Fakultet elektrotehnike i računarstva Sveučilišta u Zagrebu
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave

Laboratorijske vježbe iz kolegija
Interaktivna računalna grafika

Uporaba OpenGL-a inačice 3.x

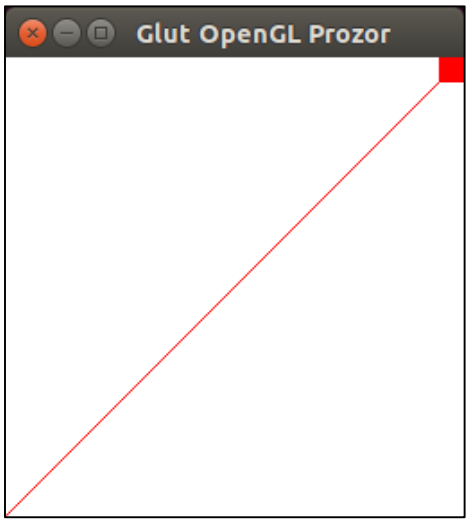
Karla Brkić
Marko Čupić
Željka Mihajlović

Zagreb, 2016.

Ovaj dokument sadrži pojašnjenja programskih primjera sa sjenčarima za OpenGL inačice 3 dostupnih na stranicama kolegija. Dokument je strukturiran tako da je za svaki primjer dan kratki opis te su uvedeni osnovni pojmovi potrebni za njegovo razumijevanje. Primjeri su zamišljeni tako da se kroz njih prolazi slijedno počevši od prvog; savjetujemo studentima da ne preskaču primjere.
Za prevođenje i pokretanje ovih primjera pod operacijskim sustavom Windows korištenjem razvojnog okruženja Microsoft Visual Studio pripremili smo zasebne upute koje su dostupne na stranicama kolegija. Za prevođenje pod operacijskim sustavom Linux uz svaki primjer nalazi se i bash skripta `prozor.sh`. Primjer možete prevesti tako da u konzoli napišete `bash prozor.sh`.

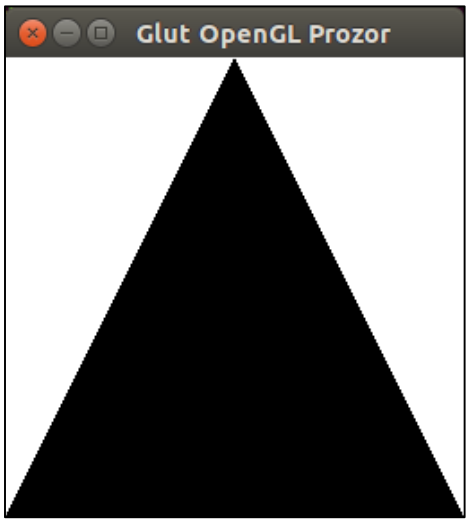
Primjer 1.

Primjer 1 je osnovni OpenGL primjer s GLUT-om u kojem se otvara prozor i iscrtava jedna linija te jedan kvadratić. Za iscrtavanje koristimo klasične OpenGL pozive (OpenGL inačica 2). Primjer je napisan u skladu s osnovnim primjerom danim u knjizi.
Pokretanjem primjera iscrtat će se sljedeće:



Primjer 2.

Primjer 2 je osnovni primjer u kojem za iscrtavanje koristimo sjenčare, odnosno OpenGL inačice 3. Ovaj primjer na ekranu iscrtava trokut korištenjem pretpostavljene funkcionalnosti sjenčara vrhova i sjenčara fragmenata koji svaki vrh ostavljaju nepromijenjenim i iscrtavaju točke crnom bojom. Pokretanjem programa dobit će se slika prikazana u nastavku:



Metode koje su ključne za razumijevanje ovog primjera su metoda `main`, metoda `init_data` i metoda `myDisplay`.

Metoda `main`

Metoda `main` započinje inicijalizacijom OpenGL-a:

```
glutInitContextVersion(3, 3);
glutInitContextProfile(GLUT_CORE_PROFILE);
```

Metoda `glutInitContextVersion` odabire inačicu OpenGLa koja će se koristiti u aktivnom OpenGL konktekstu. OpenGL kontekst možemo promatrati kao podatkovnu strukturu u kojoj se pamte sva stanja i podaci jednog primjerka OpenGL-a. Kontekst se implicitno stvara kada stvaramo OpenGL prozor. U prikazanom primjeru metodom `glutInitContextVersion` definirano je da će se koristiti OpenGL verzije 3.3.
Metoda `glutInitContextProfile` postavlja profil koji će se koristiti u OpenGL kontekstu. Profili su podskupovi funkcionalnosti OpenGL-a korisni za različite primjene (razvoj igara, računalno modeliranje i sl.). U ovom primjeru učitan je `GLUT_CORE_PROFILE`, čime imamo pristup podskupu osnovnih funkcionalnosti OpenGL-a.

