GRAFIČNI CEVOVOD

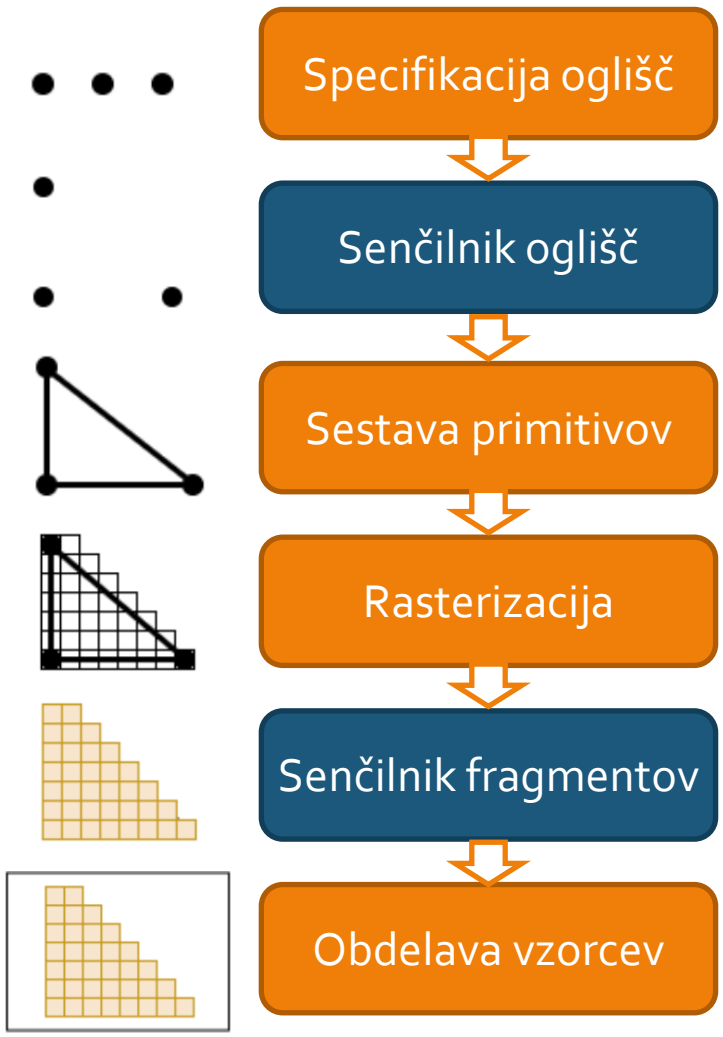
Od modela do slike (v realnem času)

- Grafični cevovod
  - pretvori (3D) predmet/sceno iz računalniškega zapisa v bitno sliko
- Sestavljen je iz več faz, ki preslikajo 3D predmete v bitno sliko (sestavljeno iz pikslov)
- Tipično je implementiran v strojni opremi
  - grafična kartica

```
v 2.712726 -2.398764 -2.492640
v 2.712726 -1.954302 -2.665440
v -5.975275 -1.954302 -2.665440
v -5.975275 -2.398764 -2.492640
v -6.113514 -1.885536 -2.803680
v 2.712726 -1.885536 -2.803680
v -5.975275 -1.372307 -2.803680
v -5.975275 -1.816770 -2.700000
v 2.712726 -1.816770 -2.700000
v 2.712726 -1.372307 -2.803680
v 4.766168 -2.256987 -2.354400
v 4.766168 -1.372307 -2.665439
```

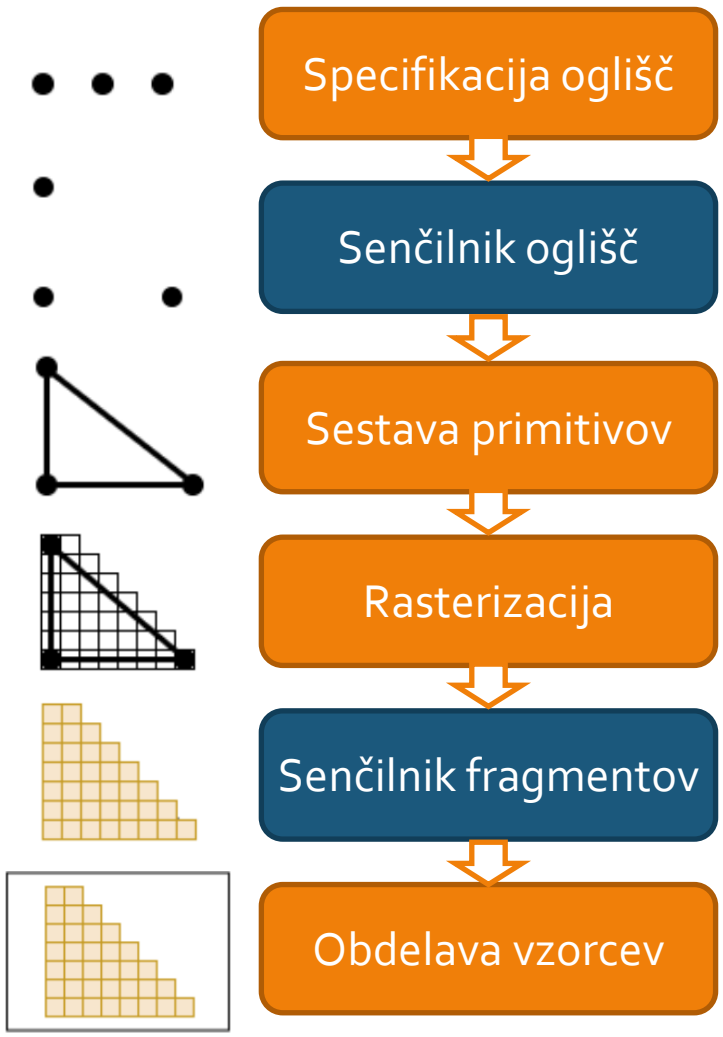
Grafični cevovod

- Primitivi (poligoni) se po fazah procesirajo od vhodnih podatkov do izhodne slike
  - vsaka faza posreduje rezultate naslednji fazi
- Cevovod lahko predstavimo na različne načine
  - to je WebGPU pogled
- Določene faze so lahko implementirane strojno, druge programsko (senčilniki / *shaders*)



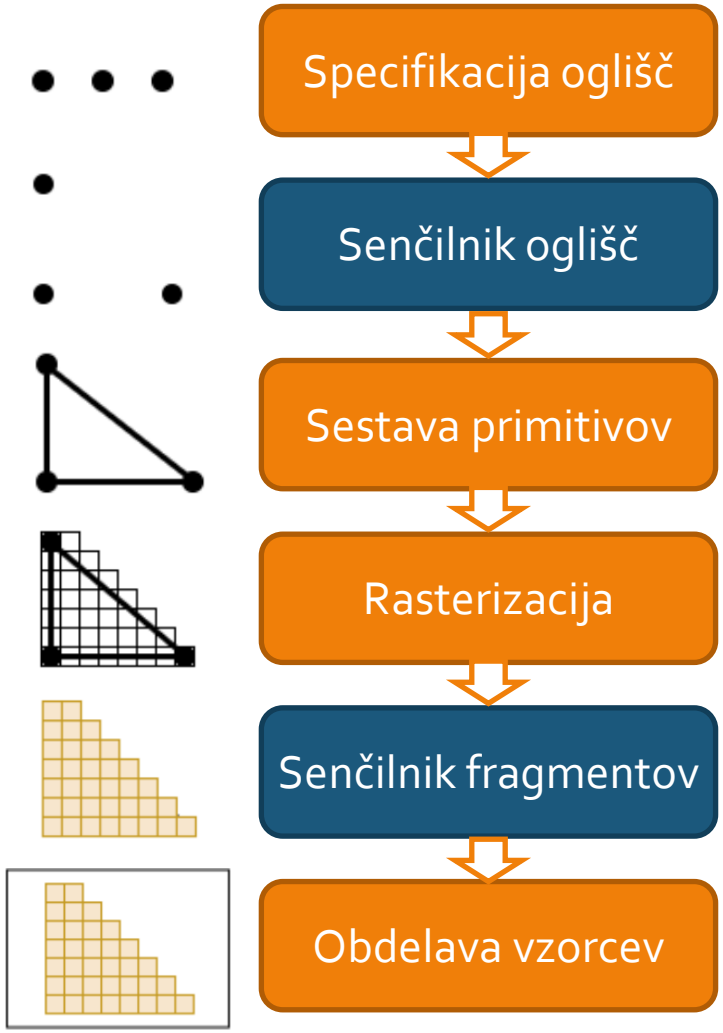
Specifikacija oglišč

- Priprava podatkov za upodabljanje
  - *Vertex Buffer* (oglišča)
  - *Index Buffer* (kako se povežejo)
  - *Render Buffers* (kam zapisujemo rezultate)
  - *Shaders* (programska koda in uniforme)



