Iscrtavanje 3D objekata koristeći Mesh Shader-e

Kodžoman, Leon

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

Sveučilišni studij

Iscrtavanje 3D objekata koristeći Mesh Shader-e

Diplomski rad

Leon Kodžoman

Osijek, 2023.

SADRŽAJ

1. UVOD

Iscrtavanje 2D i 3D objekata je u modernom svijetu postala svakodnevna stvar koja se može primijetiti na svakom koraku modernog života. Najveći primjer bi bila razna računala koje se koriste svakodnevno. Iako je velikom većinom svakodnevni život obilježen 2D grafikom on ne bi mogao biti moguć bez velike količine 3D grafike iza sebe. 3D grafika je nužan alat za mnogo ljudi koji ju koriste da bi dizajnirali razne proizvode koji će biti u uporabi u svakodnevnom životu. Djelatnosti poput arhitekture, 3D modeliranja, znanosti oko vizualizacije podataka, simulacija te naravno industrije poput video igara se ekstenzivno koriste 3D grafikom. Kako rastu zahtjevi i kompleksnosti stvari na kojima se radi u tim raznim područjima potrebno je smisliti tehnike iscrtavanja 3D grafike koje omogućuju više slobode te tehnike koje povećavaju performanse u svrhu boljeg iskorištavanja računalnih resursa. U ovom radu će se opisati implementacija te teorija koja stoji iza nove i moderne tehnike iscrtavanja grafike koja pruža veće performanse zbog više mogućnosti za optimiziranje. Da bi se opisala moderna tehnika s *mesh shaderima*, potrebno je opisati kako funkcionira klasično iscrtavanje da bi se opisali razlozi zašto se odlučilo pronaći zamjenu. Opis logike iscrtavanja je od najveće važnosti jer je logički pristup iscrtavanju najveća razlika koja razdvaja klasičnu i modernu tehniku. U prvoj polovici rada će se opisati kako funkcionira algoritam iscrtavanja u klasičnom načinu rada. Zatim će se opisati svaka faza iscrtavanja zasebno da bi se opisalo na što treba obratiti pozornost. Na kraju će biti ukratko opisano kako se iscrtavanje pokreće i što je potrebno da bi se to izvelo. Druga polovica rada će biti strukturno slična prvoj u smislu da je raspored tema jednak, ali sadržaj je vezan za modernu tehniku. Nakon opisa obje tehnike će se usporediti rezultati u raznim okolnostima.

1.1. Zadatak diplomskog rada

Osmisliti i implementirati novi, jednostavan sustav za iscrtavanje 3D objekata na bazi modernih Mesh Shadera. Osmisliti algoritme prebacivanja tradicionalnih mreža objekata u oblik pogodan za Mesh shader-e. Usporediti novi način iscrtavanja s već postojećim, tradicionalnim načinima iscrtavanja po raznim karakteristikama: kvaliteta, performanse, jednostavnosti korištenja itd.

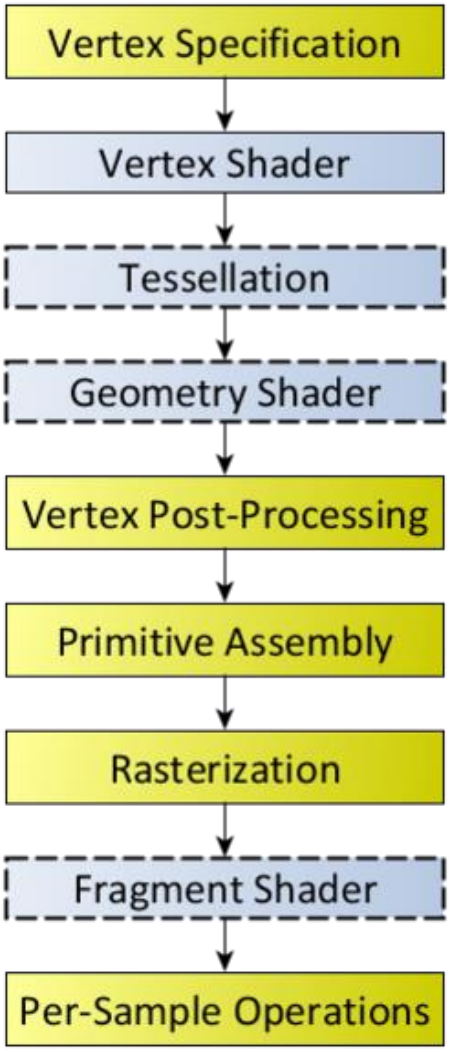
2. PREGLED PODRUČJA

Zbog činjenice da je tehnika koja će se opisati vrlo nova u svijetu grafike (predstavljeno prvi puta 2018. godine) ne postoji mnogo izvora iz kojih se mogu koristiti informacije vezane za tu tehniku. Najveći izvori za implementaciju u ovome radu su bile web stranice s dokumentacijom [1], web stranica od tvrtke Nvidia gdje je opisan opći princip rada tehnike [2] te jedan vrlo koristan github račun s primjerom koda za CAD svrhe [3]. Prethodno naveden github račun je bio vrlo koristan jer se u njemu nalaze primjeri koda. Jedini veći primjer te tehnike je Unreal Engine 5 od tvrtke Epic Games gdje je po prvi puta implementirana ta tehnika te da je dostupna javnosti [4]. Moderna tehnika iscrtavanja je deleko najfleksibilnija tehnika iscrtavanja koja trenutno postoji što znači da je velika vjerojatnost da će se kroz vrijeme razvijati metode korištenja koje trenutno ne postoje.

Slika 2.1 Prikaz detaljne geometrije u Unreal Engine 5

3. KLASIČNO ISCRTAVANJE

Područje računalne grafike se razvijalo velikom brzinom od svojih začetaka počevši pojavom prvih grafičkih sučelja u svrhu prikazivanja jednostavnog teksta na nekom obliku ekrana. Tek je pojavom dovoljno zahtjevnih programa došlo do potrebe da se na ekranima prikazuje nešto više od teksta. Kompleksniji programi koji se nisu oslanjali isključivo na tekst naišli su na problem iscrtavanja proizvoljnih oblika na nekakav ekran. Budući da proizvoljni oblici po definiciji nisu nužno jednostavni bilo je potrebno smisliti način pomoću kojega bi se takvi oblici prikazivali korisniku. Nakon ranih 2000. godina način iscrtavanja svih mogućih 2D i 3D oblika za potrebe grafike u stvarnom vremenu se ustalio i je postao standardan u cijelom području računalne grafike te će biti objašnjen ispod. Svi mogući objekti koji će se iscrtati se uvijek pretvore u trokute prije crtanja. Razlog tome je što je trokut najjednostavniji 3D oblik za iscrtati. Nije važno kakav je objekt ili što predstavlja, uvijek su prikazani trokutima. Ako objekti nisu prikazani trokutima onda se rastave na trokute te se tako nastavi dalje. Područje računalne grafike se velikom većinom koristi isključivo izrazima na engleskom jeziku te radi činjenice da za velik broj naziva ne postoje pravilni prijevodi koristiti će se engleski nazivi i u ovom radu. Budući da ima velik broj naziva, za svaki će biti objašnjeno što predstavlja. Način iscrtavanja je postao standardiziran te prati nekoliko faza prikazanih na slici 3.1 [5]. U ovom radu će se koristiti OpenGL API zbog svoje jednostavnosti i mogućnosti da se s njime implementira sve što će se koristiti u ovom radu.

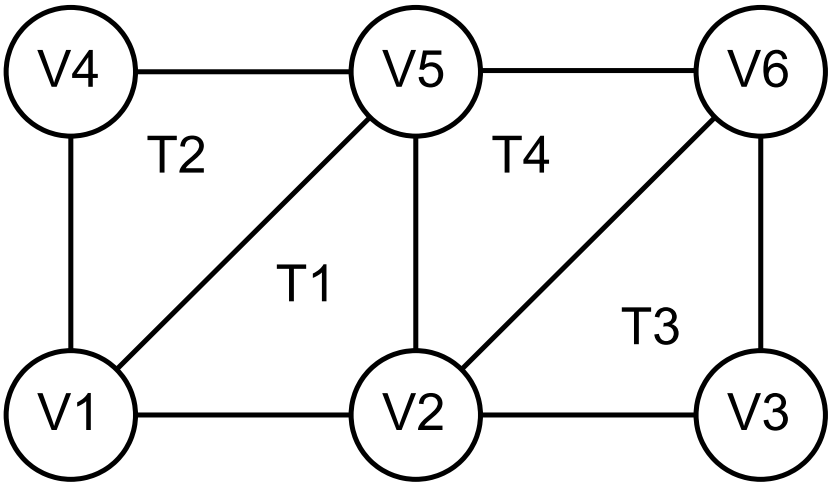


Slika 3.1 Redoslijed faza iscrtavanja 2D i 3D objekata

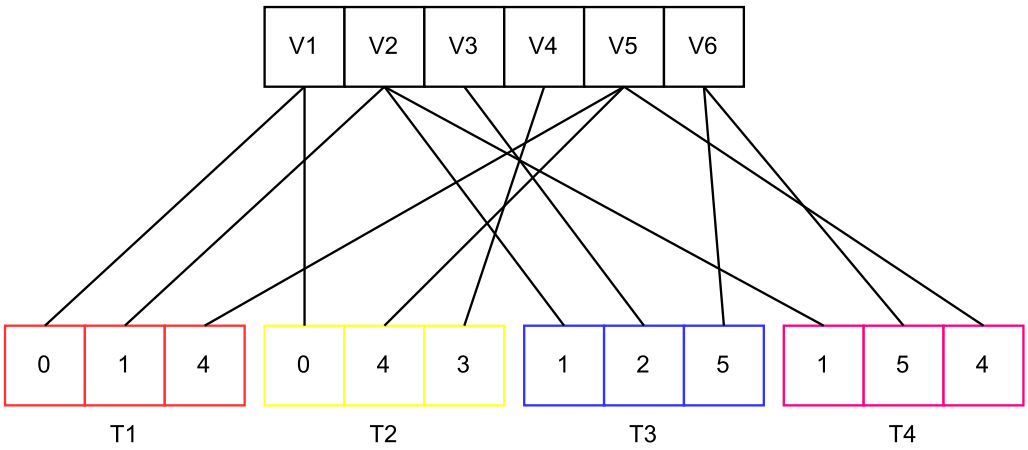
Iscrtavanje grafike se izvršava na zasebnom hardware-u koji se zove grafička kartica. Grafičke kartice se zasebne komponente čija je cijela svrha postojanja obavljanje poslova vezanih za grafiku poput prikazivanja teksta i raznih objekata. Postojanje grafičkih kartica podrazumijeva da moraju postojati programi koji se mogu izvršavati na njima. Ti programi se zovu *shader*-i (engl. *shader*). Tako su imenovani jer potiču od izraza „to shade“ što znači sjenčati jer su se ti programi standardizirali kada je sjenčanje postajalo aktualno. Na slici 3.1 se može vidjeti kako izgleda proces transformiranja nekog geometrijskog oblika iz čistih podataka sve do završne slike. Značajni dijelovi iz perspektive programera koji piše program su samo svijetlo plavi dijelovi, neki od ostalih dijelova su značajni te će se spomenuti kasnije, no trenutno nisu važni. Cijeli proces započinje *vertex shaderom*, zatim se rezultati toga prebacuju u *tessellation shadere* (na slici su prikazani kao jedan blok, ali zapravo su dva zasebna *shadera*) koji svoj rezultat prebacuju u *geometry shader* te rezultati toga na kraju

završe u *fragment shaderu* koji zapravo iscrtava piksele korisniku. *Tessellation* i *geometry shader* imaju svoje uloge, ali za potrebe ovoga rada nisu korišteni.

