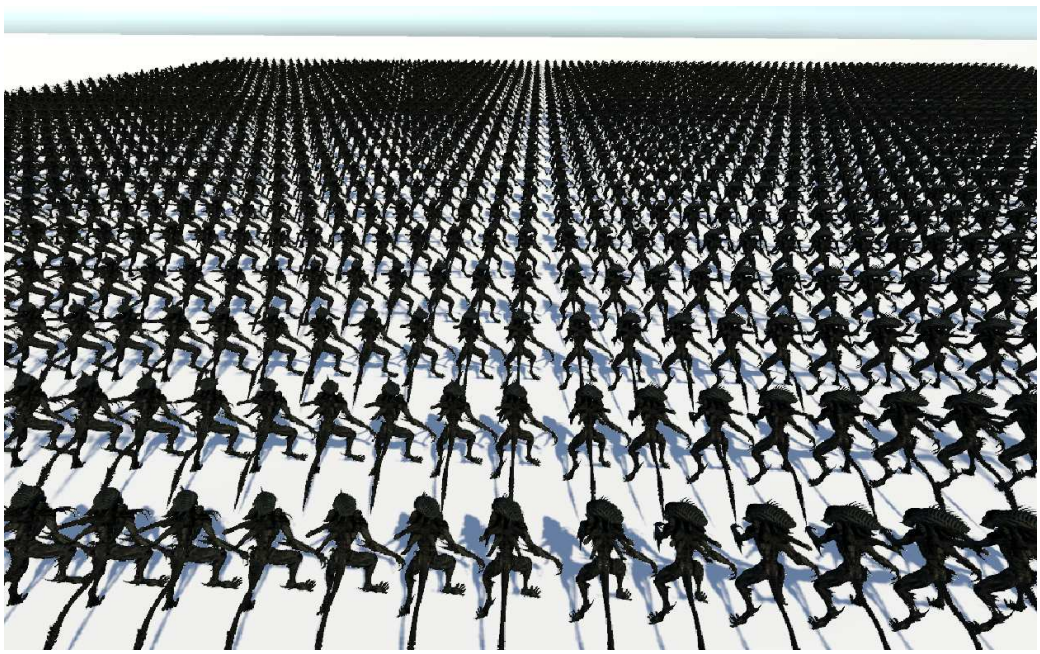
Unity koristi vlastiti ugrađeni renderer nazvan "Unity Rendering Pipeline" za renderiranje 3D grafike u stvarnom vremenu. Unity nudi nekoliko oblika prikaza u stvarnom vremenu, svaki dizajniran za različite platforme, zahtjeve učinkovitosti i grafičke značajke [21].

Kod prikaza 3D modela u stvarnom vremenu, bitno je napomenuti da poligoni od kojih je 3D model izgrađen su puno zahtijevniji za procesirati, te promjene u performansama ne možemo izmjeriti kod samo jednog modela ako skaliramo teksture. Ako zauzmemo malo memorije, broj slika po sekundi se kreće u tisućama na modernim komponentama i jako oscilira. Iz tog razloga ćemo u Unity primjerima, napuniti scenu sa 3D modelima dok se broj slika u sekundi ne stabilizira, ispod 165 slika, te usporediti performanse sa i bez optimizacije tekstura.

Komponentne uređaja na kojemu testiramo scene u Unity-u:

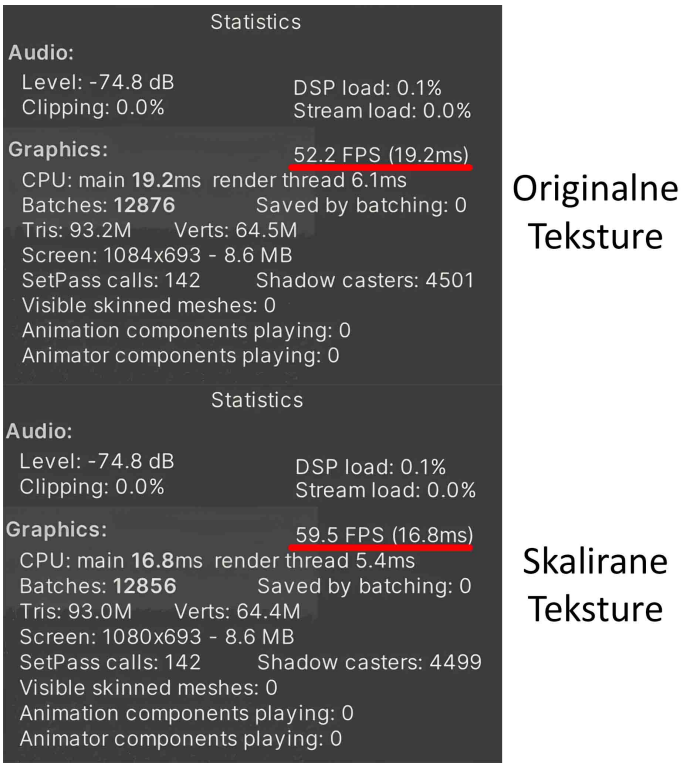
```
CPU: AMD Ryzen 9 5900HX with Radeon Graphics 3.2GHz
GPU: RTX 3080 Laptop
RAM: 32 GB
OS: Windows 11 Ghost Spectre Superlite (Custom OS)
```