Daljnja inicijalizacija odvija se na način uobičajen u ranijim inačicama OpenGL-a: funkcijom `glutInitDisplayMode` definira se način prikaza, potom se inicijalizira i stvara prozor te se definiraju funkcije za prikaz, promjenu veličine prozora, te obradu pritiska miša i tipkovnice. Konačno, poziva se:

```
glewExperimental = GL_TRUE;
glewInit();
```

Ovim se pozivom inicijalizira biblioteka GLEW koja nam omogućuje da na jednostavan način koristimo moderne API funkcije OpenGL-a. Dostupnost pojedinih funkcija OpenGL-a ovisi o konkretnom sklopovlju na kojem se program izvodi. Bez GLEW-a, u našem bismo programu morali ručno provjeravati koja je verzija OpenGL-a podržana trenutnim hardverom, koje funkcije su podržane, te bismo potom za svaku funkciju koju želimo koristiti morali inicijalizirati pokazivač. GLEW značajno pojednostavljuje ovaj proces. Da bismo omogućili korištenje GLEW-a u našem programu sve što je potrebno je pozvati funkciju `glewInit`. Linija `glewExperimental = GL_TRUE` osigurava da GLEW ispravno radi i u slučaju da se koriste eksperimentalni ili neslužbeni upravljački programi za grafičku karticu, što može biti slučaj npr. pod operacijskim sustavom Linux.

Nakon što je završena incijalizacija konteksta i GLEW-a, metoda `main` poziva korisničku metodu `init_data` kojom se inicijaliziraju sve strukture i podaci koji će se koristiti u programu. U slučaju da su podaci uspješno inicijalizirani, pokreće se iscrtavanje pozivom metode `glutMainLoop`.

Metoda `init_data`

U metodi `init_data` inicijaliziramo potrebne podatkovne strukture kojima ćemo osigurati iscrtavanje jednog trokuta na ekran. Tri su pojma važna za razumijevanje metode: VertexArrayObject, VertexBufferObject i tok vrhova.

1. VertexArrayObject

`VertexArrayObject` (VAO) enkapsulira varijetet informacija o nizu vrhova: koordinate, normale, boje, redoslijed kojim se polje vrhova pretvara u tok vrhova... Objekti tipa `VertexArrayObject` stvaraju se naredbom `glGenVertexArrays`. Naredba u memoriji stvara traženi broj ovih objekata, za svaki objekt generira jedinstveni identifikator koji se kasnije koristi za referenciranje pojedinog objekta, te pozivatelju u predano polje vraća popis tih identifikatora. Identifikatori su tipa `GLuint`. U najjednostavnijem slučaju predano polje može biti veličine 1 čime se umjesto adrese polja može predati adresa obične varijable tipa `GLuint`.

U ovom primjeru, na početku metode `init_data` sljedećim je pozivom zatraženo stvaranje jednog VAO-a:

```
glGenVertexArrays(1, &vertexArrayID);
```

Identifikator stvorenog VAO-a upisao se u varijablu `vertexArrayID`. Uočite da je u ovom primjeru varijabla `vertexArrayID` tretirana kao jednoelementno polje i stoga je predana njezina adresa.

Naredba kojom se VAO postavlja kao trenutni je `glBindVertexArray`. U našem primjeru, postavljamo upravo stvoreni VAO kao trenutni pozivom:

```
glBindVertexArray(vertexArrayID);
```

`VertexArrayObject` opisuje *koje sve podatke* imamo o nekom nizu vrhova, ali *ne sadržava eksplicitno te podatke*, već referencira druge objekte (tipa `VertexBufferObject`) koji ih čuvaju.

2. VertexBufferObject

`VertexBufferObject` (VBO) predstavlja jedan niz podataka u memoriji; to može biti polje koordinata, isprepleteno (engl. *interlaced*) polje koordinata i boja, polje indeksa koje određuje kojim se redoslijedom dohvaćaju vrhovi, itd.