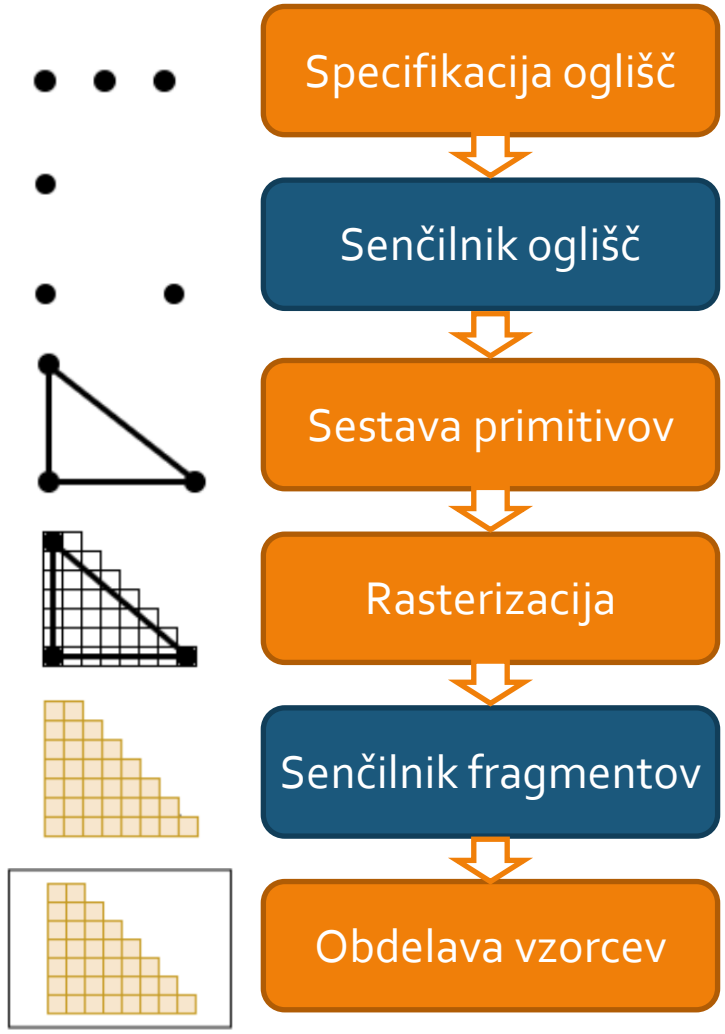
Senčilnik oglišč

- Program, ki se izvede za vsako posamezno oglišče
  - specificira ga uporabnik, je obvezen
- Tipično **transformacije** oglišč in normal, projekcija
  - lahko tudi osvetlitev itn.
- Atribute oglišč le **spreminjamo** (ne dodajamo novih oglišč ali jih brišemo)
  - vhod je posamezno oglišče z vsemi atributi
  - izhod je prav tako (transformirano) oglišče z atributi, ki se kot **interpoliranke** prenesejo v senčilnik fragmentov



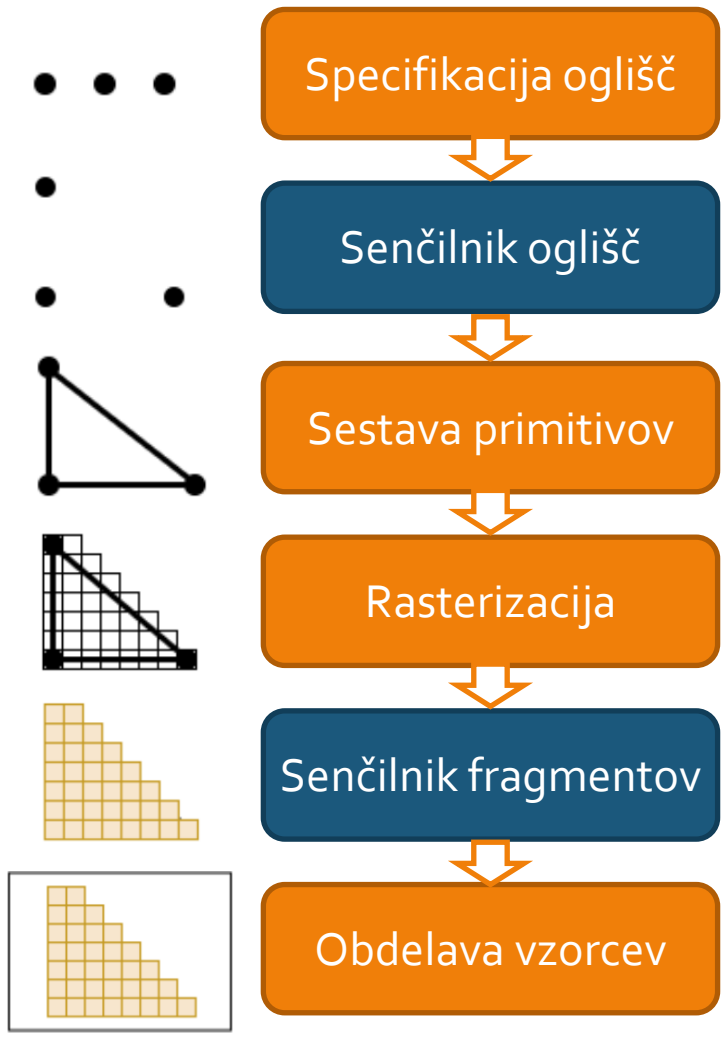
Enostaven primer – senčilnik oglišč

```
struct VertexInput {
    @location(0) position : vec3f,
    @location(1) normal : vec3f,
}

struct VertexOutput {
    @builtin(position) position : vec4f,
    @location(0) normal : vec3f,
    @location(1) vertPosition : vec4f,
}

struct ModelUniforms {
    viewModelMatrix : mat4x4f,
    projectionViewModelMatrix : mat4x4f,
    normalMatrix : mat4x4f,
}

@group(0) @binding(0) var<uniform> model : ModelUniforms;

@vertex
fn vertex(input : VertexInput) -> VertexOutput {
    var output : VertexOutput;
    output.position = model.projectionViewModelMatrix * vec4(input.position, 1);
    output.normal = (model.normalMatrix * vec4(input.normal,0)).xyz;
    output.vertPosition = model.viewModelMatrix * vec4(input.position, 1);
    return output;
}
```



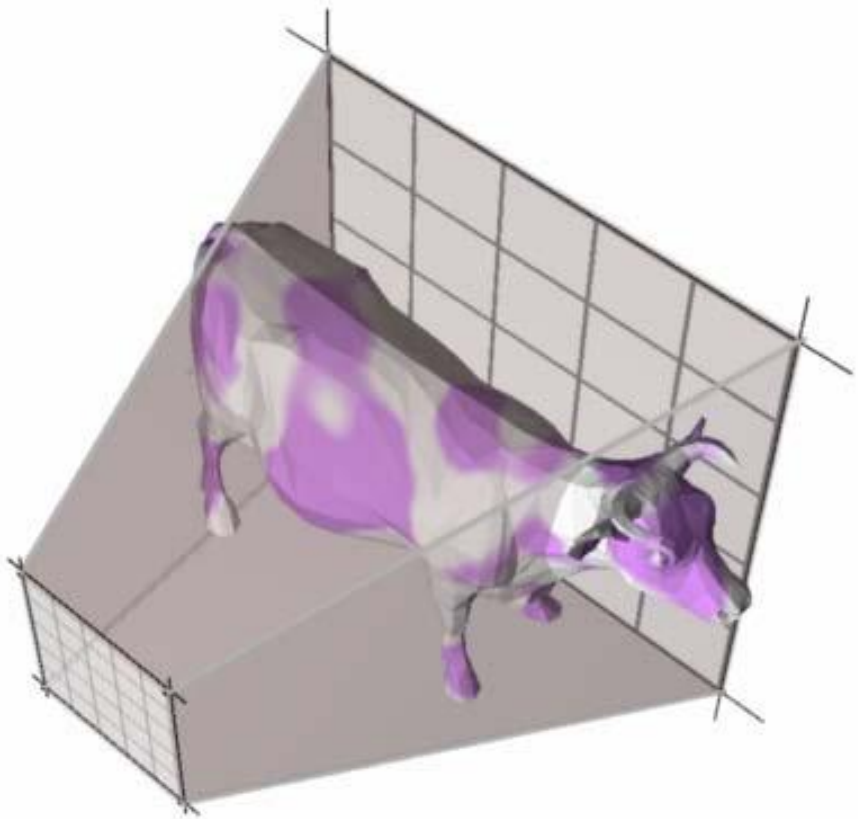
Sestava primitivov

- Oglišča se **povežejo** skupaj v posamezne primitive (trikotnike, črte …)
- **Rezanje in izločanje**
- Perspektivno **deljenje**
- Transformacija v **koordinate zaslona**
- **Izločanje ploskev**



Rezanje (*clipping*) in izločanje (culling)

- Rezanje in izločanje
  - **izločimo** (*odstranimo*) vse ploskve, ki niso v vidnem delu (prisekana piramida)
  - **porežemo** ploskve, ki so delno v, delno pa izven vidnega dela



Kaj režemo/izločimo?

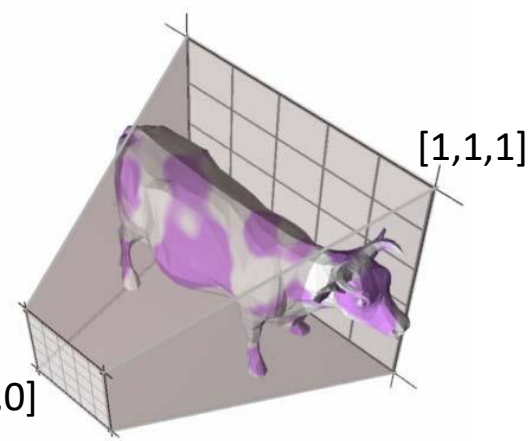
- Kako vemo kaj režemo in izločimo?
- Po projekciji, ki se zgodi v senčilniku oglišč, so oglišča v t.i. **normaliziranih koordinatah naprave**
  - x in y koordinate oglišč znotraj vidnega polja so na intervalu [-1,1]
  - z je na intervalu [0,1] (WebGPU)
- Režemo/izločimo vse kar pade izven teh vrednosti
  - zaradi hitrosti režemo/izločimo še pred perspektivnim deljenjem z z, v t.i. koordinatah rezanja (*clip coordinates*)
  - obdržimo torej oglišča:
    - $-z \leq \frac{2}{w}x \leq z$
    - $-z \leq \frac{2}{h}y \leq z$
    - $0 \leq \frac{zf}{n-f} + \frac{fn}{n-f} \leq z$

Matrika perspektivne projekcije, ki preslika točko v NDC. Pozor, v primeru gre za desnosučni k.s. (kamera gleda v negativni z, zato je d=-1).

$$M_p = \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{f}{n-f} & \frac{fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