Kao što je ranije spomenuto, svi se objekti prikazani trokutima. To znači da je potrebna nekakva struktura podataka koja nekakav objekt prikazuje u obliku velikog (ili malenog) broja trokuta. Ta struktura podataka su zapravo dva *buffera* (engl. *buffer*) koji su ulazi za prethodno spomenuti *vertex shader* te predstavljaju polja podataka od nekog objekta. Razlog zašto su dva *buffera* je sljedeći. Prvi *buffer* se zove *vertex buffer* te on predstavlja sve točke od kojih se objekt sastoji. Točke od kojih se objekt sastoji se zovu *vertexi* (engl. *vertex*) te oni nose informacije o tome gdje je ta točka u prostoru. Bitno za spomenuti je da nije ispravno smatrati da su *vertexi* samo pozicije u prostoru. *Vertexi* nose sve informacije koje se mogu dodijeliti nekoj točki u prostoru, ti podatci mogu biti različiti i mogu predstavljati razne podatke. Ono što velikom većinom nose u sebi jesu prostorne koordinate, ali mogu nositi mnogo više podataka. U ovome radu svaki *vertex* se sastoji od 3D koordinate u prostoru, 3D vektora koji predstavlja vektor normale u toj točki te 2D koordinate teksture. Prvi *buffer* ili *vertex buffer* drži podatke o točkama koje definiraju objekt što znači da nedostaju podatci koji definiraju način na koji se te točke povezuju u trokute. Drugi *buffer* ispunjava tu ulogu te se zove *index buffer* (engl. *index buffer*) jer svaka vrijednost u tom bufferu je zapravo broj koji kaže na koji vertex on pokazuje u prvom *bufferu*. *Index buffer* se sastoji od cjelih brojeva koji predstavljaju uređene trojke koje definiraju svaki trokut. Prikaz opisanoga je na slikama 3.2 i 3.3 odmah ispod.



Slika 3.2 Prikaz jednostavnog oblika od 6 *vertexa* koji su povezani u 4 trokuta



Slika 3.3 Prikaz *vertex* i *index buffera* te njihovih odnosa

Na slici 3.2 se može vidjeti jednostavan primjer oblika od 6 *vertexa* i 4 trokuta te na slici 3.3 se može vidjeti kako su točno poredani *vertex* i *index buffer*. *Vertex buffer* u sebi sadrži samo *vertexe*, u primjeru je to 6 *vertexa*, a *index buffer* sadrži trojke *indexa* koji definiraju kako povezati *vertexe* u trokute. Prva trojka u *index bufferu* govori da *vertexi* na pozicijama 0 (V1), 1 (V2) i 4 (V5) tvore prvi trokut, druga trojka govori jednako, ali za *vertexe* na pozicijama 0, 4 i 3 itd. Trojke *indexa* su razdvojene samo radi vizualizacije, u programima je to samo jedno veliko polje brojeva. Takav način definiranja objekata je vrlo jednostavan i učinkovit te su sve grafičke kartice proizvedene u posljednjih ~20 godina dizajnirane s time na umu.

3.1. Vertex shader

Vertex shader je prva faza kroz koju moraju proći podatci od objekata prije nego se mogu iscrtati. Uloga *vertex shadera* je vrlo jednostavna, za svaki *vertex* u prethodno definiranom *vertex bufferu* će se izvesti jedna takozvana „invokacija“ *vertex shadera*. To znači da ako postoji 150 *vertexa* jednako toliko puta će se pozvati *vertex shader*, jednom za svaki *vertex*. Zato što se izvodi na grafičkoj kartici to znači da će se izvoditi paralelno. Svrha tog procesa je da se pomoću određene logike i matematike unutar *vertex shadera* svaki ulazni vertex transformira te smjesti na prikladno mjesto u 3D prostor. U kontekstu OpenGL-a to znači da će svaka invokacija *vertex shadera* dobiti podatke o vertexu s kojim će raditi zajedno sa dodatnim podatcima koje se proslijede *vertex shaderu*. Da bi se nešto prikazalo na ekran potrebno je svaki *vertex* transformirati i prebaciti u poseban koordinatni sustav što OpenGL koristi. Definirani raspon unutar kojega OpenGL funkcionira je [-1,1] u svakoj od 3 koordinatne osi. Za potrebe ovog rada *vertex shader* je odmah ispod.

```
#version 460 core

layout (location = 0) in vec4 vertexPosition;
layout (location = 1) in vec4 vertexNormal;
layout (location = 2) in vec4 vertexUV;



struct Matrices {
    mat4 model;
    mat4 normal;
};

layout (binding = 8, std430) buffer MatricesBuffer {
    Matrices matrices[];
};



uniform mat4 viewProjection;



out vec3 fragmentPosition;
out vec3 fragmentNormal;
out flat int meshIndex;



void main() {

    int index = gl_DrawID;

    mat4 modelMat = matrices[index].model;
    mat4 normalMat = matrices[index].normal;

    fragmentPosition = (modelMat * vec4(vertexPosition.xyz, 1.0)).xyz;
    fragmentNormal = mat3(normalMat) * vertexNormal.xyz;
    meshIndex = index;

    gl_Position = viewProjection * modelMat * vec4(vertexPosition.xyz, 1.0);
}
```

Slika 3.4 *Vertex shader*

Definicija *vertex shadera* započinje time da se definiraju ulazi u *shader*. U *shaderu* iznad oni su definirani kao 4D vektor koji predstavlja poziciju u prostoru, 4D vektor koji predstavlja vektor normale u toj poziciji u prostoru te 4D vektor koji predstavlja koordinate teksture. Razlog zašto su 4D umjesto ranije navedenih 3D i 2D vektora je zapravo problematična situacija oko upravljačkog programa od OpenGL-a. OpenGL je samo specifikacija, što znači da svaki proizvođač mora implementirati specifikaciju, ali to ne znači da će točno implementirati. Točnije, problemi se stvore oko 3D vektora jer se oni ravnaju na 4D vektore čak i kad ne bi smjeli te je rješenje za to da su uvijek smatra kao da su 4D te da se

odbace podatci koji ne trebaju. Nakon definiranja ulaznih podataka od samog objekta definira se struktura koja sadrži *model* i *normal* matrice. *Model* matrica opisuje transformaciju svakog *vertexa* iz svog lokalnog koordinatnog sustava u globalni koordinatni sustav unutar kojega se nalazi cijeli svijet koji se iscrtava. *Normal* matrica je matrica potrebna za ispravljanje vektora normale. Struktura za matrice je smještena u svoj *buffer* koji je zapravo *buffer* za sve objekte koje želimo iscrtati. Nakon toga prosljeđuje se matrica koja definira transformaciju iz 3D svijeta na 2D ekran. Varijable koje su definirane sa „out“ su varijable koje *vertex shader* prosljeđuje u druge *shadere* tako da se oni mogu koristiti njima. U slučaju iznad navedenog *shadera* to su pozicija fragmenta u 3D prostoru, vektor normale i indeks koji predstavlja koji objekt iscrtavamo. Unutar main funkcije samog *shadera*, dio koji se zapravo izvršava, koristi se varijabla gl_DrawID koja vrati indeks objekta koji se iscrtava. Ako se iscrtava samo jedan objekt onda je vrijednost te varijable 0, a inače je vrijednost 0, 1, 2 itd. sve do maksimalnog broja objekata koji se iscrtavaju. Koristeći tu varijablu se uzmu svi nužni podatci za taj jedan *vertex* te se izračuna sve što treba i proslijedi dalje u druge faze iscrtavanja.

3.2. Fragment shader

Nakon što se izvršio *vertex shader* te potencijalno prisutni *tessellation* i *geometry shaderi* ulogu preuzima hardver grafičke kartice koji od prethodno navedenih indexa i transformiranih vertexa pravi trokute. Svaki trokut mora proći kroz jedan dio grafičke kartice koji se zove *rasterizer* (engl. *rasterizer*). Taj dio grafičke kartice je zadužen da uzima trokute koji se nalaze unutar raspona koji OpenGL koristi te ih onda pretvara u individualne piksele. U kontekstu grafike pikseli se također zovu i fragmenti, na osnovi toga je i njihov *shader* dobio ime, ali u literaturi se može pronaći ili *fragment* ili *pixel shader*. Uloga *fragment shadera* je da se pozove za svaki *fragment* (piksel) koji se generira. Npr. ako neki trokut proizvede 12,000 fragmenata onda će se toliko puta pozvati taj shader. Što točno *fragment shader* radi ovisi isključivo o tome što bi on trebao predstavljati, u ovome radu *fragment shader* će računati osvjetljenje nekoga piksela. Algoritam kako se računa osvjetljenje nije važan jer je bitan princip iscrtavanja. Implementacija *fragment shadera* je sljedeća.

```
#version 460 core

layout (location = 0) out vec4 finalColor;



struct Material {
    vec4 ambient;
    vec4 diffuse;
    vec4 specular;
};

layout (binding = 9, std430) buffer Materials {
    Material materials[];
};



uniform vec3 lightDirection;
uniform vec3 lightAmbient;
uniform vec3 lightDiffuse;
uniform vec3 lightSpecular;
uniform vec3 cameraPos;



in vec3 fragmentPosition;
in vec3 fragmentNormal;
in flat int meshIndex;



void main() {

    vec3 normal = normalize(fragmentNormal);

    vec3 viewDir = normalize(cameraPos - fragmentPosition);
    vec3 halfwayDir = normalize(lightDirection + viewDir);
    vec3 reflectDir = reflect(-lightDirection, normal);

    float diffuseScalar = max(dot(normal, lightDirection), 0.0);
    float specularScalar = pow(max(dot(normal, halfwayDir), 0.0),
materials[meshIndex].specular.w);

    vec3 color0 = materials[meshIndex].ambient.xyz;
    vec3 color1 = materials[meshIndex].diffuse.xyz;
    vec3 color2 = materials[meshIndex].specular.xyz;

    vec3 ambientColor = color0 * lightAmbient;
    vec3 diffuseColor = color1 * lightDiffuse;
    vec3 specularColor = color2 * lightSpecular;

    vec3 ambient = ambientColor;
    vec3 diffuse = diffuseColor * diffuseScalar;
    vec3 specular = specularColor * specularScalar;

    finalColor = vec4(ambient + (diffuse + specular), 1);
}
```

Slika 3.5 *Fragment shader*

Definicija *fragment shadera* započinje definiranje varijable koja predstavlja piksel koji će se proizvesti. Zatim, većina koda je definiranja podataka za osvjetljenje, ti podatci nisu važni nego su važne „in“ varijable. Kako su u *vertex shaderu* definirane „out“ varijable u *fragment shaderu* moraju biti definirane istoimene „in“ varijable jer to su varijable koje će *fragment shader* koristiti za svoje izračune. Vrijednosti tih varijabli su interpolirane vrijednosti varijabli koje izbacuje *vertex shader*. Budući da *vertex shader* izbacuje *vertexe* koji se formiraju u trokute vrijednosti „in“ varijabli su linearna interpolacija pripadajućih varijabli od sva 3 vertex od kojih se sastoji trokut. Unutar main funkcije se računa osvjetljenje svakog fragmenta pomoći Blinn-Phong modela osvjetljenja, ali detalji implementacije nisu važni u kontekstu ovoga rada.

