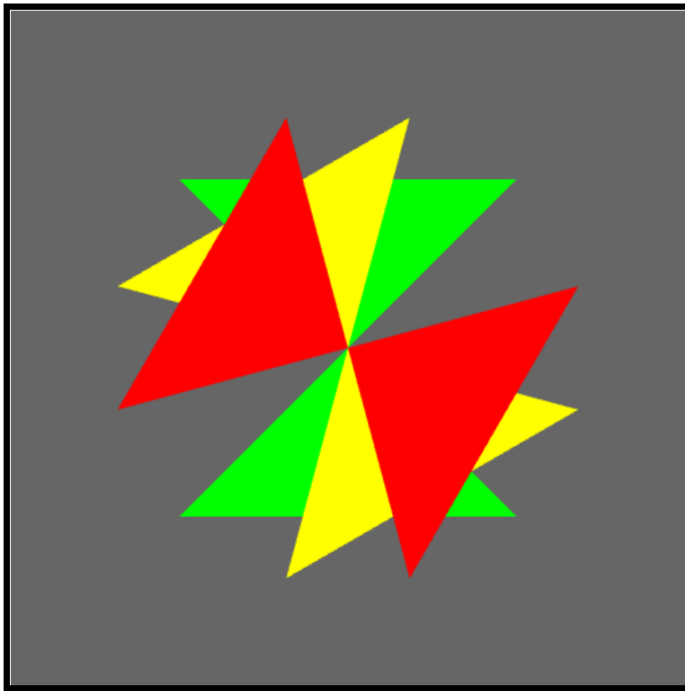


# Programiranje procesora za sjenčanje vrhova i fragmenata



1. dio:  
Geometrijska  
faza

# Iscrtavanje u stvarnom vremenu

- Za interaktivnu 3D grafiku potrebno je iscrtavati nekoliko desetaka sličica u sekundi
- Jasno je da taj proces treba maksimalno optimirati
- Tri osnovne faze kod iscrtavanja:
  - Aplikacijska faza
  - Geometrijska faza
  - Faza rasteriziranja