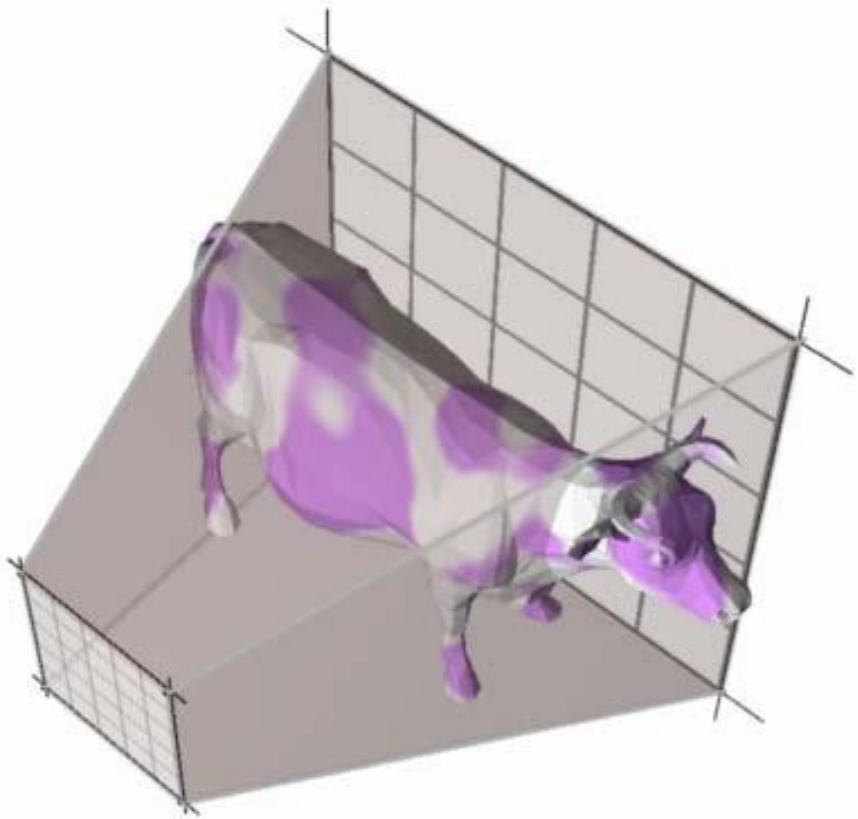
w, h: *širina in višina vidnega polja*
n, f: *bližnja in daljna ravnina*

$$x' = -\frac{1}{z}\frac{2}{w}x$$
$$y' = -\frac{1}{z}\frac{2}{h}y$$
$$z' = \frac{f}{f-n} + \frac{fn}{f-n}\frac{1}{z}$$



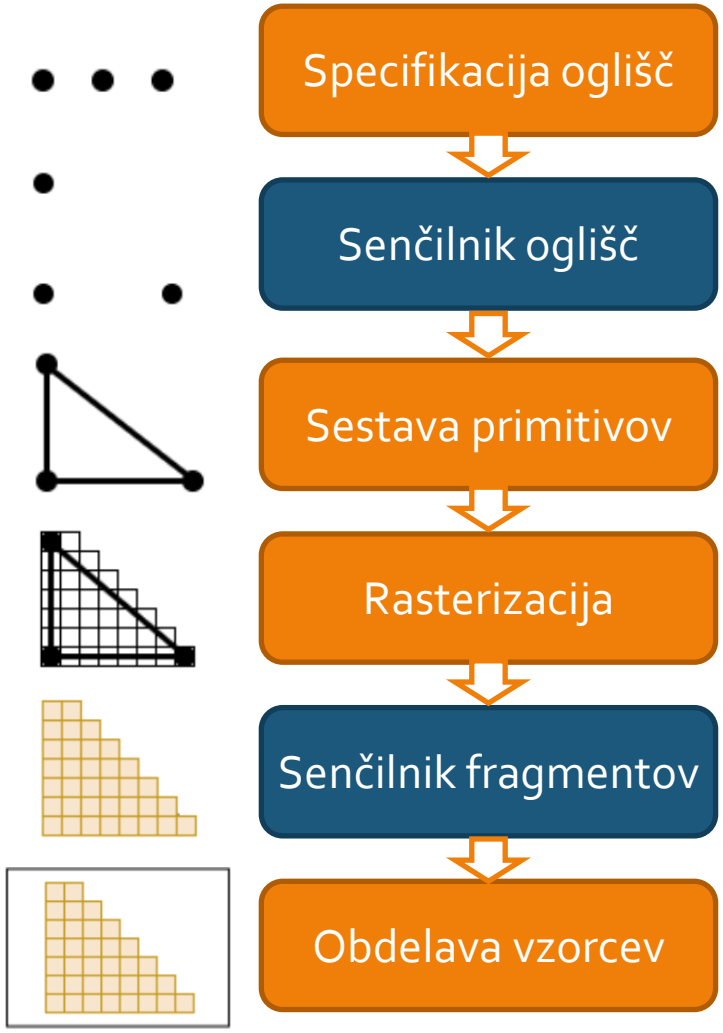
Rezanje

- Elemente, ki so delno v in delno iz vidnega polja, **porežemo**
  - v fazi sestave primitivov tipično režemo le glede na bližnjo (*near*) plane
- Točke odstranimo
- Rezanje črt
  - primer: algoritem *Cohen-Sutherland*
- Rezanje trikotnikov
  - rezultat je lahko več trikotnikov
  - primer: algoritem *Sutherland–Hodgman*



Perspektivno deljenje

- Na točkah, ki ostanejo po rezanju, izvedemo perspektivno deljenje
  - prevedemo iz koordinat rezanja v normalizirane koordinate naprave
    - $x_d = \frac{x_c}{-z}$, bo na intervalu $[-1,1]$
    - $y_d = \frac{y_c}{-z}$, bo na intervalu $[-1,1]$
    - $z_d = \frac{z_c}{-z}$, bo na intervalu $[0,1]$



Koordinate zaslona

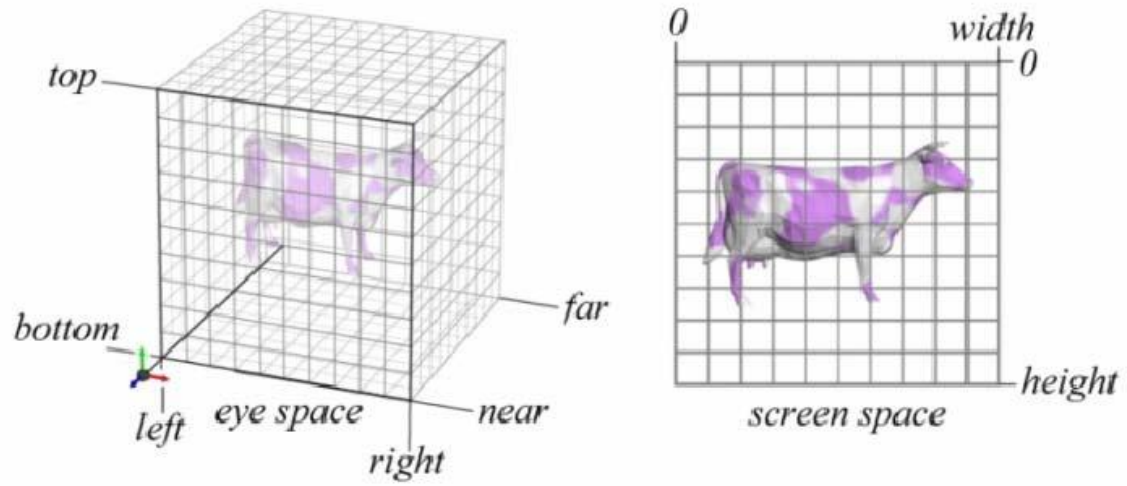
- Iz NDC pretvorimo v **koordinate zaslona**
  - *viewport (screen-space) coordinates*
  - ustrezajo ločljivosti slike, ki jo ustvarjamo (w_s, h_s)
  - pretvorba je le ustrezno skaliranje iz NDC

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{w_s}{2}(x_d+1) \\ \frac{h_s}{2}(y_d+1) \\ z_d \\ 1 \end{bmatrix}$$

$x_v \in [0, w]$

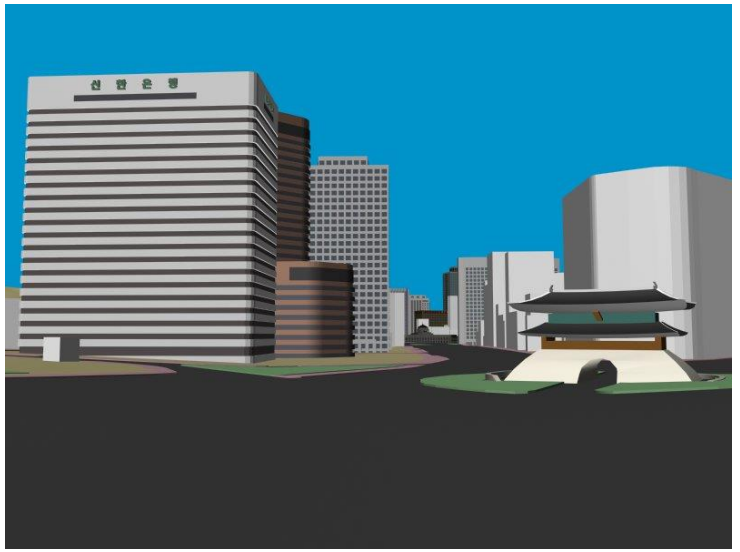
$y_v \in [0, h]$

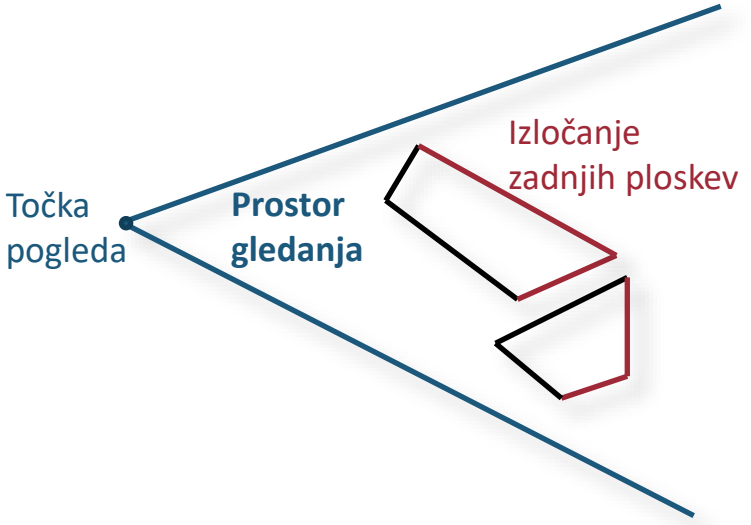
$z_v \in [0,1]$



Izločanje ploskev (*face culling*)