3.3. Implementacija

Implementacija iscrtavanja je je vrlo jednostavna. Potrebno je napraviti *buffere* koji drže podatke od objekata. To se postigne vrlo jednostavno. Klasa BufferGL je vrlo malena abstrakcija nad OpenGL funkcijama radi jednostavnosti korištenja.

```
Mesh mesh = MeshGenerator::generateIcoSphere(256);

std::vector<VertexGPU> verts(mesh.vertices.size());
for (int i = 0; i < verts.size(); i++) {
	verts[i].position = glm::vec4(mesh.vertices[i].position, 1);
	verts[i].normal = glm::vec4(mesh.vertices[i].normal, 1);
}

BufferGL vertices, indices;
vertices.allocateStorage(verts.data(), verts.size() *
sizeof(decltype(verts)::value_type));
indices.allocateStorage(mesh.indices.data(),mesh.indices.size()*
sizeof(decltype(mesh.indices)::value_type));
```

Slika 3.6 Generiranje *mesha* i *buffera*

Na početku se generira *mesh* (engl. *mesh*) koji predstavlja oplošje objekta, a to su svi *vertexi* i svi *indeksi*. Zatim se oni pretvore u verzije sa 4D vektorima kao što je navedeno ranije u radu te se na kraju alociraju *bufferi* na grafičkoj kartici te se ti podatci prosljede tim *bufferima*. Također je potreban korak definiranja kako su točno posloženi podatci u memoriji no taj korak trenutno nije važan te će se preskočiti.

Također je potrebno i napraviti *shader*. Koristeći jednostavnu abstrakciju nad *shaderima* se može u nekoliko linija definirati *shadere*.

```
ShaderGL shaderSimple;
shaderSimple.addShaderStage("shaderSimple.vert", ShaderStage::VERTEX);
shaderSimple.addShaderStage("shaderSimple.frag", ShaderStage::FRAGMENT);
shaderSimple.createProgram();
```

Slika 3.6 Generiranje *shader programa*

Shaderi kad se napravi zajedno se povežu u jedan *shader program* objekt koji se koristi dalje. Korištenje *shader programa* je također jednostavno.

```
shaderSimple.use();
shaderSimple.setVec3f("lightDirection", glm::normalize(glm::vec3(1, 0.3,
0.3)));
shaderSimple.setVec3f("lightAmbient", glm::vec3(1));
shaderSimple.setVec3f("lightDiffuse", glm::vec3(1));
shaderSimple.setVec3f("lightSpecular", glm::vec3(1));

shaderSimple.setMat4f("viewProjection",
camera.getProjectionMatrix(window.getAspectRatio()) * camera.getViewMatrix());
```

Slika 3.7 Postavljenje podataka *shaderu*

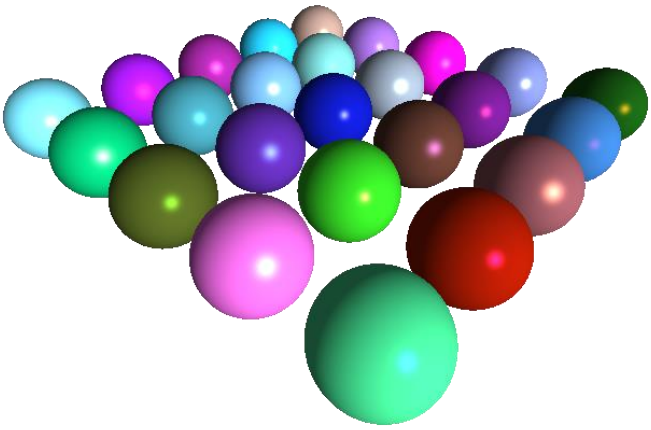
Sve uniform varijable unutar *vertex* i *fragment shadera* se mogu namjestiti te poslati u *shader*. Poziv kojim se započne iscrtavanje se nalazi ispod.

```
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, commandSimpleBuffer.getID());

shaderSimple.setVec3f("cameraPos", camera.position);
glMultiDrawElementsIndirect(GL_TRIANGLES,      GL_UNSIGNED_INT,      nullptr,
matrices.size(), sizeof(MultiDrawElementsCommand));
```

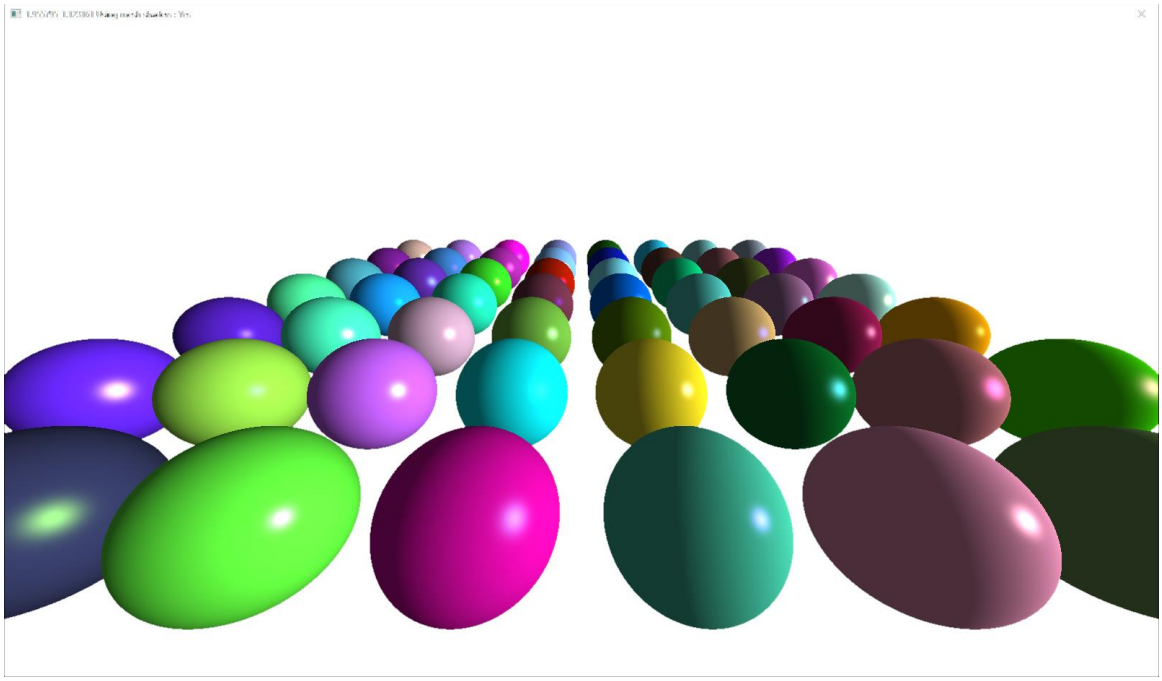
Slika 3.8 Pokratanje iscrtavanja

Iscrtavanje se u ovom radu radi pomoću indirektnog načina rada u OpenGL-u te to omogućuje da se jednostavno iscrtava više objekata. Sami poziv koji započine iscrtavanje je glMultiDrawElementsIndirect poziv.

Nakon pozive te funkcije OpenGL predaje upravljačkom programu svu kontrolu nad iscrtavanjem te on sve što je potrbno šalje na grafičku karticu i započinje iscrtavanje. Pozivom te funkcije se ujedno započinje i završava iscrtavanje objekata. Postoje i jednostavniji načini iscrtavanja, ali radi malo više performansi kada se iscrtava više objekata se koristio taj malo kompleksniji način.



