Izrada demonstracijske aplikacije za edukaciju korištenjem OpenGL biblioteke

Cvrk, Ivan

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:661490

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-23**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1553

IZRADA DEMONSTRACIJSKE APLIKACIJE ZA EDUKACIJU KORIŠTENJEM OPENGL BIBLIOTEKE

Ivan Cvrk

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1553

IZRADA DEMONSTRACIJSKE APLIKACIJE ZA EDUKACIJU KORIŠTENJEM OPENGL BIBLIOTEKE

Ivan Cvrk

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1553

Pristupnik:	**Ivan Cvrk (0036540698)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	izv. prof. dr. sc. Mirko Sužnjević

Zadatak: **Izrada demonstracijske aplikacije za edukaciju korištenjem OpenGL biblioteke**

Opis zadatka:

Područje računalne grafike bavi se stvaranjem i manipulacijom vizualnih elemenata na računalnom zaslonu radi stvaranja realističnih slika i animacija. OpenGL je biblioteka otvorenog koda za računalnu grafiku koja pruža programsko sučelje za izradu 2D i 3D grafike. Koristi se široko u industriji videoigara, simulacija, znanstvenih vizualizacija i drugim područjima gdje je potrebna visoka razina vizualne obrade na računalima. Cilj ovog rada je proučiti relevantne biblioteke za računalnu grafiku, s posebnim fokusom na OpenGL. Potom je potrebno razviti jednostavnu demonstracijsku aplikaciju koristeći navedenu OpenGL biblioteku. U okviru razvoja demonstracijske aplikacije potrebno je demonstrirati osnovne funkcionalnosti OpenGL biblioteke kao što su iscrtavanje poligona, transformacija koordinatnog sustava, svjetla i slično. Potrebno je sustavno dokumentirati svaki pojedinačni korak u izradi te jasno definirati i dokumentirati sve programske zadatke koje bi budući studenti mogli samostalno izvršiti kako bi replicirali aplikaciju. Programski kod treba biti jednostavan te izrađen u dvije varijante: kompletna i nekompletna varijanta. Iz nekompletnih verzija budući studenti će trebati razviti kompletiran programski kod. Potrebno je jasno definirati zadatke koji će biti praćeni kako bi se replicirala izrada takvog programskog koda.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

1. Uvodni pregled grafičkih programskih sučelja **2**

2. Pokrivenost i demonstracija OpenGL-a **5**

3. Tehnička izvedba demonstracijskih primjera **10**

3.1. Korištene tehnologije 10

3.2. Demonstracijske aplikacije 11

3.3. Zadatci za studente 14

4. Primjena u nastavi **19**

Literatura **21**

Sažetak **23**

Abstract **24**

A: Upute za laboratorijsku vježbu **25**

1. Uvodni pregled grafičkih programskih sučelja

Područje računalne grafike bavi se stvaranjem i manipulacijom vizualnih elemenata na računalnom zaslonu radi stvaranja realističnih slika i animacija. Manipulacija i prikaz vizualnih elemenata mogu biti računalno vrlo zahtjevni zadatci zbog velikog broja matematičkih izračuna potrebnih za transformaciju geometrije objekata i njihovo sjenčanje. U grafičkim aplikacijama, tijekom procesiranja 3D objekata često je potrebno primijeniti unarne operacije nad velikim skupom ulaznih podataka. Zbog toga je obrada podataka u grafičkim aplikacijama vrlo pogodna za paralelizaciju.

Iako su današnji procesori u računalima dovoljno brzi za izvršavanje većine svakodnevnih zadataka, nisu optimizirani za paralelnu obradu podataka u grafičkim aplikacijama. Za to nam je potrebno specijalizirano sklopovlje - grafički procesori (GPU). Moderni grafički procesori dizajnirani su oko SIMD modela paralelizacije[1]. Zbog inherentno paralelne prirode velikog dijela zadataka, grafičke aplikacije mogu vrlo efikasno koristiti SIMD model paralelizacije grafičkih procesora za izvršavanje unarnih operacija nad vrhovima objekata i slikovnim elementima.

Na tržištu postoji puno modela grafičkih kartica od različitih proizvođača. Kako bi mogli pisati prenosive programe koji koriste pogodnosti grafičkih procesora, potreban nam je standardiziran način za njihov pristup i korištenje. Upravo zbog toga napravljena su različita grafička aplikacijska programska sučelja. Aplikacijsko programsko sučelje (API) skup je pravila i specifikacija koje programeri slijede tijekom razvoja aplikacija s ciljem korištenja usluga i resursa koje API pruža. U kontekstu grafičkih programskih sučelja, API pruža uniforman pristup resursima grafičkih kartica i njihovim uslugama neovisno o njihovoj sklopovskoj implementaciji. Grafička programska sučelja pružaju mnoštvo funkcionalnosti potrebnih za prijenos podataka i njihovu manipulaciju s ciljem

stvaranja slike na računalnom zaslonu. Neka od glavnih grafičkih programskih sučelja su:

- **OpenGL** [2] - Višeplatformsko grafičko programsko sučelje otvorenog koda koje održava Khronos grupa. Koristi se za izradu igrica i CAD programa gdje je prenosivost ključna.
- **Vulkan** [3] - Višeplatformsko grafičko programsko sučelje niske razine. Programeru prepušta velik dio kontrole nad grafičkim procesorom. Koristi se za izradu aplikacija visokih performansi kao što su igrice, simulacije i virtualna stvarnost.
- **DirectX** [4] - Kolekcija Microsoftovih aplikacijskih programskih sučelja za rad s medijima od kojih je Direct3D najistaknutiji za rad s računalnom grafikom.
- **Metal** [5] - Grafičko programsko sučelje niske razine optimiziran za rad na Apple platformama.
- **WebGL** [6] - Grafičko programsko sučelje za renderiranje grafike u web-preglednicima temeljen na OpenGL ES API-ju.

Iako Vulkan danas polako zamjenjuje OpenGL, ovaj rad se fokusira na OpenGL zbog njegove široke podržanosti, zrelosti i jednostavnosti. OpenGL 1.0 izdan je 1992. godine, a verzija koju koristi ovaj rad je 4.6 izdanu 2017. godine [7]. Osim što je OpenGL podržan na gotovo svim operacijskim sustavima, ima i široku podršku na raznom sklopovlju [8].

U usporedbi s ostalim grafičkim programskim sučeljima OpenGL je relativno jednostavan za korištenje, no to ne znači da pruža manje usluga od ostalih. Većina koncepata koji postoje u grafičkom sučelju Vulkan postoje i u OpenGL-u, samo što programer ima manju kontrolu nad njima, što je za našu primjenu u edukacijske svrhe pogodno jer moramo napisati manje koda kako bi vidjeli iste rezultate.

Grafički protočni sustav glavni je dio procesa renderiranja geometrijskih primitiva na zaslon ekrana. Sastoji se od niza koraka koji transformiraju i procesiraju ulazne podatke s ciljem generiranja slike. Za izvođenje tih koraka brine se OpenGL, a na programeru je da ih konfigurira i pruži ulazne podatke.

U starijim verzijama OpenGL-a programer je imao vrlo ograničen pristup i moguć-

nosti konfiguracije grafičkog protočnog sustava i ulaznih podataka. Od verzije 3.0 OpenGL API mijenja paradigmu i grafički protočni sustav više nije predefiniran nego se programerima dopušta prilagođavanje postupka renderiranja definiranjem programa sjenčara. Također, programerima se prepušta upravljanje memorijom kako bi mogli efikasnije koristiti usluge GPU-a s obzirom na potrebe aplikacije koju razvijaju.

Tema ovog rada je proučiti, demonstrirati i dokumentirati korištenje modernog OpenGL grafičkog programskog sučelja s ciljem stvaranja edukacijskog sadržaja za buduće studente. Dokumentacija i demonstracija je napravljena u obliku laboratorijskih vježbi kako bi se studenti na praktičan način mogli detaljnije upoznati sa sučeljem. Studenti bi nakon izrade vježbi trebali biti upoznati s osnovnim konceptima OpenGL-a što će im dati dublje razumijevanje na koji se način izvode grafičke aplikacije u računalu, te ih pripremiti za susret s naprednijim i zahtjevnijim grafičkim sučeljima kao što je Vulkan ako se za to odluče.

2. Pokrivenost i demonstracija OpenGL-a

Kako sam spomenuo u prošlom poglavlju, jedan od zadataka je bio dokumentirati izradu demonstracijskih aplikacija što sam napravio u formatu laboratorijske vježbe za studente. Cijela dokumentacija postupka dostupna je u privitku gdje su detaljno objašnjeni svi obrađeni dijelovi OpenGL-a. U ovom poglavlju htio bih se osvrnuti na to koji dijelovi OpenGL-a su odabrani za demonstraciju, a koji nisu, te prokomentirati zašto je takva odluka donesena.

Kako je grafički protočni sustav ključna komponenta za renderiranje vizualnih elemenata na zaslonu računala, upute za laboratorijsku vježbu prvo objašnjavaju grafički protočni sustav u OpenGL-u na visokoj razini. Tu su ukratko opisani sjenčar vrhova, sjenčar teselacije, sjenčar geometrije, postprocesiranje vrhova, sastavljanje primitiva, rasterizacija, sjenčar fragmenata i dodatne operacije nad fragmentima. Većina ovih koraka je u potpunosti programirljiva ili konfigurabilna kroz moderni OpenGL API.

Kako bi išta mogli nacrtati na ekran, u modernom OpenGL-u nužno je definirati ulazne podatke, sjenčar vrhova i sjenčar fragmenata. Definicijom ta tri elementa uspostavlja se funkcionalni grafički protočni sustav spreman za korištenje. Ostale komponente se ili izostavljaju ili imaju pretpostavljenu konfiguraciju definiranu API-jem.

U svrhu demonstracije OpenGL-a, odabrao sam pokriti implementaciju samo nužnih dijelova grafičkog protočnog sustava; sjenčar vrhova i sjenčar fragmenata. Sjenčari u OpenGL-u pišu se u *OpenGL Shading Language* (GLSL) jeziku. Potrebno ih je napisati i učitati u aplikaciju u tekstualnom obliku, kompajlirati, te povezati u program sjenčara. Nakon toga ih možemo slobodno koristiti u grafičkom protočnom sustavu.

Zadaća sjenčara vrhova je obaviti unarne operacije nad vrhovima objekata koje ren-

deriramo. Jedan vrh sastoji se od svojih koordinata, a može sadržavati i dodatne atribute kao što su vektor smjera normale, UV koordinate teksture, boja, itd. Najčešća operacija koja se ovdje provodi je transformacija koordinata vrha iz lokalnog koordinatnog sustava objekta u koordinatni sustav scene, kamere i konačno u koordinatni sustav projekcije na ekranu.

Zadaća sjenčara fragmenata je izračunati boju fragmenta rasterizirane primitive. Ovaj izračun najčešće ovisi o boji objekta, teksturi, količini svjetlosti koja pada na fragment i mnogim drugim faktorima ovisno o efektu koji se želi postići sjenčanjem. Ulazni podatci u sjenčar fragmenata jednaki su vrijednostima izlaznih varijabli prethodnog sjenčara interpoliranim po vrhovima primitive koja se rasterizira. U demonstracijskim primjerima i uputama za laboratorijsku vježbu implementirao sam 3 različita načina sjenčanja:

1. sjenčanje fragmenata u ovisnosti o boji vrhova trokuta i udaljenosti od njih,
2. sjenčanje konstantnom bojom,
3. pojednostavljeni model Phongovog osvjetljenja [9].

Demonstrirao sam i korištenje *uniform* varijabli. Vrijednosti *uniform* varijabli postavljaju se prije pokretanja grafičkog protočnog sustava i vidljive su svim sjenčarima, te njihova vrijednost ostaje nepromijenjena sve dok ju programer sam ne promijeni.

Definicijom ova dva sjenčara pokrivena je većina funkcionalnosti potrebna za renderiranje u većini grafičkih aplikacija. Ostali sjenčari su preskočeni zato što oni služe za efikasno postizanje specijalnih efekata što nije u tema ovog rada.

Nakon što smo definirali programirljivi i obavezni dio grafičkog protočnog sustava, sljedeći koncept na koji se fokusiraju demonstracijski primjeri i upute za laboratorijsku vježbu je pohrana podataka i njihova priprema za korištenje u grafičkom protočnom sustavu.