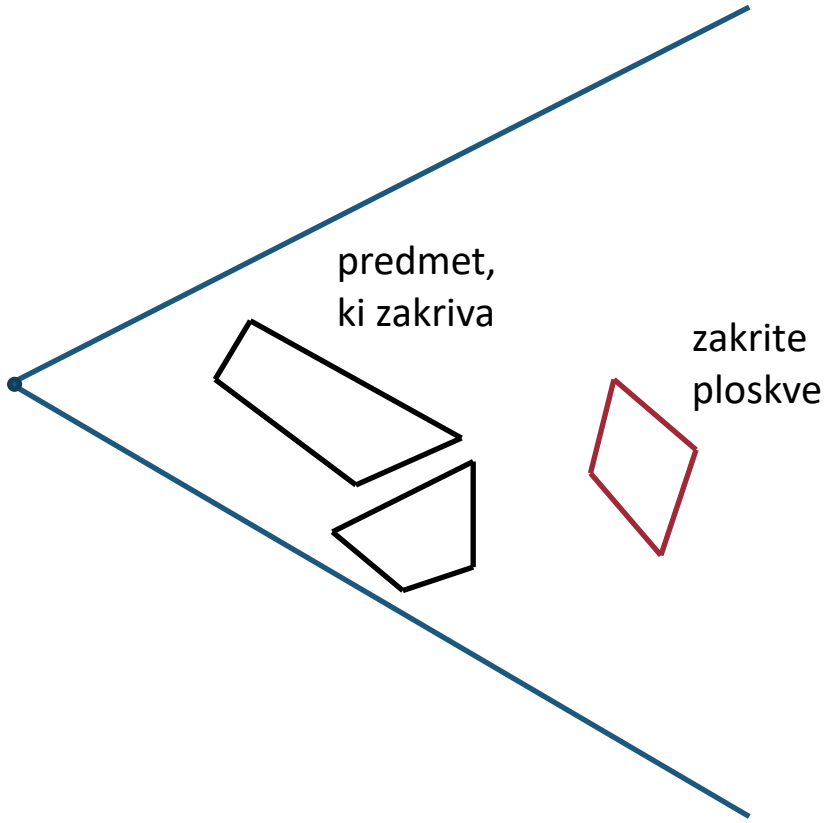
- Izločiti želimo čimveč poligonov, ki v upodobitvi ne bodo vidni
- Izločanje ploskev
  - poligone obravnavamo **enostransko** – kot da so vidni le z ene strani
  - poligon izločimo, če ga gledamo z druge strani (prednje ali zadnje)
  - lahko izločimo cca. 50% poligonov v prostoru gledanja
- Izločanje **zadnjih ploskev** (*backface culling*) - če proti kameri gleda zadnja stran poligona, ga izločimo
  - "zaprti" predmeti, v katere ne bomo vstopili
- Izločanje je enostavno
  - gleda se **vrstni red oglišč** pri izrisu trikotnika
    - prednja stran je nasprotna smeri urinega kazalca (CCW)
  - *primitive.cullMode = none/back/front*





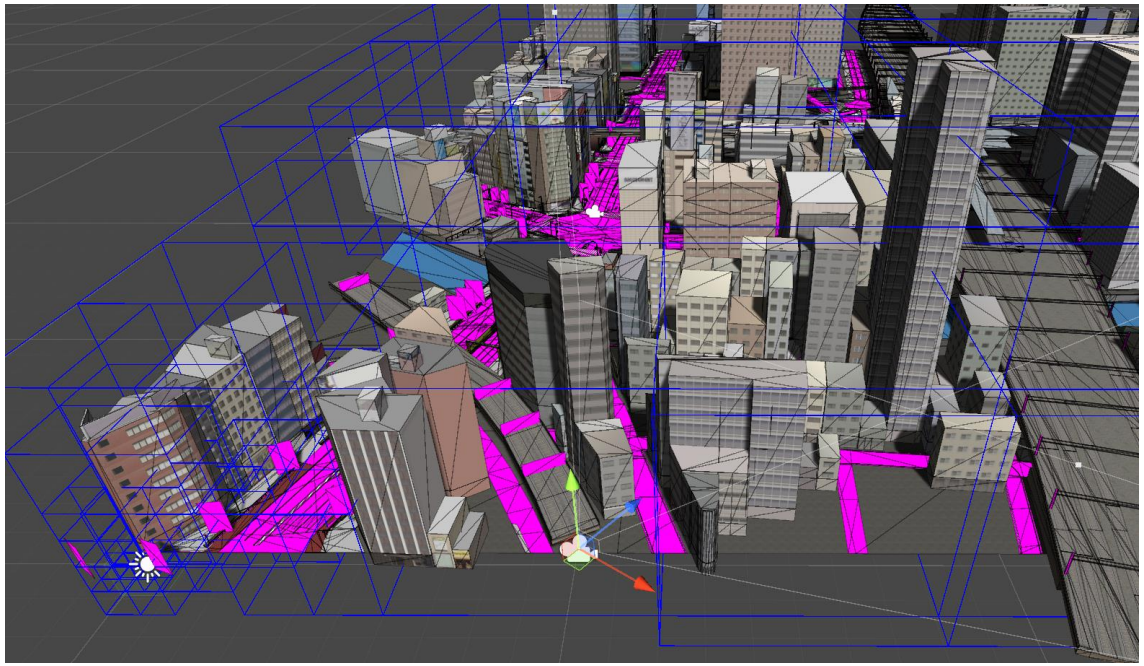
Izločanje zakritih ploskev

- *Occlusion culling, Z culling*
- Veliko različnih algoritmov
  - nekateri za specifične domene (npr. sprehajanje po sobah)
  - nekateri potrebujejo veliko predprocesiranja
- Koristno predvsem v primerih, ko je velik del scene zakrit, npr. sprehod skozi mesto, premikanje po sobah ipd.



Primer: Izločanje zakritih ploskev in Unity

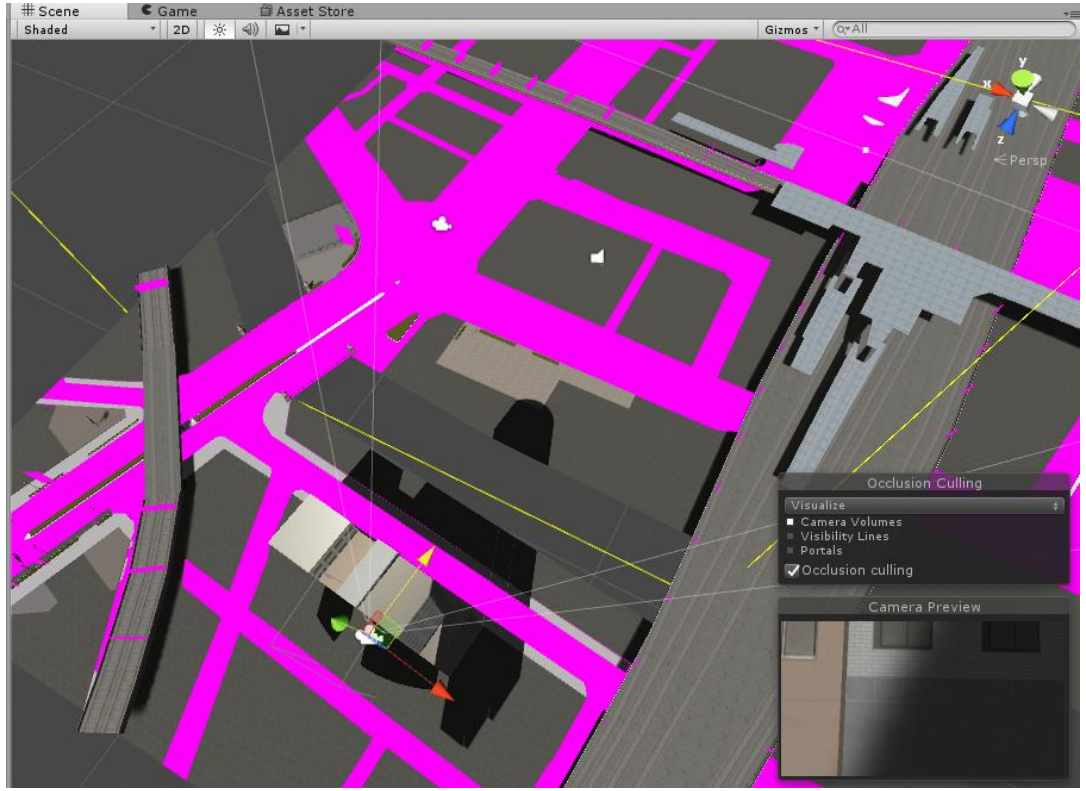
- Primer implementacije v Unity 3D
- Določimo kaj so **ploskve**, ki **zakrivajo**
  - *occluders, occludees*
  - lahko le statični predmeti
- Sceno Unity razdeli v celice in v celicah preračuna **globino** tega kar kamera vidi
  - določimo ločljivost
    - najmanjša velikost predmeta ki zakriva
    - najmanjša luknja med predmeti
- Podatki se **zapečejo** (*bake*)
  - če se scena spreminja, jih je potrebno osvežiti



Scena razdeljena na celice

Izločanje zakritih ploskev in Unity

- Glede na položaj kamere se izrišejo samo predmeti z globino manjšo od predmetov, ki zakrivajo
  - primerja se glede na izračunane celice
  - postopek ni popoln
    - če kaj preveč pošljemo v izris, ni hudo
    - predmet lahko tudi izgine, napačno nastavimo parametre izračuna celic