U sljedećoj sceni, u stvarnom vremenu učitavamo veliku količinu 3D modela Ksenomorfa sa slika 4.1 i 4.2.



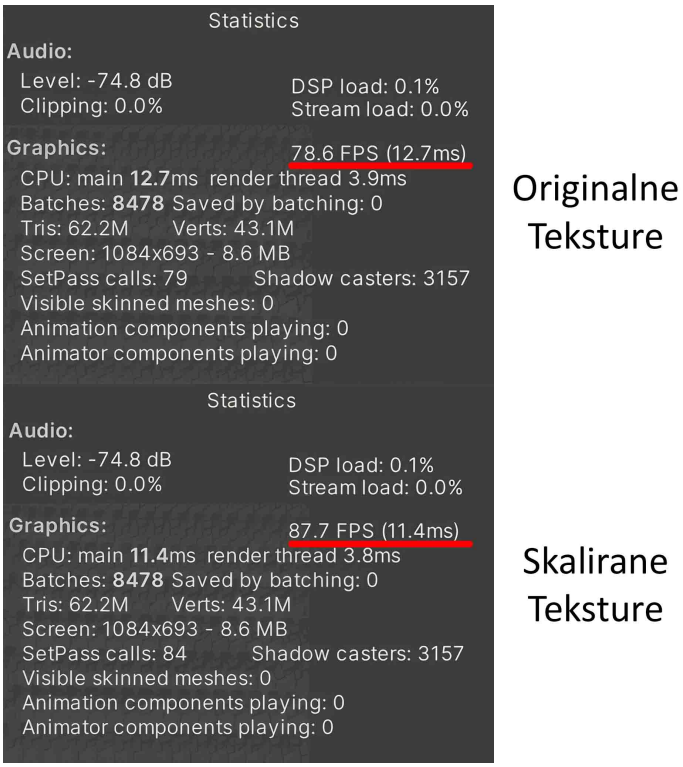
Slika 4.7 Scena puna Ksenomorfa spremna za testiranje

Iako ovako veliki broj 3D modela visoke rezolucije nije primjeren za igre gdje je kamera na ovoj udaljenosti, vrlo je dobar primjer da brzo prikažemo rezultate skaliranja tekstura na skali jedne jako detaljne videoigre.



Slika 4.8 Rezultati testiranja u Unity-u #1

U trenutku snimanja informacija na rezultatima se učitao različiti broj poligona (0.215% manje poligona u statistici skaliranih tekstura),no svejedno možemo vidjeti da na ogroman broj informacija dobivamo u prosjeku čak 7.3 slika u sekundi više pri optimizaciji. Bitno je napomenuti da u ovom primjeru se uopće nismo osvrtali na pravu udaljenost kamere od objekta, nego smo koristili veličine tekstura iz primjera slike 4.2 gdje Ksenomorf iz blizine izgleda detaljno. Daljnim stvaranjem razina detalja koji će se dinamično izmjenjivati pri različitim udaljenostima, možemo drastično povećati perfromanse videoigara. To ćemo pokazati u sljedećem primjeru, kada snimamo cijelu scenu iz zraka.



Slika 4.9 Rezultati testiranja u Unity-u #2

Iz veće udaljenosti, uspijeli smo skalirati teksture još više kako bi postigli još bolji rezultat. No, kako vidimo da su slike u sekundi porasle, očito je i sa usporedbom brojem poligona u odnosu na prošle rezultate da je Unity sam decimirao količinu poligona. U ovom testu, svi podatci o sceni su jednaki, no uspjeli smo dobiti 9.1 slika u sekundi više na udaljenosti od 17 metara od modela.