U starijim verzijama OpenGL-a, prije renderiranja svake slike bilo je potrebno definirati svaki vrh svakog objekta koji smo željeli nacrtati. Ovaj pristup definiranja ulaznih podataka u grafički protočni sustav je vrlo neefikasan zato što ne iskorištava mogućnost paralelizacije ove operacije. Kao optimizaciju ovog postupka OpenGL je definirao liste.

Liste su objekti starog OpenGL API-ja koji su držali podatke, o vrhovima geometrijskih primitiva od kojih se sastoje 3D objekti, u memoriji grafičke kartice. Na ovaj način se izbjegla potreba da procesor mora iterirati sve vrhove svih primitiva za svako iscrtavanje slike na ekran jer se programer mogao referirati na jednom pohranjene podatke na grafičkoj kartici svaki puta kada bi htio iskoristiti te primitive za renderiranje. Iako je ovaj način pohrane podataka prihvatljiv za neke primjene, on i dalje nije idealan zato što programer nema utjecaja na to u koji dio memorije na grafičkoj kartici će se podatci pohraniti, nema utjecaj na njihov raspored u memoriji i ne može ih mijenjati.

Zbog navedenih nedostataka u definiranju i korištenju memorije, OpenGL nakon verzije 3.0 u potpunosti mijenja pristup na koji način se definiraju ulazni podatci u grafički protočni sustav, te prelazi na moderni koncept *buffer* objekata. *Buffer* objekti su međuspremnici za držanje proizvoljnih podataka u memoriji grafičke kartice, a na programeru je da se pobrine za njihovo stvaranje, da definira kako se podatci čitaju i dohvaćaju iz međuspremnika u grafički protočni sustav i da se pobrine za njihovo brisanje. Na ovaj je način prebačen dio odgovornosti na programera, ali ovako programer ima i više mogućnosti za optimizaciju svoje aplikacije.

Neke od optimizacija koje su dostupne ako se koriste *buffer* objekti su da se podatci mogu pohraniti u dio memorije grafičke kartice koji nije nužno mapiran na memoriju vidljivu iz glavnog procesora. Na ovaj način se ubrzava dohvat podataka kroz memorijsku hijerarhiju do registara grafičkih procesora. Odluka o tome u koji dio memorije je efikasnije pohraniti podatke ovisi o njihovom korištenju i o tome koliko se često mijenjaju, što može znati samo programer. Broj memorijskih alokacija koji se može napraviti na grafičkoj kartici je ograničen. Zbog toga je poželjno spremiti podatke koji se uvijek zajedno koriste i jednako često mijenjaju u jedan međuspremnik.

U uputama za laboratorijsku vježbu, što se tiče memorijskih koncepata u OpenGL-u, prvo je opisan *Vertex Array object* (VAO). VAO je struktura u OpenGL API-ju koja sadrži opisnike atributa svih vrhova, odnosno opisnike svih ulaznih podataka u grafički protočni sustav potrebnih za jedan njegov prolaz. Kako su programeri slobodni rasporediti ulazne podatke u memoriji kako žele, mora postojati mehanizam kojim će definirati kako se ti podatci trebaju dohvaćati iz memorije u grafički protočni sustav. Upravo tome služi VAO. Nakon predstavljanja VAO objekata, upute objašnjavaju stvaranje *buffer* objekata i

učitavanje podataka u njih.

Sljedeći bitan koncept koji je objašnjen u uputama je indeksiranje vrhova. Općenito, ne moramo definirati vrhove geometrijskih primitiva od kojih se sastoje objekti za prikaz onim redoslijedom kojim ih želimo nacrtati. Primitive u velikim 3D modelima često dijele vrhove i ako bi te vrhove htjeli navesti onim redoslijedom kojim se trebaju iscrtati, odnosno povezati kako bi stvorili primitive, morali bi navesti puno duplikata što bi rezultiralo zauzećem puno više memorije nego što nam je zapravo potrebno. Na primjer, kocka sastavljena od trokuta sastoji se od 8 vrhova i 12 trokuta gdje jedan vrh može dijeliti čak 6 različitih trokuta. Zbog toga sam odlučio u uputama i primjerima demonstrirati indeksiranje vrhova kao jednu od osnovnih metoda optimizacije memorijske potrošnje u OpenGL-u. Upute još kratko objašnjavaju metode crtanja objekata ovisno o tome koriste li se indeksi vrhova ili ne.

Ovime su definirani svi dijelovi OpenGL API-ja potrebni za osnovno renderiranje objekata na ekran. Moderni OpenGL je dosta zahtjevniji za rad od starih verzija i studenti moraju dobro razumjeti ove osnovne koncepte OpenGL-a kako bi mogli uspostaviti funkcionalan grafički protočni sustav koji mogu koristiti za renderiranje. Zbog toga sam odlučio ograničiti opseg demonstracije OpenGL-a na ovaj opisani podskup funkcionalnosti.

Ostale funkcionalnosti OpenGL-a koje nisu opisane ali su vrijedne za spomenuti su:

- uniform buffer objekti - Omogućuju pohranu niza vrijednosti uniformnih varijabli u međuspremnike. Koriste se za instancirano renderiranje jer možemo pohraniti vrijednosti uniformnih varijabli svakog objekta, na primjer matrice modela, u polje koje predamo programu sjenčara.

- teksture - OpenGL ima podršku za razne tipove tekstura od kojih su najvažnije 1D, 2D, 3D i *cubemap* teksture. Teksture se ponajprije koriste za dodavanje detalja na objekte u sceni primjenom slika na njihove površine.

- framebuffer objekti - Fleksibilni objekti koji omogućuju konfiguriranje različitih ciljeva renderiranja. Osim renderiranja na ekran, moguće je renderirati slike u teksture ili međuspremnike koji se kasnije mogu koristiti za dodatno procesiranje

i dodavanje raznih efekata.

Upute za laboratorijsku vježbu kroz dodatne zadatke potiču studente da samostalno istraže ostale mogućnosti OpenGL-a, no to nije obavezno.

Osim što se u uputama obrađuju koncepti OpenGL API-ja, demonstrira se i primjena linearnih transformacija koordinatnog sustava kako bi se ostvarile transformacije translacije, rotacije i skaliranja. Također, kako bi lakše raspoznali dijelove 3D objekata u sceni implementiran je jednostavan primjer Phongovog modela osvjetljenja [9]. Nije implementirano Phongovo sjenčanje u kojem se računa srednja vrijednost normala u vrhovima koja rezultira glatkom, zaobljenom površinom objekata zato što smatram da bi ovaj izračun nepotrebno zakomplicirao demonstracijske programe.

3. Tehnička izvedba demonstracijskih primjera

3.1. Korištene tehnologije

OpenGL je moguće koristiti iz raznih programskih jezika. Za izradu demonstracijskih primjera odabrao sam programski jezik C++ zato što je brz i često se koristi za izradu grafičkih aplikacija gdje su performanse ključne. Također, OpenGL je izvorno dizajniran za korištenje s C i C++ programskim jezicima zbog čega se iz tih programskih jezika može koristi u svojem izvornom obliku. Za korištenje OpenGL-a iz ostalih programskih jezika potrebno je koriti vezivne biblioteke koje omogućuju pristup izvornom OpenGL-u što dodaje nepotreban sloj indirekcije.

Funkcije OpenGL API-ja pružene su od strane *driver* programa za grafičko sklopovlje. OpenGL se mora moći jednostavno i neovisno mijenjati od aplikacija koje ga koriste, zbog čega se ne povezuje statički u izvršne datoteke aplikacija, nego se aplikacije moraju pobrinuti da dinamički učitaju sve funkcije API-ja koje im trebaju. Za dinamičko učitavanje OpenGL API-ja u demonstracijskim primjerima iskoristio sam programski kod generiran na Glad web stranici [10].

Kako je OpenGL u potpunosti agnostičan od operacijskog sustava na kojem se koristi, nema nikakvu potporu za stvaranje prozora i praćenje korisničkog unosa kroz periferne uređaje miš i tipkovnicu. Za taj zadatak sam iskoristio GLFW biblioteku [11]. GLFW je višeplatformska biblioteka za upravljanje prozorima i korisničkim unosom u aplikacijama koje koriste OpenGL, OpenGL ES i Vulkan grafička programska sučelja. Biblioteka je jednostavna i intuitivna za korištenje s vrlo detaljnom dokumentacijom zbog čega mislim da će biti jednostavna studentima za korištenje i praćenje u demonstracijskim primjerima. Studentima sam priložio izvorni kod ove biblioteke zajedno s kodom

demonstracijskih aplikacija kako bi osigurao da ju svi mogu kompajlirati i koristiti neovisno o operacijskom sustavu kojeg imaju.

Posljednja biblioteka koju sam koristio je *OpenGL Mathematics* (GLM) [12]. To je C++ matematička biblioteka za grafičke programe temeljena na GLSL specifikaciji. Koristio sam ju za provedbu matematičkih izračuna transformacije koordinatnog sustava i pohranu podataka koji se prenose u sjenčare zato što se biblioteka brine za ispravno raspoređivanje podataka u memoriji kako bi se jednostavno mogli mapirati u GLSL tipove podataka prilikom kopiranja u *uniform* varijable.

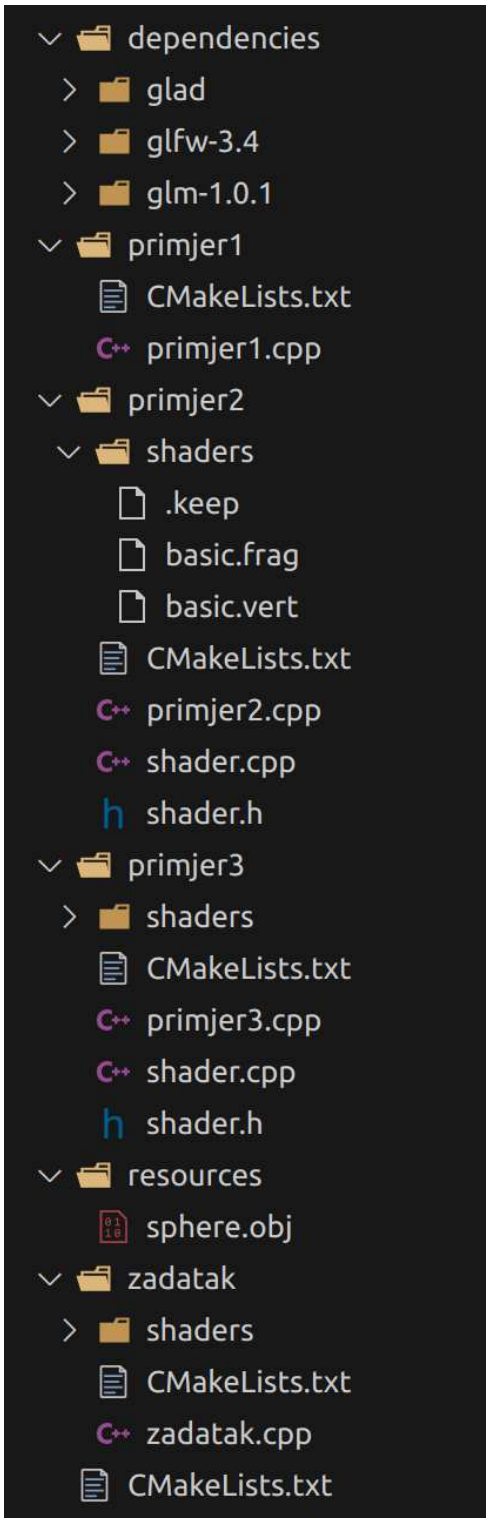
Za sustav izgradnje koristio sam CMake alat [13]. Odabrao sam CMake zato što se takav sustav izgradnje može koristiti na različitim operacijskim sustavima s različitim uređivačima teksta i okruženjima za programiranje. Bitan aspekt OpenGL-a je da se može koristiti na različitim operacijskim sustavima, te mi je zbog toga bilo bitno osigurati da studenti ne budu ograničeni sustavom izgradnje.