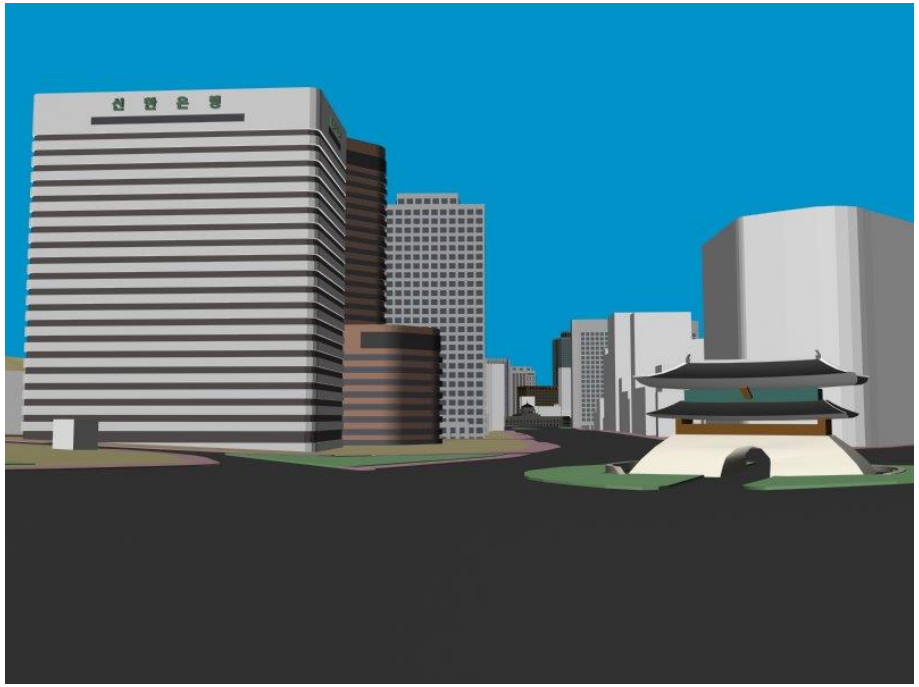
Mesto z izločanjem zakritih ploskev (večina stavb ni izrisanih)

Izločanje predmetov izven prostora pogleda

- *View frustum culling*
- Če je scena **velika** in vidimo le **majhen** del, je smiselno, da čimvec geometrije izločimo
- Izločimo predmete, ki so **izven prostora gledanja**
  - **preden** geometrijo pošljemo v cevovod

Izločanje predmetov izven prostora gledanja

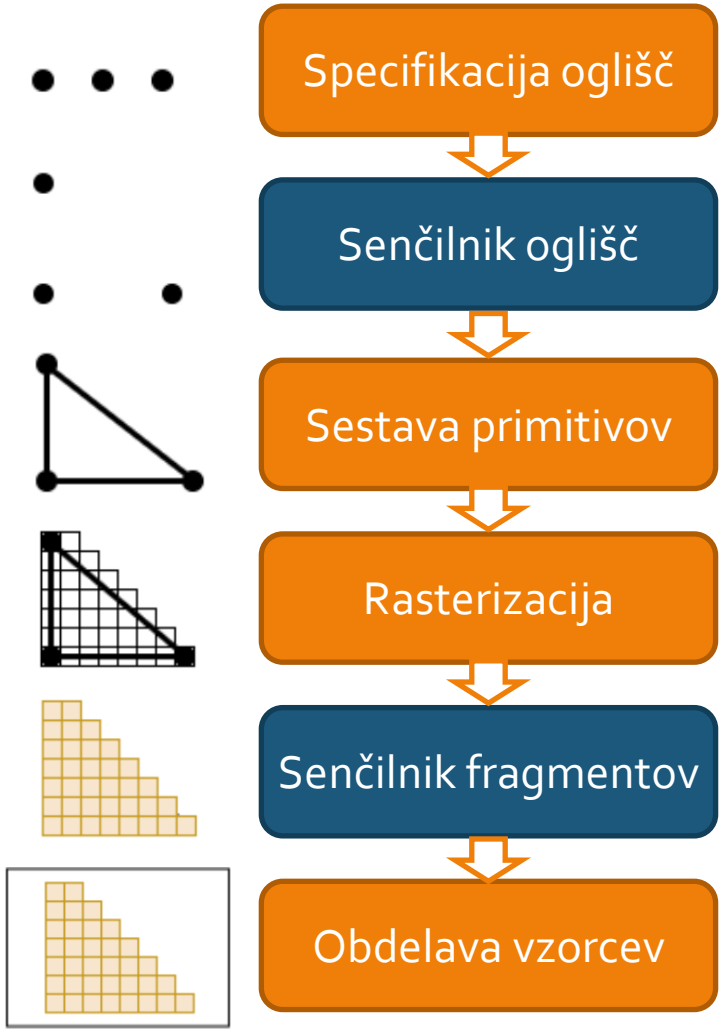
- Predmeti imajo **kompleksno geometrijo**
  - preverjanje vsakega poligona bi bilo prepočasno
  - za **hitro** izločanje uporabimo **tehnike razdelitve prostora** *(glej ustrezno poglavje)*
    - očrtana telesa in hierarhije očrtanih teles
    - BSP drevesa, osmiška drevesa ...
- Tovrstne tehnike niso del cevovoda na kartici, ampak jih izvedemo na CPE
  - na GPE pošljemo le geometrijo, ki je vidna



Rasterizacija

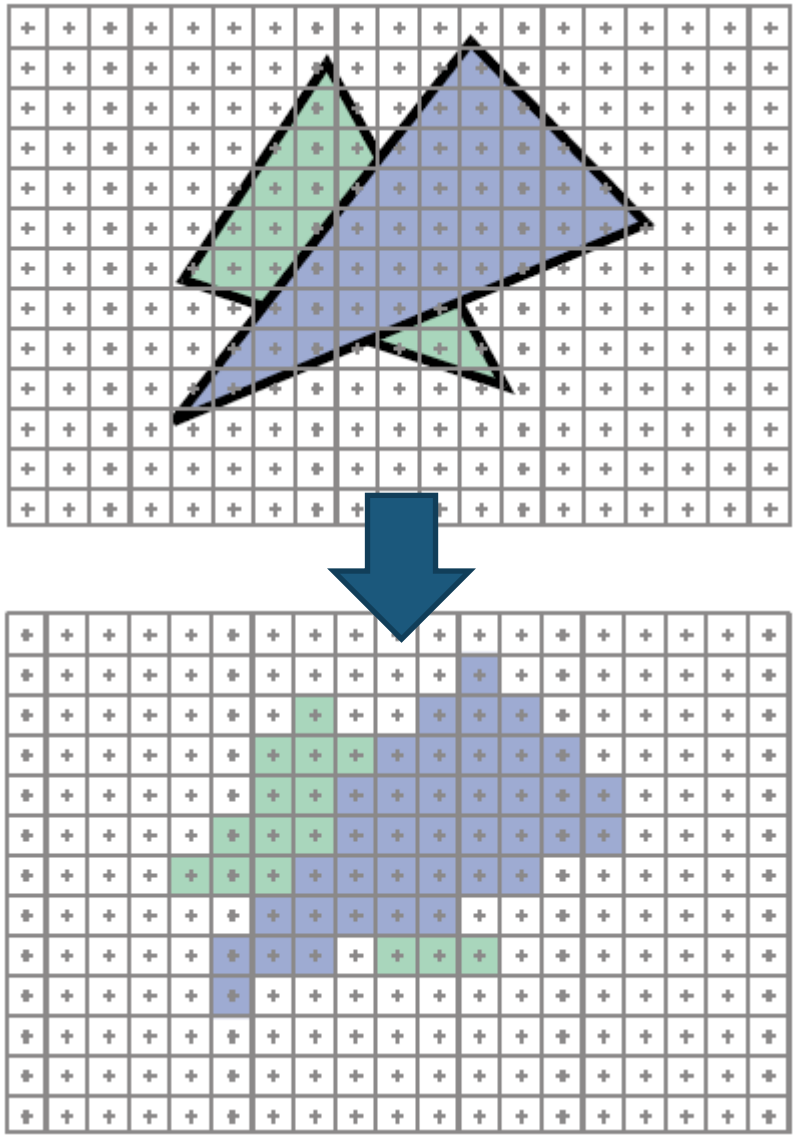
Rasterizacija

- Primitivi se pretvorijo v **diskretne elemente** (fragmente) glede na del končne slike, ki ga pokrivajo
  - najmanj en fragment na piksel
- Izvede se **interpolacija** lastnosti oglišč v posameznih fragmentih
- Nad posameznim fragmentom se izvede **senčilnik fragmentov**



Rasterizacija

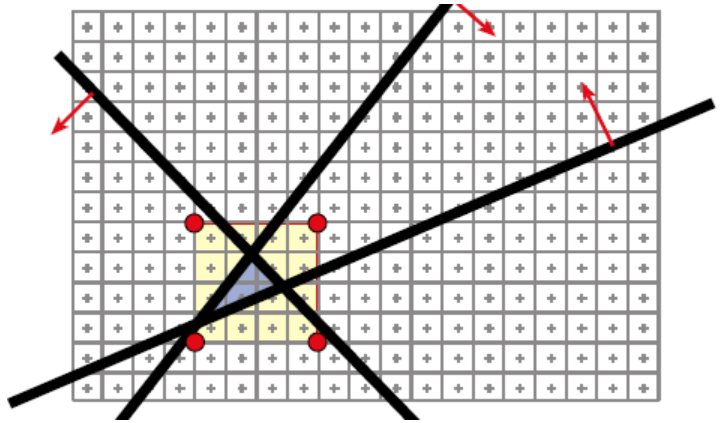
- Rasterska slika je 2D polje slikovnih elementov (pikslov) – **diskretno**
  - pikslov je končno mnogo
- Zaslonske koordinate predmetov so **zvezne**
- Rasterizacija: učinkovit izračun **pokritja pikslov**
  - katere piksle/fragmente pokriva poligon
  - interpolacija lastnosti oglišč v fragmentih



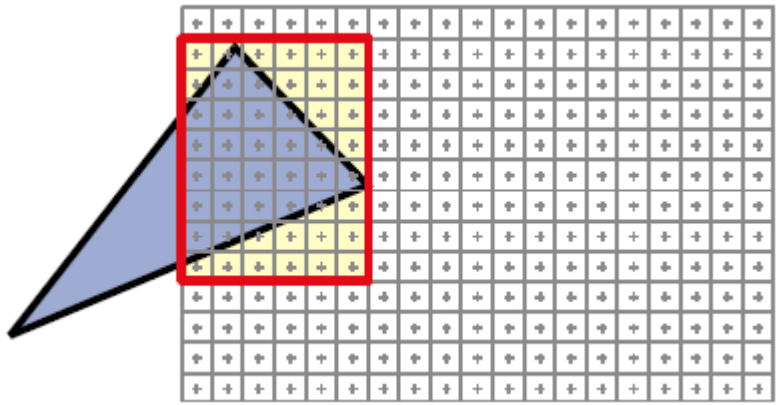
Kako poteka rasterizacija

- Za vsak piksel okoli trikotnika izračunamo, ali je znotraj trikotnika
  - uporabimo težiščne koordinate
  - računamo za piksle znotraj očrtanega pravokotnika
  - rezanje je enostavno
  - interpolacija lastnosti oglišč preko težiščnih koordinat je enostavna
- Algoritem s težiščnimi koordinatami:

```
bb = očrtan pravokotnik
for all pixels [x,y] in bb
  α,β,γ = težiščne koord.
  if 0 ≤ α,β,γ ≤ 1
   //piksel je v trikotniku
   interpoliraj
   izvedi senčilnik fragmentov
```