# Hardverska akceleracija

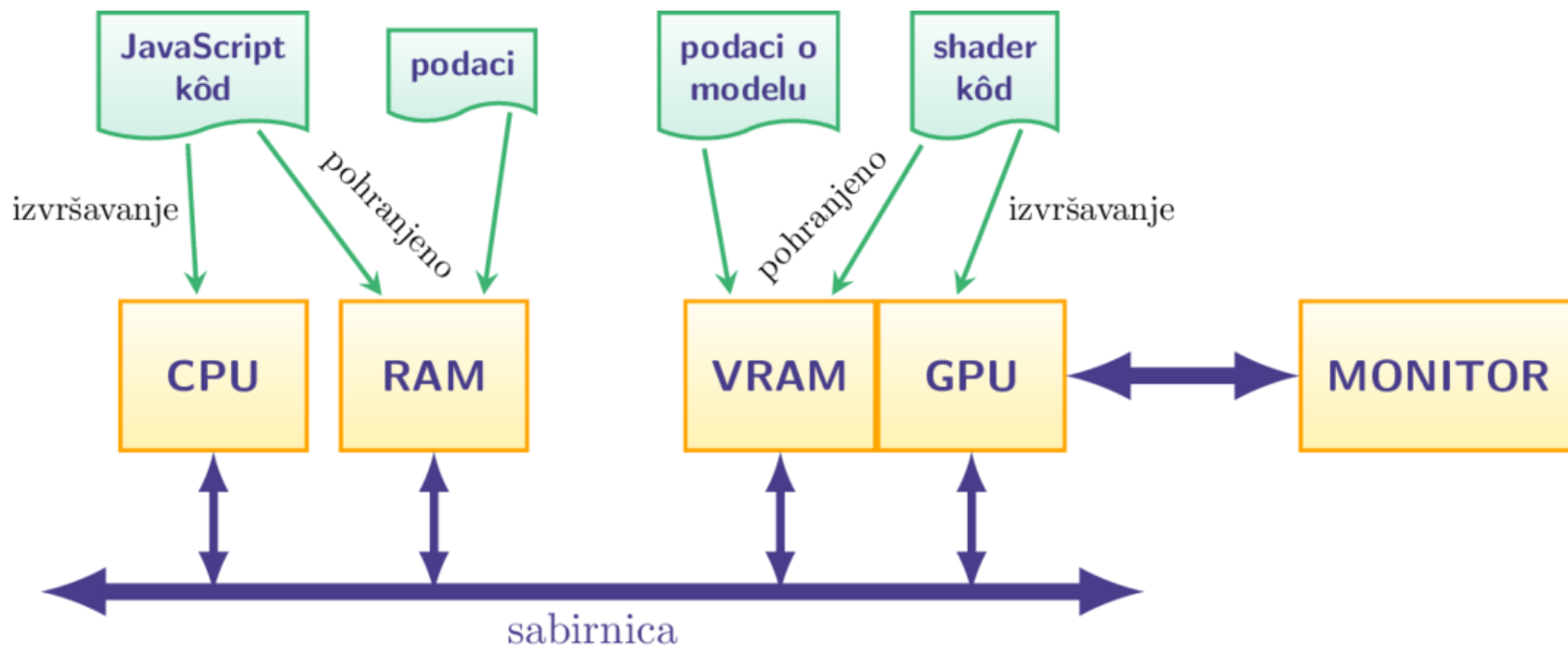
- Geometrijska faza i faza rasterizacije su šablonizirane i stoga povoljne za implementaciju u hardveru
- Operacije po fazama izvode se kao na pokretnoj traci pa se govori o **grafičkom protočnom sustavu** (engl. *graphics pipeline*)
- Dio operacija izvodi se neovisno na svakom verteksu i fragmentu što omogućuje visok stupanj paralelizacije
- **Grafički procesor** (*GPU – Graphics Processor Unit*)

# Programabilni grafički protočni sustav

- Korisnik može direktno programirati neke dijelove grafičkog protočnog sustava, konkretno **procesor vrhova** (engl. *vertex shader*) i **procesor fragmenata** (engl. *fragment shader*)
- U početku je to bilo moguće samo strojnim jezikom no u standard OpenGL 2.0 (2004. godine) uključen je i **GLSL** (Open**GL** Shader Language) sa sintaksom sličnom programskom jeziku C
- Ekvivalent u Direct3D: **HLSL** (High-Level Shading Language) nastao iz nVidijinog Cg (**C** for graphics)

# WebGL

- Razvoj OpenGL-a išao je do verzije 4.6 (2017. godine)
- OpenGL ES je podskup OpenGL-a za ugradbene sustave (engl. *embedded systems*) – verzija 1.0 objavljena 2003.
- Verzija 2.0 (2007.) uvodi programe za sjenčanje (GLSL)
- Verzija 3.0 (2012.) postaje baza WebGL-a 2.0
- WebGL je JavaScript programsko sučelje za 2D i 3D grafiku koje omogućuje korištenje GPU iz web preglednika – verzija 1.0 je iz 2011., a verzija 2.0 iz 2017.
- WebGL 2.0 aplikacija = JavaScript kod koji poziva OpenGL ES 3.0 rutine + programi za sjenčanje u GLSL-u



# Programi za sjenčanje u OpenGL-u

- Pišu se u GLSL-u u sintaksi vrlo sličnoj programskom jeziku C i **prevode se tek po pokretanju OpenGL programa** – na taj način izvlači se maksimum iz dostupnog hardvera i raspoloživih upravljačkih programa (engl. *drivera*)
- Zadaju se kao stringovi i korisnik je unutar OpenGL programa odgovoran za njihovo prevođenje (*compile*) i povezivanje (*link*) u program za sjenčanje koji se potom svaki puta izvršava kod iscrtavanja