3.2. Demonstracijske aplikacije

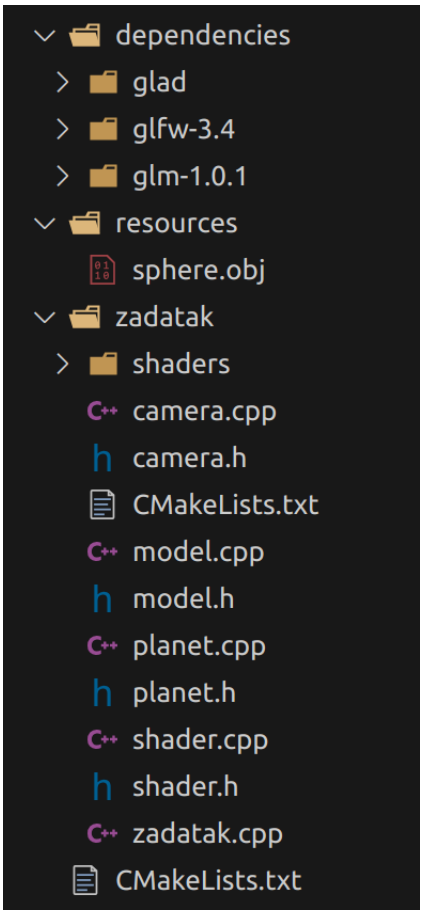
Uz upute za provođenje laboratorijskih vježbi, napravio sam i tri demonstracijske aplikacije, jednu nepotpunu aplikaciju koju studenti trebaju samostalno dovršiti i demonstracijsku aplikaciju koja nije za studente jer sadrži implementirana rješenja zadataka iz uputa.

Demonstracijske aplikacije su komplementarne uputama zato što prate jedno drugo. S obzirom na to da je mnogim studentima ovo prvi susret s OpenGL-om, kod je sažet i dobro komentiran kako bi ga studenti mogli jednostavnije pratiti. Demonstracijske aplikacije rastu u složenosti od prve do treće tako da se kompleksnost demonstracije OpenGL-a razrijedi na 3 jednostavnija primjera gdje se svaki sljedeći oslanja na prethodni.

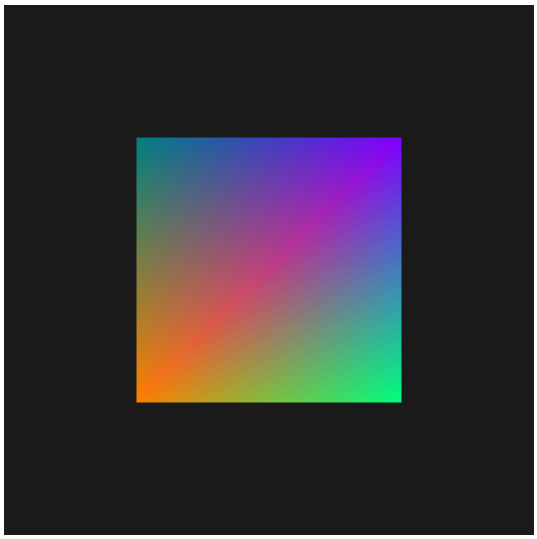
Sve demonstracijske aplikacije napravljene su unutar jednog projekta kako bi mogle dijeliti biblioteke o kojima ovise. Odvojeno sam napravio aplikaciju koja implementira sve obavezne zadatke iz uputa za laboratorijsku vježbu zato što taj kod nije namijenjen za dijeljenje studentima. Struktura projekta demonstracijskih aplikacija vidi se na slici 3.1., a struktura projekta rješenja vidi se na slici 3.2.



Slika 3.1. Struktura primjera

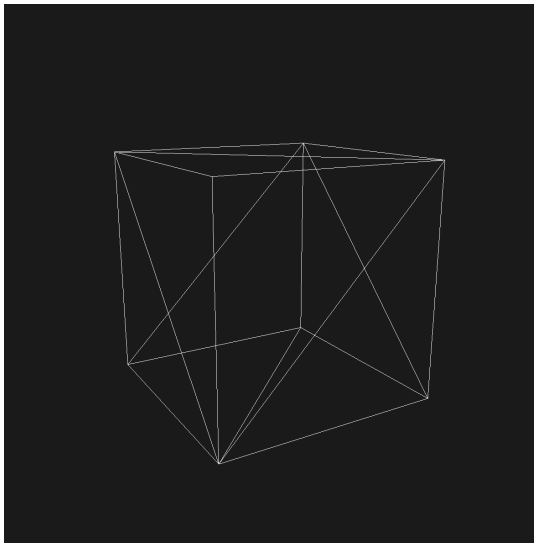


Slika 3.2. Struktura rješenja



Slika 3.3. Demonstracijska aplikacija 1

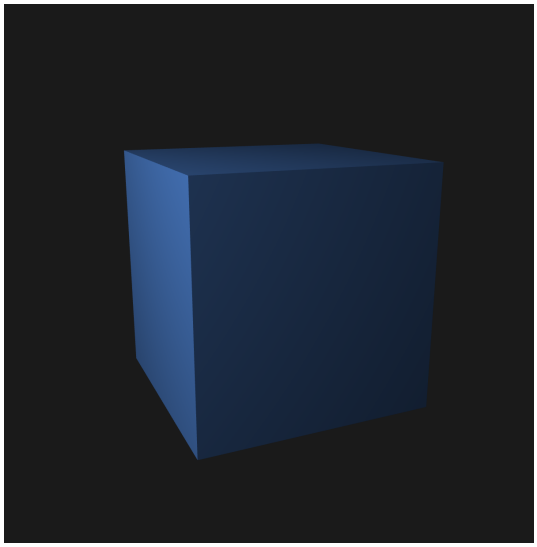
Prva demonstracijska aplikacija vrlo je jednostavna i namijenjena je prvom susretu studenata s OpenGL-om. Cijeli kod smješten je u jednu datoteku *primjer1.cpp* od 184 linije koda. Ideja ove aplikacije je demonstrirati uspostavljanje minimalnog funkcionalnog grafičkog protočnog sustava za crtanje na ekran. Programi sjenčara ne učitavaju se iz vanjske datoteke već je njihov kod hard-kodiran kao konstantna *string* varijabla u aplikaciji. Pokretanjem ove aplikacije na ekranu se prikaže šareni kvadrat (Slika 3.3.).



Slika 3.4. Demonstracijska aplikacija 2

Druga demonstracijska aplikacija demonstrira transformacije koordinatnog sustava. Grafički protočni sustav ove aplikacije je identičan prošloj aplikaciji, uz promjene da se sada sjenčari učitavaju iz vanjske datoteke i koriste se uniformne varijable za prije-

nos transformacijskih matrica koordinatnog sustava u sjenčar vrhova. Pokretanjem ove aplikacije na ekranu se prikaže kocka u žičanom formatu koja se kontinuirano rotira u smjeru kazaljke na satu oko svoje osi (Slika 3.4.).



Slika 3.5. Demonstracijska aplikacija 3

Treća demonstracijska aplikacija demonstrira Phongovo osvjetljenje. Glavni fokus ove aplikacije je demonstrirati moć sjenčara u programirljivom grafičkom protočnom sustavu. Implementacija Phongovog osvjetljenja napravljena je u sjenčaru fragmenata 4., a nadahnuta je knjigom Learn OpenGL[14]. Pokretanjem ove aplikacije na ekranu se prikaže kocka osjenčana Phongovim modelom osvjetljenja koja se kontinuirano rotira u smjeru kazaljke na satu oko svoje osi (Slika 3.5.).

3.3. Zadatci za studente

Demonstracijske aplikacije zajedno s uputama trebale bi biti dovoljne studentima da se upoznaju s odabranim dijelovima OpenGL-a opisanim u prošlom poglavlju. U uputama za laboratorijsku vježbu studentima su za kraj zadana tri zadatka kako bi se sami okušali u razvoju grafičke aplikacije korištenjem OpenGL-a.

Prvi zadatak je učitati koordinate vrhova i vektore normala 3D modela sfere iz *Wavefront .obj* datoteke. Za rješavanje ovog zadatka potrebno je biti spretan s C++ programskim jezikom zbog parsiranja datoteke. Nakon što je datoteka parsirana potrebno je stvoriti međuspremničke *buffer* objekte i prebaciti učitane podatke, te definirati atribute

```
#version 460
precision mediump float;
struct Light
{
  vec3 position;
  vec3 color;
  // attenuation
  float constant;
  float linear;
  float quadratic;
};

out vec4 FragColor;
in vec3 Normal;
in vec3 FragPos;

uniform Light light;
uniform vec3 camPos;
uniform vec3 color;

void main()
{
  // Ambient lighting
  float ambientStrength = 0.3;
  vec3 ambient = ambientStrength * color;
  vec3 normal = normalize(Normal);
  vec3 lightDir = normalize(light.position - FragPos);
  vec3 camDir = normalize(camPos - FragPos);
  // Diffuse shading
  float diff = max(dot(normal, lightDir), 0.0);
  vec3 diffuse = light.color * diff * color;
  // Speculat shading
  vec3 reflectDir = reflect(-lightDir, normal);
  float spec = pow(max(dot(camDir, reflectDir), 0.0), 32.0);
  vec3 specular = light.color * spec * color;
  // attenuation
  float distance = length(light.position - FragPos);
  float attenuation = 1.0 / (
            light.constant +
            light.linear * distance +
            light.quadratic * (distance * distance));
  FragColor = vec4((ambient + diffuse + specular) * attenuation, 1.0);
}
```

Isječak koda 3.1. Phongovo osvjetljenje u sjenčaru fragmenata

vrhova u VAO objektu kako bi se učitani podatci mogli koristiti u grafičkom protočnom sustavu.

Za rješavanje ovog primjera predložio sam stvaranje razreda Model. To je vrlo jednostavan razred koji u konstruktor prima ime datoteke koju odmah parsira i učitava podatke. Dodatne metode koje taj razred ima su `draw` i dekonstruktor. U metodi `draw` model se iscrtava pozivom jedne od `glDraw` naredbi ovisno o tome kako su podatci pohranjeni, a u dekonstruktoru se otpuštaju zauzeti resursi.

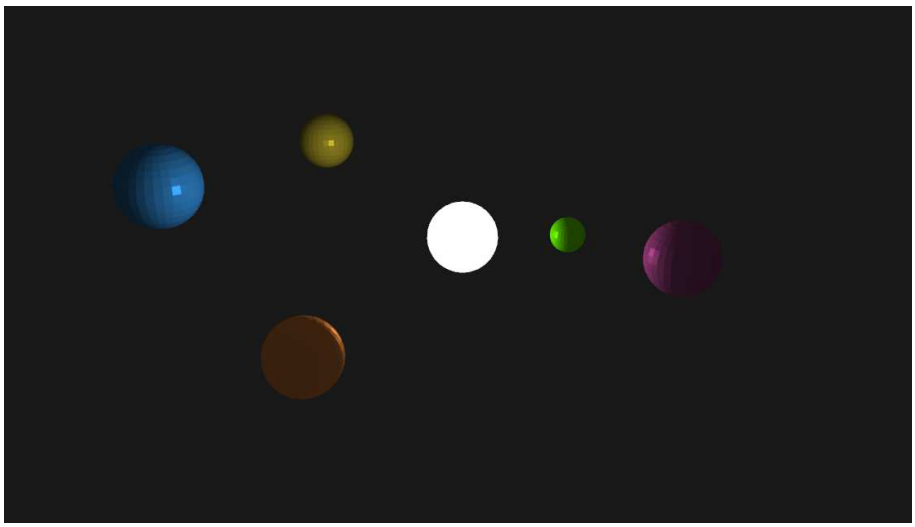
Tema drugog zadatka su transformacije koordinatnog sustava. Ideja je ostvariti vrlo jednostavnu kameru koja se može pomicati u prostoru scene. Kamera bi uvijek trebala gledati u ishodište globalnog koordinatnog sustava i moći micati u xz ravnini rotacijom oko ishodišta. Ovo je relativno jednostavan zadatak jer sve što je potrebno napraviti je osvježiti `LookAt` matricu prije svakog renderiranja slike na ekran ovisno o položaju kamere. Odlučio sam se za ovaj zadatak zato što mislim da je iznimno bitno znati raditi s transformacijama koordinatnog sustava, te smatram da je doživljaj rada s OpenGL-om puno ljepši jednom kad kamera prestane biti statična i dok se omogući kretanje kroz prostor scene. Implementacija potpuno slobodne kamere je dosta složenija i mislim da je nepotrebna za potrebe laboratorijske vježbe. Tako da sam odlučio napraviti kompromis i zadati zadatak s implementacijom slobodne kamere ali pod strogim ograničenjem da uvijek gleda u ishodište koordinatnog sustava.

