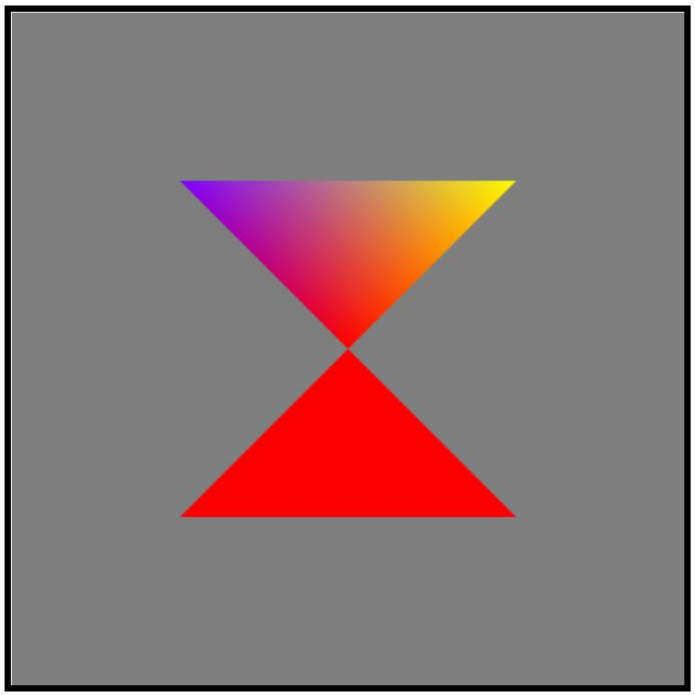
Programiranje procesora za sjenčanje vrhova i fragmenata



2. dio:

Rasterizacija

Rasterizacijska faza

- nakon što su u geometrijskoj fazi vrhovi transformirani u normirane koordinate sljedeći korak je rasterizacija linija i trokuta koji su tim vrhovima definirani
- rasterizacija se provodi automatski no moguće je utjecati na boju svakog pojedinog piksela, tj. fragmenta (u 3D sceni neki se objekti nalaze iza drugih objekata pa se ne iscrtavaju – fragment još nije piksel!)

Fragment vs. piksel

- piksel je ono što se konačno vidi na zaslonu (skraćeno od engl. *picture element*)
- rasterizacijom linija ili trokuta dobivaju se fragmenti no to još ne znači da će se svaki fragment prikazati na zaslonu:
  - ovisno o z koordinati možda neće biti iscrtan jer se nalazi iza nekog drugog fragmenta
  - ovisno o zadanoj prozirnosti može biti kombiniran s bojom nekog drugog fragmenta

Program za sjenčanje fragmenata

- izvodi se posebno za svaki fragment!
- postavlja se boja svakog fragmenta, ali može se i intervenirati u spremnik dubine ili manipulirati teksturama
- očigledno da je nužno masovno-paralelno izvođenje da bi to bilo efikasno i primjenjivo u praksi (na primjer, svaka slika u HD rezoluciji ima oko 2 milijuna piksela!)

Primjer 6.1. – fragment shader

- iz RG-primjer6-1-crtanje-linija-i-trokuta.html

```
#version 300 es
precision mediump float;
out vec4 bojaPiksela;

void main() {
  bojaPiksela = vec4(0, 1, 0, 1); //RGBA
}
```

Deklariranje verzije

- **`#version 300 es`** znači da se radi o OpenGL ES verziji 3.0
- deklaracija verzije mora biti u prva linija koda programa za sjenčanje i ne smije sadržavati dodatne znakove (osim razmaka)
- ako nema deklaracije verzije podrazumijeva se (default) da je verzija 1.0 – naši primjeri možda neće raditi!