# Program za sjenčanje vrhova

- Programski segment koji se izvodi na procesoru vrhova i to **za svaki pojedini vrh!**
- U njemu treba obaviti geometrijske transformacije i proračune vezane uz model osvjetljavanja
- Na kraju, obavezno je dodijeliti vrijednost ugrađenoj (engl. *built-in*) varijabli **gl\_Position** u koju treba spremiti konačnu poziciju vrha u normiranim koordinatama (x i y se iscrtavaju, a z je vrijednost koja se koristi u spremniku dubine)



# Primjer 6.1. – vertex shader

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

```
#version 300 es
in vec2 a_vrhXY;

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

# Vektori i matrice

- GLSL podržava posebne tipove podataka prilagođene 3D računalnoj grafici i registrima grafičkog procesora:
  - vec2, vec3, vec4** – vektor sa 2, 3 ili 4 komponente
  - mat2, mat3, mat4** – 2 x 2, 3 x 3 ili 4 x 4 matrica
- Podržava i odgovarajuće operacije, množenje matrica ili množenje vektora matricom

# Primjer 6.2. – vertex shader

- iz RG-primjer6-2-uniform-varijable.html

```
#version 300 es
in vec2 a_vrhXY;
uniform mat2 u_mTrans;

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

# 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

# uniform variable

- u programima za sjenčanje deklariraju se kao globalne varijable i oznaka **uniform** ide ispred tipa varijable
- dozvoljeni su praktički svi tipovi podataka koji postoje u programima za sjenčanje, na primjer:

```
uniform vec4 u_boja;
```

- posebno često i korisno je prenijeti matricu transformacije, na primjer:

```
uniform mat3 u_mTrans;
```

# Inicijalizacija u glavnom programu

- u glavnom programu trebamo lokaciju **uniform** varijable u programu za sjenčanje, na primjer:

```
GPUprogram1.u_mTrans =  
    gl.getUniformLocation(GPUProgram1, "u_mTrans");
```

- metodu **getUniformLocation** dovoljno je pozvati samo jednom pa to treba učiniti tijekom inicijalizacije

# Dodjeljivanje vrijednosti

- vrijednost **uniform** varijabli treba postaviti prije pozivanja metoda za crtanje (**drawArrays** ili **drawElements**)
- tijekom izvođenja metoda za crtanje, dakle u programima za sjenčanje, vrijednost **uniform** varijabli je konstantna (ista je za sve vrhove i fragmente!)
- međutim, slobodno se može mijenjati između poziva rutina za crtanje

# Dodjeljivanje vrijednosti

- različitim tipovima podataka prilagođene su različite metode za dodjeljivanje vrijednosti **uniform** varijablama, na primjer za tip podataka `vec4`:

```
gl.uniform4fv(GPUprogram1.u_boja, [1.0, 1.0, 0.0, 1.0]);
```

(uočite da nije potrebna konverzija u `Float32`)

- ili za tip podataka `mat3`:

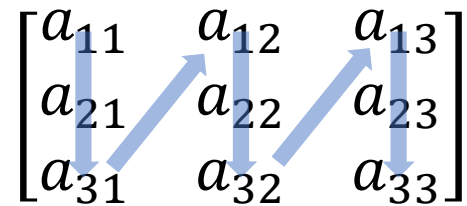
```
gl.uniformMatrix3fv(GPUprogram1.u_mTrans, false,  
                    [1, 0, 0, 0, 1, 0, 0, 0, 1]); // jedinična matrica
```

(uočite da je matrica ispisana kao lista, tj. jednodimenzionalno polje)



# Matrica kao lista

- kod pretvaranja matrice u listu koristi se konvencija da se ispisuje stupac po stupac (engl. *column major order*)