Kao rješenje ovog zadatka napravio sam razred `Camera` koji enkapsulira položaj kamere i provjerava položaj tipki za kretanje (W, A, S, D) korištenjem GLFW biblioteke prije svakog renderiranja slike na ekran. Implementacijski je ovo jednostavniji zadatak od prvog i trećeg. Stvaranje razreda `Camera` čak nije bilo potrebno već se sve moglo napraviti u glavnoj petlji programa, ali ovako je implementacija ljepša i čišća.

Treći zadatak je simulacija sunčevog sustava u kojoj je potrebno vizualizirati sunce kao izvor svjetlosti i planete koji se okreću oko njega i rotiraju oko svoje osi. Ovdje nema ništa novo što nije viđeno u prošla dva zadatka. Idealno bi bilo ne učitavati više od jednog 3D modela jer svi planeti i sunce su zapravo isti model. Mogu se razlikovati po poziciji u sceni, veličini i boji, a sve to je moguće mijenjati kroz uniformne varijable sjenčara. Model sunčevog sustava pogodan je za vizualizaciju zato što model osvjetljenja koji je

implementiran u trećem demonstracijskom primjeru nema sjene. Vizualizacija bilo koje druge scene bila bi neprirodna bez sjena. U ovoj simulaciji je prihvatljivo da planeti ne bacaju sjenu jedni na druge.

U rješenju trećeg zadatka napravio sam razred `Planet` koji enkapsulira podatke o jednom planetu. Ti podatci su VAO objekt 3D modela, sjenčar, boja, polumjer planeta i udaljenost od sunca. Glavna petlja aplikacije poziva metodu `update` svakog planeta s proteklim vremenom od prošlog poziva te metode. Tu planeti imaju priliku promijeniti svoje 3D transformacije s obzirom na brzinu rotacije oko sunca i oko svoje osi. Zatim se poziva metoda `draw` koja iscrtava 3D model korištenjem predanog sjenčara, no prije toga podešavaju se vrijednosti uniformnih varijabli u sjenčaru jedinstvenima za taj planet. Za sunce nije moguće koristiti isti sjenčar kao za planete zato što bi se izvor svjetlosti nalazio unutar modela sfere. Zbog toga je potrebno napraviti poseban sjenčar za sunce koji samo oboja model u konstantnu bijelu boju. Težina ovog zadatka leži u tome da je potrebno dobro razumjeti sve naučene koncepte OpenGL-a kako bi efikasno mogli doći do rješenja.



Slika 3.6. Demonstracijska aplikacija rješenja

Rješavanjem svih zadataka u sklopu jedne aplikacije, dobit će se zanimljiva dinamična scena koja podsjeća na sunčev sustav. Iako će ukupno rješenje bit nešto veći program, niti jedan zadatak nije previše zahtjevan. Za rješenje svakog zadatka ne bi trebalo više od 100 linija koda. Sav ostali kod koji je potreban za izradu aplikacije studenti mogu preuzeti iz demonstracijskih primjera, a to je učitavanje funkcija OpenGL API-ja, stva-

ranje prozora i učitavanje sjenčara. Svaki zadatak je samostalan i ne ovisi o prethodnim zadatcima. Ovo svojstvo zadataka mi je bilo bitno tijekom njihovog kreiranja zato što nisam htio da ako netko zapne na jednom zadatku bude u nemogućnosti riješiti ostale.

Dodatni zadatci nisu implementirani zato što nisu obavezni. Oni su napravljeni sa svrhom da motiviraju studente na samostalno istraživanje dodatnih metoda renderiranja ako im se svidjelo do sada naučeno.

4. Primjena u nastavi

Cilj ovog rada je proučavanje OpenGL-a i izrada laboratorijske vježbe za kolegij *Osnove virtualnih okruženja*. Na tom kolegiju već postoji laboratorijska vježba iz OpenGL-a ali ona koristi staru verziju API-ja koja se danas više ne koristi za razvoj novih aplikacija i bitno se razlikuje od modernog OpenGL-a. Iako se i moderni OpenGL smatra zastarjelim za današnje standarde, mislim da je idealan za učenje osnovnih koncepata u modernim grafičkim programskim sučeljima. Moderna grafička programska sučelja idu prema tome da programerima daju što više slobode u kreiranju svojih aplikacija. Moderni OpenGL je na tragu toga i svi koncepti koji postoje u njemu postoje i u složenijem i novijem grafičkom sučelju Vulkan. Vulkan je po mojoj procijeni pretežak da bi se radio u sklopu laboratorijske vježbe. Dobar primjer omjera težine je taj da je za izradu replike OpenGL aplikacije od par stotina linija koda u Vulkan-u potrebno tisuće linija za istu funkcionalnost. Cilj laboratorijskih vježbi je edukacija studenata, a ne izrada visoko optimiziranih 3D aplikacija. Nakon što se studenti upoznaju s modernim OpenGL-om, samostalno učenje Vulkan API-ja bi trebalo ići puno jednostavnije ako se za to odluče.

U usporedbi s trenutnom verzijom laboratorijske vježbe iz OpenGL-a, nova verzija napravljena u ovom radu je ipak nešto zahtjevnija. Osim što je potrebno teorijsko poznavanje API-ja, potrebno je uspostaviti funkcionalan grafički protočni sustav prije nego što je moguće bilo što renderirati na ekran. Mislim da je kompliciranje ove laboratorijske vježbe neizbježno ako se želi održati ju relevantnom. No kako se ne bi značajno otežala izvedba laboratorijske vježbe, ključno je pripremiti detaljne upute s dobrim demonstracijskim primjerima što mislim da je to ovaj rad uspio postići.

Moram priznati da još nitko nije pokušao riješiti novu verziju laboratorijske vježbe tako da nemam povratnu informaciju o tome koliko je teška. Bilo bi idealno pronaći nekoga tko nije radio s modernim OpenGL-om da ih pokuša riješiti. S obzirom na to da

se težina ove laboratorijske vježbe temelji isključivo na mojoj procjeni, moguće je da će ju trebati korigirati prije uvođenja u nastavu.

Smatram da se laboratorijska vježba dobivena kao produkt ovog rada treba što prije uključiti u nastavu kako bi kolegij *Osnove virtualnih okruženja* ostao u korak s vremenom i kako bi se studenti bolje pripremili za profesionalni rad u tom području.

Literatura

[1] M. J. Flynn, “Some computer organizations and their effectiveness”, *IEEE Transactions on Computers*, sv. C-21, br. 9, str. 948–960, 1972. https://doi.org/10.1109/TC.1972.5009071

[2] The Khronos Group, https://www.khronos.org/opengl/wiki/Main_Page, [mrežno; stranica posjećena: lipanj 2024.].

[3] ——, https://www.vulkan.org/, [mrežno; stranica posjećena: lipanj 2024.].

[4] Microsoft, https://learn.microsoft.com/en-us/windows/win32/directx, [mrežno; stranica posjećena: lipanj 2024.].

[5] Apple, https://developer.apple.com/metal/, [mrežno; stranica posjećena: lipanj 2024.].

[6] Mozilla, https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API, [mrežno; stranica posjećena: lipanj 2024.].

[7] The Khronos Group, https://www.khronos.org/opengl/wiki/History_of_OpenGL, [mrežno; stranica posjećena: lipanj 2024.].

[8] ——, https://www.khronos.org/conformance/adopters/conformant-products/opengl, [mrežno; stranica posjećena: lipanj 2024.].

[9] B. T. Phong, “Illumination for computer generated pictures”, u *Seminal graphics: pioneering efforts that shaped the field*, 1998., str. 95–101.

[10] David Herberth, https://glad.dav1d.de/, [mrežno; stranica posjećena: lipanj 2024.].

[11] GLFW, https://www.glfw.org/, [mrežno; stranica posjećena: lipanj 2024.].

[12] GLM, https://github.com/g-truc/glm, [mrežno; stranica posjećena: lipanj 2024.].

[13] Kitware, https://cmake.org/, [mrežno; stranica posjećena: lipanj 2024.].

[14] J. de Vries, *Learn OpenGL - Graphics Programming*. Kendall & Welling, 2020.

Sažetak

Izrada demonstracijske aplikacije za edukaciju korištenjem OpenGL biblioteke

Ivan Cvrk

Tema ovog rada je upoznavanje s OpenGL grafičkim aplikacijskim programskim sučeljem. Cilj je napraviti demonstracijske aplikacije i detaljno dokumentirati tijek njihovih izrada kako bi se kao rezultat dobile upute za novu laboratorijsku vježbu na predmetu *Osnove virtualnih okruženja*. U radu su napravljene tri demonstracijske aplikacije tako da svaka demonstrira određenu funkcionalnost; osnovno renderiranje na ekran, transformacije koordinatnog sustava i osvjetljenje. Osim uputa za korištenje modernog OpenGL API-ja i primjera, sastavljeni su zadatci za studente čijim se rješavanjem trebaju okušati u samostalnoj izradi grafičke aplikacije. Također, napravljena je i dodatna demonstracijska aplikacija koja implementira rješenja svih zadataka.

Ključne riječi: virtualna okruženja; računalna grafika; OpenGL; laboratorijske vježbe; 3D transformacije; osvjetljenje; C++; GLSL

Abstract

Creation of a demonstration application for education using the OpenGL library

Ivan Cvrk

The theme of this paper is an introduction to the OpenGL graphical application programming interface. The aim is to create demonstration applications and thoroughly document their creation process to produce guidelines for a new laboratory exercise in the course *Osnove virtualnih okruženja*. Three demonstration applications have been developed in this work, each demonstrating a specific functionality: basic screen rendering, coordinate system transformations, and lighting. In addition to instructions for using the modern OpenGL API and examples, tasks have been composed for students to try their hand at independently creating a graphical application. Additionally, an extra demonstration application has been developed that implements the solutions to all the tasks.

Keywords: virtual environments; computer graphics; OpenGL; laboratory assignments; 3D transformations; lighting; C++; GLSL

Privitak A: Upute za laboratorijsku vježbu

Osnove virtualnih okruženja

4. Laboratorijska vježba
Programsko sučelje OpenGL

Pripremio: Ivan Cvrk

1. Uvod

OpenGL (Open Graphics Library) je aplikacijsko programsko sučelje (API) za iscrtavanje 2D i 3D vektorske grafike. Ono je hardverski neovisno sučelje predviđeno za korištenje na bilo kojoj platformi, a tipično se koristi za interakciju s grafičkom karticom kako bi ostvarili hardverski ubrzano renderiranje.

U sklopu OpenGL-a ne postoje naredbe za rad sa prozorima ili korištenje ulaznih parametara zadanih od strane korisnika; za to se koriste specifične naredbe vezane uz određeni hardware koji se koristi. OpenGL ne pruža naredbe više razine za opis modela sastavljenih od 3-dimenzionalnih objekata.

OpenGL radi kao automat stanja. Prolazeći različitim stanjima, ona ostaju nepromijenjena sve dok ih sam korisnik ne promijeni.

Cilj vježbe je upoznavanje sa sintaksom OpenGL-a kroz ove upute i službenu dokumentaciju, a rješavanjem konkretnih zadataka dobit ćete bolji uvid u iscrtavanje poligona, geometrijske transformacije, osnovne animacije, rad sa sjenčarima i uvid u jednostavan model osvjetljenja.

Za detaljan opis OpenGL funkcija i njihovih parametara, pogledajte dokumentaciju na Khronos stranicama `https://registry.khronos.org/OpenGL-Refpages/gl4/` ili na neslužbenoj, ali možda bolje strukturiranoj stranici za Opengl reference `https://docs.gl/`.

VAŽNO: Tijekom čitanja uputa pratite ponuđene primjere 1-3 jer tamo možete vidjeti primjere korištenja koncepata OpenGL-a koji su objašnjeni u ovoj uputi.

2. Alati potrebni za izvođenje

Sve potrebne datoteke za izvođenje vježbe nalaze se u **Vježba4.zip** arhivi. Za izgradnju projekta potrebno je koristiti **cmake** alat. Možete ga preuzeti sa službene stranice `https://cmake.org/download/` ili korištenjem paket managera na svojem operacijskom sustavu.