Preciznost

- u fragment shaderu preciznost se mora obavezno postaviti jer nije automatski zadana
- najbolje je postaviti istovremeno za sve float varijable: **precision mediump float**
- alternativa je postaviti za svaku varijablu posebno, na primjer: **out mediump vec4 bojaPiksela**
- ostale opcije su **highp** i **lowp**
- u vertex shaderu default vrijednost je **highp**
- *napomena: **highp** ne znači double precision*

Postavlja se boja piksela

- u starijim varijantama OpenGL-a postojala je ugrađena varijabla **gl_FragColor** u koju je na kraju trebalo postaviti boju piksela (slično kao **gl_Position** kod sjenčanja vrhova u koji se spremaju konačne koordinate vrhova)
- sadašnja varijanta je da se boja piksela postavlja u varijablu proizvoljnog imena, ali koja je deklarirana kao **out** u programu za sjenčanje fragmenata

Format boje

Boja piksela postavlja se u RGBA formatu:

- prva tri parametra se intenzitet crvene, zelene i plave boje u rasponu od 0 do 1
- četvrti parametar je takozvani *alpha*, također u rasponu od 0 do 1, koji predstavlja prozirnost (0 znači potpuno prozirno, 1 potpuno neprozirno)
- ako je bilo koji parametar manji od 0 uzima se da je 0, ili ako je veći od 1 uzima se da je 1, bez da se javlja pogreška

Sjenčanje (nijansiranje)

- postavlja se pitanje kako postići fine prijelaze svjetline i tonova boja koji su karakteristični za stvarne objekte koji su pod različitim vrstama osvjetljenja
- to se može postići korištenjem takozvanih `varying` varijabli

Specifične vrste varijabli u GLSL-u

- programski jezici za sjenčanje *GL Shading Language* i *GL Shading Language ES*, kao njegov podskup, koriste osim običnih i tri specifične vrste varijabli:
- `attribute` – služe za prijenos podataka o vrhovima u procesor za sjenčanje vrhova
- `uniform` – služe za prijenos podataka koji su zajednički svim vrhovima ili fragmentima
- `varying` – služe za prijenos i interpolaciju podataka za pojedine fragmente u procesor za sjenčanje fragmenata

varying varijable

- vrijednosti varying varijabli se interpoliraju za svaki fragment, ovisno o položaju fragmenta prema vrhovima u kojima je vrijednost varijable definirana
- varying varijabla se u programu za sjenčanje vrhova deklarira kao out
- ta ista varijabla mora u programu za sjenčanje fragmenata biti deklarirana kao in

Primjer 7.1. – vertex shader

- iz `RG-primjer7-1-bojanje-piksela.html`

```
#version 300 es
in vec2 a_vrhXY;
out vec2 v_vrhXY;

void main() {
  gl_Position = vec4(a_vrhXY, 0, 1);
  v_vrhXY = a_vrhXY;
}
```