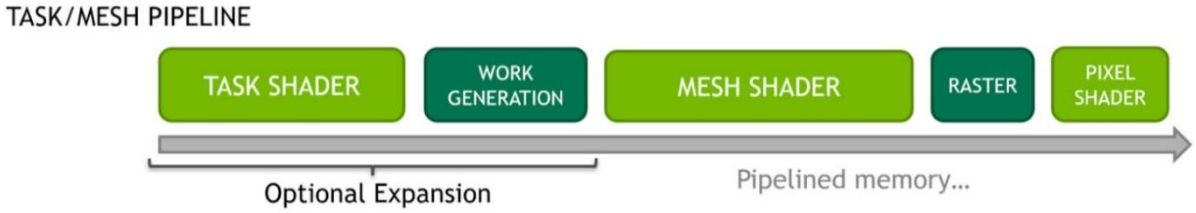
Slika 3.9 Primjer iscrtanih sfera



Slika 3.10 Primjer iscrtanih sfera

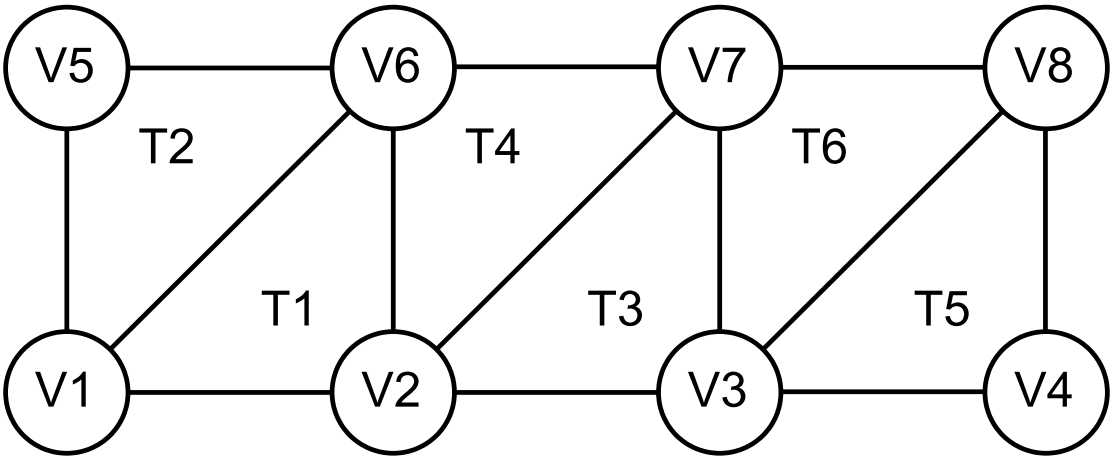
4. MODERNO ISCRTAVANJE

Iscrtavanje grafike se svelo na klasični pristup opisan u ranijim poglavljima zbog svoje jednostavnosti i učinkovitosti. Razvojem boljeg hardvera za rad s 3D grafikom nije bilo potrebe tražiti novi način rada jer su grafičke kartice bile dovoljno snažne da mogu podnijeti sve što se tražilo od njih. S vremenom, najčešće u video igrama, konzumeri su težili sve više prema većim rezolucijama i većim količinama detalja u video igrama što je zahtjevalo bolji hardver. Velikom većinom je hardver uspjevao sustizati zahtjeve, no kroz posljednjih desetak godina razlika između očekivanja i mogućnosti hardwarea sve više raste. Godine 2018. tvrtka Nvidia po prvi puta predstavlja novi način iscrtavanja klasične grafike koji se zove *mesh shaders* [2]. Te iste godine predstavljaju i novi princip iscrtavanja koji se zove *ray tracing* gdje se iz virtualne kamera ispaljuju zrake u scenu te se pomoću tih zraka računa osvjetljenje. *Ray tracing* može dati veoma realistične rezultate pa je pozornost javnosti bila isključivo na tome umjesto na *mesh shaderima*. Budući da je pozornost velikom većinom bila na *ray tracingu* nije se mnogo uzimalo u obzir što mogu *mesh shaderi* sve dok tvrtka Epic Games nije predstavila jedan video gdje demonstriraju sposobnosti njihovog Unreal Engine 5 *enginea* za 3D grafiku [6]. Objavom tog videa se javnost znatno zainteresirala za sposobnosti *enginea* (i indirektno za *mesh shadere*) jer omogućuje nevjerovatnu realističnost i performanse. *Mesh shaderi* su novi skup *shadera* koji su dizajnirani u potpunosti oko modernih grafičkih kartica da mogu iskoristiti hardware maksimalno. *Mesh shaderi* i dalje rade s trokutima, ali način na koji su podatci strukturirani je drukčiji te upravo ta promjena omogućuje drastična poboljšanja u performansama. Također *shaderi* koji se pokreću su poprilično drukčiji od klasičnog pristupa. U klasičnom načinu rada postoje *vertex shader,* dva *tessellation shadera, geometry shader* i na kraju *fragment shader* dok u modernom načinu postoje samo 3 shadera, a oni su *task shader, mesh shader* i *fragment shader*. Prikaz novog načina iscrtavanja je ispod na slici 4.1 [2].



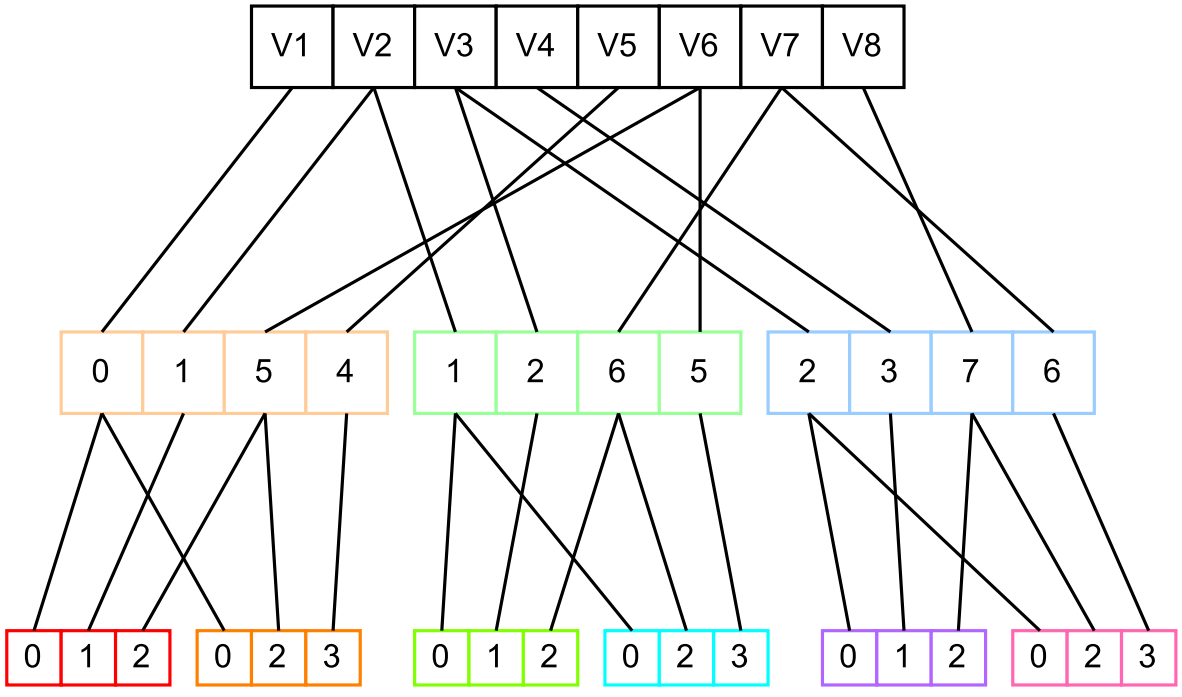
Slika 4.1 Faze iscrtavanja kod *mesh shadera*

Iscrtavanje s *mesh shaderima* započinje neobaveznim dijelom koji se zove *task shader*, nakon njega ide *mesh shader* i na kraju standardni *fragment shader*. Konceptualno se može zamisliti *mesh shader* kao *shader* koji objedinjuje *vertex shader*, oba *tessellation shadera* i *geometry shader* odjednom, ali istovremeno je moćniji i fleksibilniji od svih njih zajedno. Razlog zašto je moćniji je zato što nije vezan za specifičnu količinu podataka, kao što je *vertex shader* vezan za broj *vertexa*. *Mesh shader* je dizajniran u stilu *compute shadera* što im daje izvrsnu fleksibilnost i mogućnosti što *vertex shader* nikad nije imao. Razlog zašto su *mesh shaderi* bolji je drukčija struktura podataka koja drži podatke o objektu koji se treba iscrtati. *Mesh shaderi* zahtjevaju 3 *buffera* umjesto 2 kao što bude u klasičnom načinu rada. *Vertex buffer* je nepromijenjen, i dalje drži sve *vertexe* objekta. Ono što je drukčije je *index buffer*, točnije, sada su dva. Razlog za postojanje dva *buffera* je zbog razlike u logičkom pristupu kako je objekt definiran. U klasičnom načinu rada svi podatci su u 2 *buffera* što znači da ne posoji fina kontrola nad iscrtavanjem, samo se pokrene i dobiju se rezultati što nije uvijek optimalno. Npr. veoma detaljan objekt, poput sfere koja se sastoji od 10 milijuna trokuta. Grafička kartica mora proći kroz svaki trokut i iscrtati ga. Ako se razmisli o postupku iscrtavanja, nije uopće potrebno iscrtati trokute koji se ne vide, a u prethodno navedenom primjeru to je ~5 milijuna trokuta. Grafičke kartice imaju sposobnost određivanja da li je trokut vidljiv ili ne. Ako se ta opcija uključi, onda grafička kartica može odbacivati te trokute, no to je i dalje ~5 milijuna trokuta, što je znatan broj trokuta čak i za brze grafičke kartice. *Mesh shaderi* imaju drukčiji pristup, oni definiraju da je jedan objekt zapravo skup velikog broja manjih dijelova tog istog objekta. Na taj način se *mesh* objekta rastavi na velik broj manjih *mesheva* koji se zovu *meshlets* (maleni mesh na engleskom) te se svaki od tih manjih *mesheva* iscrtava nazavisno od ostalih. Upravo zbog tog rastavljanja većeg *mesha* na manje se može dobiti veliko ubrzanje. Razlog tome je što se zajedno s *task shaderom* može na razini samog *shadera* odlučivati da li je neki *meshlet* vidljiv ili ne što predstavlja iznimno veliko ubrzanje. Ako se pretpostavi da se jedan maleni *meshlet* sastoji od 100 trokuta i da se mogu odbacivati cijeli *meshleti* onda iz prethodnog primjera umjesto da hardver mora prvo izračunati svih 10 milijuna trokuta samo da bi odbacio ~5 milijuna trokuta, softver sada prvo odbaci ~50,000 *meshleta* koji predstavljaju ~5 milijuna trokuta te tek onda hardver izračuna preostale trokute i iscrta ih. Takav princip rastavljanja velikog *mesha* na *meshlete* je razlog zašto su potrebna 2 *index buffera*. Jedan od njih će indeksirati *vertexe* na isti način kao i u klasičnom načinu, ali razlika je da sada neće biti uređena trojka koja definira trokut nego će biti uređena n-torka koja definira od kojih se vertexa sastoji jedan *meshlet*. Drugi *index buffer* će zatim indeksirati taj prvi na isti način kao i u klasičnom radu, da bi indeksirao individualne trokute. Prikaz navedenoga je na slikama ispod.



Slika 4.2 Primjer *mesha* za *mesh shadera*

Na slici 4.2 je primjer *mesha* koji će se koristiti. Rastavljanje na *meshlete* će biti napravljeno tako da su trokuti T1 i T2 dio prvog *meshleta*, T3 i T4 su dio drugog i T5 i T6 su dio trećeg.



Slika 4.3 Vizualni prikaz indeksiranja buffers sa mesh shaderima

Na slici 4.3 je prikazan odnos sva 3 *buffera* u radu sa *mesh shaderima*. Razmaci izmedju vrijednosti su samo vizualni, kada se implementira to su samo veliki *bufferi* podataka bez

razmaka. Prvi *index buffer*, koji će se u ovom radu zvati *vertex index buffer*, je zaslužan da za svaki *meshlet* definira koji *vertexi* pripadaju tom *meshletu*. Drugi *index buffer*, koji će se zvati *primitive index buffer*, zatim indexira *vertex index buffer* te definira trokute. Ako se uzme drugi *meshlet*, iz slike se može vidjeti da se on sastoji od *vertexa* na indeksima 1, 2, 6 i 5 što su zapravo *vertexi* V2, V3, V7 i V6. Zatim *primitive index buffer* kaže kojim redosljedom da se uzimaju indeksi iz *vertex index buffera* da bi se napravili trokuti. Za prvi trokut se uzimaju *vertex indexi* na indeksima 0, 1 i 2 unutar tog *meshleta* što su zapravo *vertex indexi* 1, 2 i 6. Uzimajući *vertexe* na tim pozicijama se dobije da je prvi trokut trokut definiran *vertexima* V2, V3 i V7. Jednako vrijedi i za drugi trokut. *Primitive indexi* su 0, 2 i 3 koji indeksiraju *vertex index buffer* na 1, 6 i 5 što indeksira *vertexe* V2, V7 i V6. Taj dvostruki sloj indirekcije je čak poprilično moćan sam po sebi. Jedna stvar koja se postiže time je da se koristi manje memorije. Ako se ograniči maksimalna veličina jednog *meshleta* onda se mogu postići određene garancije. U slučaju *mesh shadera* maksimalni broj trokuta je 256 što znači da se vrijednosti u *primitive index bufferu* u rasponu od 0 do 255 što je zapravo samo jedan bajt. Budući da *primitive index buffer* drži svoje podatke kao niz individualnih bajtova time se uštedi na memoriji. Čak se može i na tom malenom primjeru to primijetiti. Ako bi se iscrtavao prvi *meshlet* s trokutima T1 i T2 na klasičan način bilo bi potrebno držati 4 bajta za svaki indeks, a indeksa je 6 zbog dva trokuta što je ukupno 4 * 6 = 24 bajta. S *meshletima* prvi *index buffer* bi imao 4 indeksa po 4 bajta što je 16 bajtova, a drugi *index buffer* bi imao 2 trokuta a svaki je 3 bajta što je zajedno 6 bajtova. Ukupno su 16 + 6 = 22 bajta. Razlika nije velika na ovom malenom primjeru, ali kako su objekti veći sve više se memorije može uštediti.