A 3x3 matrix is shown with elements  $a_{11}, a_{12}, a_{13}$  in the first row,  $a_{21}, a_{22}, a_{23}$  in the second row, and  $a_{31}, a_{32}, a_{33}$  in the third row. Blue arrows indicate the traversal order for column-major storage: a vertical arrow from  $a_{11}$  to  $a_{21}$  to  $a_{31}$ , a diagonal arrow from  $a_{31}$  to  $a_{12}$ , a vertical arrow from  $a_{12}$  to  $a_{22}$  to  $a_{32}$ , a diagonal arrow from  $a_{32}$  to  $a_{13}$ , and a vertical arrow from  $a_{13}$  to  $a_{23}$  to  $a_{33}$ .

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$\Rightarrow [a_{11}, a_{21}, a_{31}, a_{12}, a_{22}, a_{32}, a_{13}, a_{23}, a_{33}]$$

# attribute variable

- u programu za sjenčanje vrhova deklarira se oznakom **in** ispred tipa varijable
- pomoću njih prenose se podaci o vrhovima, na primjer koordinate:

```
in vec2 a_vrhXY;
```

- ali mogu se prenositi i boje pojedinih vrhova ili vektori normale:

```
in float a_boja;
```

```
in vec3 a_normala;
```

# Inicijalizacija u glavnom programu

- u glavnom programu trebamo lokaciju **attribute** varijable u programu za sjenčanje, na primjer:

```
GPUprogram1.a_vrhXY =  
    gl.getAttributeLocation(GPUProgram1, "a_vrhXY");
```

- metodu **getAttributeLocation** dovoljno je pozvati samo jednom

# Dodjeljivanje vrijednosti

- vrijednost **attribute** varijabli treba pohraniti u odgovarajući spremnik (engl. *buffer*) prije pozivanja metoda za crtanje **drawArrays** ili **drawElements**
- sadržaj spremnika prenosi se u memoriju GPU
- tijekom izvođenja metoda za crtanje, program za sjenčanje vrhova poziva se onoliko puta koliko ima pohranjenih podataka o vrhovima u spremniku
- u programu za sjenčanje vrhova, vrijednost **attribute** varijable ima kod svakog poziva drugu vrijednost!

# Punjenje spremnika

```
spremnikVrhova = gl.createBuffer();

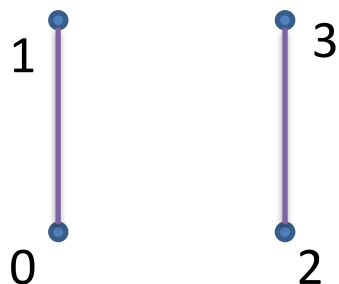
gl.bindBuffer(gl.ARRAY_BUFFER, spremnikVrhova);
gl.enableVertexAttribArray(GPUprogram1.a_vrhXY);
gl.vertexAttribPointer(GPUprogram1.a_vrhXY, 2,
    gl.FLOAT, false, 0, 0);
// punjenje spremnika - podaci koji se šalju na GPU
gl.bufferData(gl.ARRAY_BUFFER,
    new Float32Array(vrhovi), gl.STATIC_DRAW);
```

# Crtanje linija i trokuta

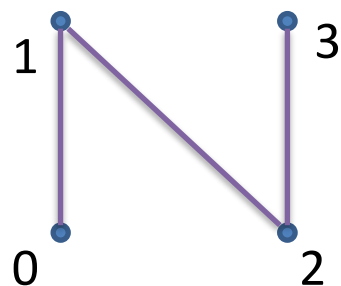
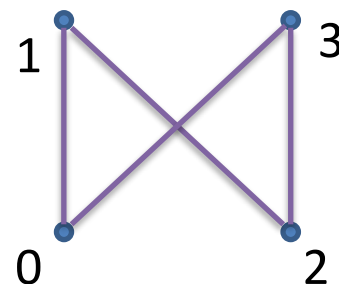
- kad su vrhovi jednom pohranjeni na GPU pomoću njih se mogu na razne načine crtati linije i trokuti
- poziva se metoda **drawArrays** i prvi parametar je konstanta koja određuje što će se iscrtati
- predefinirane konstante za crtanje linija: **LINES**, **LINE\_STRIP** i **LINE\_LOOP**
- predefinirane konstante za crtanje trokuta: **TRIANGLES**, **TRIANGLE\_FAN** i **TRIANGLE\_STRIP**