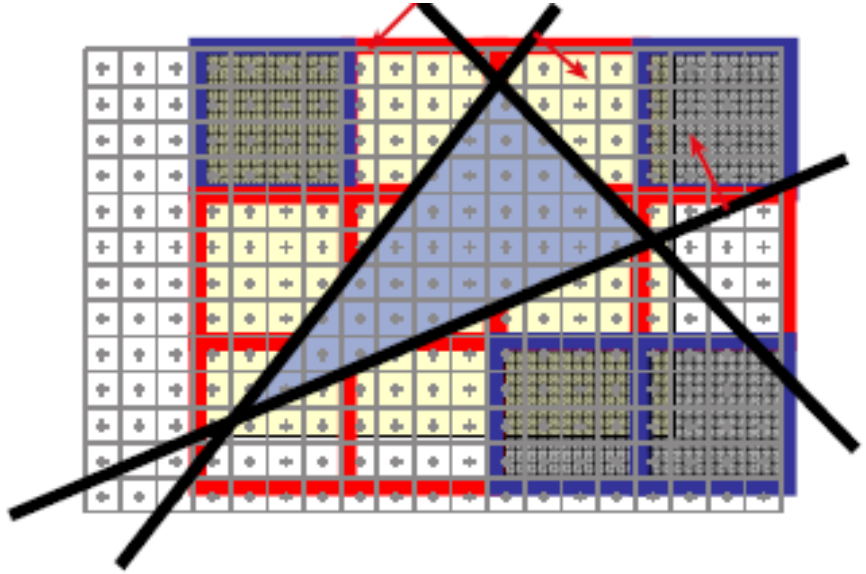


rezanje je enostavno:



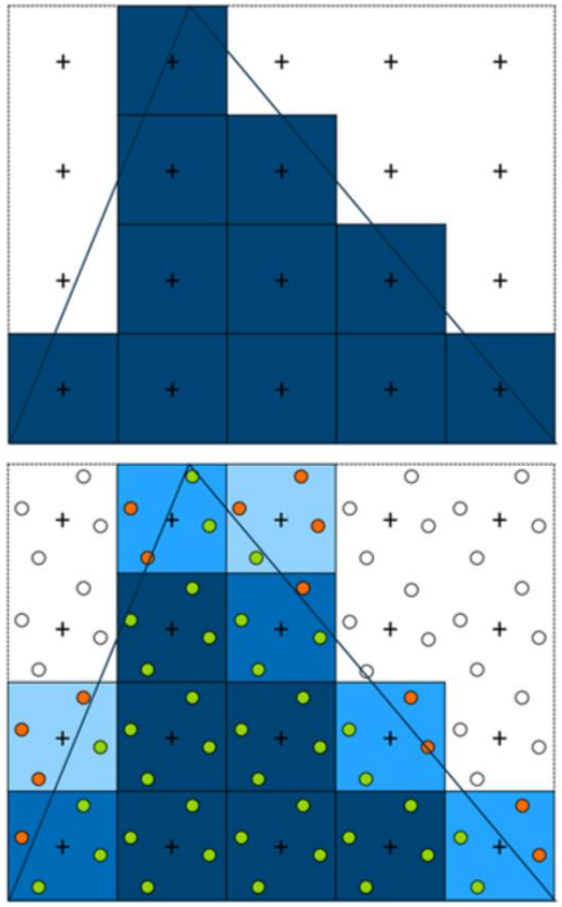
Rasterizacija

- Da ne računamo preveč nepotrebnih pikslov, očrtan pravokotnik lahko razbijemo na **podpodročja**
  - če je celo podpodročje izven trikotnika, ga izpustimo



Mehčanje robov

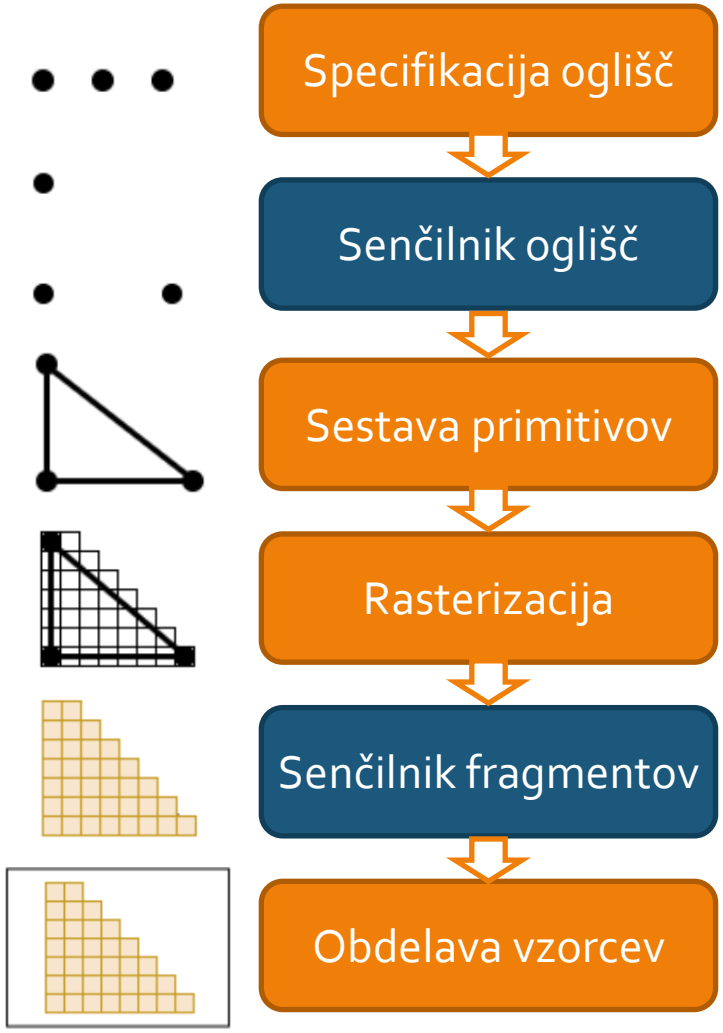
- Privzeto rasteriziramo en fragment na sredini piksla
  - to lahko vodi do ostrih prehodov na robovih – aliasing
- Robove lahko **mehčamo** z izbiro **več fragmentov** na piksel
  - *multisample antialiasing* (MSAA)
  - *multisample.count > 1*
- Postopek:
  - izračunamo **barvo** fragmentov **znotraj trikotnika**:
    - kot barvo **središča** piksla ali središča trikotnika v pikslu (*center, centroid*) – hitreje, 1x kličemo senčilnik fragmentov
    - barvo izračunamo posebej **za vsak** fragment (*sample*) – počasneje, za vsak fragment kličemo senčilnik fragmentov
  - končna barva piksla bo **povprečje** barv fragmentov



Vir: MSAA (Multisample anti-aliasing)

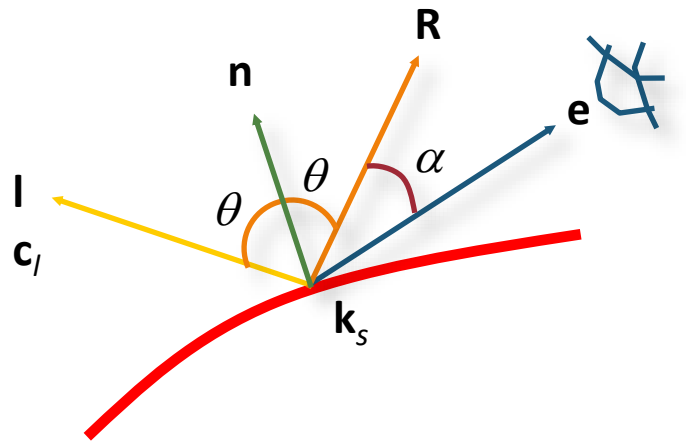
Senčilnik fragmentov

- Program, ki se izvede nad **fragmenti** trikotnika (seznam fragmentov je rezultat rasterizacije)
- Glavni namen je izračun **barve** fragmenta
  - iz teksture, osvetljevanje …
- Lahko pa modificira tudi globino, šablono (*stencil*) itn.