4.1. Compute shader

Jedna stvar koju je bitno objasniti prije opisivanja individualnih shadera su *compute shaderi* jer se i *task shader* i *mesh shader* oslanjaju na njih. *Compute shaderi* su posebna vrsta *shadera* koja uopće nema poveznice s iscrtavanjem objekata. Njihova svrha je raditi nekakav općeniti posao na grafičkoj kartici iskorištavajući velik broj jezgri koje su dostupne [7]. Zbog činjenice da nisu vezani za iscrtavanje mora im se definirati što bi svaki *thread* trebao raditi. *Compute shaderi* rade na principu da se definira koliko se individualnih *threadova* želi pokrenuti te se svakom *threadu* dodijeli jedinstven indeks koje ga definira. Indeksi se definiraju kao 3D koordinate jer time se omogući velika fleksibilnost u slučajima gdje treba moći na takav način podijeliti posao te takvo dijeljenje omogućuje da se vrlo precizno određuje koji thread će što raditi. *Threadovi* se također slažu u skupine po 32 threada koji se izvode istovremeno jer to

je kako ih hardware izvršava. *Compute shaderi* imaju mnogo drugih prednosti, ali ona koja je najbitnija za ovaj rad je činjenica da se proizvoljno određuje posao koji će se raditi.

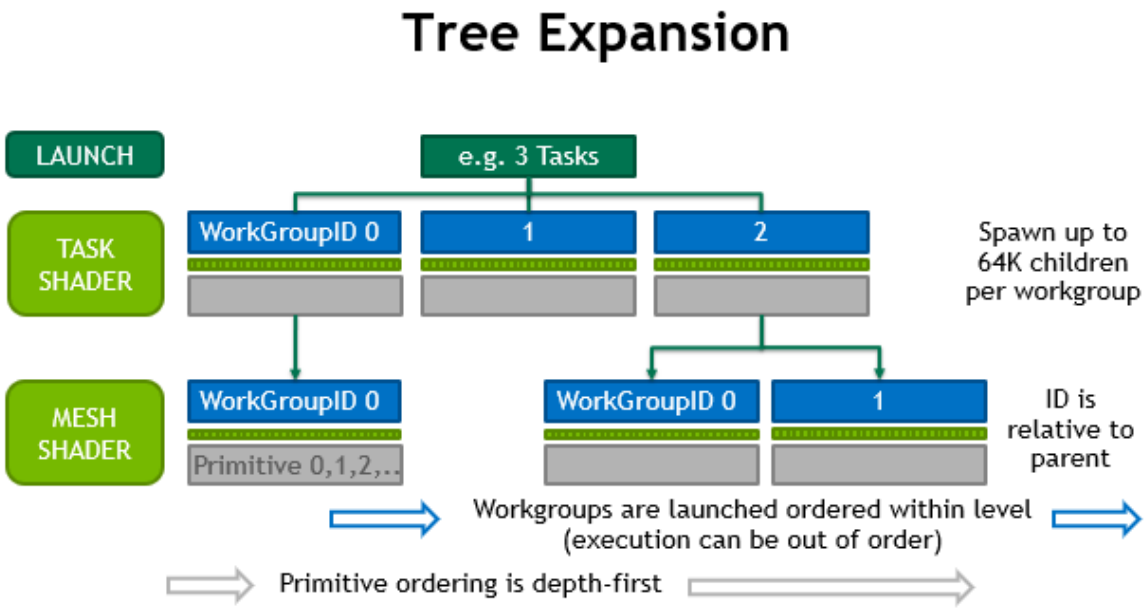
4.2. Task shader

Task shader je prvi korak u iscrtavanju grafike na moderan način. *Task shader* zapravo nije u stanju ništa iscrtavat. Ono što definira *task shader* je njegovo svojstvo da može pokretati proizvoljan broj *mesh shadera*. Razlog zašto je značajan ovaj shader je upravo ta mogućnost da pokreće *mesh shadere* jer taj shader odlučuje koliko i koje će *mesh shadere* pokrenuti. To znači da ako taj *shader* iz nekoga razloga odluči da se neki *meshlet* ne treba iscrtati onda ga neće ni pokrenuti. Ovaj korak je zapravo najviše značajan jer tu će se odvijati svo odbacivanje *meshleta* prije iscrtavanja.

```
#version 460 core

#extension GL_NV_gpu_shader5 : enable
#extension GL_NV_mesh_shader : enable
#extension GL_NV_shader_thread_group : enable

layout (local_size_x = 32) in;

…

taskNV out Task {
	uint meshletIDs[32];
} Offsets;

…

void main() {

	uint meshletID = 32 * gl_WorkGroupID.x + gl_LocalInvocationID.x;

	if (meshletID >= meshletCount) { return; }

	MeshletDescription description =
meshletDescriptions.description[meshletID];

	if (earlyCull(description)) { return; }

	uint bitmask = ballotThreadNV(true);
	int validThreads = bitCount(bitmask);

	bitmask &= (2 << gl_LocalInvocationID.x) - 1;
	int index = bitCount(bitmask) - 1;

	Offsets.meshletIDs[index] = meshletID;

	gl_TaskCountNV = validThreads;
}
```

Slika 4.4 *Task shader*

Kod od *shadera* je relativno jednostavan, na početku je potrebno upaliti neke dodatke na standardni GLSL jezik za pisanje *shadera*, zatim se definira u jednakom stilu kao i *compute shaderi* od koliko *threadova* će se sastojati jedan blok *threadova*. *Task shader* je definiran da se sastoji od 32 *threada*. Svaki *thread* će zatim izračunati da li je *meshlet* koji pripada tom *threadu* uopće vidljiv te će odlučiti hoće li ga uopće iscrtati. Zatim je prekočen dio koda od nešto manje važnosti gdje su definirani ulazi u *shader*. Jedini malo značajniji dio koji je preskočen je definiranje *buffera* koji sadrže informacije o tome kako su strukturirani *meshleti*. Preskočeno je jer način definiranja *meshleta* ovisi o tome što se želi postići. Struktura koja je definirana kao „out“ je maleni blok podataka koji će se proslijediti svakom *mesh shaderu* koji se pozove iz *task shadera*. U ovom slučaju to je maleno polje indeksa koji govore koji *meshlet* će se iscrtavati. U main funkciji je prvo potrebno izračunati indeks svakog *meshleta*. Iscrtavanje radi na principu da nakon što se mesh podijeli u *meshlete* da se pokrene dovoljno invokacija ili poziva *task shadera* tako da broj *task shader* invokacija pomnoženo sa 32 bude veći ili jednak broju *meshleta*. Na taj način se garantira da će se svaki *meshlet* obraditi. Zatim, ako je neki *thread* dobio indeks koji je veći ili jednak broju *meshleta* taj *thread* se ugasi. Nakon toga se učita definicija *meshleta* pomoću koje se određuje da li je *meshlet* vidljiv. Upravo taj dio je ono gdje se najviše performansi može dobiti, jer na taj način se mogu odbacivati velike skupine trokuta koje nisu uopće vidljive. Ako su neki *threadovi* prošli kroz to sve onda je samo potrebno zapisati koji su točno *meshleti* prošli provjeru vidljivosti. Poziv funkciji ballotThreadNV vrati cjeli broj od 32 bita gdje svaki bit predstavlja da li se taj *thread* i dalje izvodi ili ne. Npr. ako se uzme broj od 8 bitova radi jednostavnosti, broj 00110001 zapravo govori da samo 3 *threada* i dalje rade te svaki bit koji je 1 govori da je *thread* na tom mjestu i dalje aktivan. Ta informacija se može iskoristiti da se na prikladna mjesta u „out“ bloku zapišu indeksi *meshleta* koji su vidljivi te zatim *task shader* pokreće onoliko *meshleta* koliko je *threadova* bilo aktivno na kraju. Činjenica da svaki *task shader* može pokrenuti varijabilni broj *mesh shadera* je velika prednost jer na taj način se postiže velika fleksivilnost sustava. Vizualno se može opisati kao na slici 4.5 ispod.



Slika 4.5 Prikaz strukture pokretanja *task* i *mesh shadera* [2]

4.3. Mesh shader

Mesh shader je glavni dio cijelog načina modernog iscrtavanja objekata. Ovaj *shader* objedinjuje *vertex shader*, *tessellation shadere* i *geometry shader* te sve njihove funkcionalnosti odjednom. Ono što je posebno oko ovog *shadera* jer to da za razliku od *vertex shadera* koji ima točno određeno što mora učiniti, ovaj shader je u stanju odlučiti što želi napraviti i koliko toga da napravi. U iscrtavanju objekata to znači da je ovaj *shader* u stanju proizvoditi trokute koji nisu nužno definirani prijevremeno, ili jednostavno odbaci trokute ako mu ne trebaju ili nešto slično. Ono što čini ovaj *shader* jakim je upravo ta velika flaksibilnost. Primjer kako mesh shader izgleda u radu je odmah ispod.

```
#version 460 core

#extension GL_NV_mesh_shader : enable
#extension GL_NV_gpu_shader5 : enable

#define VERTICES 64
#define PRIMITIVES 84

layout (local_size_x = 32) in;
layout (triangles, max_vertices = VERTICES, max_primitives = PRIMITIVES) out;

struct Vertex {
	vec4 position;
	vec4 normal;
	vec4 uv;
};
layout (std430, binding = 0) readonly buffer vertexBuffer {
	Vertex vertices[]; };
layout (std430, binding = 1) readonly buffer vertexIndexBuffer {
	int vertexIndices[]; };
layout (std430, binding = 2) readonly buffer localIndexBuffer {
	uint8_t localIndices[]; };

…

taskNV in Task {
	uint meshletIDs[32];
} Offsets;

layout (location = 1) out VertexData {
	vec3 fragmentPosition;
	vec3 fragmentNormal;
	flat int meshIndex;
} vertexData[VERTICES];



uniform mat4 viewProjection;
uniform vec3 bbMin;
uniform vec3 bbMax;
```