# Crtanje linija kroz vrhove

LINES



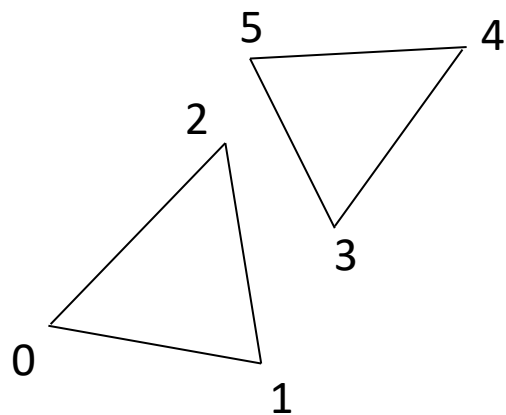
LINE\_LOOP



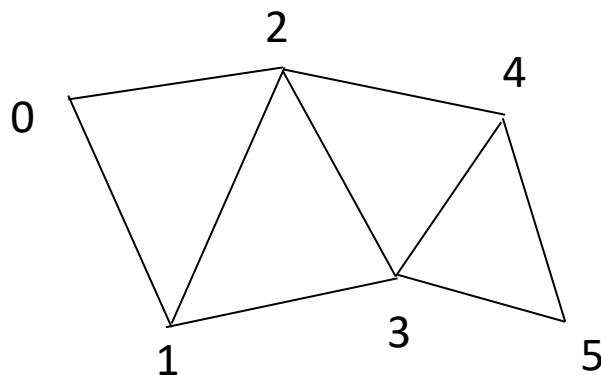
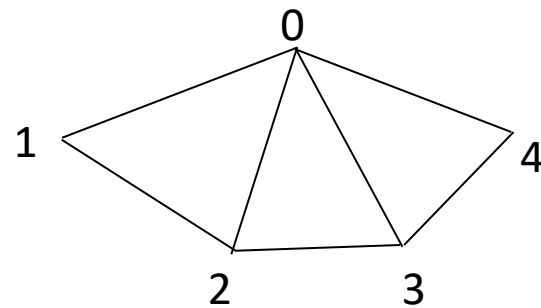
LINE\_STRIP