4.3 Više razina detalja

U ovom poglavlju diskutirati ćemo generiranje statične razine detalja koje vrlo brzo možemo kreirati sa programskim rješenjem, djelomično eliminirajući dva nedostatka takve vrste razine detalja. Započeti ćemo stvaranjem nulte razine detalja s obzirom na udaljenost kamere gdje cijela stolica stane u kadar i s obzirom na oplošje stolice.

Slika 4.10 3D model stolice prema odabiru veličina tekstura umjetnika

Slika 4.11 3D model stolice nakon skaliranja tekstura programskim rješenjem

Iz priloženoga, u slikama 4.10 i 4.11 vidimo da skaliranje nije promijenilo izgled teksture koje ljudsko oko može zamijetiti. Slika 4.11 prikazuje skaliranu verziju tekstura na nultoj razini. Skaliranu teksturu programskim rješenjem stvorili smo na udaljenosti 3 metra od centra težišta stolice. Zatim, možemo izmjenom udaljenosti na 6, 12 i 24 metra stvoriti još tekstura koje će nam služiti kao razine niže rezolucije između kojih dinamično možemo prelaziti u stvarnom vremenu. Različite udaljenosti uzimamo proizvoljno, prema potrebama našeg projekta. U svrhu prikazivanja rezultata veličine datoteka, proizvoljno smo odabrali ove udaljenosti. U nekoliko klikova, mijenjanjem opcija udaljenosti, putem programskog riješenja, generiramo 3 nove slike za svaki tip teksture.

Već smo prijašnjim primjerom Ksenomorfa u Unityu zaključili da veličina tekstura poprilično utječe na broj slika u sekundi. Manjim teksturama, odnosno nižim razinama taj rezultat postaje još bolji. Usporedimo veličine datoteka tekstura koje je umjetnik odabrao za nultu razinu sa nultom razinom skaliranih tekstura i novo generiranim razinama.

Tablica 4.1 Statistika originalnih tekstura stolice

Tip teksture	Razina detalja	Zauzeta memorija (KB)	Rezolucija
Albedo	0	9799	4096 × 4096
Hrapavost	0	3546	4096 × 4096
Normala	0	22042	4096 × 4096

Ukupna zauzeta memorija (KB)
35387

Tablica 4.2 Statistika skaliranih tekstura stolice

Tip teksture	Razina detalja	Zauzeta memorija (KB)	Rezolucija
Albedo	0	2570	2058 × 2058
Hrapavost	0	1379	2058 × 2058
Normala	0	4154	2058 × 2058
Albedo	1	815	1112 × 1112
Hrapavost	1	430	1112 × 1112
Normala	1	1309	1112 × 1112
Albedo	2	171	478 × 478
Hrapavost	2	88	478 × 478
Normala	2	270	478 × 478
Albedo	3	17	133 × 133
Hrapavost	3	9	133 × 133
Normala	3	26	133 × 133

Ukupna zauzeta memorija (KB)
11238

Iz priloženoga vidimo 3 teksture napravljene od strane umjetnika zauzimaju ukupno 35.387 MB memorije, dok sve razine kod skaliranog slučaja (ukupno 12 tekstura), zauzimaju 11.238 MB, što je 315% manje zauzetog prostora na disku. Time ne samo da eliminiramo problem statičnih razina tekstura o zauzimanju više memorije od dinamičnih razina tekstura, već smanjujemo veličinu igre i dobivamo sve prednosti optimizacije koje nam statične razine pružaju. Bitno je napomenuti da programsko rješenje ne koristi kompresiju formata *JPG*, već smo izvezli teksture u istom formatu kao originalne teksture (*JPG*), bez gubitka kvalitete. Naravno, prema potrebi, faktor kompresije se može promijeniti. Sa većom kompresijom, možemo postići još bolje rezultate, no moramo biti vrlo oprezni kako ne bi izgubili previše informacija. Rezultati će biti različiti ovisno o veličini modela, i kod površinski malih modela dobiti ćemo najbolje rezultate. Što se tiče većih modela poput cijelih zgrada i kuća u videoigrama, u posljednje vrijeme proces modularne izgradnje istih, od manjih modela postao je jako popularan - time i kod takvih konstrukcija možemo dobiti jako visoku razinu kvalitete uz jako malo zauzetog prostora, i visokog broja slika po sekundi [22].