Slika 4.6 *Mesh shader* prije *main* funkcije

```
void main() {

	uint groupID = Offsets.meshletIDs[gl_WorkGroupID.x];
	uint threadID = gl_LocalInvocationID.x;

	uint meshID = gl_DrawID;
	mat4 model = matrices[meshID].model;
	mat4 normal = matrices[meshID].normal;

	MeshletDescription description =
meshletDescriptions.description[groupID];

	const int vertexLoops = (VERTICES + 31) / 32;
	for (int i = 0; i < vertexLoops; i++) {

		uint vertex = threadID + 32 * i;
		vertex = min(vertex, VERTICES - 1);

		uint index = vertexIndices[groupID * VERTICES + vertex];

		gl_MeshVerticesNV[vertex].gl_Position = viewProjection * model *
vec4(vertices[index].position.xyz, 1);
		vertexData[vertex].fragmentPosition = (model *
vec4(vertices[index].position.xyz, 1)).xyz;
		vertexData[vertex].fragmentNormal = mat3(normal) *
vertices[index].normal.xyz;
		vertexData[vertex].meshIndex = gl_DrawID;
	}

	const int indexLoops = (PRIMITIVES * 3 + 31) / 32;
	for (int i = 0; i < indexLoops; i++) {

		int index = int(threadID) + 32 * i;
		index = min(index, PRIMITIVES * 3 - 1);

		gl_PrimitiveIndicesNV[index] = localIndices[index + PRIMITIVES *
3 * int(groupID)];
	}

	gl_PrimitiveCountNV = description.pc;
}
```

Slika 4.7 *Mesh shader main* funkcija

Kod za *mesh shader* na početku ima definiranje dodataka na GLSL jezik za *shadere* koji omogućuju korištenje *mesh shadera*. Odmah nakon toga se definiraju *vertices* i *primitives*. *Vertices* u ovom kontekstu, u kontekstu mesh shadera specifično, je definicija koliko *vertexa* je *mesh shader* u stanju proslijediti dalje u *rasterizer* i *fragment shader*. *Primitives* je definicija koliko se maksimalno trokuta može generirati i proslijediti dalje. Način na koji se odabiru vrijednosti je poprilično eksperimatalan za broj *vertexa*, ali za broj trokuta ima izračun. Hardver alocira blokove podataka za trokute višekratnicima od 128 bajtova. Budući da je potrebno 3 bajta za svaki trokut i dodatno 4 bajta za ukupan broj indeksa to znači da je optimalni broj

trokuta (128 – 4) / 3 = 41. Zato što hardveru više odgovaraju parni brojevi krajnji broj je zapravo 40. Jednako se primijenjuje ako se želi više trokuta, da bi se iskoristilo 256 bajtova onda je broj trokuta jednak 84, a za 512 bajtova je 168. Odabir broja *vertexa* i broja trokuta pravi veliku razliku za performanse jer se time određuje od koliko se trokuta može sastojati svaki *meshlet*. Nakon definiranja tih vrijednosti su definirani svi podatci od objekta, a to su svi *vertexi*, *vertex index buffer* i *primitive index buffer*. Također odmah nakon je je definiran ulaz iz *task shadera* koji govori koji *meshleti* se trebaju iscrtati. Ono što je novo je odmah nakon, potrebno je definirati izlaze iz mesh shadera. Ti izlazi su bitni jer oni predstavljaju identične vrijednosti kao i zasebne „out“ varijable u *vertex shaderu* u klasičnom načinu rada. Na kraju je nekoliko *uniform* varijabli.

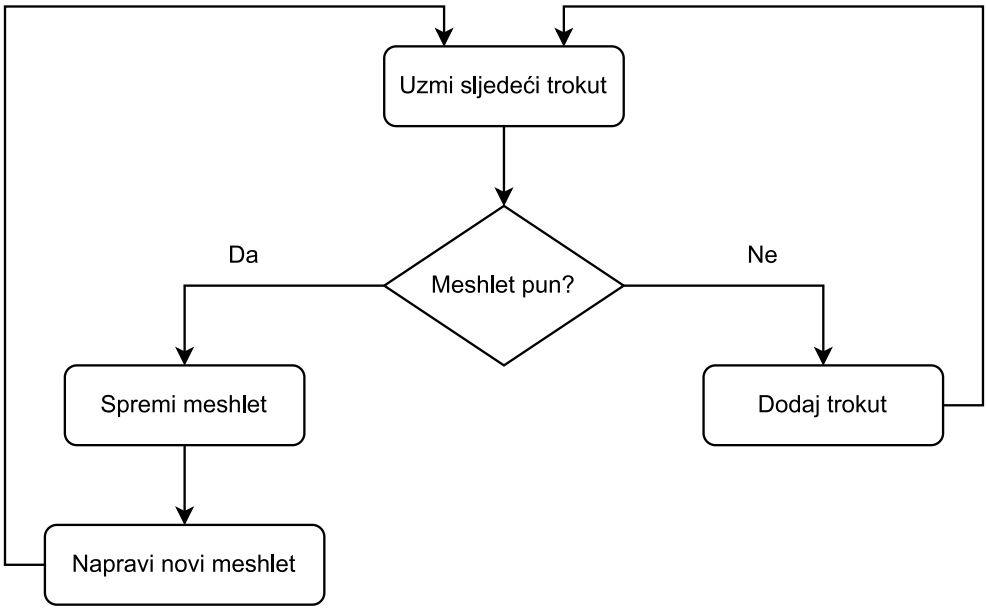
Tijelo main funkcije je mjesto gdje se po prvi puta formiraju trokuti i šalju dalje. Prvo je potrebno odrediti kojem *meshletu* pripada trenutna invokacija ili poziv *mesh shadera*. Nakon toga se učitaju podatci koji kažu koliko *vertexa* i koliko trokuta se nalazi u trenutnom *meshletu*. Odmah nakon toga slijedi petlja koja za učitava *vertexe* u grupama po 32, jednako kao i veličina grupa *mesh shadera*, te ih sprema u gl_MeshVerticesNV varijablu koja je zapravo maleno privremeno polje gdje se spreme *vertexi* za iscrtavanje. Ako se obrati pažnja na tijelo for petlje može se vidjeti dvostruka indirekcija zbog dva *index buffera*. Nakon te petlje slijedi petlja koja radi istu stvar kao i prethodna, ali za indekse u *primitive index bufferu*. Na kraju se varijabli gl_PrimitiveCountNV dodijeli vrijednost koje govori koliko od koliko indeksa se sastoji trenutni *meshlet*. Time završava postupak za iscrtavanje. Nakon toga se poziva *fragment shader* koji iscrtava individualne piksele. Jedina razlika u *fragment shaderu* spram prije je da se „in“ varijable zamjene sa sljedećim kodom.Funkcionalno je identično kao i u klasičnom načinu rada.

```
layout (location = 1) in VertexData {
      vec3 fragmentPosition;
      vec3 fragmentNormal;
      flat int meshIndex;
};
```

Slika 4.8 Ulazi u *fragment shader*

4.4. Generiranje meshleta

U prethodnim poglavljima je opisano kako funkcioniraju *task* i *mesh shader* no nije objašnjeno kako se poslože podatci da se to može primijeniti. Najjednostavniji algoritam za pretvaranje klasičnog *mesha* u *meshlete* je vrlo jednostavan, no nije najučinkovitiji. Algoritam radi na sljedećem principu opisanim odmah ispod.



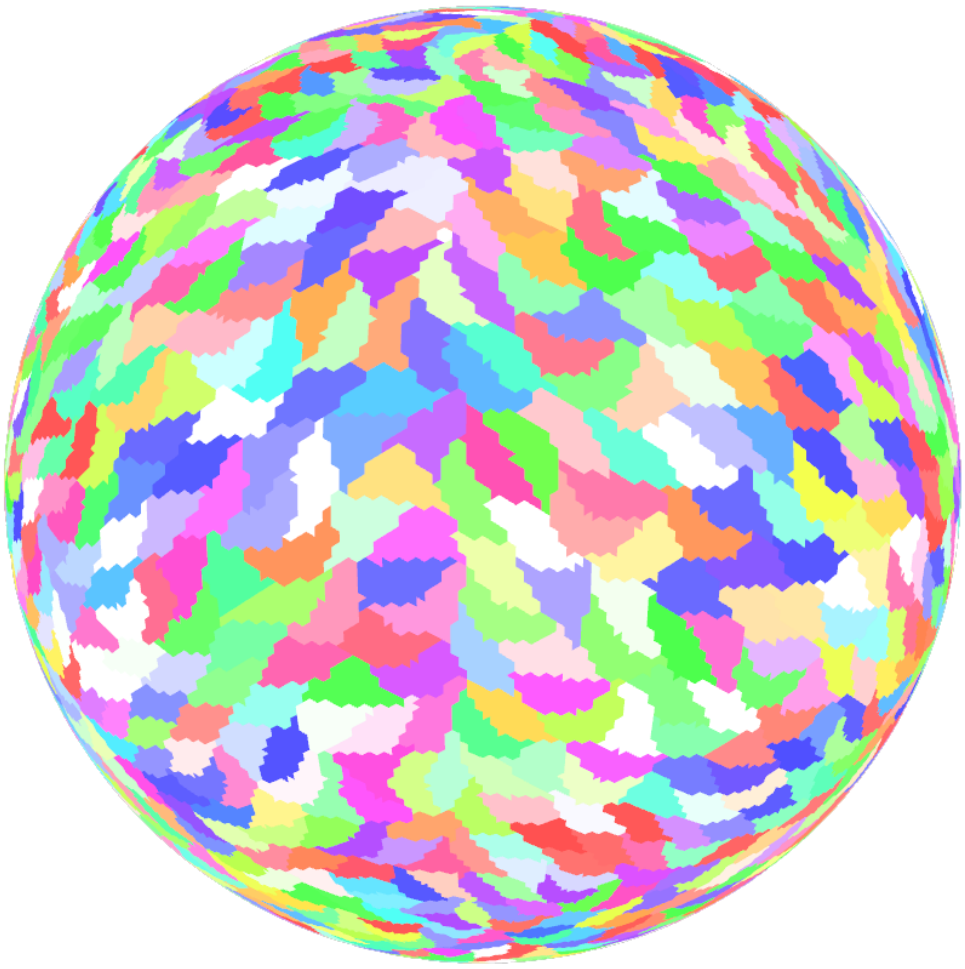
Slika 4.9 Algoritam za generiranje *meshleta*

Algoritam je vrlo jednostavan, petlja se pokreće za svaki trokut u *meshu* te se svaki puta pokuša dodati trokut u *meshlet*, ako se *meshlet* popuni, tj. dosegne se maksimalni broj trokuta ili *vertexa*, onda se sprema *meshlet*, napravi se novi te se nastavi dalje. U trenutku kada algoritam završi tada je cijeli originalni *mesh* spremljen u obliku koji može koristiti sa *mesh shaderima*. Učinkovitost ovog algoritma ovisi u velikoj mjeri o tome kojim redoslijedom su definirani trokuti u *index bufferu*. Ako trokuti imaju dobru prostornu lokalnost onda će učinkovitost biti veća nego kada nisu dobro posloženi. Učinkovitost algoritma varira, ali je uglavnom od 60% do 80%. Učinkovitost se mjeri kao prosječan broj trokuta po *meshletu* spram maksimalnog broja trokuta po *meshletu*, npr. 76 trokuta od maksimalnih 84 je ~90%.