Objekti tipa `VertexBufferObject` stvaraju se pozivom metode `glGenBuffers`:

```
glGenBuffers(kolikoSpremnikaTrebaStvoriti, &identifikatorSpremnika)
```

pri čemu se može tražiti stvaranje jednog ili više objekata, a naredba u predano polje identifikatora za svaki stvoreni objekt upisuje jedinstveni identifikator koji se kasnije koristi za referenciranje tog objekta. U ovom primjeru zatraženo je stvaranje jednog spremnika čiji se identifikator upisuje u varijablu `vertexbuffer`:

```
glGenBuffers(1, &vertexbuffer);
```

Nakon što je spremnik stvoren, potrebno je definirati koju će ulogu taj spremnik imati u programu, tj. koji tip podataka će čuvati. Ovo radimo pozivom metode `glBindBuffer` koja ima sljedeći potpis:

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

Prvi argument označava vrstu spremnika, a drugi argument predstavlja jedinstveni identifikator spremnika koji je vratila metoda `glGenBuffers`.

Važno: Metoda `glBindBuffer` određuje da se za zadanu vrstu spremnika u trenutnom VAO koristi spremnik čiji je identifikator drugi argument. Jedan VAO u jednom trenutku može za istu vrstu spremnika koristiti jedan podatkovni spremnik. Poseban slučaj je vrsta `GL_ARRAY_BUFFER` koja nije vezana uz trenutni VAO, već se ponaša kao "globalna" varijabla.

U ovom primjeru poziva se:

```
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
```

U ovom primjeru vrsta je `GL_ARRAY_BUFFER`, što označava da će naš spremnik čuvati podatke o vrhovima koji će kasnije biti obrađeni odgovarajućim primitivom za iscrtavanje. Kao što je već rečeno, `GL_ARRAY_BUFFER` se ponaša kao "globalna" varijabla na razini programa; ovim pozivom definirat ćemo da je trenutan spremnik s podacima o vrhovima sadržan u spremniku čiji je identifikator pohranjen u varijabli `vertexbuffer`. Za razliku od vrste `GL_ARRAY_BUFFER`, sve ostale vrste spremnika bit će vezane uz trenutni VAO (onaj kojeg smo zadnje postavili kao trenutnog pozivom `glBindVertexArray(…)`).

Kako bismo spremnik vrhova napunili podacima, potrebno je pozvati funkciju `glBufferData`. U ovom primjeru se poziva:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
g_vertex_buffer_data, GL_STATIC_DRAW);
```

Pri tome je `g_vertex_buffer_data` polje s podacima koje je korisnik pripremio i koje je potrebno prekopirati u aktivni spremnik vrste `GL_ARRAY_BUFFER`. Kod metode `glBufferData` prvi argument određuje vrstu spremnika u koji je potrebno pohraniti podatke (koji je to točno spremnik, prethodno je trebalo biti postavljeno pozivom `glBindBuffer`), drugi argument ukupnu veličinu podataka u oktetima koje je potrebno prekopirati u spremnik,

treći argument predstavlja adresu od koje se u korisnikovoj memoriji nalaze pripremljeni podaci, a četvrti argument definira način uporabe ovih podataka (koristimo `GL_STATIC_DRAW`).

3. Tok vrhova

Jednom kada je spremnik vrhova definiran, kako bi se pokrenulo iscrtavanje potrebno je poslati tok (stream) vrhova sjenčaru vrhova na obradu. Program sjenčara vrhova definira niz atributa koje prima za svaki vrh. Atributi su numerirani od 0 na više. Primjerice, program sjenčara može tražiti da kao nulti atribut dobije koordinate vrha a kao prvi atribut boju vrha.

Kad se pokrene program sjenčara vrhova, za svaki vrh, od svih atributa koji su poznati za taj vrh, programu se dostavljaju samo atributi koji su omogućeni. Omogućavanje slanja atributa se radi s:

```
glEnableVertexAttribArray(indeksAtributa);
```

a onemogućavanje s:

```
glDisableVertexAttribArray(indeksAtributa);
```

U općem slučaju, spremnik s podacima može za svaki vrh sadržavati različite informacije. U najjednostavnijem slučaju, spremnik s podacima sadrži samo slijedno zapisane koordinate vrhova. U složenijem slučaju, spremnik može sadržavati isprepleteno najprije koordinate prvog vrha, pa boju prvog vrha, potom koordinate drugog vrha, pa boju drugog vrha itd. Stoga je potrebno definirati na koji će se način iz spremnika rekonstruirati određeni atributi.
Kako se to točno radi za određeni atribut, određuje se metodom `glVertexAttribPointer(indeks,…)`. Ova metoda u VAO upisuje da se atribut rednog broja `indeks` dohvaća iz spremnika koji je trenutno vezan pod `GL_ARRAY_BUFFER` na način definiran ostatkom argumenata. Kako se ovo zapisuje u VAO, korisnik kasnije pod `GL_ARRAY_BUFFER` može povezati neki drugi spremnik i to ništa ne mijenja - i dalje se za zadani atribut koristi spremnik koji je bio "trenutni" u trenutku poziva ove metode.

Nakon što korisnik stvori jedan VAO i postavi ga kao aktivnog, može stvoriti jedan ili više VBO-ova (gdje svaki čuva određenu vrstu informacija) te potom funkcijom `glVertexAttribPointer` pojasniti na koji se način za trenutni VAO podatci za atribut zadanog indeksa dohvaćaju iz trenutnog VBO-a. Ta se informacija pohranjuje u sam VAO.

Potpis metode `glVertexAttribPointer` je sljedeći:

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

U ovom primjeru, niz `g_vertex_buffer_data` sadrži 9 elemenata i predstavlja tri vrha s po tri koordinate. Kako bi se sjenčaru slale tri po tri koordinate, definiramo:

```
glVertexAttribPointer(
         0,  // indeks atributa, koji mora odgovarati indeksu u sjenčaru
         3,  // broj koordinata u vrhu - mora biti 1, 2, 3 ili 4
         GL_FLOAT, // tip koordinate
         GL_FALSE, // automatska normalizacija na raspon [-1, 1]
         0,        // razmak u oktetima između vrhova; 0: slijedni vrhovi
         (void*)0  // indeks okteta u nizu na kojem počinju vrhovi
      );
```

Predzadnji parametar (`stride`) predstavlja ukupan broj okteta između početaka uzastopnih zapisa atributa. U ovom primjeru spremnik sadrži koordinate vrhova koje su tipa GL_FLOAT (4 okteta); kako svaki vrh ima tri koordinate stride bi bio 4*3=12 jer podaci o sljedećem vrhu u spremniku kreću neposredno nakon. Kada je to slučaj, stride je dozvoljeno postaviti na vrijednost 0, čime će OpenGL sam izračunati ovu vrijednost.

Pretvorba u `void*` posljednjeg parametra potrebna je zbog kompatibilnosti sa starijim verzijama OpenGL-a, no posljednji parametar predstavlja običan redni broj okteta u nizu na kojem počinju podaci o vrhovima. Kako u prikazanom slučaju podaci o vrhovima počinju od početka polja, ovaj parametar postavljen je na 0. Više informacija o ovoj metodi moguće je pronaći u dokumentaciji metode.

Po završetku metode `init_data` sve podatkovne strukture potrebne za iscrtavanje trokuta na ekran su inicijalizirane. Definiran je jedan VAO i postavljen je kao trenutni. Potom je stvoren jedan VBO i napunjen je podacima o 3 vrha. Konačno, definirano je na koji način se podaci trebaju čitati iz VBO-a (počevši od početka, čitaju se po 3 koordinate koje su decimalni brojevi, te nije potrebno raditi normalizaciju). Jednom kad su inicijalizirane podatkovne strukture, moguće je iscrtati trokut na ekran. Iscrtavanje se odvija u metodi `myDisplay`.

Metoda `myDisplay`

Metoda `myDisplay` započinje brisanjem ekrana. Nakon toga kao trenutni VAO postavlja se VAO kojeg se inicijaliziralo u metodi `init_data`. Kako bi se to postiglo, koristi se njegov identifikator pohranjen u varijabli `vertexArrayID` koji je popunjen u metodi `init_data`:

```
glBindVertexArray(vertexArrayID);
```

U metodi `init_data` metodom `glVertexAttribPointer` definirali smo vrhove kao atribut s indeksom 0 koji se sastoji od 3 koordinate. Prije no što se zatraži crtanje, potrebno je definirati koji se sve atributi šalju sjenčaru. Omogućavanje slanja atributa obavlja se naredbom `glEnableVertexAttribArray` koja kao argument dobiva redni broj atributa. U slučaju da je potrebno za svaki vrh slati više od jednog atributa, svaki od atributa omogućava se zasebno. Kako se u slučaju iz primjera koristi samo atribut s indeksom 0, odgovarajući poziv je:

```
glEnableVertexAttribArray(0);
```

gdje je kao indeks atributa predano 0, što je u skladu s ranije definiranim indeksom u funkciji `glVertexAttribPointer`.

Nakon ovoga pristupa se iscrtavanju jednostavnim pozivom:

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

čime je zatraženo iscrtavanje **toka vrhova** pri čemu se taj tok tretira kao trokute (definirano konstantom `GL_TRIANGLES`). Potpis metode `glDrawArrays` je sljedeći:

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

Parametar `mode` određuje koji primitiv želimo crtati korištenjem vrhova. Primjeri podržanih primitiva opisani su u knjizi i obrađeni na predavanjima; neki od njih su `GL_TRIANGLES`, `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, itd. Parametar `first` određuje indeks prvog elementa u svakom omogućenom toku koji će se obrađivati, dok parametar `count` određuje koliko će se elemenata obraditi. U ovom primjeru, omogućeno je slanje atributa 0 (trojki koordinata vrhova), te je pozivom metode `glDrawArrays(…)` zatraženo da se za dani tok počevši od indeksa 0 uzme 3 vrha (točno toliko ih i imamo u toku) te da se korištenjem ta 3 vrha iscrta trokut.

Ako se crtanje radi metodom `glDrawArrays(…)`, tok vrhova generira se izravno prema redoslijedu kojim su vrhovi spremljeni u spremnik vrhova. Ako je u VAO povezan spremnik indeksa kao GL_ELEMENT_ARRAY_BUFFER, metodom `glDrawElements(…)` može se tražiti generiranje i obrada toka vrhova koji nastaje prema zadanim indeksima (ovo je ilustrirano u primjeru 2c).

U ovom primjeru nije posebno podešavan pogled, pa je važno znati sljedeće: bez ikakvih podešavanja, kamera je smještena u ishodište te je usmjerena prema negativnoj *z*-osi, view-up vektor je pozitivna *y*-os, volumen pogleda određen je po *x*, *y* i *z*-osima od -1 do +1; *viewport* je čitav prozor. Ovime će ishodište koordinatnog sustava (pozicija kamere) biti u centru prozora; +*x* os gleda u desno, +*y* os prema gore a +*z* os po pravilu desne ruka gleda iz ekrana prema korisniku.

Više detalja o ovoj temi može se pročitati na adresi:
https://www.opengl.org/wiki/Vertex_Specification

Primjer 2b.

Primjer 2b je proširenje primjera 2 u kojem po prvi puta radimo s vlastitim programima za sjenčare vrhova i fragmenata. Ovi se programi pišu u posebnom programskom jeziku GLSL (engl. *GL Shading Language*) koji je sintaksom sličan programskom jeziku C. Ono što je pomalo neuobičajeno je da je programe sjenčara potrebno **programski učitavati, prevoditi i povezivati** izravno kroz programski kod. Ovo je važna distinkcija u odnosu na klasične programe koji se izvode na operacijskom sustavu i koji se iz izvornog koda prevode u strojni kod kroz razvojnu okolinu ili izravno uporabom naredbenog retka. U ovom primjeru, učitavanje, prevođenje i povezivanje programa za sjenčare se radi na kraju funkcije `init_data`, pozivom:

```
programID = loadShaders("SimpleVertexShader.vertexshader",
"SimpleFragmentShader.fragmentshader");
```

Metoda `loadShaders` je metoda koju smo sami napisali, smještena je u pomoćnoj biblioteci `util.cpp` i slobodno je možete koristiti u daljnjem radu pri izradi laboratorijskih vježbi. Metodom `loadShaders` učitat će se sjenčar vrhova iz datoteke `SimpleVertexShader.vertexshader` i sjenčar fragmenata iz datoteke `SimpleFragmentShader.fragmentshader`. Ukoliko sve bude u redu, programski kod oba sjenčara će biti preveden i povezan u jedinstven program kojemu će biti dodijeljen jedinstveni identifikator koji pohranjujemo u varijablu `programID`. Naknadno referenciranje na taj program obavlja se uporabom ovog identifikatora.

Važna napomena: uz ovakav poziv metode `loadShaders` program će očekivati da su datoteke sjenčara vrhova `SimpleVertexShader.vertexshader` i sjenčara fragmenata `SimpleFragmentShader.fragmentshader` smještene u istoj mapi u kojoj je i izvršna datoteka programa. Ukoliko prevodite program korištenjem razvojne okoline Microsoft Visual Studio to neće biti slučaj, pa je potrebno promijeniti putanje do datoteka tako da su apsolutne (u putanjama koristite dvostruku obrnutu kosu crtu). Primjer:

```
programID = loadShaders("C:\\IRG\\SimpleVertexShader.vertexshader",
"C:\\IRG\\SimpleFragmentShader.fragmentshader");
```

Kako bi se omogućilo da se za iscrtavanje koriste učitani sjenčari, unutar metode `myDisplay` dodana je naredba:

```
glUseProgram(programID);
```

Datoteke sjenčara vrhova `SimpleVertexShader.vertexshader` i sjenčara fragmenata `SimpleFragmentShader.fragmentshader` su obične tekstovne datoteke koje možete otvoriti unutar razvojne okoline ili u uređivaču teksta po želji. U njima je smješten GLSL programski kod sjenčara.

Datoteka SimpleVertexShader.vertexshader