# Crtanje trokuta

TRIANGLES



TRIANGLE\_FAN



TRIANGLE\_STRIP



# 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-trokuta-i-linija.html

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

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

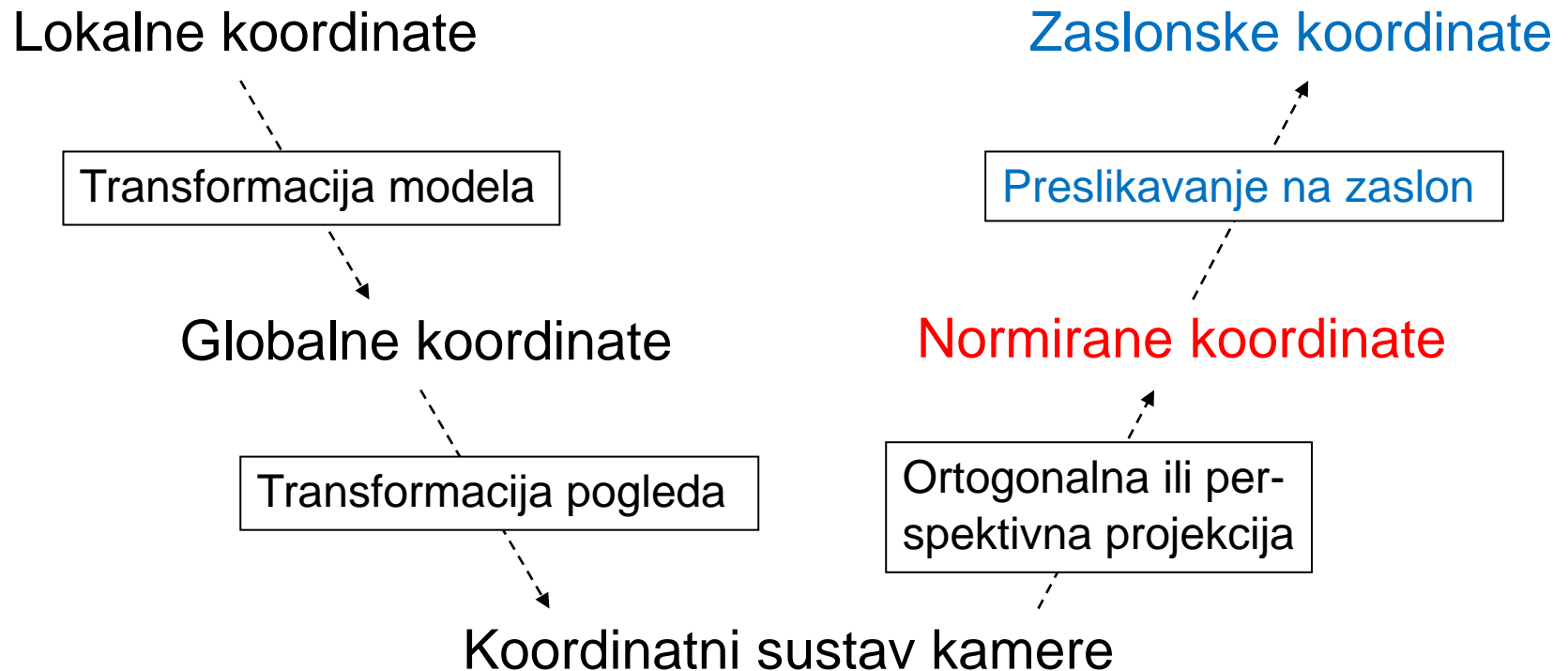
# Postavlja se boja piksela

- 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

# Geometrijska faza

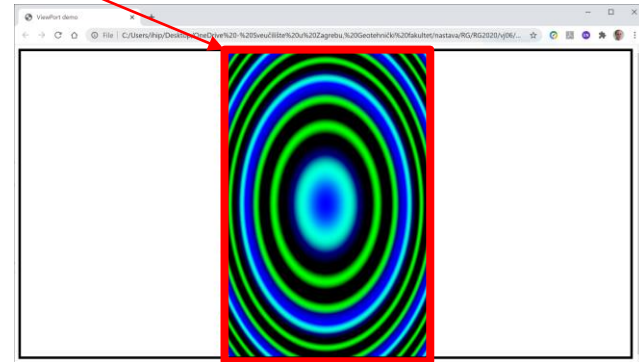
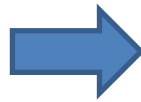
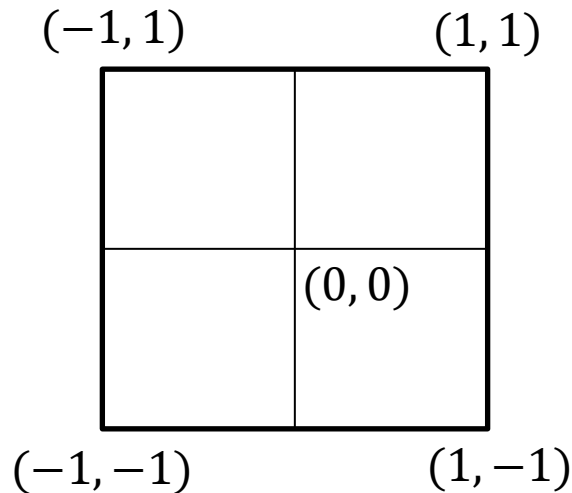
- Transformacija modela i pogleda
  - Iz lokalnih koordinata najprije u globalne pa u koordinatni sustav kamere
- Sjenčanje vrhova
  - Vrhovima se pridjeljuje boja izračunata korištenjem nekog modela osvjetljavanja
- Projekcija (ortogonalna ili perspektivna)
- Odrezivanje
- Preslikavanje na ekran