Nakon što preuzmete alat, pozicionirajte se u direktorij raspakirane arhive i pokrenite ga s naredbom:

```
cmake -G <generator> .
```

Kako bi provjerili koji sve generatori su vam dostupni možete iskoristiti `cmake -help` naredbu. Primjeri dva generatora:

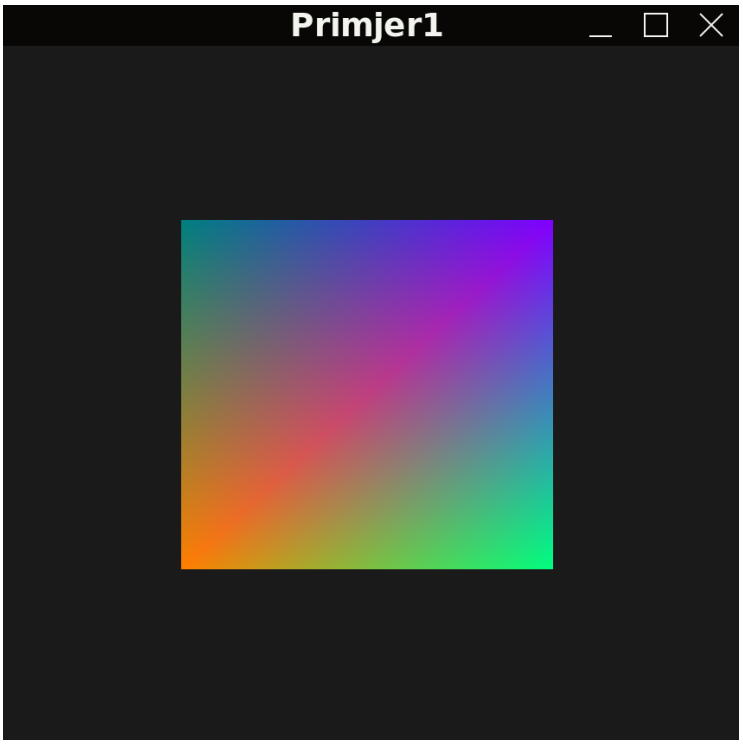
- Windows i microsoft visual studio: `cmake -G "Visual Studio 17 2022" .`
- Unix makefile: `cmake -G "Unix Makefiles" .`

Ovisno o opciji koju ste odabrali, pokrenite *.sln* datoteku ili koristite `make` za kompajliranje datoteka.

Ako ste odabrali Visual Studio, nakon što pokrenete *.sln* datoteku, trebate pritisnuti desni klik na zadatak koji želite pokrenuti u izborniku projekata i odabrati opciju "Set as StartUp Project", te nakon toga možete kompajlirati i pokrenuti odabrani zadatak.

Kako bi osigurali da se laboratorijska vježba može izvoditi na što više platforma, ponuđen je izvorni kod biblioteka koje ćemo koristiti u sklopu vježbi. Dodatne biblioteke nalaze se u *dependencies* direktoriju, a one su:

- glad - Generirana datoteka za pronalazak adresa OpenGL funkcija u sklopu operacijskog sustava.



Slika 1 Obojeni kvadrat na crnoj podlozi

- glfw-3.4 - Biblioteka za upravljanje prozorima i korisničkim unosom (tipkovnica, miš) neovisna o operacijskom sustavu.
- glm-1.0.1 - Matematička biblioteka prilagođena za rad s OpenGL API-jem.

3. OpenGL teorijska podloga

3.1. Jednostavan OpenGL program

Kako bi se upoznali sa strukturom OpenGL programa, vaš prvi zadatak je pokrenuti *primjer1* program. Nakon što pokrenete program, trebali bi vidjeti šareni kvadrat kao na Slici 1.

Prolaskom kroz izvorni kod možete vidjeti da je za prikaz ovog kvadrata potrebno:

- inicijalizirati glfw biblioteku i napraviti prozor
- dohvatiti lokacije OpenGL funkcija
- napraviti program sjenčara
- napraviti vertex array objekt
- učitati koordinate vrhova, boja vrhova i indekse redoslijeda vrhova
- iscrtati kvadrat.

Slika 2 Grafički protočni sustav

Inicijalizaciju glfw biblioteke i stvaranje prozora nećemo detaljno razmatrati, dovoljni su komentari u kodu za razumijevanje funkcionalnosti. Ako vas zanimaju detalji, pogledajte službenu dokumentaciju biblioteke: `https://www.glfw.org/`.

Za određivanje lokacija OpenGL funkcija u adresnom prostoru programa iskoristili smo glad.cpp datoteku.

3.2. Grafički protočni sustav

Grafički protočni sustav u OpenGL-u sastoji se od 5 koraka i oni se izvode sljedećim redoslijedom:

1. Procesiranje vrhova
   - 1.1 Nad svakim vrhom iz **vertex array objekta (VAO)** primjenjuje se **sjenčar vrhova**. Na taj način svaki vrh transformira se u izlazni vrh koji se dalje koristi u grafičkom protočnom sustavu.
   - 1.2 Opcionalni sjenčar teselacije (Tessellation Shader) - služi za podjelu vrhova u manje primitive.
   - 1.3 Opcionalni sjenčar geometrije (Geometry Shader) - služi za procesiranje primitiva, može stvoriti nula ili više primitiva.
2. Postprocesiranje vrhova - sastavljanje primitiva, mapiranje viewport-a na prostor ekrana.
3. Rasterizacija - svaka primitiva (najčešće trokut) rastavlja se na pojedine piksele. Interpolacija parametara primitive.
4. **Sjenčar fragmenata** - bojanje fragmenata.
5. Dodatno procesiranje izlaznih fragmenata - scissor test, stencil test, depth test, blending...

Na Slici 2 nalazi se skica grafičkog protočnog sustava. Žute elemente moramo sami definirati, sivi nisu nužni, ali su programirljivi, a plavi elementi nisu direktno programirljivi. Eksplicitna definicija sjenčara vrhova i sjenčara fragmenata obavezna je od verzije 3.0 OpenGL-a.

U VAO objektu definiramo ulazne podatke u protočni sustav. U njemu se nalazi sve što je potrebno da bi obavili iscrtavanje objekta na ekran (koordinate vrhova, normale vrhova, teksture, indeksi vrhova...). Dodatno procesiranje u zadnjem koraku možemo prilagoditi zvanjem OpenGL funkcija kao što su glEnable(...), glDepthFunc(...) itd.

3.3. Kreiranje programa sjenčara u OpenGL-u

Resursi u OpenGL-u generalno prate model korištenja tako da ga stvorimo, inicijaliziramo s podatcima, koristimo i obrišemo nakon što nam više ne treba. Prema tome, kako bi stvorili svoje programe sjenčara,

prvo je potrebno stvoriti ih pozivom funkcije `glCreateShader()`. Funkcija vrača identifikator resursa tipa GLuint[1]koji trebamo koristiti kada se želimo referirati na taj resurs.

Nakon što stvorimo shader objekt, potrebno je učitati njegov izvorni kod što činimo pozivom funkcije `glShaderSource`, a zatim taj kod kompajliramo pozivom `glCompileShader` funkcije. Na isti način stvaramo i sjenčar fragmenata.

OpenGL Shading Language (GLSL) je jezik visoke razine za pisanje sjenčara, a temelji se na sintaksi programskog jezika C. Programi sjenčara izvode se na jezgrama grafičke kartice i svi rade paralelno nad različitim ulaznim podatcima u grafički protočni sustav prema Single instruction multiple data (SIMD) modelu paralelizacije.

Sjenčar vrhova

Sjenčar vrhova je mali "program" koji se izvodi za svaki definirani vrh objekta koji crtamo. Pogledajmo kako izgleda sjenčar vrhova u prvom primjeru.

```
#version 460
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
out vec3 color;
void main()
{
   color = aColor;
   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Isječak koda 1 Sjenčar vrhova

Prvo što moramo napraviti u svakom sjenčaru je definirati koju verziju GLSL-a koristimo direktivnom `#version <verzija>`. Primijetite da koristimo OpenGL verziju 4.6 i da je odabrana verzija GLSL-a 460. To nije slučajnost, OpenGL i GLSL nakon verzije 3.3 odnosno 330 se podudaraju tako da ovisno o verziji OpenGL-a koju koristite možete jednostavno odrediti koja verzija GLSL-a vam treba.

Vrhovi 3D modela općenito se sastoje od više podataka koji ih opisuju. Na primjer, uz vrh možemo vezati njegove koordinate, boju poligona u tom vrhu, vektor normale, UV koordinate teksture itd. Podatci koje vežemo uz taj vrh zovemo atributi. Kako svaki prolaz sjenčara vrhova obrađuje jedan vrh, potrebno je definirati koje atribute vežemo uz vrh i na kojoj lokaciji se svaki atribut nalazi kako bi ih mogli dohvatiti kao ulazne parametre u sjenčar. Atribute vrha definiramo sljedećom sintaksom:

```
layout(qualifier1 , qualifier2 = value, ...)  variable definition
```

U sjenčaru vrhova definirali smo dva ulazna atributa `aPos` i `aColor` koji definiraju poziciju koordinate vrha i boju poligona u tom vrhu. O tome kako smo odabrali koje lokacije trebamo postaviti za svaki atribut bit će više rečeno kasnije kada ćemo objasniti kako se kreira VAO objekt. Ukratko, svaki VAO

[1]GLuint je 32-bitni pozitivni cijeli broj, slično kao *unsigned int* u C-u i C++-u. OpenGL definira svoje primitivne tipove podataka kako bi postigao prenosivost, zato što primitivni tipovi podataka tipa *unsigned int* ne moraju biti iste veličine na svim platformama. Na jednom računalu *unsigned int* može imati 16 bitova, dok na drugom može imati 32 bita.

objekt sastoji se od sirovih podataka u međuspremnicima (buffer objektima) i opisa kako dohvaćati te podatke iz međuspremnika, odnosno opisa atributa i njihovih lokacija. Lokacija odgovara indeksu vertex attribute pointer-a koji opisuje taj vrh.

Zatim definiramo `out vec3 color;` varijablu. Vrijednost koja će se nalaziti u ovoj varijabli nakon što završi izvođenje sjenčara bit će proslijeđena sljedećim sjenčarima u grafičkom protočnom sustavu. Pomoću te varijable proslijedit ćemo informaciju sjenčaru fragmenata kako da oboji poligon.

Nakon toga imamo vrlo jednostavnu `main` funkciju. U ovom slučaju ne radimo ništa komplicirano, jednostavno prosljeđujemo ulazni atribut `aColor` na izlaznu varijablu Color i postavljamo `gl_Position` varijablu. Postavljanje `gl_Position` varijable je obavezno za svaki prolaz sjenčara. Inače, u ovom sjenčaru transformiramo koordinate vrhova množenjem s matricom modela (primjer2).

Izlazni vrh ima 4 koordinate jer radimo u s homogenim koordinatama kako bi pojednostavnili izračune transformacija koordinata vrha.Translacija nije linearna transformacija i ne možemo ju izvesti množenjem matrice i vektora koordinata vrha ako smo u 3-dimenzionalnom prostoru, ali ako prijeđemo u 4-dimenzionalni prostor, tada translacija postaje linearna transformacija. Kasnije (4. korak grafičkog protočnog sustava) se x, y, i z koordinate dijele s četvrtom koordinatom w kako bi prešli u koordinate radnog prostora. Zbog toga smo koordinatu w postavili na 1.0 kako ne bi promijenili ostale koordinate tijekom dijeljenja.

Primijetimo da se za svaki vrh moraju pohraniti njegovi atributi memoriju. Što kada želimo definirati varijablu koja će imati istu vrijediti za sve vrhove i prolaske sjenčara iscrtavanja jednog objekta, na primjer matricu transformacije modela? Bilo bi nepraktično za svaki vrh pohraniti kopiju matrice transformacije da bi ju mogli učitati kao atribut svakog vrha. Zbog toga postoje uniformne varijable. Vrijednosti uniformnih varijabli postavljamo iz glavnog programa i one ostaju fiksirane za sve prolaze sjenčara dok ih ne promijenimo. Uniformne varijable moguće je koristiti u svim sjenčarima. Primjer korištenja uniformne varijable možete vidjeti u *primjer2/shaders/basic.vert* datoteci. Da bi postavili vrijednost uniformne varijable potrebno je dohvatiti njenu lokaciju i zatim postaviti vrijednost. Na primjer za postavljanje uniformne varijable matrice 4x4:

```
GLint transformUniformPos = glGetUniformLocation(shaderProgamID, "transformMatrix");
glUniformMatrix4fv(transformUniformPos, 1, GL_FALSE, &transformMatrix[0][0]);
```

Isječak koda 2 Postavljanje uniformne varijable: `uniform in mat4 transformMatrix;`

Sjenčar fragmenata

Sjenčar fragmenata poziva se za izračun boje svakog slikovnog elementa (pixela). Pogledajmo kako izgleda sjenčar Fragmenata.

U sjenčaru fragmenata moramo definirati verziju GLSL-a isto kao i u sjenčaru vrhova. Zatim definiramo izlaznu boju fragmenta kao `out vec4 FragColor`. Kako smo u sjenčaru vrhova definirali izlaznu varijablu `out vec3 color`, u sjenčaru vrhova moramo definirati istu takvu ulaznu varijablu `in vec3 color` kako bi u njoj mogli uhvatiti izlaz iz sjenčara vrhova.

Primijetite da svaki trokut, koji je osnovna primitiva u računalnoj grafici, ima 3 vrha i može se sastojati

```
#version 460
out vec4 FragColor;
in vec3 color;
void main()
{
    FragColor = vec4(color, 1.0f);
}
```

Isječak koda 3 Sjenčar fragmenata

od puno više piksela, ovisno o veličini međuspremnika za renderiranje. Zbog toga nije moguće direktno prenijeti izlazne parametre iz sjenčara vrhova u sjenčar fragmenata. Vrijednost proslijeđenih varijabli iz prethodnih sjenčara prenose se u sjenčar fragmenata na način da se vrijednost izlaznih parametra prethodnih sjenčara interpolira između vrijednosti definiranih u vrhovima primitive, u našem slučaju trokuta. Ako pogledate kod kojim smo stvorili kvadrat u *primjer1* primjetit ćete da je kvadrat sastavljen od dva trokuta i da je boja definirana samo u vrhovima. Zbog interpolacije izlaznih parametara iz prethodnih sjenčara u sjenčar fragmenata, svi pikseli kvadrata su obojeni i njihova boja ovisi o udaljenosti od vrhova trokuta kojem pripadaju.

Kako se sjenčar fragmenata pokreče za izračun svakog slikovnog elementa, u njemu možemo izvesti vrlo precizne izračune za prikaz 3D objekata. Na primjer, možemo izračunati boju svakog slikovnog elementa ovisno o rasvjetljenju scene, koliko je udaljena svjetlost, pod kojim kutem padaju zrake svjetlosti na objekt, kako se reflektiraju upadne zraka svjetlosti s obzirom na položaj kamere, od kakvog materijala je objekt itd. Pogledajte `primjer3` s Phongovim modelom osvjetljenja kocke.

Program sjenčara

Kompajlirani sjenčari konceptualno odgovaraju objektnim datotekama kompajliranog C i C++ koda. Kako bi stvorili program potrebno je povezati objektni kod korištenjem linkera. Na isti način stvaramo program sjenčara. Prvo je potrebno stvoriti resurs program, a zatim dodamo kompajlirane shader programe i povežemo ih.

```
GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragmentShader);
glLinkProgram(program);
```

Isječak koda 4 Stvaranje programa sjenčara

Nakon stvaranja programa možemo obrisati sjenčare jer nam više ne trebaju, spojili smo ih u program. Kako bi koristili program u grafičkom protočnom sustavu, potrebno je osigurati da je željeni program aktivan prije poziva `glDrawArrays` ili `glDrawElements` funkcije za crtanje. Samo jedan program sjenčara može biti aktivan u nekom trenutku, a postavljamo ga aktivnim pozivom `glUseProgram` funkcije.

```
GLfloat vertices[] = {...};
GLuint indices[] = {};

GLuint VAO;
GLuint VBO;
GLuint EBO;

glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);

glCreateVertexArrays(1, &VAO);
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (void *)0);
glEnableVertexAttribArray(0);
```

Isječak koda 5 Primjer učitavanja koordinata vrhova i kreiranje VAO objekta.

3.4. Vertex Array Objekt

Vertex Array Object (VAO) je OpenGL objekt koji sadrži sve potrebno za opis ulaznih podataka u grafički protočni sustav. Konkretno, VAO se sastoji od opisa atributa vrhova koji su spomenuti u sjenčaru vrhova. Oni opisuju gdje se nalaze podatci o vrhovima i kako im pristupati. Vertex array objekt kreiramo pozivom glCreateVertexArrays funkcije.

```
void glCreateVertexArrays(GLsizei n, GLuint *arrays);
```

- `n` - broj VAO objekata koji želimo stvoriti
- `buffers` - pokazivač na polje GLuint elemenata veličine `n`.

Međuspremnici (Buffer objekti)

Kad učitamo podatke o vrhovima nekog 3D objekta iz datoteke ili ih definiramo u kodu kao u našim primjerima, ti podatci se nalaze u RAM memoriji najčešće u obliku polja. Da bi grafička kartica mogla efikasno koristiti te podatke, potrebno ih je prebaciti u VRAM memoriju grafičke kartice. Buffer objekti su reference na blokove memorije grafičke kartice.

Kako bi prebacili podatke iz RAM memorije u VRAM memoriju, prvo je potrebno stvoriti buffer objekt pozivom OpenGL API-ja. Za to nam služi funkcija `glGenBuffers`.

```
void glGenBuffers(GLsizei n, GLuint * buffers);
```

- `n` - broj buffer objekata koji želimo stvoriti
- `buffers` - pokazivač na polje GLuint elemenata veličine `n`.

Funkcija je optimizirana za svaranje više buffer objekata odjednom, zbog tog toga prima broj buffer objekata koji želimo stvoriti i pokazivač na polje GLuint kako bi mogla zapisati identifikatore novostvorenih buffer objekata. Te identifikatore možemo koristiti kad ćemo se htjeti referencirati na pojedini buffer objekt. Isto vrijedi i za `glCreateVertexArrays` funkciju.

Nakon što smo stvorili buffer objekt ili više njih, možemo ih napuniti s podatcima. Prije punjenja, potrebno je je odabrani buffer objekt postaviti aktivnim. To radimo pozivom glBindBuffer funkcije.

```
void glBindBuffer(GLenum target, GLuint buffer);
```

- `target` - Za prvi element odabiremo target `GL_ARRAY_BUFFER`. To znači da će podatci u međuspremniku sadržavati vrijednosti atributa vrhova.
- `buffer` - Drugi element je identifikator buffer objekta koji želimo aktivirati na tom targetu.

Samo jedan buffer objekt može biti aktivan za odabrani target.

Spremni smo prebaciti podatke u međuspremnik u VRAM memoriju. Tome služi `glBufferData` funkcija.

```
void glBufferData(GLenum target, GLsizeiptr size, const void * data, GLenum usage);
```

- `target` - target na kojem je vezan odabrani buffer objekt.
- `size` - Veličina podataka u bajtovima.
- `data` - Pokazivač na podatke koje želimo prebaciti.
- `usage` - Obrazac korištenja.

Prvi argument je target. Vezali smo odabrani buffer objekt na `GL_ARRAY_BUFFER` target zato što želimo učitati podatke o vrhovima, tako da ćemo tu opet odabrati isti target. Za zadnji argument odabiremo `GL_STATIC_DRAW` što daje uputu grafičkom API-ju da podatke pohrani u dio VRAM memorije koji je najefikasniji za korištenje grafičkoj kartici. Pohranjivanje podataka u taj dio memorije je najsporije, ali jednom kad su podatci pohranjeni možemo ih vrlo efikasno koristiti. Suprotnost tome je `GL_DYNAMIC_DRAW`.

Vertex Array Atributi

Vertex attribute pokazivači opisuju gdje se nalaze vrijednosti atributa vrhova i kako im pristupati. Svaki VAO objekt ima najviše 16 atributa. Atribut sadrži informaciju u kojem buffer objektu se nalaze vrijednosti atributa, na kojem odkamu od početka međuspremnika se nalazi prvi atribut i koliko se bajtova potrebno pomaknuti unutar međuspremnika da bi došli do sljedeće vrijednosti atributa (stride). Ovi pokazivači nisu povezani s klasičnim pokazivačima u C/C++ programskim jezicima. Oni opisuju i pokazuju na sadržaj memorije (vrijednosti atributa vrhova) u buffer objektima.

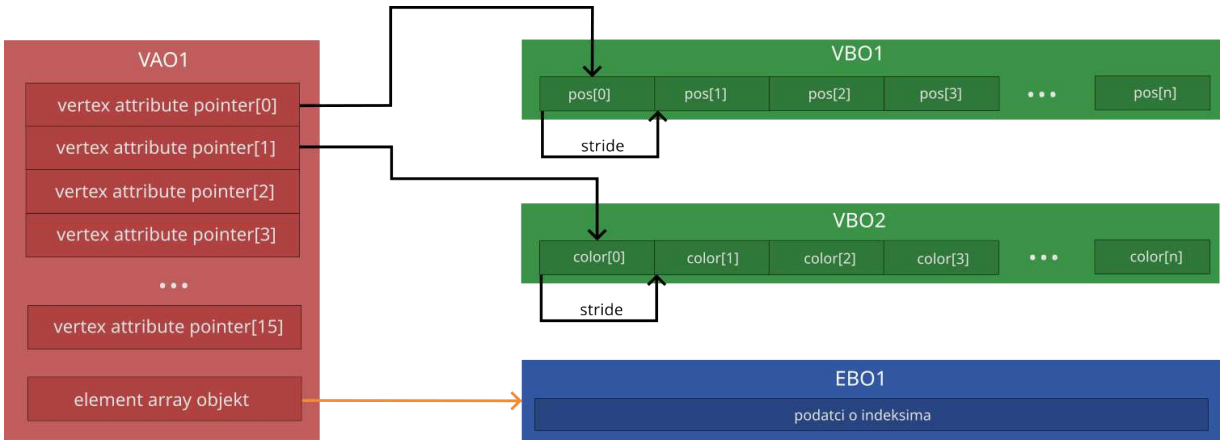
Vertex attibute pokazivače stvaramo pozivom funkcije glVertexAttribPointer. OpenGL API zahtjeva da je prije poziva ove funkcije na `GL_ARRAY_BUFFER` targetu nalazi aktiviran buffer objekt koji sadrži podatke atributa kojeg želimo opisati stavanjem ovog pokazivača.