Enostaven primer – senčilnik fragmentov

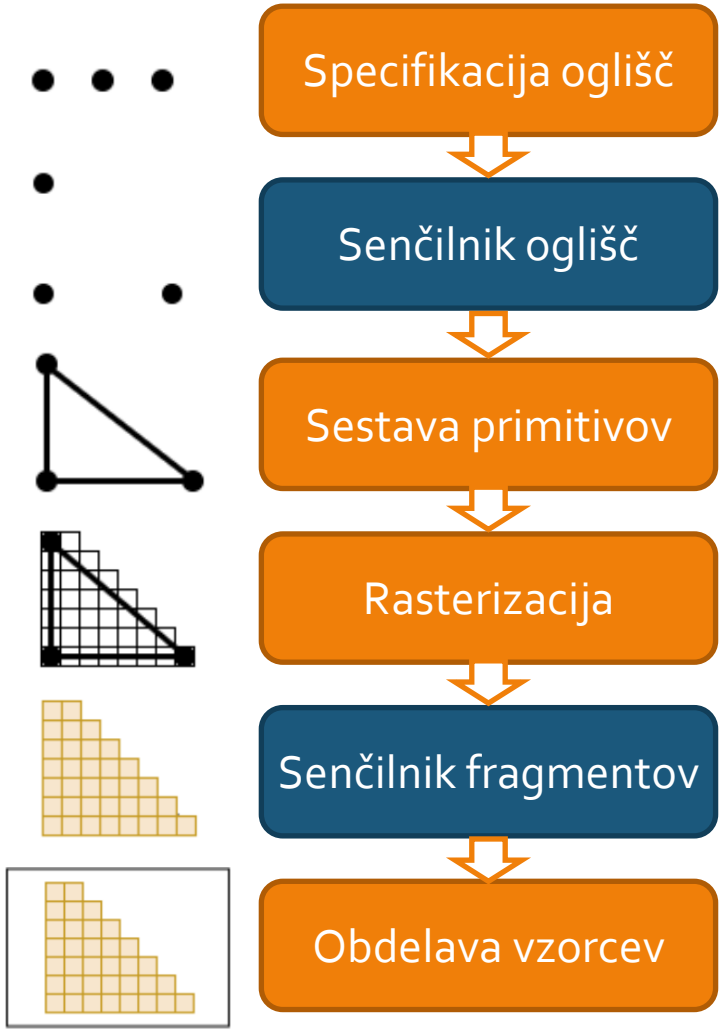
```
struct FragmentInput {
    @location(0) @interpolate(perspective) normal : vec3f,
    @location(1) @interpolate(perspective) fragPosition: vec4f,
}
struct FragmentOutput {
    @location(0) color : vec4f,
}

@fragment
fn fragment(input : FragmentInput) -> FragmentOutput {
    var output : FragmentOutput;

    var c: vec3f = vec3(1,1,1);
    var lightPos: vec3f = vec3(0,5,0);
    var l: vec3f = normalize(lightPos - input.fragPosition.xyz);

    var n: vec3f = normalize(input.normal);

    var kd: vec3f = vec3(1,0,1);
    var difIntensity = max(dot(n, l), 0.0);

    var ks: vec3f = vec3(1,1,1);
    var R: vec3f = reflect(-lightVec, n);
    var e: vec3f = normalize(-input.fragPosition.xyz);
    var specIntensity = pow(max(dot(R, e), 0.0), 20.0);

    var ka: vec3f = vec3(0.1,0.1,0.1);

    output.color = vec4(ka + c * kd* difIntensity + c * ks * specIntensity ,1);
    return output;
}
```



$$\mathbf{c}_a \mathbf{k}_a + \mathbf{c}\left(\mathbf{k}_d(\mathbf{l}\cdot\mathbf{n}) + \mathbf{k}_s(\mathbf{R}\cdot\mathbf{e})^p\right)$$

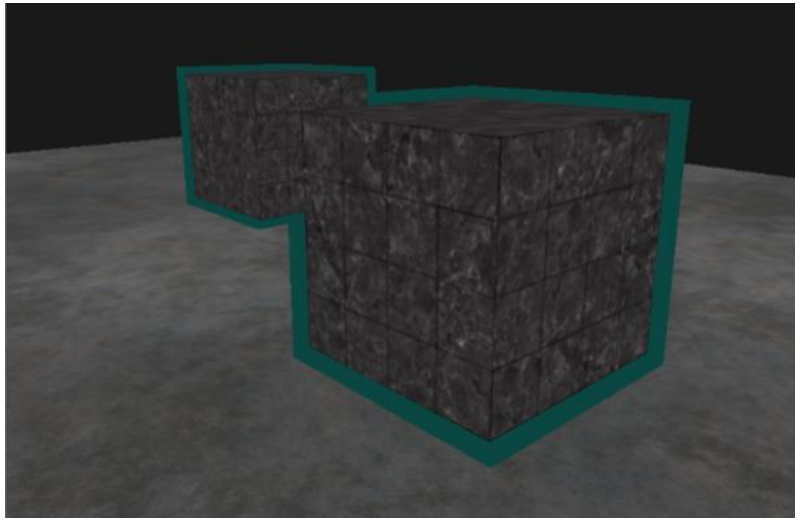
Obdelava vzorcev

- Sestavi končno sliko
  - test šablone (*stencil test*)
  - test globine (*depth test*)
  - mešanje (*blending*)
  - zapisovanje vrednosti



Test šablone

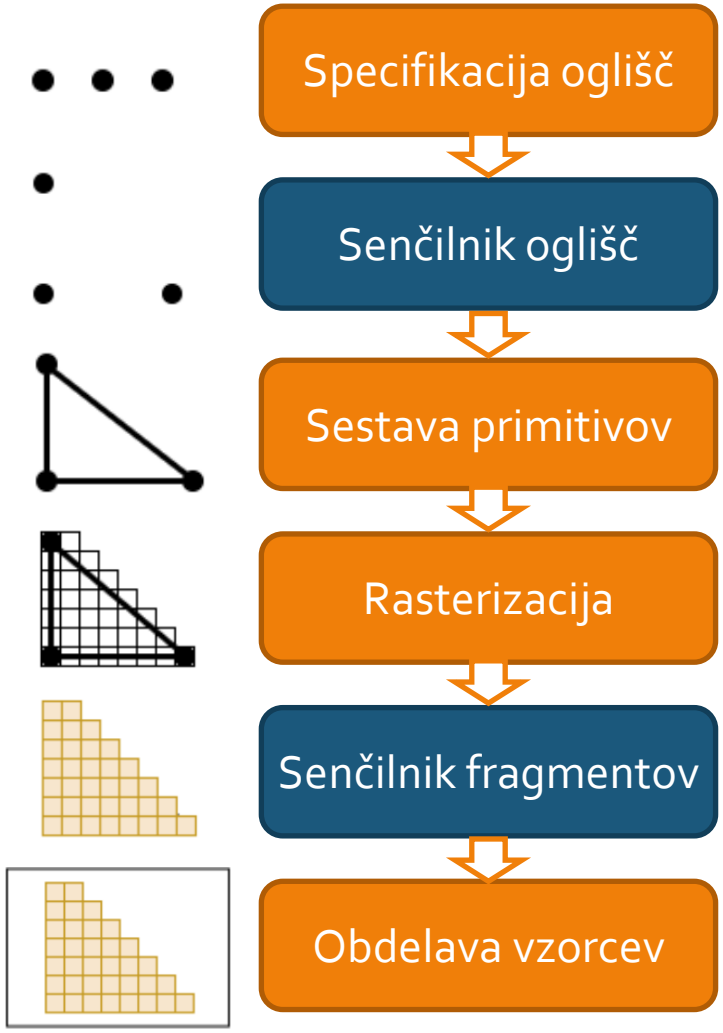
- *Stencil test*
- Vsak fragment ima lahko dodatno **vrednost šablone** (celo število)
- Vrednost se primerja s trenutno vrednostjo v **medpomnilniku šablone** (*stencil buffer*)
  - glede na definirano **operacijo** lahko fragment obdržimo ali zavržemo
    - <, >, =, <=, >=, !=
- Uporaba za omejitev izrisa na področje, sence, mehke prehode, poudarjanje robov itn.
- Primer: izris obrisa (*outline*) predmeta
  - predmet izrišemo in hkrati v medpomnilnik šablone zapišemo 1 za vse izrisane fragmente
  - predmet povečamo (skaliranje)
  - ponovno izrišemo predmet, tokrat z barvo obrisa, vendar le za vrednosti šablone, ki niso 1



Stencil testing

Test globine

- *Depth test*
  - določa vidnost (kateri fragmenti bodo vidni)
- Vsak fragment ima **globino**
  - z vrednost v koordinatah zaslona
    - lahko jo tudi spremenimo v senčilniku fragmentov
  - $z \in [0,1]$
- Na končni sliki navadno želimo videti fragmente, ki so najbližji kameri in zakrivajo fragmente za njimi

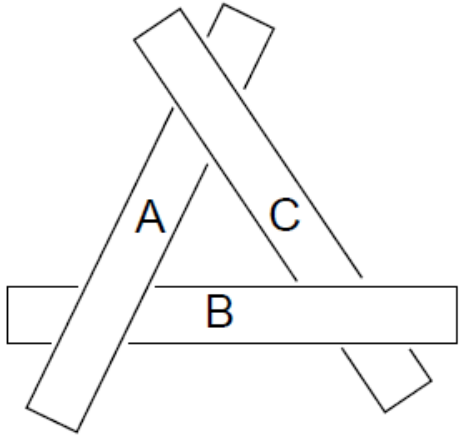


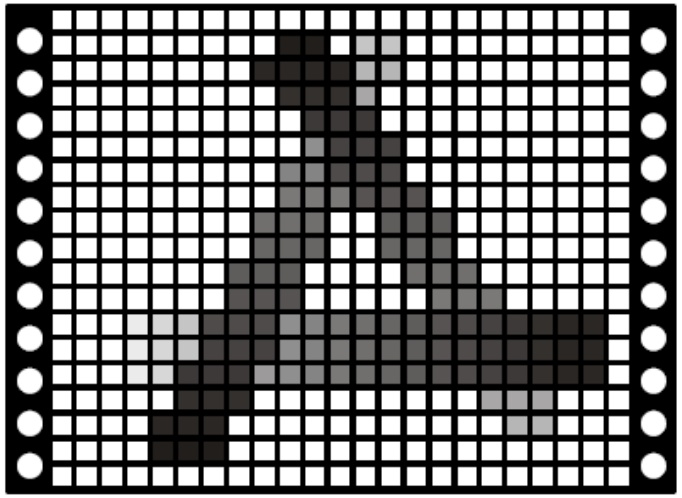
Depth buffer

- Kako vemo kateri fragment bo najbližji kameri?
- **Medpomnilnik globine** (*depth buffer, z buffer*)
  - za vsak **fragment** hranimo z vrednost (globino) trenutno najbližjega izrisanega fragmenta
- **Test globine** (*depth test*):
  - če je z vrednost fragmenta manjša od vrednosti v medpomnilniku, fragment ohranimo, sicer zavržemo
  - Poenostavljeno:

```
(x,y,z)=trenutni fragment
if z < depthBuffer[x,y]
   // fragment ohranimo
   depthBuffer[x,y]=z
else
   removeFragment // fragment zavržemo
```

- Operacijo primerjave lahko tudi zamenjamo
  - *depthStencil.depthCompare*





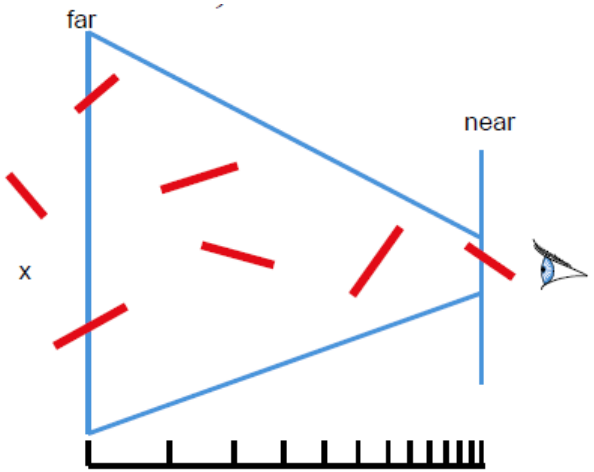
Kaj je že Z?