# Koordinatni sustavi i transformacije u 3D grafici - OpenGL

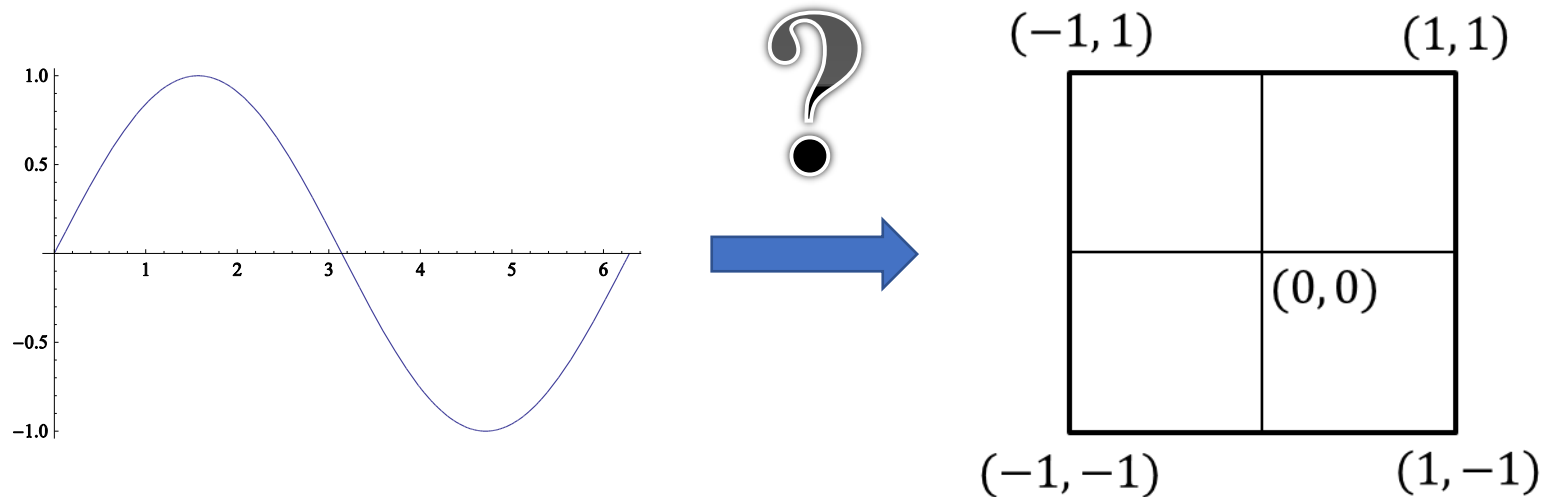


# Normirane koordinate - OpenGL

- Projekcija u normirane koordinate:
  - normirane koordinate imaju raspon od -1 do 1
  - z koordinata ima ulogu spremnika dubine
  - x i y koordinata preslikavaju se na pravokutni dio zaslona, takozvani *viewport*