Moguće je napraviti i kompleksniji algoritam koji se pobrine da *meshleti* budu skoro idealno rapoređeni. Algoritam je sporiji od prethodno navedenog, ali mu je učinkovitost mnogo veća. Algoritam radi na sljedeći način. Prvo je potrebno pretvoriti mrežu trokuta u graf. Time se apstraktna reprezentacija trokuta u *index bufferu* pretvori u nešto što ima inznimno veliku prostornu komponentu, svaki susjed trokuta je odmah neposredno pored. Zatim se odabere

neka točka u grafu, točnije to je neki trokut, te se koristi algoritam obilaska po širini sve dok se ne popuni neki *meshlet*. U tom trenutku se sprema *meshlet*, generira se novi, te algoritam nastavlja raditi gdje je stao, ali uz bitnu napomenu da prethodne trokute ignorira potpuno. Na taj način se nikada isti trokut neće ubaciti u dva nezavisna *meshleta*. Učinkovitost tog algoritma je redovito preko 95%.

Nakon što se taj algoritam pokrene i vizualizira dobije se sljedeći prikaz. Na prikazu ispod svaka nakupina iste boje je jedan *meshlet*. Kao što se može vidjeti, distribucija meshleta je vrlo dobra.



Slika 4.10 Prikaz kako je jedan *mesh* podijeljen na *meshlete*

4.5. Implementacija

Implementacija modernih *shadera* je velikom većinom u potpunosti jednaka kao i klasičnima uz naravno promjene u *shaderima* i *bufferima* te načinu pozivanja funkcije za iscrtavanje.

```
MeshletMesh meshletMesh = MeshConverter::generateMeshletMesh(mesh);

std::vector<VertexGPU> verts(mesh.vertices.size());
for (int i = 0; i < verts.size(); i++) {
	verts[i].position = glm::vec4(mesh.vertices[i].position, 1);
	verts[i].normal = glm::vec4(mesh.vertices[i].normal, 1);
}

BufferGL vertices;
vertices.allocateStorage(verts.data(), verts.size() *
sizeof(decltype(verts)::value_type));
```

Slika 4.11 Generiranje meshleta i slanje *vertex* podataka na grafičku karticu

Prvi korak je da se uzme originalni *mesh* te da se pretvori u novu reprezentaciju. Nakon toga se alocira memorija za *vertexe* te se pošalje na grafičku karticu. Potrebno je i napraviti *shader program* koji će koristiti prethodno navedene *shadere*.

```
ShaderGL shaderMesh;
shaderMesh.addShaderStage("shaderSimpleMesh.task", ShaderStage::TASK);
shaderMesh.addShaderStage("shaderSimpleMesh.mesh", ShaderStage::MESH);
shaderMesh.addShaderStage("shaderSimpleMesh.frag", ShaderStage::FRAGMENT);
shaderMesh.createProgram();
```

Slika 4.12 Generiranje *shader programa*

Također se trebaju pripremiti i oba *index buffera*. *Meshlet buffer* je poseban *buffer* gdje se nalaze definicije *meshleta*, on je jedan veoma varijabilan dio programa pa nije opisan jer će previše se razlikovati od programa do programa.

```
BufferGL vertexIndexBuffer, meshletIndexBuffer, meshletBuffer;
vertexIndexBuffer.allocateStorage(meshletMesh.vertexIndices.data(),
meshletMesh.vertexIndices.size() * sizeof(int));
meshletIndexBuffer.allocateStorage(meshletMesh.meshletIndices.data(),
meshletMesh.meshletIndices.size() * sizeof(uint8_t));
meshletBuffer.allocateStorage(meshletMesh.meshletDescriptions.data(),
meshletMesh.meshletDescriptions.size() * sizeof(MeshletDescription));
vertexIndexBuffer.bindToIndex(1, IndexType::SHADER_STORAGE);
meshletIndexBuffer.bindToIndex(2, IndexType::SHADER_STORAGE);
meshletBuffer.bindToIndex(3, IndexType::SHADER_STORAGE);
```

Slika 4.13 Generiranje ostalih *buffera*

U glavnoj petlji gdje se izvršava iscrtavanje potrebno je poslati odgovarajuće *uniform* varijable *shader programu*. Na kraju se poziva funkcija koja pokreće određen broj *task shadera*.

```
shaderMesh.use();
shaderMesh.setMat4f("viewProjection",
camera.getProjectionMatrix(window.getAspectRatio()) * camera.getViewMatrix());
shaderMesh.setInt("meshletCount", meshletMesh.meshletDescriptions.size());
shaderMesh.setVec3f("bbMin", mesh.boundingBox.minimum);
shaderMesh.setVec3f("bbMax", mesh.boundingBox.maximum);
shaderMesh.setVec3fArray("planes", (glm::vec3*)&planes, 12);

shaderMesh.setVec3f("lightDirection", glm::normalize(glm::vec3(1, 0.3, 0.3)));
shaderMesh.setVec3f("lightAmbient", glm::vec3(1));
shaderMesh.setVec3f("lightDiffuse", glm::vec3(1));
shaderMesh.setVec3f("lightSpecular", glm::vec3(1));

shaderMesh.setVec3f("cameraPos", camera.position);
glMultiDrawMeshTasksIndirectNV(0, commandsMesh.size(),
sizeof(MultiDrawMeshCommand));
```

Slika 4.14 Postavljanje varijabli *shaderu* te pokretanje iscrtavanja

5. USPOREDBA ALGORITAMA

A sljedećim tablicama će biti uspoređeni klasični način iscrtavanja i moderni u raznim okolnostima, različite količine detalja objekata, različiti broj objekata i slično. Prve 4 tablice (tablice 5.1, 5.2, 5.3, 5.4) predstavljaju podatke skupljene na prijenosnom računalu s Ryzen R5 4600H procesorom i Nvidia GTX1650 grafičkom karticom. Druge 4 tablice (tablice 5.5, 5.6, 5.7, 5.8) su podatci skupljeni na računalu s Intel i7 9700K procesorom i RTX 3080 grafičkom karticom. Odmah nakon prikazanih tablica se nalaze grafički prikazani podatci iz tablica. Vremena prikazana u svim tablicama i grafovima su u milisekundama (ms).

Tablica 5.1 Vrijeme potrebno za iscrtati sfere klasičnim načinom

		Broj sfera				
		1	4	9	16	25
Broj trokuta	81,920	0.25	0.29	0.45	0.70	1.00
	5,242,880	2.5	9.4	20.7	36.4	56.50

Tablica 5.2 Vrijeme potrebno za iscrtati sfere modernim načinom gdje je veličina *meshleta* maksimalno 32 *vertexa* i 40 trokuta

		Broj sfera				
		1	4	9	16	25
Broj trokuta	81,920	0.20	0.21	0.35	0.45	0.60
	5,242,880	0.9	3.3	7.1	12.6	19.50

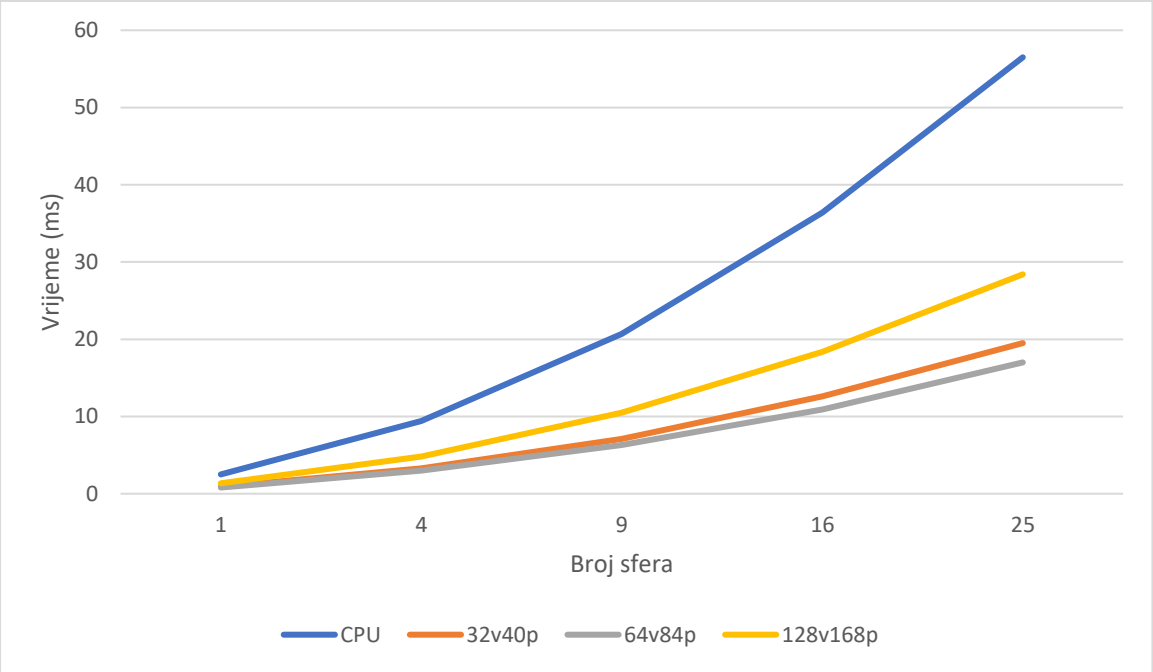
Tablica 5.3 Vrijeme potrebno za iscrtati sfere modernim načinom gdje je veličina *meshleta* maksimalno 64 *vertexa* i 84 trokuta

		Broj sfera				
		1	4	9	16	25
Broj trokuta	81,920	0.25	0.3	0.4	0.5	0.6
	5,242,880	0.8	3	6.3	10.9	17

Tablica 5.4 Vrijeme potrebno za iscrtati sfere modernim načinom gdje je veličina *meshleta* maksimalno 128 *vertexa* i 168 trokuta

		Broj sfera				
		1	4	9	16	25
Broj trokuta	81,920	0.2	0.25	0.4	0.52	0.74
	5,242,880	1.35	4.8	10.5	18.35	28.4

Na grafu ispod je prikazana ovisnost vremena iscrtavanja u milisekundama o broju sfera u sceni. Prikazane su sve četiri metode iscrtavanja u slučaju kada su sfere velike (5,242,880 trokuta). Radi vizualne jednostavnosti kod modernih *shadera* oni koji imaju veličina *meshleta* 32 *vertexa* i 40 trokuta, umjesto da se zapisuje sve, skraćeno je na naziv „32v40p“ u grafovima. Naziv je identičan i za druge vrijednosti.