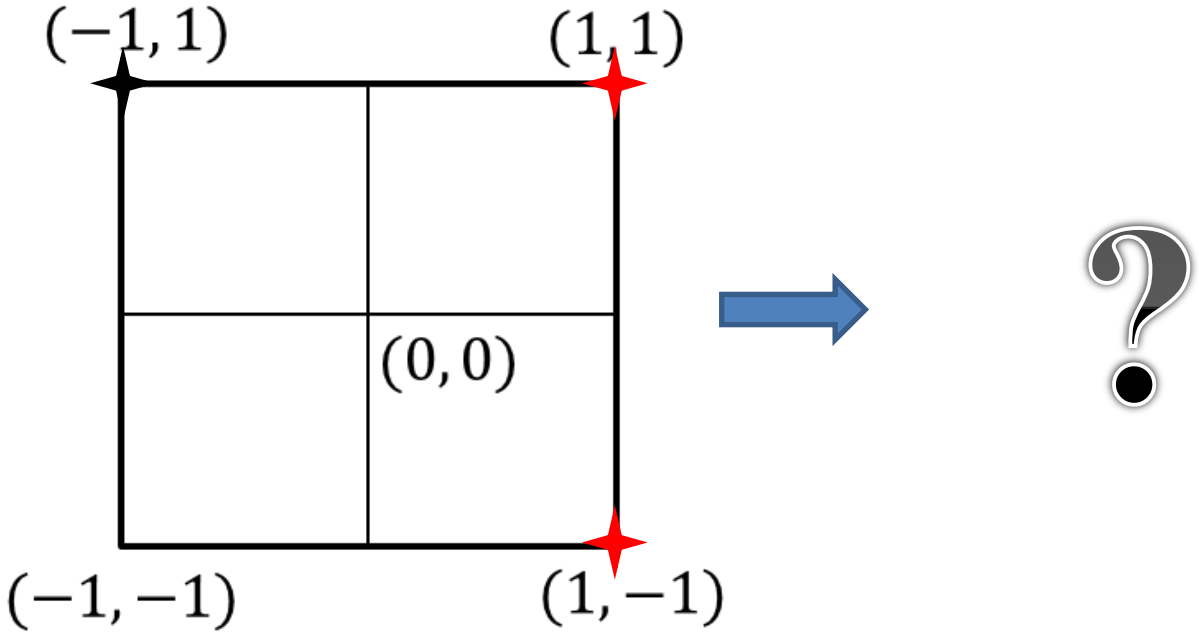
Primjer 7.1. – fragment shader

- iz `RG-primjer7-1-bojanje-piksela.html`

```
#version 300 es
precision highp float;
in vec2 v_vrhXY;
out vec4 bojaPiksela;

void main() {
  bojaPiksela = vec4(v_vrhXY.x, 0, 0, 1);
}
```