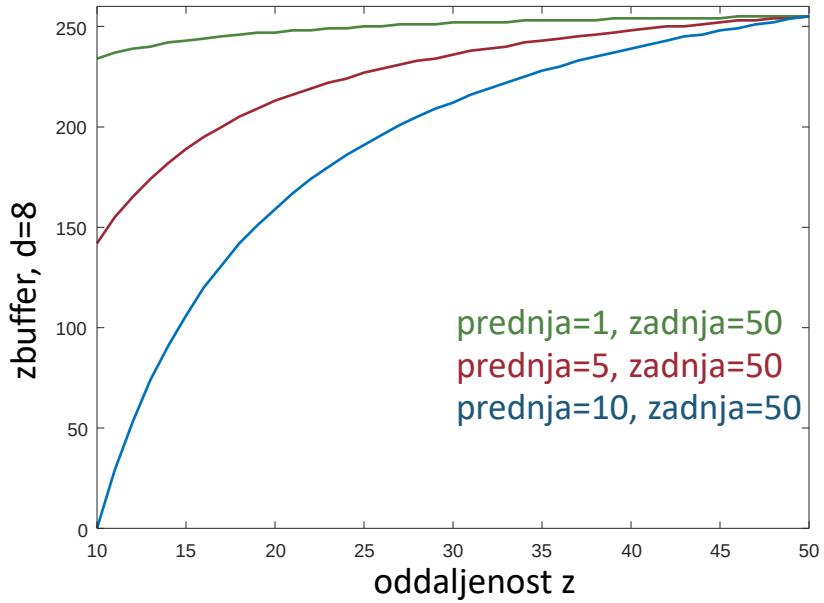
- Z v koordinatah zaslona je:
  - $z_v = \frac{f}{f-n} + \frac{fn}{f-n}\frac{1}{z}$
    - $z_v \in [0,1]$
- Znotraj trikotnikov interpoliramo med z v ogliščih, da dobimo globino v poljubnem fragmentu
- z ni linearno povezan z globino, $\frac{1}{z}$

$$M_p = \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{f}{n-f} & \frac{fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Test globine

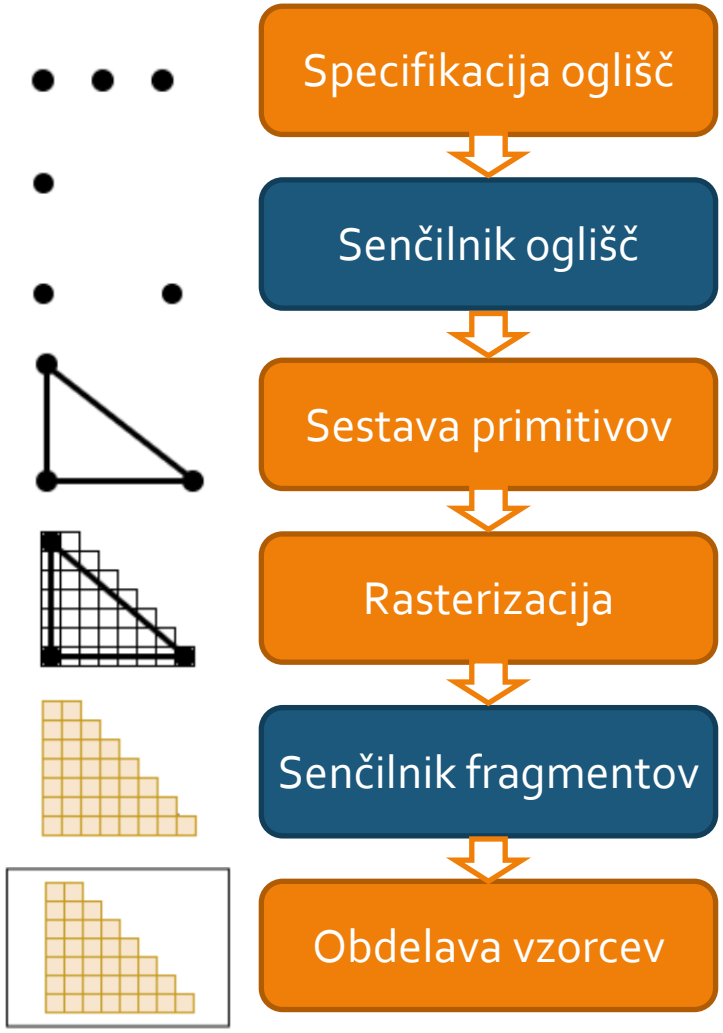
- $1/z$ da večji poudarek na bližnje predmete
  - medpomnilnik globine ima omejeno ločljivost (npr. 24 bitov)
  - če je prednja ravnina preblizu kameri, bomo porabili zelo velik del zaloge vrednosti za predstavitev bližnjih predmetov
  - posledica je, da imata 2 oddaljena predmeta lahko enako globino, kar pripelje do utripanja pri izrisu
    - ***z fighting***
    - naključno se najprej spredaj enkrat izriše en, drugič drug fragment predmeta
    - primer





Mešanje

- *Blending*
- **Mešanje barve** fragmenta (C_{src}) z barvo slike (C_{dst})
- Privzeto:
  - ko fragment preide test globine, se **prepiše**
    - $C_{dst} = C_{src}$
- Z *blend* lahko nastavimo različne načine kombiniranja barve slike in fragmenta:
  - $C_{dst} = \pm F_{src} C_{src} \pm F_{dst} C_{dst}$
    - F_{src} in F_{dst} lahko nastavimo na 0, 1, α_{src}, α_{dst}, $1 - \alpha_{src}$ ...



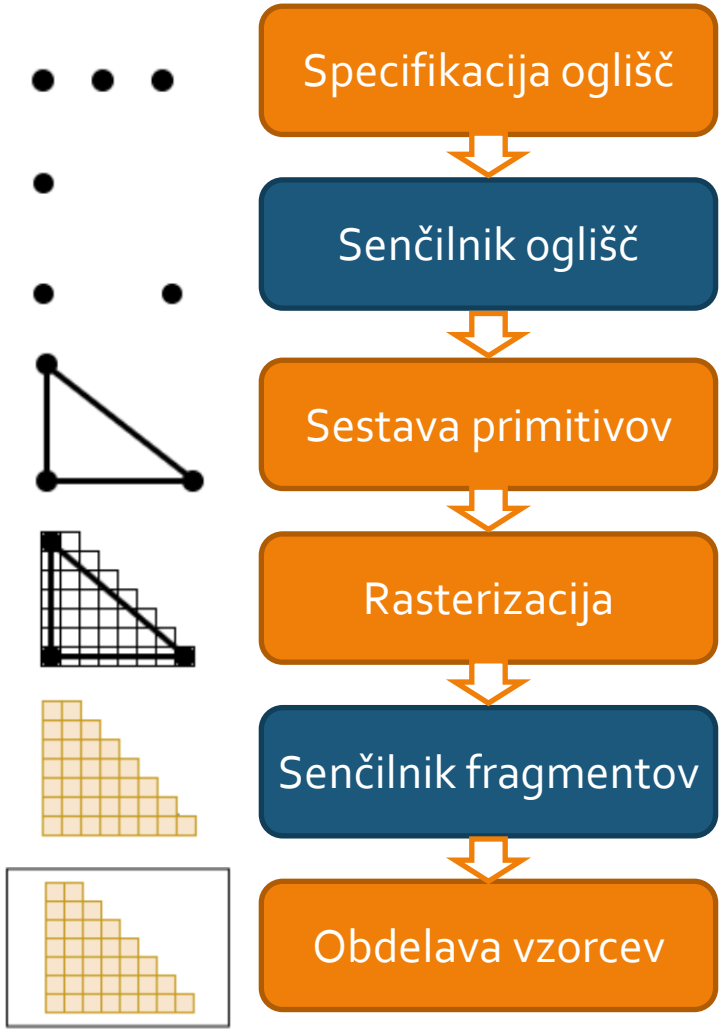
Mešanje

- Nastavitve mešanje lahko uporabimo za izris **prosojnih** predmetov
- **alfa mešanje**:
  - $C_{dst} = \alpha_{src} C_{src} + (1 - \alpha_{src}) C_{dst}$
  - $\alpha_{dst} = \alpha_{src} + (1 - \alpha_{src}) \alpha_{dst}$
- Za pravilen izris moramo izrisovati poligone od zadnjega proti prvemu!
  - sicer fragmenti ne preživijo nujno testa globine in ne pridejo do zlivanja

```
fragment: {
   module,
   entryPoint: 'fragment',
   targets: [{ format,
      blend: {
      // operation can be: "add", "subtract",
      // "reverse-subtract", "min", "max"
      // factor can be: "zero", "one", "src", "one-minus-src",
      // "src-alpha", "one-minus-src-alpha", "dst",
      // "one-minus-dst", "dst-alpha", "one-minus-dst-alpha",
      // "src-alpha-saturated", "constant", "one-minus-constant",
         color: {
            operation: 'add’,
            srcFactor: ‘src-alpha',
            dstFactor: ‘one-minus-src-alpha',
         },
         alpha: {
            operation: 'add',
            srcFactor: 'one',
            dstFactor: ‘one-minus-src-alpha',
         }
      }
   }],
},
```

Zapisovanje vrednosti

- Vrednosti se zapišejo v končne medpomnilnike
  - sliko (barva, alfa)
  - globinski medpomnilnik (nova globina)
  - medpomnilnik šablone (nova šablona)



REFERENCE

- WebGPU documentation
- C. Kubisch: Life of a Triangle – NVIDIA‘s logical pipeline
- Implementacija A-Bufferja: K. Myers, L. Bavoil: “Stencil Routed A-Buffer”, Nvidia
- NVidia GF-100 Whitepaper
- Rendering with conviction – Hierarhični Z Buffer