```
#version 330 core


// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
```

```
void main(){
    gl_Position.xyz = vertexPosition_modelspace;
    gl_Position.w = 1.0;
}
```

Program za sjenčar vrhova je vrlo jednostavan. Započinje deklaracijom verzije OpenGL-a za koju je sjenčar pisan. Potom se deklariraju ulazni podatci. Poziv `layout(location=0)` znači da želimo nulti atribut iz ulaznog spremnika (podsjetimo se, ranije je funkcijom `glVertexAttribPointer` definirano da su vrhovi koje šaljemo sjenčaru atribut indeksa 0); `in` označava da je podatak ulazni, `vec3` označava da je podatak vektor od 3 elementa, a `vertexPosition_modelspace` je varijabla u koju će podatak biti pohranjen.

U programu za sjenčar vrhova postoji jedna implicitno deklarirana varijabla `gl_Position`, i u nju program treba zapisati konačnu poziciju vrha. Naš jednostavni sjenčar samo prosljeđuje ulaz na izlaz: ulazni vrh nalazi mu se u varijabli `vertexPosition_modelspace`, i ta se vrijednost pridjeljuje varijabli `gl_Position`. Vrijednost homogene koordinate postavlja se na 1.0.
Program može deklarirati i dodatne varijable ako želi, i to kao `uniform` varijable ili pak kao `out` varijable. Varijable tipa `uniform` su globalne GLSL varijable kojima se prenose parametri iz glavnog programa u sjenčar (primjerice, programu koji će obavljati transformaciju pogleda na ovaj ćemo način predati transformacijsku matricu). Varijable tipa `out` su izlazne varijable. Vrijednosti koje se zapišu u `out` varijable interpoliraju se između vrhova i šalju u sjenčar fragmenata gdje ih se može dohvatiti tako da se deklarira varijabla tipa `in` istog imena. Ovo ćemo vidjeti kasnije u primjeru 4.

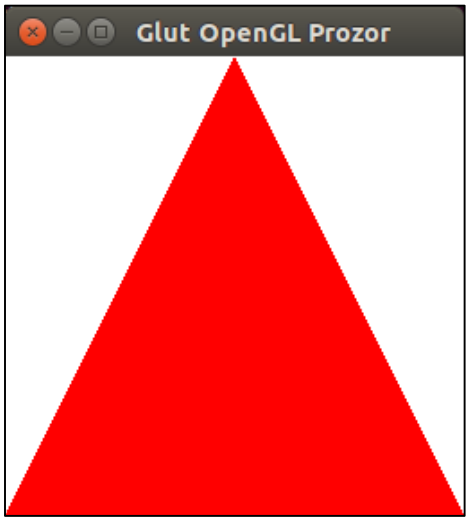
Datoteka SimpleFragmentShader.fragmentshader

```
#version 330 core

// Ouput data
out vec3 color;

void main()
{
      // Output color = red
      color = vec3(1,0,0);
}
```

Program za sjenčar fragmenata deklarira jednu izlaznu vrijednost tipa `vec3` te postavlja tu vrijednost na (1,0,0), što odgovara crvenoj boji. Sadržaj te varijable interpretirat će se kao boja kojom je potrebno prikazati taj fragment.
Pokretanjem ovog primjera dobit će se sljedeće:



Primjer 2c.

Primjer 2c je nadogradnja primjera 2b u kojem smo promijenili način na koji radimo s tokom vrhova. Umjesto da se tok vrhova izravno generira kao slijed vrhova iz predanog spremnika, u VAO dodajemo i spremnik indeksa kojim se vrhovi šalju kartici. Ovime iste vrhove možemo slati više puta, čime možemo ostvariti značajnu uštedu memorije i dobiti na brzini.
Prva promjena nalazi se u metodi `init_data`. Inicijalizacija VAO-a i VBO-a koji sadrži vrhove ostala je ista kao u prethodnim primjerima, ali nakon inicijalizacije VBO-a s koordinatama vrhova dodano je i stvaranje i inicijalizacija VBO-a s indeksima vrhova:

```
// Stvaramo polje za 6 indeksa
static const ushort vindices[6] = { 0, 1, 2, 0, 1, 3};

// Identifikator za indeksni spremnik
GLuint indexbuffer;
// Stvori 1 spremnik i pohrani identifikator u indexbuffer
glGenBuffers(1, &indexbuffer);
// postavi naš spremnik kao aktivni GL_ELEMENT_ARRAY_BUFFER
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexbuffer);
// popuni naš spremnik elementima iz polja indeksa
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(ushort)*6, vindices,
GL_STATIC_DRAW);
```

Druga promjena nalazi se u metodi `myDisplay`, gdje je metoda `drawArrays` za iscrtavanje trokuta zamijenjena metodom `drawElements`:

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, (GLvoid*)0);
```

Ovaj poziv znači da se kao tok vrhova šalju oni vrhovi čiji su indeksi specificirani spremnikom `GL_ELEMENT_ARRAY_BUFFER`, da se koristi šest indeksa počevši od nultog, te da su indeksi tipa `GL_UNSIGNED_SHORT`.