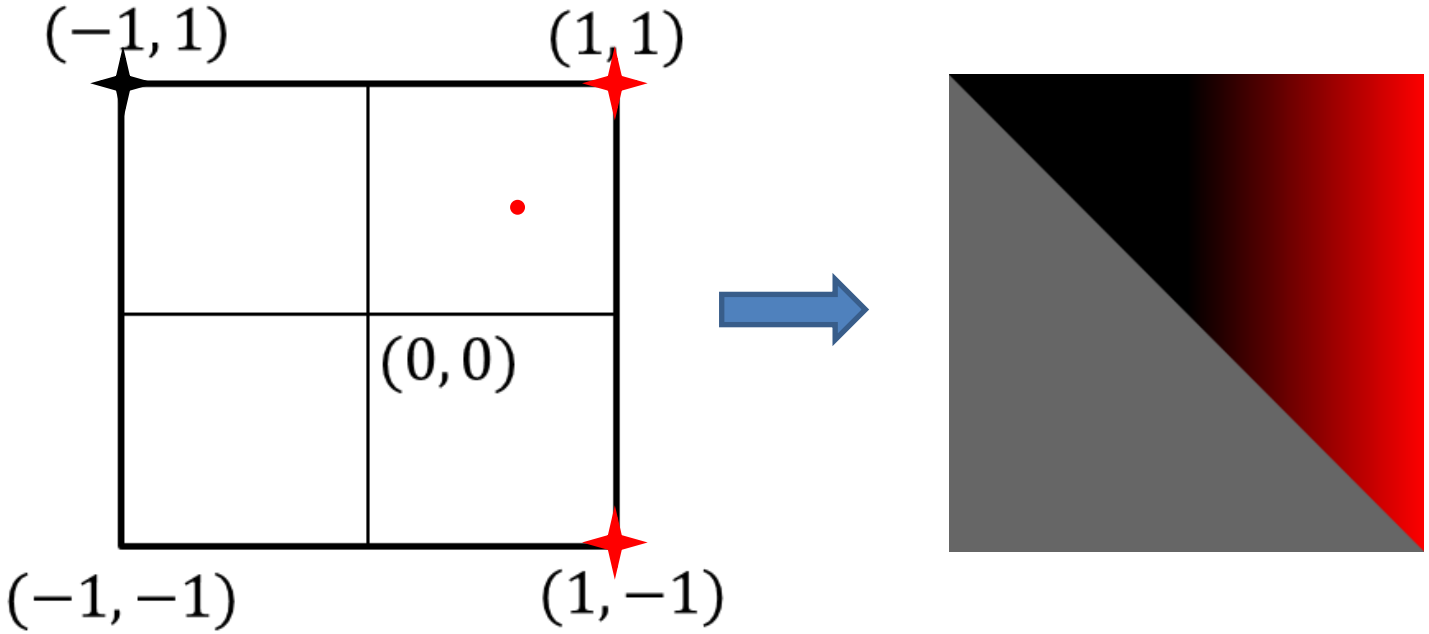
Primjer 7.1. – boja ovisna o koordinati

- imamo informaciju samo o tri vrha koja su procesirana u programu za sjenčanje vrhova



Primjer 7.1. – boja ovisna o koordinati

- intenzitet crvene boje raste ovisno x-koordinati fragmenta/piksela



Primjer 7.2. – vertex shader

- iz `RG-primjer7-2-bojanje-vrhova.html`

```
#version 300 es
in vec2 a_vrhXY;
in vec3 a_boja;
out vec3 v_boja;

void main() {
  gl_Position = vec4(a_vrhXY, 0, 1);
  v_boja = a_boja;
}
```