Poglavlje 5

Zaključak

Iako iz godine u godinu programeri razvijaju različite metode za optimizaciju tekstura, očito je da kako bi postigli što veće performanse uz što veću kvalitetu, moramo pažljivo pristupiti svakom 3D modelu. U svijetu tekstura, 3D modela i videoigri, ne postoji univerzalno riješenje za sve probleme, no dobrim shvaćanjem dane teme, nije teško ići generalno ispravnim putem. Nažalost, kao ljudi, često ćemo napraviti krivu procjenu, ali tada nam programska rješenja mogu pomoći i ispraviti nas. S obzirom da na temu veličine 3D objekata, njihove površine i veličine tekstura nema sveopćeg pravila, emprijski pristup rješenjima je jedini koji nas istraživanjem može dovesti do ravnoteže kvalitete i optimizacije. Stotinama iteracijama korištenjem programskog riješenja, vidimo da je moguće stvoriti jednake detalje tekstura kao što je umjetnik zamislio, uz 2-5 puta manje memorije, dobivajući do desetak više slika po sekundi u videoigrama kada pažljivo pristupimo svakom 3D modelu. Ako stavimo inicijalnu premisu u perspektivu, gdje teksture zauzimaju do sto puta više memorije od 3D modela u današnjim AAA igrama, vidimo da ovom tehnikom također možemo smanjiti veličinu videoigara upola s obzirom da u takvim igrama 30 do 50 posto memorije čine samo teksture. U današnje vrijeme prevladava ideja da su računalne komponente dovoljno moćne da bi procesirale gomilne količine informacije, i dok je to istina, vrijedno je zamijetiti da se u proizvodnji video igara često žrtvuje optimizacija kako bi se uštedilo na vremenu; no u samo par klikova mišem, s pravim programskim rješenje, možemo dopustiti da čak i vrlo "slabe" računalne komponente u stvarnom vremenu prikazuju vrlo detaljne svjetove.

Bibliografija

[1] J. Dobrilović, "Proces," Ph.D. dissertation, University of Zagreb. Academy of Fine Arts. Department of Art Education, 2020.

[2] K. Hormann and M. S. Floater, "Mean value coordinates for arbitrary planar polygons," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 4, pp. 1424–1441, 2006.

[3] W. Brainerd, "Catmull-clark subdivision surfaces," *GPU Pro 7: Advanced Rendering Techniques*, 2016.

[4] M. Rossoni, S. G. Barsanti, G. Colombo, G. Guidi *et al.*, "Retopology and simplification of reality-based models for finite element analysis," *Computer-Aided Design and Applications*, vol. 17, no. 3, pp. 525–546, 2020.

[5] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz, "Designing a pc game engine," *IEEE Computer Graphics and Applications*, vol. 18, no. 1, pp. 46–53, 1998.

[6] A. Paquette, "Clean geometry," *Computer Graphics for Artists: An Introduction*, pp. 33–64, 2008.

[7] P. Pokornỳ and M. Birošík, "The preparation of graphic models for a virtual reality application in unity," in *Intelligent Algorithms in Software Engineering: Proceedings of the 9th Computer Science On-line Conference 2020, Volume 1 9*. Springer, 2020, pp. 331–340.

[8] H. P. Lensch, M. Goesele, Y.-Y. Chuang, T. Hawkins, S. Marschner, W. Matusik, and G. Mueller, "Realistic materials in computer graphics," in *ACM SIGGRAPH 2005 Courses*, 2005, pp. 1–es.

[9] A. Roth and D. Lichtman, "The community game development toolkit," in *Proceedings of the 28th ACM Symposium on Virtual Reality Software and Technology*, 2022, pp. 1–2.

[10] K. Halladay and K. Halladay, "Normal mapping," *Practical Shader Development: Vertex and Fragment Shaders for Game Developers*, pp. 181–200, 2019.

[11] L. Flavell, "Uv mapping," in *Beginning Blender: Open Source 3D Modeling, Animation, and Game Design*. Springer, 2010, pp. 97–122.

[12] A. L. Possemiers and I. Lee, "Fast obj file importing and parsing in cuda," *Computational Visual Media*, vol. 1, pp. 229–238, 2015.

[13] H. Thomas, "Simplicity from complexity," *The APPEA Journal*, vol. 56, no. 2, pp. 542–542, 2016.

[14] T. Nöll and D. Strieker, "Efficient packing of arbitrary shaped charts for automatic texture atlas generation," in *Computer Graphics Forum*, vol. 30, no. 4. Wiley Online Library, 2011, pp. 1309–1317.