```
void glVertexAttribPointer(GLuint index, GLint size, GLenum type,
  GLboolean normalized, GLsizei stride, const void * pointer);
```

- `index` - indeks/lokacija vertex attribute pokazivača kojeg želimo postaviti. Ova lokacija odgovara `layout (location = <n>)` direktivi u sjenčaru vrhova.

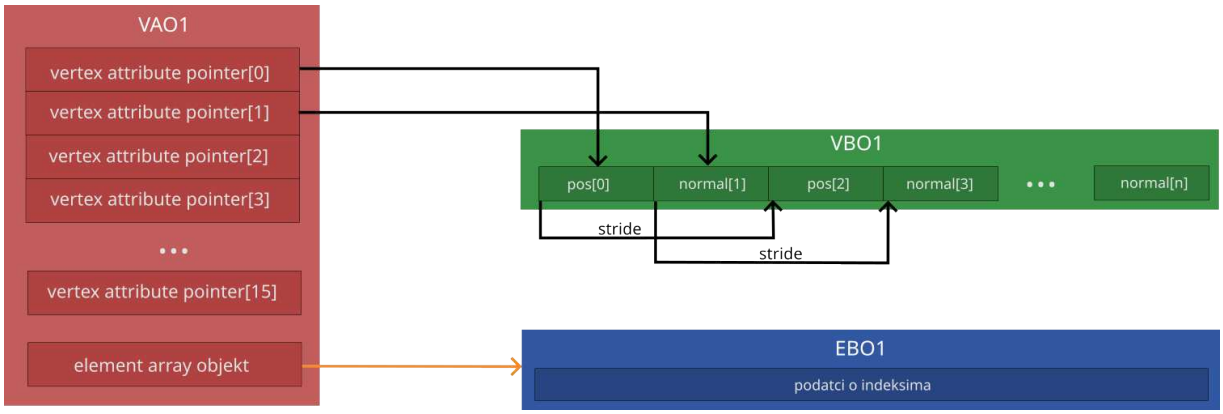
- `size` - od koliko elemenata se sastoji atribut. Na primjer ako atribut opisuje koordinate vrha, onda se sastoji od 3 elementa - x, y i z koordinata.
- `type` - kojeg tipa su elementi. Za naše potrebe koristimo samo `GL_FLOAT`.
- `normalized` - za nas nebitan parametar. Postavite ga uvijek na `GL_FALSE`.
- `stride` - broj bajtova za koji se potrebno pomaknuti da bi došli do sljedeće vrijednosti atributa. Na primjer ako smo imali 3 elementa tipa `GL_FLOAT`, da bi došli do sljedećeg atributa u međuspremniku, potrebno se pomaknuti `3 * sizeof(GLfloat)` bajtova. Ovo ovisi o tome kako smo rasporedili podatke u međuspremniku.
- `pointer` - broj bajtova za koji se potrebno pomaknuti od početka međuspremnika da bi došli do vrijdnosti atributa prvog vrha.

U primjeru 1 koristimo dva buffer objekta gdje prvi sadrži vrijednosti atributa koordinata `n` vrhova, a drugi sadrži vrijednost atributa boje za `n` vrhova. To su dva atributa i zbog toga koristimo dva vertex attribute pokazivača gdje prvi ima vrijednosti:
`index` = 0, `size` = 3, `type` = `GL_FLOAT`, `normalized` = `GL_FALSE`, `stride` = `3 * sizeof(GLfloat)`, `pointer` = (void*)0,
drugi vertex attribute pokaizvač ima vrijednosti:
`index` = 1, `size` = 3, `type` = `GL_FLOAT`, `normalized` = `GL_FALSE`, `stride` = `3 * sizeof(GLfloat)`, `pointer` = (void*)0.



Slika 3 Odnos VAO objekta i buffer objekata u primjer1

U primjeru 3 imamo jedan buffer objekt u kojem su podatci isprepleteni. I dalje učitavamo dva atributa; koordinate vrhova i vektor normale. Zbog toga moramo koristiti dva vertex attribute pokazivača gdje prvi ima vrijednosti:
`index` = 0, `size` = 3, `type` = `GL_FLOAT`, `normalized` = `GL_FALSE`, `stride` = `6 * sizeof(GLfloat)`, `pointer` = (void*)0,
a drugi vertex attribute pokazivač ima vrijednosti:
`index` = 1, `size` = 3, `type` = `GL_FLOAT`, `normalized` = `GL_FALSE`, `stride` = `6 * sizeof(GLfloat)`, `pointer` = (void*)(3 * sizeof(GLfloat)).



Slika 4 Odnos VAO objekta i buffer objekata u primjer3

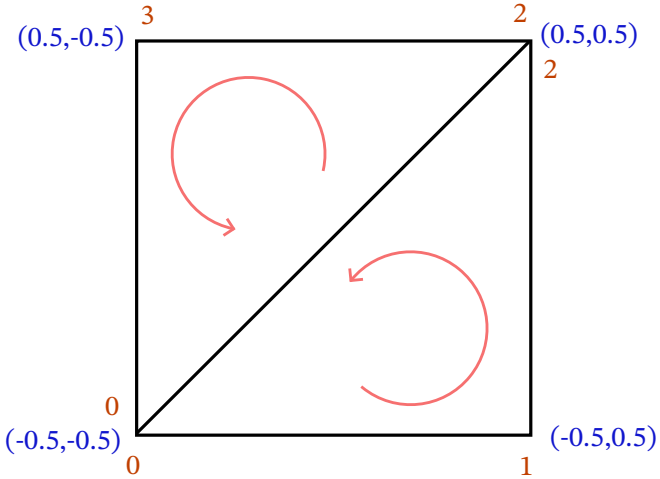
Nakon što definiramo vertex attibute pokazivač, potrebno ga je aktivirati pozivom funkcije:

```
glEnableVertexAttribArray(GLuint index)
```

Funkcija prima samo jedan argument, a to je indeks/lokacija vertex attibute pokazivača u trenutno aktivnom VAO objektu koji želimo aktivirati.

Redoslijed definiranja indeksa vrhova

Indeksi vrhova su polje `GLuint` brojeva koji definiraju kojim redoslijedom je potrebno povezati definirane vrhove kako bi dobili primitive. U primjerima koristimo trokute, tako da svaka tri indeksa definiraju jedan trokut. Korištenje indeksa vrhova je praktično kad više poligona dijeli zajedničke vrhove. U prvom primjeru gdje crtamo kvadrat imamo dva trokuta koja dijele vrh dolje-lijevo i gore-desno. Da ne bi morali duplicirati podatke tih vrhova (koordinate vrhova i boja u njima), jednostavno smo definirali da se trokuti sastave od 4 definirana vrha tako da se spoje u trokute redosljedom 0, 1, 2, 2, 3, 0 (vidi sliku 5).



Slika 5 Spajanje vrhova kvadrata u trokute pomoću indeksa u primjeru 1

Bitno je naglasiti da su vrhovi trokuta definirani u smjeru suprotnom od kazaljke na satu. Svi trokuti

moraju biti definirani u istom smjeru kako bi grafički protočni sustav mogao odrediti koji trokuti su okrenuti od nas i nisu vidljivi kako bi ih mogao odbaciti i ubrzati proces renderiranja.

Indeksi vrhova učitavaju se u buffer objekte jednako kao i podatci o vrhovima. Jedina razlika je da za `target` koristimo `GL_ELEMENT_ARRAY_BUFFER`. Time učitavamo indekse vrhova za trenutno aktivni VAO objekt, pa je potrebno pobrinuti se da smo prije punjenja buffer objekta aktivirali željeni VAO objekt pozivom funkcije `glBindVertexArray`.

Indeksiranje vrhova je opcionalno, uvijek možete definirati vrhove (u buffer objektu) redoslijedom kojim ih želite pospajati u trokute i nacrtati. Tako bi za ovaj primjer definirali vrhove: [(-0.5, -0.5), (-0.5, 0.5), (0.5, 0.5), (0.5, 0.5), (0.5, -0.5), (-0.5, -0.5)].

3.5. Crtanje objekata

Sada kad su svi podatci postavljeni i definirani, možemo ih koristiti za crtanje. Kako su svi podatci potrebni za iscrtavanje nekog objekta definirani u VAO objektu, prvo je potrebno aktivirati taj VAO objekt. Zatim je potrebno aktivirati program sjenčara koji želimo koristiti. Time smo definirali sve potrebno grafičkom sustavu za crtanje. Funkcija koju ćemo koristiti za crtanje ovisi o tome koristimo li indekse vrhova ili ne.

Ako ne koristimo indekse vrhova nego želimo proći kroz vrhove onim redoslijedom kojim su definirani u buffer objektima koje smo napunili s targetom `GL_ARRAY_BUFFER`, tada koristimo:

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- `mode` - U koje primitive želimo povezati vrhove. Mi radimo s trokutima, zato ćemo definirati `GL_TRIANGLES`.[2]
- `first` - Indeks prvog vrha kojeg želimo nacrtati. Uglavnom je vrijednost 0.
- `count` - Broj vrhova koji želimo koristiti u crtanju.

Ako smo definirali indekse vrhova i želimo tim redoslijedom sastavljati vrhove u primitive. Tada ćemo koristiti:

```
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const void * indices);
```

- `mode` - isto kao kod glDrawArrays
- `count` - broj indeksa
- `type` - kojeg tipa su indeksi. Za naše potrebe uvijek `GL_UNSIGNED_INT`
- `indices` - pokazivač odmaka od kuda želimo krenuti čitati indekse. Za naše potrebe uvijek 0.

[2] Moguće vrijednosti su: `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_LINE_STRIP_ADJACENCY`, `GL_LINES_ADJACENCY`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY` i `GL_PATCHES`

4. Geometrijske transformacije

4.1. Biblioteka GLM

Za izvođenje svih matematičkih izračuna i reprezentaciju podataka u RAM memoriji koristit ćemo **OpenGL Mathematics (GLM)** biblioteku. Vektori i matrice, koje prenosimo u grafički protočni sustav preko uniformnih varijabli imaju točno zadanu shemu kako moraju biti pohranjeni u memoriji da bi im programi sjenčara mogli ispravno pristupati. GLM biblioteka se brine za ispravan raspored podataka u memoriji i implementira matematičke operacije za rad s njima.

GLM ima intuitivne nazive za sve korisnički definirane tipove podataka i funkcije za rad s njima. Također, GLM definira operatore za rad s tim korisničkim tipovima. GLM podrazumijeva konvenviju množenja matrice vektorom, tako da su vektori zapravo jednostupčaste matrice.

```
glm::vec3 vektor;
glm::mat3 matrica1;
glm::mat3 matrica2;
// množenje vektora i matrice
glm::vec3 rezultat =  matrica2 * matrica1 * vektor;
// inverz matrice
glm::mat3 inverz1 = glm::inverse(matrica1);
```

Isječak koda 6 Primjer korištenja GLM biblioteke

4.2. Matrica modela

Kako bi pomaknuli, zarotirali ili skalirali neki objekt u sceni potrebno je izgraditi njegovu matricu modela. Ta matrica akumulira sve geometrijske transformacije modela i množi se s koordinatama svakog vrha. Time se koordinate vrhova prenose iz lokalnog geometrijskog sustava objekta u koodrinate scene. Kada dodajemo novu transformaciju u matricu, zapravo ju množimo s desna predanoj matrici kao

```
// Stvori jediničnu matricu
glm::mat4 matricaModela{1};

// Translacija x = 1, y = 2, z = 3
matricaModela = glm::translate(matricaModela, glm::vec3{1.0, 2.0, 3.0});

// Rotiraj za 30 stupnjeva oko lokalnog vektora 1.0, 1.0, 1.0
matricaModela = glm::rotate(matricaModela,
                      glm::radians(30.0f),
                      glm::vec3{1.0, 1.0, 1.0});

// Skaliraj po svim osima 2 puta
matricaModela = glm::scale(matricaModela, glm::vec3{2.0, 2.0, 2.0});
```

Isječak koda 7 Primjer izgradnje matrice modela

prvi argument. Tako da je rezultat primjena transformacija iz isječka 7 zapravo: `matricaModela = jedinična_matrica * skaliranje * rotacija * translacija` što rezultira sljedećim redoslijedom primjena transformacija: 1. skaliranje, 2. rotacija, 3. translacija.

Bitno je primijetiti da je matrica modela tipa $4x4$ zato što radi s homogenim koordinatama. Kada ne bi radili s homogenim koordinatama ne bi mogli prikazati translaciju kao linearnu transformaciju.

4.3. Matrica pogleda

Matrica pogleda transformira koordinate vrhova iz koordinatnog sustava scene u koordinatni sustav kamere. Koordinatni sustav kamere je desni koordinatni sustav i kamera gleda u negativnom smjeru Z osi. S obzirom na položaj i rotaciju kamere u sceni, X os gleda u desnom smjeru kamere, a Y prema gore s obzirom na kameru. Ova transformacija je nužna prije primjene projekcijske matrice.

Za izgradnju matrice pogleda, GLM pruža funkciju

```
glm::mat4 LookAt(glm::vec3 eye, glm::vec3 center, glm::up)
```

- `eye` - pozicija kamere u sceni
- `center` - smijer u kojem kamera gleda
- `up` - vektor u smjeru gore s obzirom na kameru, uglavnom koristimo globalni vektor [0, 1, 0].

4.4. Projekcijska matrica

Projekcijska matrica radi transformaciju koordinata 3D objekata u sceni u 2D koordinate ekrana. Projekcijska matrica može biti perspektivna ili ortografska. Mi ćemo koristiti perspektivnu projekcijsku matricu jer ona odgovara našem sustavu vida. U perspektivnoj projekciji objekti koji su dalje od projekcijske ravnine izgledaju manje nakon projekcije, a bliži objekti su veći. Nakon projekcije, koordinate u rasponu od (-1, -1) do (1, 1) su vidljive na ekranu. Koordinate (0,0) odgovaraju centru ekrana. Primijetite da u prvom primjeru nismo koristili matrice transformacije nego smo odmah zadali koordinate kvadrata u sustavu projekcije.

Za izgradnju perspektivne projekcijske matrice GLM nudi funkciju:

```
glm::perspective(float fovy, float aspect, float zNear, float zFar);
```

- `fovy` - vertikalni kut vidnog polja projekcijske piramide
- `aspect` - omjer visine i širine slike zaslona
- `zNear` - udaljenost do projekcijske ravnine
- `zFar` - udaljenost do gdje vidimo

5. Phongov model osvjetljenja

Kako bi mogli jasnije vidjeti 3D objekte u sceni potrebno nam je osvjetljenje, zbog toga ćemo u okviru ove laboratorijske vježbe implementirati pojednostavljeni model Phongovog osvjetljenja. Također, na

ovaj način ćemo demonstrirati fleksibilnost i moć programirljivih programa sjenčara.

Ukratko ćemo ponoviti kako radi Phongov model osvjetljenja, a na vama je da pročitate i razumijete implementaciju u sjenčarima. Otvorite *primjer3/shaders/lighting.frag* sjenčar i pratite implementaciju uz nastavak ponavljanja Phongovog sjenčanja.

Intenzitet svjetlosti označen je kao vektor $\vec{I}$ zato što je u sjenčarima intenzitet svjetlosti zapravo boja svjetlosti koja se sastoji od r, g i b komponente.

5.1. Ambijentalna komponenta

Phongov model osvjetljenja sastoji se od tri komponente. Prva komponenta je ambijentalno osvjetljenje. Ambijentalna komponenta rezultat je interakcije svjetlosti među svim objektima u sceni i pretpostavlja da je reflektirana svjetlost koja dolazi do promatranog fragmenta od svih ostalih površina konstantnog iznosa. Za intenzitet ambijentalnog osvjetljenja koristit ćemo sljedeći izraz:

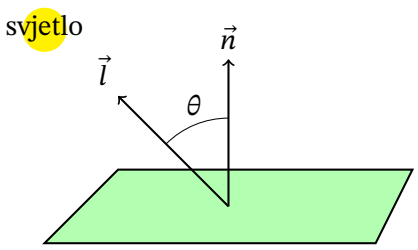
$$\vec{I_a} = k_a \cdot \vec{I_c},$$

gdje je k_a konstanta ($0 \leq k_a \leq 1$), a I_c intenzitet komponenti boja izvorne svjetlosti.

5.2. Difuzna komponenta

Difuzna komponenta određuje jačinu reflektirane svjetlosti od površine s obzirom na kut upadne zrake svjetlosti i normale površine. Jačinu difuzne komponente računat ćemo prema izrazu:

$$\vec{I_d} = \vec{I_c} \cdot cos(\theta)$$



Slika 6 Difuzna komponenta svjetlosti

Ako normiramo vektore l i n, možemo iskoristiti skalarni umnožak vektora za računanje difuzne komponente svjetlosti:

$$\vec{I_d} = max(\hat{l} \cdot \hat{n}, 0) \cdot \vec{I_c}.$$

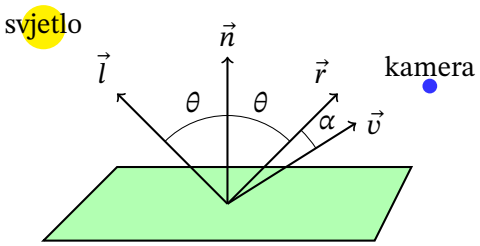
Funkcija max sprječava da obojimo fragment u slučaju da je kut veći od 90°

5.3. Zrcalna komponenta

Zrcalna komponenta određuje osvjetljenje fragmenta koje vidimo zbog reflektirane svjetlosti od tog fragmenta. Što je smjer reflektirane zrake svjetlosti bliži smjeru prema kameri, to je intenzitet reflektirane komponente jači. Intenzitet zrcalne komponente svjetlosti računat ćemo prema izrazu:

$$\vec{I_s} = max(\hat{r} \cdot \hat{v}, 0)^n \cdot \vec{I_c}$$

Velik iznos potencije n stvara dojam sjajnije površine zato što se vektor smjera od fragmenta prema kameri $\vec{v}$ smije razlikovati za mali kut od smjera reflektirane zrake $\vec{r}$ da bi vidjeli zrcalnu komponentu osvjetljenja. To stvara dojam manje disperzije svjetlosti, odnosno glađe površine.



Slika 7 Zrcalna komponenta svjetlosti

5.4. Ukupno osvjetljenje fragmenta

Intenzitet osvjetljenja fragmenta određen je zbrojem intenziteta svih komponenti osvjetljenja, a konačna boja fragmenta je određena bojom materijala, te bojom i intenzitetom svjetlosti koja ga obasjava.

$$\vec{I} = \vec{I_a} + \vec{I_d} + \vec{I_s}$$

Komponente konačne boje $\vec{C}$ računamo kao umnožak komponente boje materijala $\vec{M}$ i komponente boje rezultantnog intenziteta svjetlosti koji ga obasjava $\vec{I}$.

$$C_r = M_r \cdot I_r$$

$$C_g = M_g \cdot I_g$$

$$C_b = M_b \cdot I_b$$

6. Zadatci

Zadatci iz ove laboratorijske vježbe sastoje se od teorijskih pitanja na koja trebate moći dati odgovor i radnih zadataka koje trebate implementirati.

Pitanja:

Ako niste sigurni u odgovore na ova pitanja, izmijenite ponuđene primjere kako bi vidjeli što se događa u danim situacijama.

1. Utječe li redoslijed linearnih transformacija na konačni položaj objekta?
2. Tijekom primjena transformacija modela na vektore normala, zašto koristimo matricu 3×3, a ne matricu 4×4?
3. Zaključite što je `attenuation` varijabla u sjenčaru fragmenata kod osvjetljenja i kako ona utječe na rezultat sjenčanja poligona?
4. Ima li boja svjetla utjecaj na boju predmeta kojeg obasjava?

Radni zadatci:

Za implementaciju radnih zadataka iskoristite projekt `zadatak` koji je prazan. Za početak, možete tamo kopirajti sve što vam je potrebno iz prethodnih primjera, a zatim napišite implementaciju rješenja zadataka.

1. (35% bodova) Stvorite razred `Model` koji učitava koordinate vrhova, položaj normala i indekse vrhova trokuta modela iz *.obj* datoteke.

   Struktura OBJ datoteke je vrlo jednostavna i zbog toga ju možemo parsirati bez potrebe korištenja dodatnih biblioteka. Retci koji počinju s **v**, na primjer `v 0.123 0.234 0.345` označavaju x, y i z koordinate vrha. Retci koji počinju s **vn** označavaju vektore normale, na primjer `vn 0.707 0.000 0.707`. Konačno retci koji počinju s **f** definiraju naše primitive, trokute. Na primjer `f 6/4/1 3/5/3 7/6/5` nam daje definiciju 3 vrha trokuta na sljedeći način *indeks vrha/indeks uv koordinate/indeks normale*. Indeksi počinju od 1 i odgovaraju redoslijedu definicija vrhova i normala. UV koordinate nećemo koristiti. Za više detalja možete pogledati `https://en.wikipedia.org/wiki/Wavefront_.obj_file`

   Vaš razred `Model`, treba učitati podatke objekta iz datoteke, stvoriti Vertex Array Objekt i popuniti potrebne buffer objekte. Korištenje indeksa vrhova je izborno.

   Učitajte *sphere.obj* model, a možete probati učitati i neki svoj model koji ste napravili u Blenderu za testiranje. Nemojte predavati *.obj* datoteke kao rješenja zadataka. Za kraj ostavite da se koristi *sphere.obj* model.

2. (25% bodova) Implementirajte rotaciju kamere u XZ ravnini oko ishodišta koordinatnog sustava. Radi jednostavnosti postavite početnu poziciju kamere na koordinate (4.0, 7.0, 5.0) i postavite da uvijek gleda u ishodište globalnog koordinatnog sustava. Omogućite da se pritiskom na tipku `A` kamera rotira u smjeru kazaljke na satu oko globalne y osi (y os u OpenGL-u pokazuje prema gore), a pritiskom na tipku `D` u suprotnom smjeru. Također, dodajte da se pritiskom tipki `W` i `S` kamera približava, odnosno udaljava od centra rotacije. Za dohvat stanja tipki na tikovnici koristite GLFW biblioteku. Pogledajte kako se u primjerima provjerava je li pritisnuta tipka *escape* korištenjem funkcije `glfwGetKey`.

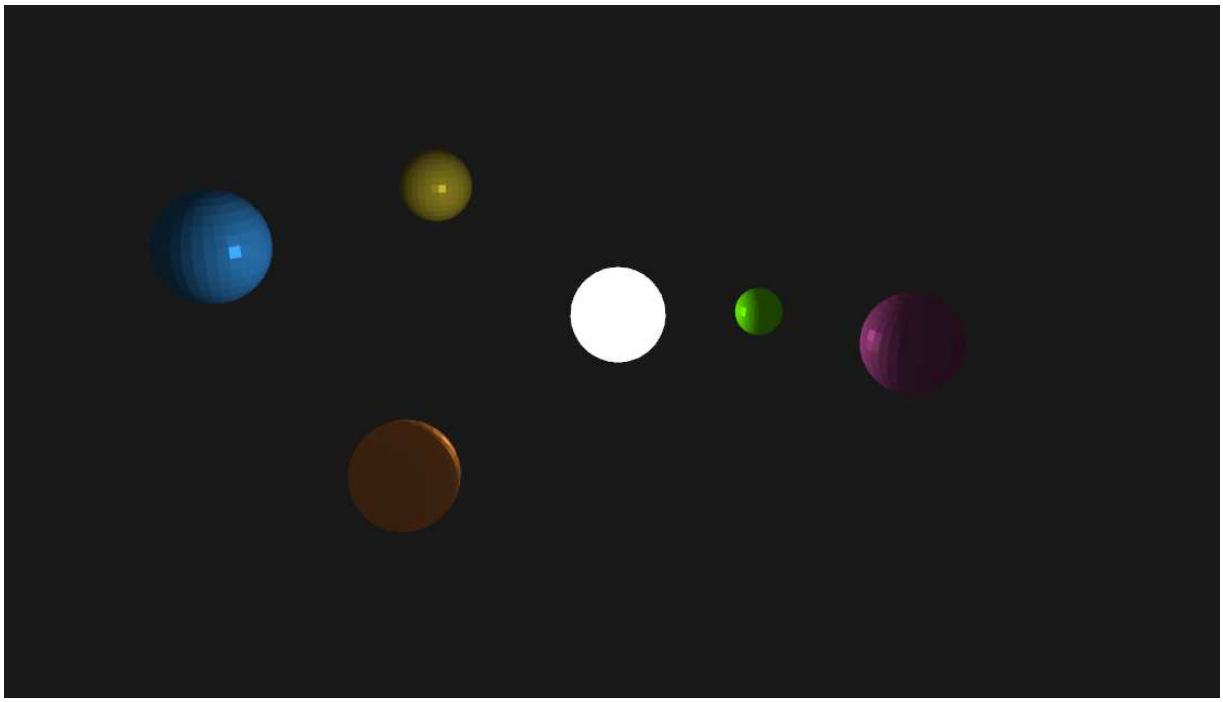
3. (40% bodova) Simulacija sunčevog sustava:
   Napomena: pod objekt se smatra ponuđeni model sfere, ali ako niste riješili 1. zadatak, možete koristiti i kocku iz primjera.

   3.1 Stavite jedan objekt u centar globalnog koordinatnog sustava i neka on bude bijele boje.

   3.2 Druge objekte rasporedite u sceni tako da se okreću oko bijelog objekta u centru koordinatnog sustava, ali na različitim udaljenostima. Objekti se trebaju rotirati oko centralnog objekta različitim brzinama. (Rotacija planeta oko sunca)

   3.3 Svaki objekt se dodatno mora rotirati oko svoje lokalne Y osi. (Rotacija planeta oko svoje osi)

   3.4 Sjenčajte planete na način da se izvor svjetlosti nalazi u centru globalnog koordinatnog sustava. Neka svjetlost bude bijele boje. Svjetlost ne bi trebala utjecati na bijeli objekt u centru koordinatnog sustava (Sunce).



Slika 8 Primjer rješenja zadataka

Dodatni zadatci:

Dodatni zadatci ne donose bodove i nisu obavezni ali ste ih slobodni implementirati:

1. Znanjem o tome da sjenčar fragmenata interpolira izlaze iz prethodnih sjenčara na svome ulazu, pokušajte sjenčati sferu tako da izgleda glatko.
2. Implementirajte FPS kameru kojom se slobodno možete kretati kroz prostor scene.
3. Istražite kako bi dodali teksture na modele, što su mipmape i uv koordinate. Učitajte neki model s teksturom u scenu.
4. Trenutno osvjetljenje ne baca sjene s jednog modela na drugi već se svaki fragment sjenča neovisno o svojem okruženju. Istražite kako dodati sjene u scenu, što su mape sjena i implementirajte ih.