Da pojasnimo, ako se ne koristi spremnik indeksa, ako smo u spremnik vrhova povezali polje:

```
{V0,V1,V2}
```

gdje je svaki V_i primjerice tri floata, upravo se tim redoslijedom vrhovi šalju kartici, pa je tok vrhova koji se obrađuje:

```
V0 -> V1 -> V2
```

Ako s druge strane u spremnik vrhova stavimo polje:

```
{V0,V1,V2,V3}
```

tj. na poziciji 0 je V_0, na poziciji 1 je V_1, itd., te u spremnik indeksa stavimo npr:

```
{0,1,2,0,1,3}
```

tok koji se šalje čine vrhovi na pozicijama određenim ovim indeksima. Stoga će kartici biti poslan tok:

```
V0 -> V1 -> V2 -> V0 -> V1 -> V3
```

U našem primjeru je primitiv za crtanje `GL_TRIANGLES`, a on grupira po tri vrha u jedan trokut, pa će se nacrtati dva trokuta: T_1 određen vrhovima V_0, V_1, V_2 te T_2 određen vrhovima V_0, V_1, V_3. Na ovaj način u polju vrhova ne moramo ponavljati vrhove. Ovu ideju koristi i OBJ datotečni format kada definira poligone.

Konačni rezultat će biti da će se iscrtati dva trokuta koja dijele bazu i slika koju ćemo dobiti je zarotirani kvadrat, kao što je prikazano na slici u nastavku.

Primjer 3.

U ovom primjeru počinju se koristiti matrice: radi se transformacija pogleda i perspektivna projekcija.

U program sjenčara vrhova u popis varijabli dodana je jedna matrica (varijabla je nazvana MVP) kao `uniform` varijabla (podatak koji se ne mijenja od vrha do vrha). Kako bi odredio konačni položaj, program koji je u okviru ovog primjera napisan za sjenčar vrhova će dobivenu koordinatu proširiti homogenom i pomnožiti s transformacijskom matricom MVP.

Datoteka SimpleVertexShader.vertexshader

```
#version 330 core


layout(location = 0) in vec3 vertexPosition_modelspace;
uniform mat4 MVP;


void main(){
    gl_Position = MVP * vec4(vertexPosition_modelspace, 1);
}
```

Vrijednost matrice MVP postavlja se iz **glavnog programa** (jednom, prije nego što započne obrada vrhova, i matrica ostaje nepromijenjena za sve vrhove u toj obradi). U tu svrhu, u metodi `init_data` nakon prevođenja programa sjenčara koji deklarira tu varijablu potrebno je zatražiti njezin identifikator (engl. *handle*). To se radi metodom `glGetUniformLocation` koja kao argument dobije identifikator programa te ime varijable u programu sjenčara, a vraća nam traženi identifikator koji će se kasnije koristiti kada će se htjeti u tu varijablu upisati podatak. Ovo je ilustrirano u nastavku.

```
MVPMatrixID = glGetUniformLocation(programID, "MVP");
```

Matricu koja obavlja zadanu transformaciju možemo računati ili prilikom crtanja, ili već u inicijalizaciji ako se položaj kamere ne mijenja. U ovom primjeru, izračunavanje matrice obavlja se u metodi `myDisplay`. Koriste se pomoćne metode iz biblioteke glm.

Prilikom crtanja, prije no što se pokrene samo crtanje, podatke iz izračunate matrice pozivom metode `glUniformMatrix4fv` prebacujemo u lokalnu varijablu programa sjenčara (dajemo identifikator varijable koji smo prethodno pripremili te adresu podataka koje tamo prebacujemo):

```
glUniformMatrix4fv(MVPMatrixID, 1, GL_FALSE, &mvp[0][0]);
```

Prvi parametar označava identifikator matrice, drugi koliko matrica želimo modificirati (moguće je raditi i s nizovima matrica), treći definira treba li transponirati matricu, a četvrti je adresa podataka.

Nakon ovakvog postavljanja matrice na uobičajeni način pokreće se crtanje.