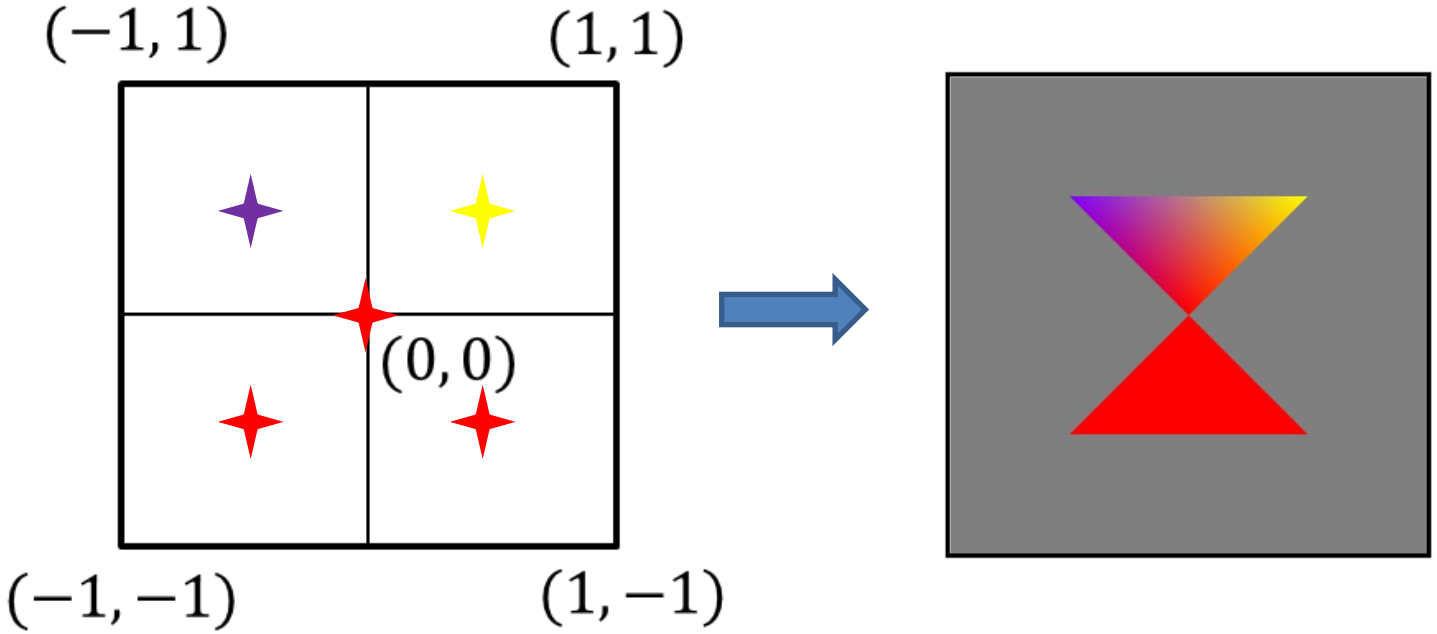
Primjer 7.2. – fragment shader

- iz `RG-primjer7-2-bojanje-vrhova.html`

```
#version 300 es
precision highp float;
in vec3 v_boja;
out vec4 bojaPiksela;

void main() {
  bojaPiksela = vec4(v_boja, 1); //RGBA
}
```

Primjer 7.2. – interpolacija boje

- boje vrhova su zadane, za svaki fragment se provodi interpolacija boje!



Isprepleteni spremnik

Kako prenijeti informaciju o boji svakog pojedinog vrha u program za sjenčanje vrhova?

- korištenjem `attribute` varijable `a_boja`
- ne treba novi spremnik, može se koristiti takozvani **isprepleteni spremnik** (engl. *interleaved buffer*) u koji se mogu zajedno pohraniti podaci više attribute varijabli (na primjer: koordinate i boja u RGB formatu)