Graf 5.1 Ovisnost vremena izvođenja o načinu iscrtavanja na prijenosnom računalu

Tablica 5.5 Vrijeme potrebno za iscrtati sfere klasičnim načinom

		Broj sfera						
		1	4	9	16	25	64	256
Broj trokuta	81,920	0.13	0.15	0.16	0.21	0.23	0.45	1.55
	5,242,880	0.53	1.9	4.17	7.32	11.35	27	99.25

Tablica 5.2 Vrijeme potrebno za iscrtati sfere modernim načinom gdje je veličina *meshleta* maksimalno 32 *vertexa* i 40 trokuta

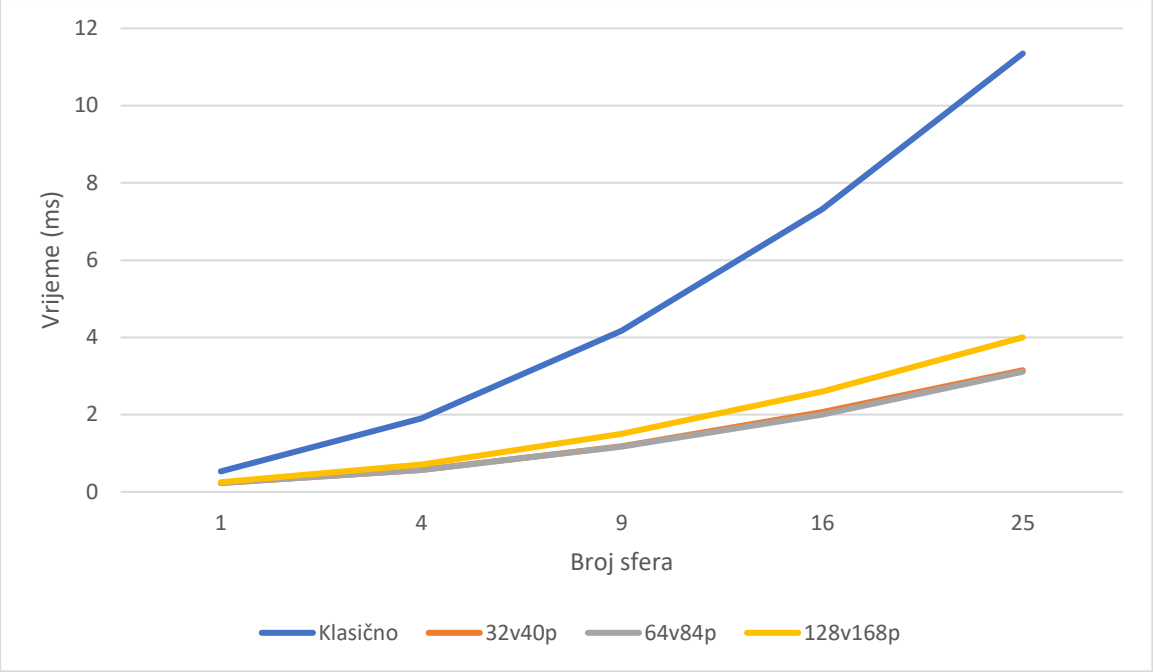
		Broj sfera						
		1	4	9	16	25	64	256
Broj trokuta	81,920	0.11	0.12	0.13	0.16	0.2	0.29	0.47
	5,242,880	0.23	0.57	1.18	2.07	3.15	6.9	19.35

Tablica 5.2 Vrijeme potrebno za iscrtati sfere modernim načinom gdje je veličina *meshleta* maksimalno 64 *vertexa* i 84 trokuta

		Broj sfera						
		1	4	9	16	25	64	256
Broj trokuta	81,920	0.11	0.12	0.13	0.16	0.2	0.29	0.5
	5,242,880	0.23	0.57	1.17	2	3.11	6.8	18.33

Tablica 5.2 Vrijeme potrebno za iscrtati sfere modernim načinom gdje je veličina *meshleta* maksimalno 128 *vertexa* i 168 trokuta

		Broj sfera						
		1	4	9	16	25	64	256
Broj trokuta	81,920	0.1	0.12	0.14	0.16	0.2	0.3	0.54
	5,242,880	0.25	0.71	1.5	2.6	4	8.6	22.6



Graf 5.2 Ovisnost vremena izvođenja o načinu iscrtavanja na stolnom računalu

Na grafovima 5.1 i 5.2 se vidi da je izvođenje pomoću klasičnih shadera znatno sporije od modernih zbog prethodno spomenute nemogućnosti odbacivanja velikog broja trokuta. Implementacije s *mesh shaderima* su osjetno brže, no i u njima se vidi razlika u vremenima izvođenja. Najbrža varijanta je ona s 64 *vertexa* i 84 trokuta. Razlog tome je vrlo vjerovatno to što se memorija zauzima u blokovima koji su višekratnici od 128 bajtova. Što se uzima više trokuta po *meshletu* time količina memorije koja je potrebna raste. U određenom trenutku veličina *meshleta* ima negativan učinak na performanse.

Jedina stvar u ovom radu koja je zaslužna za toliku razliku u performansama je mogućnost da se uklanjaju *meshleti* koji nisu vidljivi. Implementacija samo toga je zaslužna za razliku u performansama. Može se i dalje nastaviti čak, moguće je da se u *mesh shaderu* prepozna koliko je *meshlet* velik na ekranu (po broju piksela) pa da se ne mora cijeli iscrtati nego samo jedan maleni dio. Također se mogu dinamički birati modeli s manje detalja ovisno o udaljenosti od kamere. Postoji velik broj optimizacija koje se mogu izvesti, uklanjanje *meshleta* koji se ne vide znatno poboljšava performanse, implementacijom i ostalih metoda se sigurno može daleko više postići nego u ovom radu.

6. ZAKLJUČAK

Iscrtavanje 3D objekata koristeći klasičan način iscrtavanja je vrlo jednostavan proces iscrtavanja. Proces ne zahtjeva veliki trud da se upogoni i da funkcionira kako se očekuje. Zbog te jednostavnosti taj postupak ima nekoliko mana. Upravo zato što je jednostavan to znači da ne postoji fina kontrola nad cijelim procesom, samo određenim djelovima. To znači da u kompleksnijim scenama ta činjenica ograničava performanse zbog nemogućnosti prilagodbe uvjetima. Implementiranje moderne tehnike znatno poboljšava fleksibilnost i mogućnosti sustava do te mjere da se mogu primijetiti znatna poboljšanja u performansama. Poboljšavanje performansi dolazi zbog činjenice da je novi oblik prikazivanja objekata objektivno bolji te omogućuje mnogo više prilika za optimiziranje. Jedna od optimizacija koja je bila glavna u ovom radu je to da se može odbacivati velik broj trokuta prije iscrtavanja što sveukupno uštedi mnogo vremena. Za manje detaljne scene ili za sustave gdje nisu potrebne visoke performanse klasični način crtanja je i dalje odličan jer je on poznata i ustaljena tehnologija, no ako je potrebno ciljati na visoke performanse onda moderni način iscrtavanja pruža znatno više mogućnosti i prilika za optimiziranje.

LITERATURA

[1] https://registry.khronos.org/OpenGL/extensions/NV/NV_mesh_shader.txt (zadnji pristup 9.2023)

[2] https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/ (zadnji pristup 9.2023)

[3] https://github.com/nvpro-samples/gl_vk_meshlet_cadscene (zadnji pristup 9.2023)

[4] https://www.unrealengine.com/en-US/unreal-engine-5 (zadnji pristup 9.2023)

[5] https://www.khronos.org/opengl/wiki/File:RenderingPipeline.png (zadnji pristup 9.2023)

[6] https://www.youtube.com/watch?v=qC5KtatMcUw (zadnji pristup 9.2023)

[7] https://www.khronos.org/opengl/wiki/Compute_Shader (zadnji pristup 9.2023)

SAŽETAK

Iscrtavanje 3D objekata je postupak koji ima veliku važnost u modernom svijetu. Mnogo tehnologija i grana znanosti se oslanja na iscrtavanja da bi bile korisne. Iscrtavanje se velikom većinom standardiziralo u vrlo jednostavan postupak. Sve započinje *vertex shaderom* čija je zadaća da preuzme podatke objekta koji se iscrtava te da ih određenim postupcima transformira u nešto korisno. Nakon vertex shadera se izvršava *fragment shader* koji uzme podatke proizvedene u *vertex shaderu* te ih iskoristi da bi se proizveli pikseli koji se iscrtaju i prikažu korisniku. Takav klasični pristup iscrtavanju je veoma jednostavan no ima nedostatke kada su u pitanju modeli s velikim brojem trokuta. Rješenje za razne probleme klasičnog iscrtavanja daju modeni *mesh shaderi*. Oni mijenjaju logički slijed iscrtavanja te daju drastično veće mogućnosti i sposobnosti korisniku. Moderno iscrtavanje započine s *task shaderom* koji uzme podatke objekta te odlučuje koji dijelovi objekta se trebaju iscrtavati, a koji ne. *Task shader* zatim pokreće jedan *mesh shader* za svaki dio objekta koji je vidljiv. *Mesh shader* zatim izvodi učitavanje i postavljanje podataka za iscrtavanje. Nakon *mesh shadera* se poziva *fragment shader* koji proizvodi piksele identično kao i u klasičnom slučaju. Budući da je struktura podataka drukčija u modernom slučaju napisan je algoritam koji pretvara obične *mesheve* objekata u oblik pogodan za moderne *shadere*. Rezultati modernih shadera su vizualno identični kao i klasični, ali postupak je višestruko učinkovitiji te ima mnogo bolje performanse. Performanse moderne metode su redovito više od 3 puta veće od klasičnog iscrtavanja.

Ključne riječi: algoritam, iscrtavanje, mesh shaderi, OpenGL, performanse

ABSTRACT

Rendering 3D objects is a precedure of great value in the modern world. Many technologies and sciences rely on rendering in order to be useful. Rendering has largely standardized into a very simple procedure. It begins with a vertex shader whose job it is to take the object data and process them in some way in order to transform them into something useful. After it comes the fragment shader which takes the vertex shader data and uses it to produce individual pixels which are then rendered and displayed to the user. The classic way of rendering is very simple but has some downsides when it comes to models with a large number of triangles. The solution to the problems come in the form of modern mesh shaders. They completely change the logic of rendering and in doing so have far more features and capabilities. Modern rendering starts with the task shader which uses the object data to determine which parts are visible and which are not. It then starts executing a mesh shader for every part of the object that is visible. The mesh shader then loads and sets the data that needs to be rendered. After it comes the fragment shader which is virtually identical to the classic case. Considering that the data structure is different an algorithm was written which converts the usual object mesh into a form usable by the modern shaders. The resulta of the shaders are visually identical to the classic case, however the procedure is much more efficient and has better performance. The performance is regularly over 3 times greater than the classic case.

Keywords: algorithm, mesh shaders, OpenGL, performance, rendering