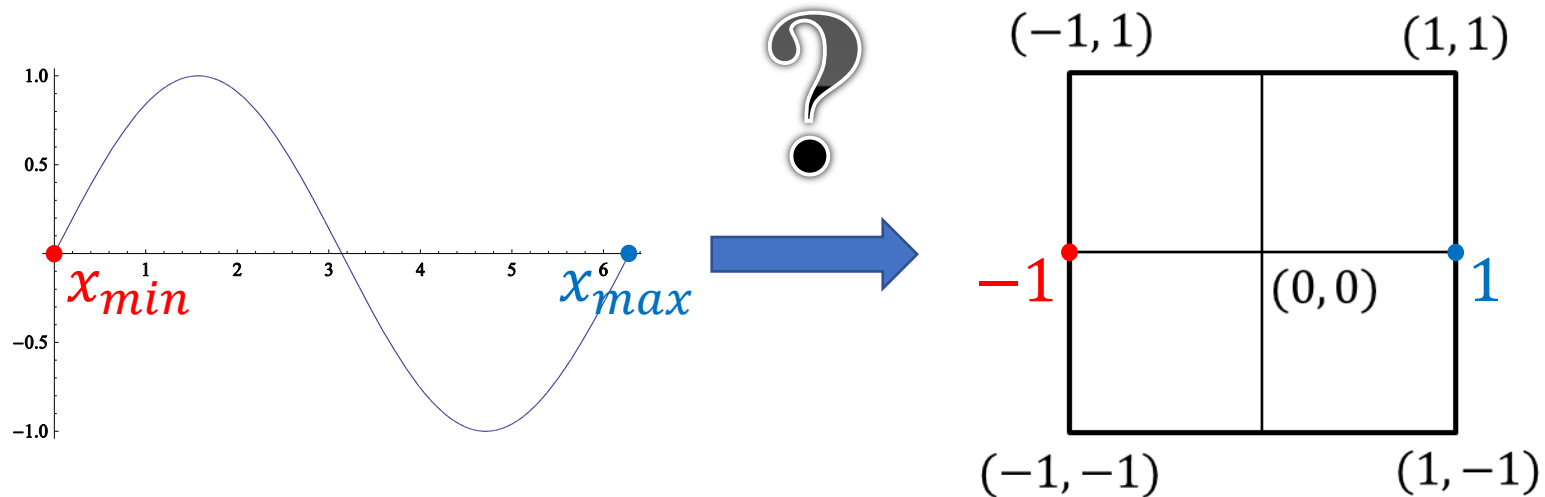


# Projekcija u normirane koordinate



- kako ćemo proizvoljni raspon koordinata od  $x_{min}$  do  $x_{max}$  i od  $y_{min}$  do  $y_{max}$  projicirati u normirane koordinate?

# Projekcija u normirane koordinate



- moramo preslikati  $x_{min} \rightarrow -1$  i  $x_{max} \rightarrow 1$ , dakle:

$$-1 = s_x x_{min} + p_x \quad (1)$$

$$1 = s_x x_{max} + p_x \quad (2)$$



# Projekcija u normirane koordinate

- ovo je sustav od dvije jednadžbe s dvije nepoznanice  $s_x$  i  $p_x$

$$-1 = s_x x_{min} + p_x \quad (1)$$

$$1 = s_x x_{max} + p_x \quad (2)$$

- ako od (2) oduzmemo (1) slijedi izraz za faktor skaliranja  $s_x$

$$1 - (-1) = s_x x_{max} + p_x - (s_x x_{min} + p_x)$$

$$2 = s_x (x_{max} - x_{min})$$

$$s_x = \frac{2}{x_{max} - x_{min}} \quad (3)$$

# Projekcija u normirane koordinate

- pomak  $p_x$  može se izraziti iz (1) ili (2)

$$p_x = -s_x \textcolor{red}{x}_{min} - 1$$

$$p_x = -s_x \textcolor{blue}{x}_{max} + 1$$

- isti izvod vrijedi i za  $y$  koordinatu pa slijedi

$$s_y = \frac{2}{\textcolor{blue}{y}_{max} - \textcolor{red}{y}_{min}}$$

$$p_y = -s_y \textcolor{red}{y}_{min} - 1$$

$$p_y = -s_y \textcolor{blue}{y}_{max} + 1$$