Kako bi se moglo koristiti metode iz biblioteke glm, potrebno je u program uključiti zaglavne datoteke `glm/glm.hpp` i `glm/gtc/matrix_transform.hpp`. Na operacijskom

sustavu Linux ove su datoteke dio biblioteke `libglm-devel` koju po potrebi treba instalirati (`sudo apt-get install ime_biblioteke`).

Pokretanjem ovog primjera iscrtat će se sljedeće:

Primjer 4.

U ovom primjeru za svaki od vrhova definira se boju, a OpenGL radi interpolaciju boja za svaki fragment. Proširuje se primjer 3 na način da se u metodi `init_data` uvodi novi spremnik koji će čuvati informacije o bojama (nazvan je `colorbuffer`). Taj se spremnik popunjava podacima o boji na isti način na koji se i spremnik vrhova popunjava podacima o vrhovima.

Nakon pojašnjavanja kako se iz jednog spremnika dohvaćaju pozicije (za atribut 0), pojašnjava se i kako se iz drugog spremnika dohvaćaju boje (tom atributu pridijeljen je indeks 1):

```
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glVertexAttribPointer(
        1,                  // attribute 1.
        3,                  // size
        GL_FLOAT,           // type
        GL_FALSE,           // normalized?
        0,                  // stride
        (void*)0            // array buffer offset
);
```

Obavezno je prije poziva metode `glVertexAttribPointer` postaviti `colorbuffer` kao trenutni `GL_ARRAY_BUFFER` pozivom metode `glBindBuffer`.

Prije iscrtavanja u metodi `myDisplay` obavezno je uključiti i slanje atributa 1 (tj. boje iz spremnika):

```
glEnableVertexAttribArray(1);
```

Sjenčar vrhova izgleda ovako:

Datoteka SimpleVertexShader.vertexshader

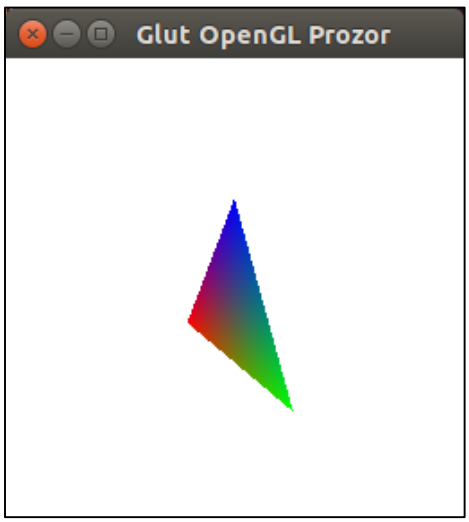
```
#version 330 core
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec3 vertexColor;

out vec3 fragmentColor;
uniform mat4 MVP;

void main(){
    gl_Position = MVP * vec4(vertexPosition_modelspace, 1);
    fragmentColor = vertexColor;
}
```

U sjenčaru vrhova dodana je lokalna varijabla tipa `in` za drugi atribut te jedna varijabla tipa `out` u koju se zapisuje primljena boja. OpenGL će za svaki od vrhova podatak koji se zapiše u

tu izlaznu varijablu interpolirati po slikovnim elementima između vrhova (kada radi popunjavanje). Tu interpoliranu vrijednost možemo primiti u sjenčaru fragmenata na način da tamo deklariramo istoimenu varijablu tipa `in`. **VAŽNO: tamo stiže interpolirana vrijednost!** U ovom primjeru napisan je najjednostavniji sjenčar fragmenata koji sadržaj te varijable samo proslijedi dalje kao boju fragmenta.
Pokretanjem primjera iscrtat će se sljedeće:



Primjer 5.

U ovom primjeru scena je proširena na dva trokuta koji se probadaju. Kako bi se dobio korektan prikaz uzimajući u obzir činjenicu da se trokuti probadaju, uključen je Z-spremnik.

U metodu `main` dodane su sljedeće naredbe:

```
// Omogući uporabu Z-spremnika
glEnable(GL_DEPTH_TEST);
// Prihvaćaj one fragmente koji su bliže kameri u smjeru gledanja
glDepthFunc(GL_LESS);
```

U metodi `myDisplay` obavezno je pri brisanju brisati i Z-spremnik, pa se stoga u `glClear` dodaje i konstanta `GL_DEPTH_BUFFER_BIT`:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Ostatak programa isti je kao u prethodnom primjeru. Pokretanjem programa iscrtat će se sljedeće:

Literatura

Više informacija o sjenčaru vrhova može se pronaći na adresi:
https://www.opengl.org/wiki/Vertex_Specification

Prikazani primjeri dijelom su zasnovani na primjerima sa adrese:
http://www.opengl-tutorial.org/beginners-tutorials/