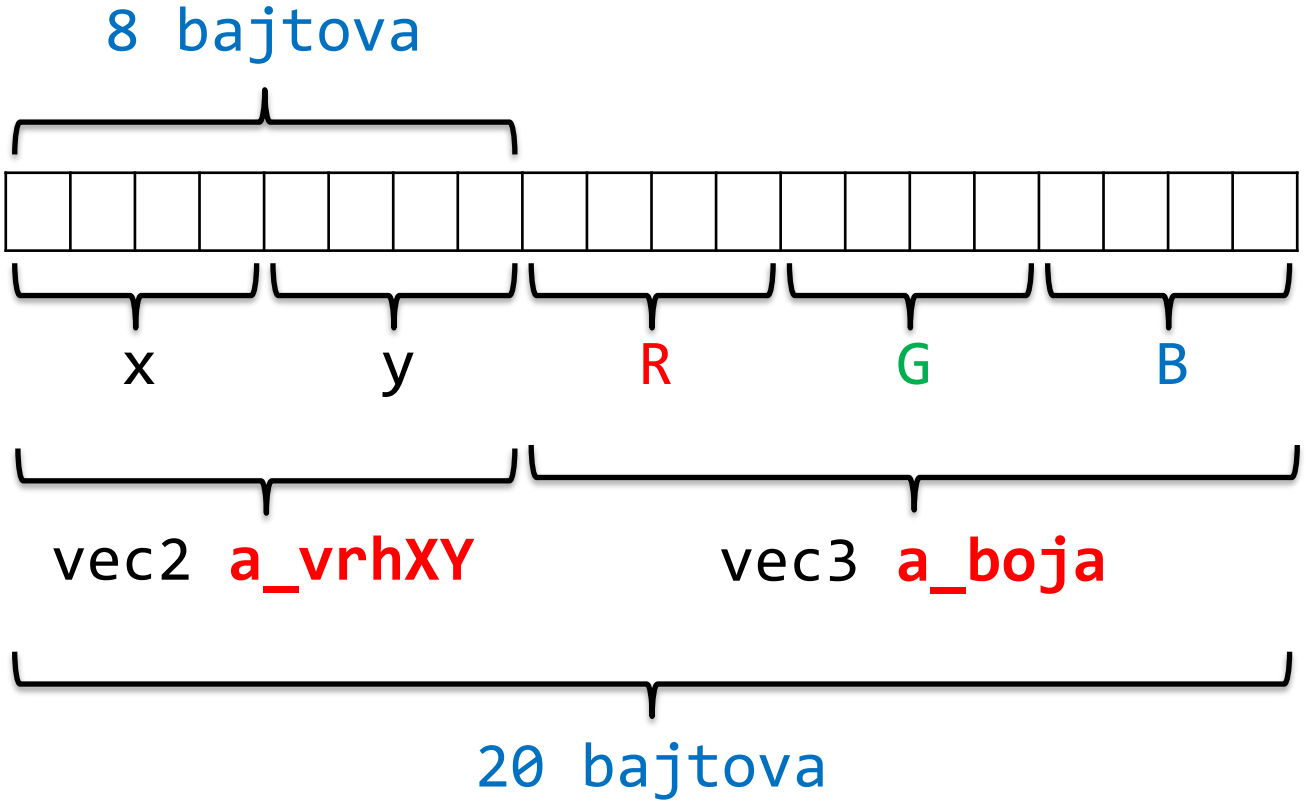
Primjer 7.2. – priprema spremnika

- iz `RG-primjer7-2-bojanje-vrhova.html`

```
// definiranje geometrije preko javascript polja
let a = 0.5;
vrhovi = [[ 0,  0,  1, 0, 0],  // crveno
          [-a, -a,  1, 0, 0],  // crveno
          [ a, -a,  1, 0, 0],  // crveno
          [ 0,  0,  1, 0, 0],  // crveno
          [ a,  a,  1, 1, 0],  // žuto
          [-a,  a,  0.5, 0, 1]]; // ljubičasto
```

Primjer 7.2. – priprema spremnika

- podaci o pojedinom vrhu zapisani su u 5 `float` varijabli:



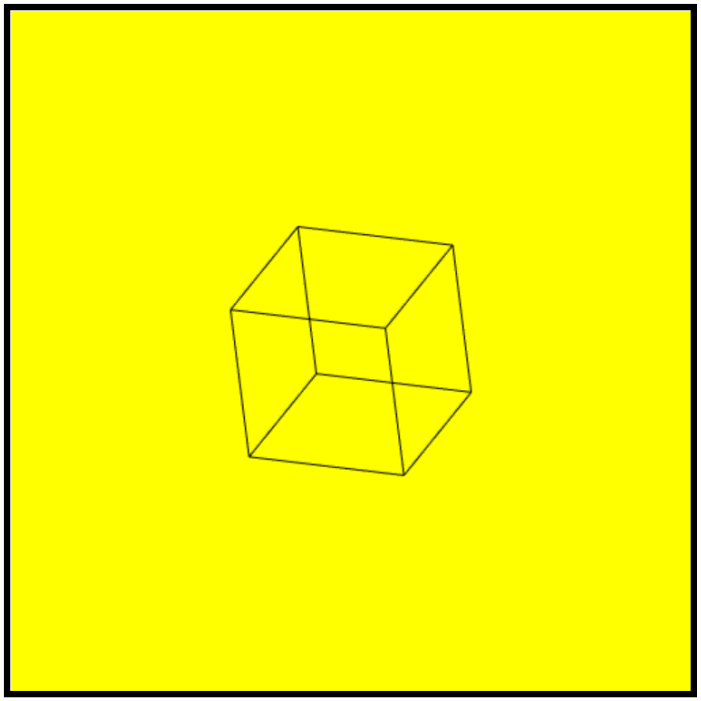
Primjer 7.2. – priprema spremnika

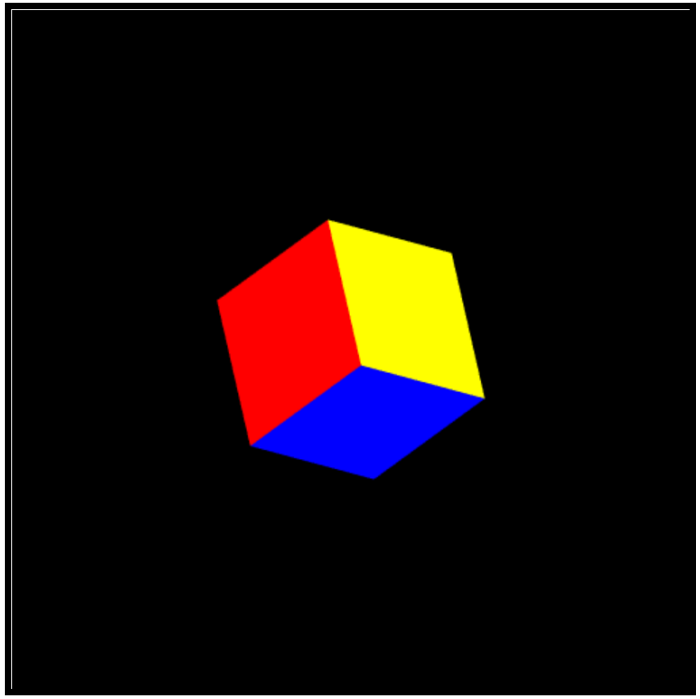
- iz `RG-primjer7-2-bojanje-vrhova.html`

```
gl.vertexAttribPointer(GPUprogram1.a_vrhXY,
  2, gl.FLOAT, false, 20, 0);

gl.vertexAttribPointer(GPUprogram1.a_boja,
  3, gl.FLOAT, false, 20, 8);

gl.bufferData(gl.ARRAY_BUFFER, new
  Float32Array(vrhovi.flat()),
  gl.STATIC_DRAW);
```

WebGL & OpenGL ES

Uvod u 3D





Podrška za 3D

- u WebGL-u (odnosno OpenGL ES), ključna podrška za realizaciju interaktivne 3D grafike su ugrađeni tipovi `vec4` i `mat4` koji su predviđeni za rad s homogenim koordinatama
- realizacija 3D geometrijskih transformacija u program za sjenčanje vrhova tako postaje gotovo trivijalna:

```
#version 300 es
in vec4 a_vrhXYZ;
uniform mat4 u_mTrans;

void main() {
  gl_Position = u_mTrans * a_vrhXYZ;
}
```

Podrška za homogene koordinate

- zadaća programera je pripremiti opis objekata (tj. vrhove i pripadajuće podatke kao što su boja, normale ili tekstura) i odgovarajuću matricu transformacije
- u homogenim koordinatama četvrta (.w) komponenta obavezno je 1 i uzima se kao predefinirana vrijednost pa je u program za sjenčanje vrhova dovoljno slati samo 3 koordinate (iako je vektor u programu za sjenčanje definiran s vec4!)

```
var vrhovi = [0, 0, 0, … ];
gl.vertexAttribPointer(program.a_vrhXYZ, 3, …);
```

Uklanjanje nevidljivih ploha

- kod 3D grafike realizirane uz pomoć ploha najveći je problem detektirati koje su plohe ispred, a koje iza
- predefinirano je da se svaka sljedeća ploha iscrtava preko svih prethodnih, dakle morali bi stalno podešavati poredak iscrtavanja ovisno o tome kako se objekti transformiraju, ili alternativno, imati algoritme koji bi određivali koje plohe ne treba iscrtavati
- izuzetno složen problem koji se najlakše rješava uz pomoć spremnika dubine (engl. *depth buffer*, *z-buffer*)