[15] P. K. Krause, "Texture compression," 2007.

[16] J. Moritz, S. James, T. S. Haines, T. Ritschel, and T. Weyrich, "Texture stationarization: Turning photos into tileable textures," in *Computer Graphics Forum*, vol. 36, no. 2. Wiley Online Library, 2017, pp. 177–188.

[17] W. H. De Boer, "Fast terrain rendering using geometrical mipmapping," *Unpublished paper, available at http://www. flipcode. com/articles/article geomipmaps. pdf*, 2000.

[18] D. Luebke, *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.

[19] G. Antonutto and A. McNeil, "Radiance primer," 2013.

[20] I. A. Astuti, I. H. Purwanto, T. Hidayat, D. A. Satria, R. Purnama *et al.*, "Comparison of time, size and quality of 3d object rendering using render engine eevee and cycles in blender," in *2022 5th International Conference of Computer and Informatics Engineering (IC2IE)*. IEEE, 2022, pp. 54–59.

[21] F. Messaoudi, G. Simon, and A. Ksentini, "Dissecting games engines: The case of unity3d," in *2015 international workshop on network and systems support for games (NetGames)*. IEEE, 2015, pp. 1–6.

[22] N. Statham, J. Jacob, and M. Fridenfalk, "Game environment art with modular architecture," *Entertainment Computing*, vol. 41, p. 100476, 2022.

Pojmovnik

BCn Block Compression. 12

CPU Centralna procesorska jedinica. 14

DXT DirectX Texture Compression. 12

ETC Ericsson Texture Compression. 12

GPU grafička procesorska jedinica. 1

JPG/JPEG Joint Photographic Experts Group. 4

LOD Level of Detail. 13

NURBS Non-Uniform Rational B-Spline. 3

OBJ Wavefront .obj format. 6

PNG Portable Network Graphics. 4

TIFF Tag Image File Format. 4

Sažetak

U ovom radu, proispitali smo generalne tehnike 3D industrije videoigara i aplikacije održavanja visokih performansi uz održavanje visoke kvalitete tekstura koje opisuju 3D modele. Također, proučili smo efekt oplošja 3D modela koji je često zanemaren, te inkorporirali ga u generalne tehnike, te stvorili programsko riješenje u programskom jeziku Java koji rješava zanemareni problem. Uz 3D modele, na kojima je najveći fokus u industriji videoigara kada govorimo o optimizaciji, uočili smo da su teksture najintenzivnije za procesirati. Uzimajući oplošje 3D objekata u obzir, vidjeli smo da je moguće stvoriti teksture jednake kvalitete, sa 2 do 5 puta manje memorije, ovisno o veličini modela, te daljnjom kompresijom smanjiti memoriju još više postižući isti efekt.

Ključne riječi **— Poligon, Tekstura, Retopologija, UV Mapa, Oplošje 3D objekta, Udaljenost Kamere, Razina detalja, Veličina datoteke, Memorija, Slike po sekundi**

Abstract

In this paper, we have examined general techniques of the 3D gaming industry and their applications for maintaining high quality of textures which define the color and attributes of 3D models. Additionally, we have studied the effect of 3D model area, which is often overlooked, and incorporated it into the general techniques. We have created a software solution in the Java programming language to address this neglected problem. Along with 3D models, which are the primary focus in the video game industry when it comes to optimization, we have noticed that textures are the most intensive to process. Taking the surface areas of 3D objects into consideration, we have seen that it is possible to create textures of equal quality with 2 to 5 times less memory, depending on the size of the model. Furthermore, by applying additional compression techniques, we can further reduce memory usage while achieving the same effect.

Keywords **— Polygon, Texture, Retopology, UV Map, Area of a 3D object, Camera Distance, Level of Detail (LOD), File size, Memory, Frames per second (FPS)**

Dodatak A

Vanjske poveznice

Potpuni kod programskog rješenja dostupan je na linku:

```
https://github.com/acvitkovic/Texture_Optimization
```

3D modeli korišteni u radu:

```
Ksenomorf: https://sketchfab.com/3d-models/xeno-raven-
    e444a88e999549d99eacb1ea0f8e04e4

Nosorog: https://sketchfab.com/3d-models/model-56a-southern-
    white-rhino-8e97b62a90f44ce19ea9e3fd421f55b4
```

Svi ostali modeli u vlasništvu su tvrtke Quixel:

```
https://quixel.com/
```

Napomena: Svi modeli korišteni u radu dostupni su za nekomercijalne svrhe.