Spremnik dubine

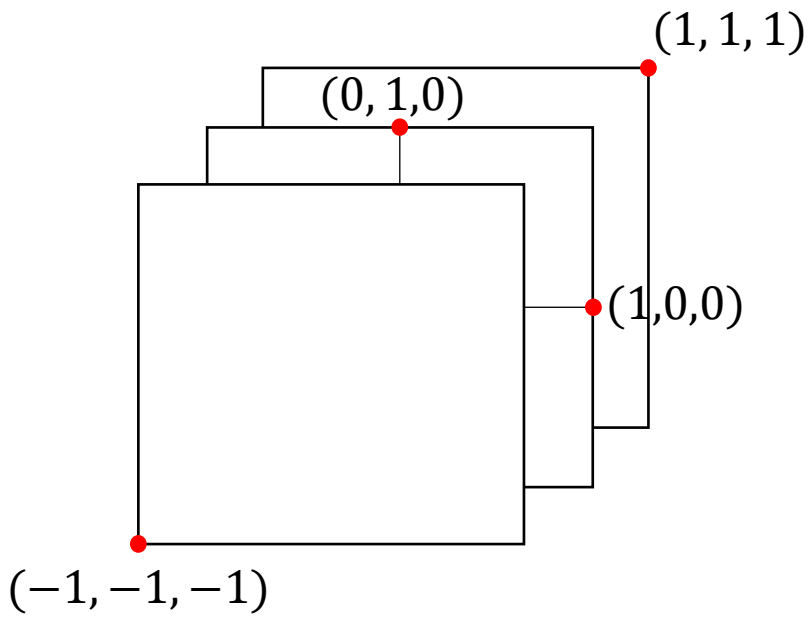
- OpenGL ES podržava spremnik dubine i njegovo korištenje je vrlo jednostavno
- spremnik dubine aktivira se samo jednom pozivom metode `gl.enable(gl.DEPTH_TEST);`
- slično kao sa spremnikom boje treba ga očistiti svaki puta prije iscrtavanja scene

  ```
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
  ```

- na kraju će kao piksel biti iscrtan fragment s najmanjom z koordinatom!

Normirane koordinate

- sve što u normiranim koordinatama izlazi izvan raspona $[-1, 1]$ je odrezano, tj. ne iscrtava se (vrijedi i za z koordinatu!)



- spremnik dubine manje vrijednosti z koordinate interpretira kao bliže, a veće z koordinate kao dalje

Selektivno odbacivanje

- selektivno odbacivanje (engl. *culling*) metoda je koja koristi orijentaciju trokuta za odbacivanje (neiscrtavanje)
- ako je smjer obilaženja vrhova trokuta pozitivan (tj. suprotan smjeru kazaljke na satu) smatra se da gledamo prednju stranu trokuta – i obrnuto, ako je u smjeru kazaljke na satu smatramo da gledamo stražnju stranu trokuta

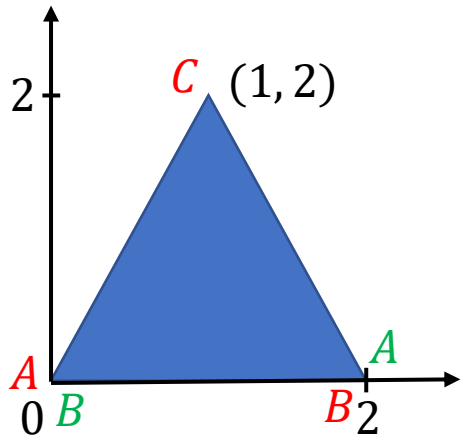
Selektivno odbacivanje

- ključno svojstvo je da ako je trokut na površini nekog geometrijskog tijela pozitivne orijentacije, tj. najprije vidimo njegovu prednju stranu, kad se tijelo zarotira za 180° orijentacije će „automatski” biti negativna!
- u tom slučaju najčešće nema potrebe iscrtavati taj trokut
- selektivno odbacivanje uključuje se pozivom metode `gl.enable(gl.CULL_FACE);`
- predefinirana vrijednost je da se odbacuju trokuti koji pokazuju stražnju stranu (`gl.BACK`)
- to je moguće promijeniti s `gl.cullFace(gl.FRONT);`

Selektivno odbacivanje

- kako računalo „zna” koji je smjer obilaženja?
- primjer: *površina trokuta*

$$P = \frac{1}{2}[x_A(y_B - y_C) + x_B(y_C - y_A) + x_C(y_A - y_B)]$$



$A(0,0) \quad B(2,0) \quad C(1,2)$

$$P = \frac{1}{2}[0(0-2) + 2(2-0) + 1(0-0)] = 2$$

$A(2,0) \quad B(0,0) \quad C(1,2)$

$$P = \frac{1}{2}[2(0-2) + 0(2-0) + 1(0-0)] = -2$$